Case Studies in Program Design for Heterogeneous Parallelism

Editors: Emil Sekerinski[1] and Tian Zhang[2]

emil@mcmaster.ca, tianz@ca.ibm.com
[1]Department of Computing and Software, McMaster University
[2]Canada Research and Development Centre, IBM Canada

March 2014

McMaster University
Department of Computing and Software

Technical Report
CAS-14-03-ES

Contents

Introduction

This collection is the outcome of students projects of the graduate course CAS 766 Concurrent Programming at McMaster University in the fall of 2013. The projects are on *heterogeneous computing*, the execution of program on a combination of CPUs, FPGAs and GPUs. With processor frequencies having reached a ceiling (for a number of years already!) and power consumption limiting the complexity of single (multi-core) CPUs, heterogeneous computing has the promise of speeding up computations through parallelism while reducing power consumption. The last point is essential for everything from mobile devices to server farms. However, heterogeneous computing requires a different programming model than the traditional models of shared variables and message passing.

The course had three parts. To start with, the instructor covered classic concepts of concurrency, like atomicity, synchronization, safety, liveness, as well as programming with semaphores and monitors. The second part was on design patterns for parallelism. For this, participants were asked to give a seminar on parts of *Structured parallel programming: patterns for efficient computation* by McCool, Robison, and Reinders [MRR12]:

1. Adam Lazzarato, *Chapter 2: Background*
2. Yihao Fang, *Chapter 3: Patterns*
3. Misha Virk, *Chapter 4: Map*
4. Ishwaree Argade, *Chapter 5: Collectives*
5. Robert Fuller, *Chapter 6: Data Reorganization* and *Chapter 7: Stencil and Recurrence*
6. Apurva Kumar, *Chapter 8: Fork-Join*
7. Sun Yue, *Chapter 9: Pipeline*

In the third part, students completed an implementation using Lime, a heterogeneous computing language being developed by IBM Research. A difficulty with FPGAs and GPUs is that the languages needed for programming those, e.g. Verilog and VHDL, are very different from those used to program CPUs and interfacing involves distracting technicalities. Furthermore, designs may not work well across different FPGA and GPU chips. By contrast, the Lime language gives programmers a uniform language for

CPU and FPGA programming; the technicalities of mapping to a specific processor and of interfacing are taken over by the compiler. As an extension of Java, Lime supports the whole development process, starting with an object-oriented design. At the time of writing, Lime supports FPGAs better than GPUs.

Tian Zhang gave guest lectures for introducing students to Lime. Students were given the freedom to select a problem to be implemented in Lime and were encouraged to consider one they are familiar with. However, students were asked to argue why the problem is particularly suited for being expressed with the Lime-specific constructs for parallelism. The list of selected topics is indeed remarkable! At the end of the course, each student presented their chosen problem and their implementation. Due to technical issues, a machine with FPGA acceleration was not available before the end of the course, so all code was tested on a CPU only and no measurements of efficiency could be taken.

Lime is not yet available to the public. It was made available to students of this course by IBM Thomas J. Watson Research Center, Yorktown Heights, NY., which is gratefully acknowledged. For the SOSCIP project[1], this collection helps in further research on effective design patterns for heterogeneous computing.

The first chapter, by Tian Zhang, was used in the course as a brief introduction to Lime; it also contains examples of how to use the six parallel design patterns from McCool, Robison, and Reinders [MRR12] in Lime. The remaining chapters are the reports of the seven student projects, each applying one or more of the six parallel patterns.

Emil Sekerinski
Tian Zhang

Bibliography

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier/Morgan Kaufmann, Amsterdam, 2012.

[1] `http://www.soscip.org/`

1 Lime in a Nutshell

Tian Zhang

1.1 What is Lime?

Lime is a heterogeneous computing language developed by IBM Research. It is designed to be executable across a broad range of architectures, including CPUs, GPUs, and FPGAs. The novel features of Lime include limiting side-effects and integration of the streaming paradigm into an object-oriented language.

Lime extends Java 1.6 as specified by [GJSB05]. As a result, all Java programs either are legal Lime programs, or can be converted to legal Lime programs with minor changes. There are 13 new keywords in Lime:

```
1 value,universal,var,typedef,extendedby,task,split,join,message,sends,local,global,glocal
```

The meaning and usage of some new keywords will be discussed in later sections. Below is a trivial Lime "Hello World" program, which is identical to its Java version:

<src/intro/HelloWorld.lime 1>

```
1 package intro;
2
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         System.out.println("hello, world!");
7     }
8
9 }
```

1.2 Some Key Features for Parallelism

- *Value types*: immutability
- *Ordinals*: finite non-negative integral types
- *Bounded arrays* (indexed by ordinals) and value arrays
- *Ranges*: for iteration over subranges of value types with ordering operations

Here is an example illustrating these features:

- Double brackets "[[" and "]]" denote *immutable* arrays
- Double colon "::" denotes range

<src/intro/Key.lime 2>

```
package intro;

import lime.util.Tasks;

public class Key {

    public static void main(String[] args) {
        int[[3]] d = { (3) 1 }; // repeats integer 1 three times
        int[] a = { 1, 2, 3 };
        for (int i: 0::2) { // from 0 to 2
            a[i] = a[i] + d[i];
        }
        System.out.print(a.contentToString());
    }

}
```

1.3 "Hello World"s

1.3.1 A "Hello World" That Uses MapReduce Pattern

The words are defined as an array of strings:

- *Lime string* "words" instead of Java String

<Words Definition 3>

```
        string [[]] words = { "hello", "you", "beautiful", "world", "!" };
```

USED IN: src/intro/HelloWorld2.lime on page 5

Insertion of spaces, concatenation, and output:

- *Map operation*: "words @+ #" "" applies the "+" operator (here string concatenation) to every element of the array "words" using scalar operator "#", with " " as the second operand, yielding a new array { "hello ", "you ", "beautiful ", "world ", "! " }, which has all the original strings with a space appended to each one
- *Reduction*: the "!" meta-operator indicates the "+" string concatenation operator is applied to the elements of the string array, producing a single string "hello you beautiful world ! "

<Concatenate and Output 4>

```
1        System.out.println(+ !  (words @+ #" "));
```

USED IN: src/intro/HelloWorld2.lime on page 5

The main program:

- The Lime strings array "args" is also immutable

<src/intro/HelloWorld2.lime 5>

```
1  package intro;
2
3  class  HelloWorld2 {
4
5      public static  void main(string[[]]  args)  {
6          <<Words Definition 3>>
7          <<Concatenate and Output 4>>
8      }
9
10 }
```

INCLUDED BLOCKS: 3 on page 4, 4 on page 5

1.3.2 A "Hello World" That Uses *Stream Computation*

The constant of the difference between upper and lower case letters:

<Case Difference Definition 6>

```
1      static  final  char caseDifference = ('a'  − 'A');
```

USED IN: src/intro/HelloWorld4.lime on page 7

The conversion to lower case:

- *Conditional expression* "b ? x : y"

<Conversion to Lower Case 7>

```
1  static local char toLowerCase(char c) {
2      return ('A' <= c && c <= 'Z') ? (char)(c + caseDifference) : c;
3  }
```

USED IN: src/intro/HelloWorld4.lime on page 7

The message as an immutable array of characters:

<Message Definition 8>

```
1      char[[]] msg = { 'H','E','L','L','O',',',' ',
2                       'W','O','R','L','D','!','\n' };
```

USED IN: src/intro/HelloWorld4.lime on page 7

Here "hello" is defined as a *task*:

- The **var** keyword is for local *type inference*
- The array "msg" is taken as the *source*, the elements of which will be emitted in order
- Keyword **task** is used to create tasks from methods "toLowerCase" and "System.out.print(char)"
- *Ensure brackets* "([" and "])" denote FPGA acceleration
- Arrow "=>" is used to *connect* tasks
- The task from "System.out.print(char)" is a *sink* since it does not produce further value for stream computation
- Inline comments for annotating the type of stream elements

<Output Task Declaration 9>

```
1      var hello = Tasks.source(msg)
2                  // char
3                  => ([ task toLowerCase ])
4                  // char
5                  => task System.out.print(char);
```

USED IN: src/intro/HelloWorld4.lime on page 7

Invocation of the method “finish()” starts the execution of task “hello”:

<Output Task Execution 10>

```
1       hello.finish();
```

USED IN: src/intro/HelloWorld4.lime on page 7

The main program:

<src/intro/HelloWorld4.lime 11>

```
1  package intro;
2
3  import lime.util.Tasks;
4
5  public class HelloWorld4 {
6
7      <<Case Difference Definition 6>>
8
9      <<Conversion to Lower Case 7>>
10
11     public static void main(String[] args) {
12         <<Message Definition 8>>
13         <<Output Task Declaration 9>>
14         <<Output Task Execution 10>>
15     }
16
17 }
```

INCLUDED BLOCKS: 6 on page 5, 7 on page 6, 8 on page 6, 9 on page 6, 10 on page 7

1.4 Merge Sort with Acceleration

Input as an immutable array of integers:

<Test Case 12>

```
1      static final int[[16]] data = {
2          503, 87, 512, 61, 908, 170, 897, 275, 653, 426, 154, 509, 612, 677, 765, 703
3      };
```

USED IN: src/intro/MergeSort.lime on page 10

Recursive sort:

- Keyword **split** denotes a *splitter* that distributes its input to multiple tasks, in a round-robin style
- Keyword **join** denotes a *joiner* that aggregates output streams from multiple tasks into a single new stream, in a round-robin style
- A "#" operator denotes a match expression

<Sort 13>

```
1   task local static LFilter<int [[]], int [[]]> sort(int N) {
2       if (N > 2) {
3           LFilter<int, int> mergesort = (int # int [[]], size N / 2)
4                                         => sort(N / 2)
5                                         => (# int);
6
7           return (int [[]] #)
8               => task split '(int, int)
9               // '( int [[N / 2]], int [[N / 2]])
10              => task [ mergesort, mergesort ]
11              // '( int [[N / 2]], int [[N / 2]])
12              => task join '(int, int)
13              => (# int [[]], size N)
14              // int [[N]]
15              => task merge(int[[]]);
16      }
17      else {
18          return task merge(int[[]]);
19      }
20  }
```

USED IN: src/intro/MergeSort.lime on page 10

Merging the two sorted sub-arrays indexed by even and odd integers respectively (as the result of joiner) into a single sorted array:

<Merge 14>

```
1   local static int [[]] merge(int[[]] input) {
2       int i = 0;
3       int j = 1;
4       int n = input.length;
5       int c = 0;
```

```
6           int[] result = new int[n];
7
8           while (i < n && j < n) {
9               int lv = input[i];
10              int uv = input[j];
11              if (lv < uv) {
12                  result[c++] = lv;
13                  i += 2;
14              }
15              else {
16                  result[c++] = uv;
17                  j += 2;
18              }
19          }
20
21          int r = i < n ? i : j;
22          while (c < n) {
23              result[c++] = input[r];
24              r += 2;
25          }
26
27          return new int[][]( result );
28      }
```

USED IN: src/intro/MergeSort.lime on page 10

Output the result of merge sort with the result of sort method in the standard library, and compare them:

<Verification 15>

```
1       static void verify(int [[]] data) {
2           int[] sortedData = new int[](data);
3           sortedData.sort();
4           System.out.println("result: " + data.contentToString());
5           System.out.println("expect: " + sortedData.contentToString());
6           if (data.contentEquals(sortedData))
7               System.out.println("TEST PASSED");
8           else
9               System.out.println("TEST FAILED");
10      }
```

USED IN: src/intro/MergeSort.lime on page 10

Construct the task for merge sort and execute:

- The "#" operator alone aggregates the integers into an array

<Run Merge Sort 16>

```
1        var ms =  Tasks.source(data)
2                => #
3                => ([  sort(data.length)  ])
4                => task verify;
5        ms.finish();
```

USED IN: src/intro/MergeSort.lime on page 10

The main program:

<src/intro/MergeSort.lime 17>

```
1  package intro;
2
3  import lime.util.Tasks;
4
5  public class  MergeSort {
6
7      <<Test Case 12>>
8
9      <<Verification  15>>
10
11     <<Sort 13>>
12
13     <<Merge 14>>
14
15     public static  void main(String[] args)  {
16         <<Run Merge Sort 16>>
17     }
18
19 }
```

INCLUDED BLOCKS: 12 on page 7, 15 on page 9, 13 on page 8, 14 on page 8, 16 on page 10

1.5 Pattern Examples

In this section we show some concrete examples of how to use the six parallel design patterns from book [MRR12] in Lime. We start with some predefined data and opera-

tions:

<src/patterns/Predefined.lime 18>

```
package patterns;

public class Predefined {

    static final int N = 3;

    static final int[[N]] row = { 1, 2, 3 };

    static final int L = 8;

    static final int[[L]] longRow = { 1, 2, 3, 4, 5, 6, 7, 8 };

    static final int delta = 1;

    static local int inc(int i) {
        return i + delta;
    }

    // outputs an integer array
    static void output(int[] a) {
        System.out.println(a.contentToString());
    }

    // outputs an integer value array
    static void outputV(int[[]] a) {
        System.out.println(a.contentToString());
    }

}
```

1.5.1 Fork-Join Pattern

The fork-join pattern lets control flow fork into multiple parallel flows that rejoin later [MRR12]. In Lime it can easily be implemented by using splitter and joiner:

<src/patterns/ForkJoin.lime 19>

```
package patterns;

import lime.util.Tasks;

```

```
5  public class ForkJoin {
6
7      public static void main(String[] args) {
8
9          // serial version
10         var t1 = Tasks.source(Predefined.row)
11                 // int
12                 => ([ task Predefined.inc ])
13                 // int
14                 => (# int[[Predefined.N]])
15                 // int[[ Predefined.N]]
16                 => task Predefined.outputV(int[[]]);
17         t1.finish();
18
19         // N-parallel version
20         var t2 = Tasks.source(Predefined.row)
21                 // int
22                 => #
23                 => task split int[[Predefined.N]]
24                 // '(int, int, ..., int)
25                 => ([ task [ (Predefined.N) task Predefined.inc ] ])
26                 // '(int, int, ..., int)
27                 => task join int[[Predefined.N]]
28                 // int[[ Predefined.N]]
29                 => task Predefined.outputV(int[[]]);
30         t2.finish();
31
32     }
33
34 }
```

1.5.2 Map Pattern

In a map pattern, a function is applied to all elements of a collection, usually producing a new collection with the same shape as the input [MRR12]. The elementary function must be pure (without side effects).

<src/patterns/Map.lime 20>

```
1  package patterns;
2
3  public class Map {
4
5      public static void main(String[] args) {
```

```
6
7           // for loop version
8           int[] r = new int[](Predefined.row);
9           for (int i: 0::Predefined.N − 1) {
10              r[i] = r[i] + Predefined.delta;
11          }
12          System.out.println(r.contentToString());
13
14          // Lime map version
15          Predefined.outputV(Predefined.row @+ #Predefined.delta);
16
17      }
18
19  }
```

1.5.3 Reduction Pattern

A reduction combines all the elements in a collection into a single element using an associative combiner function. Because the combiner function is associative, many orderings are possible [MRR12].

<src/patterns/Reduction.lime 21>

```
1   package patterns;
2
3   public class Reduction {
4
5       public static void main(String[] args) {
6
7           // Lime reduction version
8           Predefined.outputV(Predefined.row);
9           System.out.println(+ ! Predefined.row);
10
11          // Lime map & reduction version
12          Predefined.outputV(Predefined.row @+ #Predefined.delta);
13          System.out.println(+ ! (Predefined.row @+ #Predefined.delta));
14
15      }
16
17  }
```

1.5.4 Recurrence Pattern

A recurrence is like a map but where elements can use the outputs of adjacent elements as inputs [MRR12]. Here we show how iteration can be done in this style.

<src/patterns/Recurrence.lime 22>

```
1  package patterns;
2
3  import lime.util.Tasks;
4
5  public class Recurrence {
6
7      // recursively construct the task
8      task static local LFilter<int, int> incMultiple(int i) {
9          if (i > 1) {
10             return task Predefined.inc
11                 => incMultiple(i - 1);
12         }
13         else {
14             return task Predefined.inc;
15         }
16     }
17
18     public static void main(String[] args) {
19
20         // for loop version
21         int[] r = new int[](Predefined.row);
22         for (int i: 0::Predefined.N - 1) {
23             r[i] = r[i] + Predefined.N;
24         }
25         Predefined.output(r);
26
27         // recurrence task version
28         var t = Tasks.source(Predefined.row)
29                 => ([ incMultiple(Predefined.N) ])
30                 // int
31                 => (# int[[Predefined.N]])
32                 // int [[ Predefined.N]]
33                 => task Predefined.outputV(int[[]]);
34         t.finish();
35
36     }
37
38 }
```

1.5.5 Scan Pattern

Scan computes all the partial reductions of a collection. For every output position, a reduction of the input up to that point is computed [MRR12]. This example is similar to the merge sort one, but added reordering.

<src/patterns/Scan.lime 23>

```
1  package patterns;
2
3  import lime.util.Tasks;
4
5  public class Scan {
6
7      // reorder the array before split
8      local static int[[]] reorder(int[[]] input) {
9          int[] i = new int[](input);
10         int m = input.length / 2;
11         for (int j: 0::m - 1) {
12             i[2 * j] = input[j];
13             i[2 * j + 1] = input[j + m];
14         }
15         return new int[[]](i);
16     }
17
18     // reorder the array after join
19     local static int[[]] restore(int[[]] input) {
20         int[] i = new int[](input);
21         int m = input.length / 2;
22         for (int j: 0::m - 1) {
23             i[j] = input[2 * j];
24             i[j + m] = input[2 * j + 1];
25         }
26         return new int[[]](i);
27     }
28
29     // merge the two scanned parts
30     local static int[[]] merge(int[[]] input) {
31         int[] i = new int[](input);
32         int m = i.length / 2;
33         // the second half increased by i[m - 1]
34         i[m::2 * m - 1] = i[m::2 * m - 1] @+ #i[m - 1];
35         return new int[[]](i);
36     }
37
38     task local static LFilter<int[[]], int[[]]> scan(int N) {
```

```
39          if (N > 2) {
40              LFilter<int, int> subscan = (int # int [[]], size N / 2)
41                                          => scan(N / 2)
42                                          => (# int);
43
44              return ([ task reorder ])
45                  // int [[N]]
46                  => (int [[]] #)
47                  => task split `(int, int)
48                  // `( int , int )
49                  => task [ subscan, subscan ]
50                  // `( int , int )
51                  => task join `(int, int)
52                  => (# int [[]], size N)
53                  // int [[N]]
54                  => ([ task restore ])
55                  => (# int[[N]])
56                    // int [[N]]
57                  => task merge(int[[]]);
58          }
59          else {
60              return task merge(int[[]]);
61          }
62      }
63
64      public static void main(String[] args) {
65
66          // scan
67          var t = Tasks.source(Predefined.longRow)
68                  // int
69                  => (# int[[Predefined.L]])
70                  // int [[ Predefined .L]]
71                  => ([ scan(Predefined.longRow.length) ])
72                  // int [[ Predefined .L]]
73                  => task Predefined.outputV;
74          t.finish();
75      }
76
77  }
```

1.5.6 Stencil Pattern

The stencil pattern is a generalization of the map pattern where the elemental function can access not only a single element in an input collection but also a set of "neighbors". Here shift in stream is used to insert elements' neighbors with themselves.

<src/patterns/Stencil.lime 24>

```
1  package patterns;
2
3  import lime.util.Tasks;
4
5  public class Stencil {
6
7      // compute the average and increase by delta
8      static local int avgInc(int[[Predefined.N]] a) {
9          return (+ ! a) / Predefined.N + Predefined.delta;
10     }
11
12     // add the head and tail back in the array
13     static local int[[Predefined.L]] restore(int[[Predefined.L - 2]] a) {
14         return Predefined.longRow[0::0] + a + Predefined.longRow[Predefined.L - 1::
               Predefined.L - 1];
15     }
16
17     public static void main(String[] args) {
18
19         // create the shifted source
20         LSource<int[[Predefined.N]]> shifted = Tasks.source(Predefined.longRow)
21                                                 => (# int[[Predefined.N]], shift 1);
22
23         // stencil
24         var t = shifted
25                 // int
26                 => ([ task avgInc ])
27                 // int
28                 => (# int[[Predefined.L - 2]])
29                 // int[[L - 2]]
30                 => task restore
31                 // int[[L]]
32                 => task Predefined.outputV(int[[]]);
33         t.finish();
34
35     }
36
37 }
```

Bibliography

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java language specification, thrid edition. `http://docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf`, May 2005.

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier/Morgan Kaufmann, Amsterdam, 2012.

2 Machine Learning: Clustering

Ishwaree Argade

2.1 Introduction

Machine learning which is a part of artificial intelligence is closely related to the area of data mining. Machine learning focuses mainly on prediction of knowledge based on known properties from the data while data mining focuses on discovery of knowledge by finding its unknown properties from a dataset [CFZ09]. This report presents an introduction to machine learning and its categories [Mit97]. One area where data mining algorithms are used in machine learning is unsupervised learning which performs knowledge discovery [Gha04]. The focus of this report is mainly on the topic of clustering, which is an example of unsupervised learning. Clustering is a process of grouping similar data points together based on some similarity or dissimilarity matrix. It is widely used in many different areas and helps to improve the performance of the applications [JMF99]. Clustering algorithms have three main categories, namely partitional clustering, hierarchical clustering and model based clustering [CSPK09]. This report discusses these categories along with metrics used for clustering followed by some applications of clustering and overview of their parallel implementation. It further explains K-means clustering algorithm along with the approach to parallelize it [ZMH09]. Final section of presents an implementation of the K-means algorithm using parallel patterns in Lime [ABCR10]. The implementation parallelizes distance computations involved in the algorithm using the reduction pattern in Lime and then sequentially executes the iterative phase of the algorithm. This is the most common approach used to parallelize K-means algorithm as it involves a lot of distance computations which can be executed independently.

Machine Learning

Machine learning is the construction and study of systems that can learn from data and is a branch of artificial intelligence. The formal definition of machine learning is provided by Tom M. Mitchell [Mit97].:
A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.
This means that a system can be trained in such a way that it learns from its experience and then derives certain rules or patterns related to the behaviour of the system which are later used by the system. Some examples of machine learning systems include email spam filtering systems which identifies spam and non-spam messages, optical character recognition in which printed characters are recognized automatically by learning previous examples and a system to recognize spoken words [WYB+10].

One important property of machine learning is *generalization*. It refers to the ability of the system to learn from its experience, say a set of task T and use this experience to perform accurately against a set of new unseen tasks. Next characteristic of machine learning is representation which handles representation, of data instances and functions evaluated on those. The representation property decides the training dataset for the system and the learner has to build a general model which would enable the machine to produce accurate predictions in future unseen cases.

Machine learning and data mining are closely related to each other. Machine learning focuses on predictions for unseen tasks when known properties about the data learned from previous experiences are given whereas data mining focuses on discovery of knowledge like unknown patterns or rules on a data set which can be further analyzed. So, data mining process is evaluated by the ability to produce unknown knowledge discovery of data and machine learning is evaluated by the ability to reproduce the known knowledge. However, machine learning uses many data mining algorithms in one of its category called as "unsupervised learning" [CFZ09]. Machine learning categories are described as follows.

1. *Supervised Learning:* The task of inferring a function by analyzing a training data set. The training data contains training examples. Each training example accepts an input value and expects a desired output value. The learning algorithm in this case produces generalization which is used for unseen cases [Mit97].
2. *Unsupervised Learning:* Tries to find hidden structure of the data. It is the task of finding some patterns related to the data that were previously unknown. Unlike supervised learning, some desired output is not known, instead it is derived [Mit97].
3. *Reinforcement Learning:* An area of machine learning inspired by behaviourist psychology. It includes the study of behaviour of software agents in an environment

which can provide the agent some rewards or punishments. The goal is to train the machine to learn in order to maximize the amount of future rewards. The problems from various areas like game theory, control theory, operations research, information theory, decision theory are considered under reinforcement learning [Mit97].

4. *Learning to learn:* Learns the problem together with other related problems at the same time using a shared representation. This is also called multi-task learning. It emphasizes more on the main task learning model which finds the commonalities between the tasks. One example of this is spam filter [Mit97].
5. *Transduction:* A reasoning performed on specific training and test cases. New outputs are predicted on specific training and test cases [Mit97].

Unsupervised Learning

This area of machine learning highly overlaps with data mining and hence it tends to use many data mining algorithms in this area. As mentioned above, unsupervised learning tries to find the hidden structure of the data. A set of inputs $x_1, x_2, ...x_n$ is supplied to the machine. However, no outputs are supplied. This means that the machine doesn't get feedback from its environment in any form of reward. The machine has to analyze the data and then provide the representations of the input that can be used for decision making, predicting future inputs and effectively providing inputs to some other machine. In other words, unsupervised learning is a process of finding patterns in unstructured and possibly noisy data in order to increase its usefulness for future tasks. Unsupervised learning provides a probabilistic model of the data. Given n inputs without desired output, the learning system can develop a probabilistic model for a new input. Some examples of data are supermarket data where sample data can be analyzed to identify some patterns which are useful to determine trends, stock market data, weather data, business data etc. Unsupervised learning also includes many other techniques which summarize and explain key features of the data. Clustering, hidden Markov models, dimensionality reduction are some key examples of unsupervised learning. However, we are going to focus on the clustering aspect of unsupervised learning in later sections [Gha04].

2.2 Clustering

Collection of large data is commonly used for analysis today. So, clustering provides a conceptual and algorithmic framework for data analysis and interpretation. Clustering imposes abstraction by discovering structures in collection of data. Clustering algorithms determine the similarities and differences between the data samples in a dataset and accordingly classifies them in a cluster. Data samples in one cluster are "close" to each other

and "far" from samples in other clusters. Thus, clustering is a process of grouping the relevant data in one cluster to make its analysis and interpretation easy. Various algorithms follow various methods of finding similarities between the data samples. We are going to see categories of clustering algorithms in next section [CSPK09].

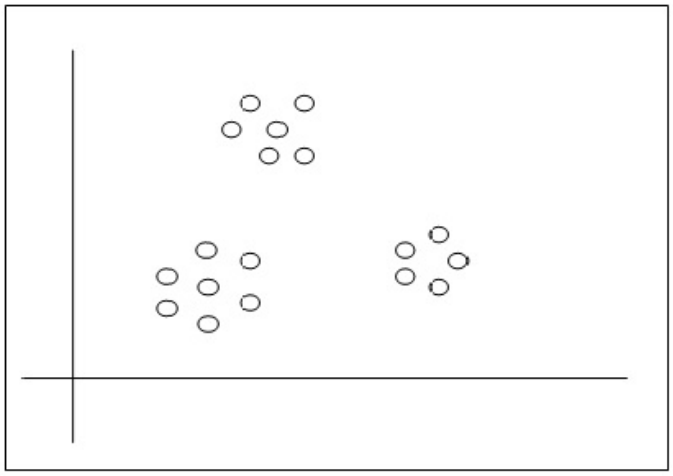


Figure 2.1: Clustering Example in 2D Space - 3 Clusters

Different ways to measure the distance between the two clusters can be classified in two categories, namely *Graph Metrics*, *Geometric Metrics*. [Ols95]. The graph metric represents the data in the form of a graph where all the vertices are the data samples to be clustered and all the edges have some cost equivalent to the Euclidean distance between the samples. Figure 2.2 shows samples 1, 3, 4, 5 grouped into two clusters. Edges have weights equal to the euclidean distance between the samples. There are three metrics used to calculate inter cluster distance.

1. *Single Link*- The distance between two clusters is the minimum cost edge between points in the two clusters. So, in Figure 2.2, the inter-cluster distance is 30 which is the minimum cost edge between the clusters.
2. *Average Link*- The distance between two clusters is the average of all of the edge costs between points in the two clusters.
3. *Complete Link*- The distance between two clusters is given by the maximum cost edge between points in the two clusters. Again, with reference to Figure 2.2 the inter-cluster distance using this metric would be 100 which the maximum weight edge.

Geometric metrics use cluster centers to determine the distances between the clusters. The distance between the two clusters is the euclidean distance between the two cluster centers. There are two ways to calculate the cluster centers.

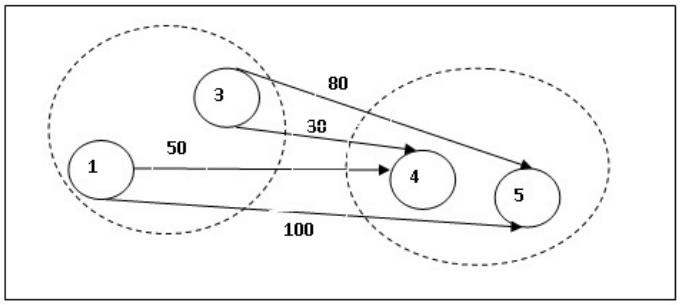


Figure 2.2: Example - Graph Metric

1. *Centroids*- Each cluster has a centroid which acts as a cluster center. It is calculated by using the formula -
   Centroid c = $((c_{11} + c_{21} + ...c_{n1})/n), \ldots ((c_{1n} + c_{2n} + ...c_{nn})/n)$
   where, $c_1....c_n$ are the data samples in a cluster having n attributes.
2. *Median*- It is used in case of hierarchical clustering. Cluster center is the average of the cluster centers which are merged together to form that particular cluster. Refer Figure 2.3. Cluster C3 is formed by merging two clusters C1 and C2. Thus, cluster center of C3 will be the average of cluster centers of C1 and C2.

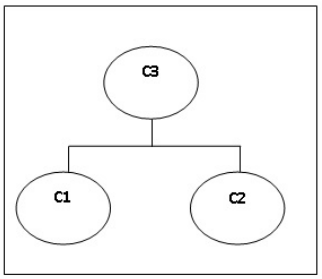


Figure 2.3: Example - Geometric Metric

Several parallel implementations of clustering algorithms are performed before for each of these metrics. However, their implementation is problem specific and machine dependent. Some parallel implementations do not decrease the time required by serial implementation while some show speed up to some extent [Ols95]. Implementing parallel clustering algorithms is realistic and some specialized hardware chips can also be built for clustering. One example of this is a systolic system proposed by Ni and Jain for partitional clustering with a potential performance gain of 1300 times over a serial processor [LF89]. Clustering algorithms can be categorized as follows:

1. *Partition based clustering*

2. *Hierarchical clustering*
3. *Model based clustering*

Partition based clustering relies on an objective function called the *distance function*. The distance function decides the similarity or dissimilarity between the data samples. The aim of partition clustering is to maximize the inter cluster distance and minimize the intra-cluster distance. The quality of the clustering depends on the algorithm, distance function and the application. Here, the number of clusters is predefined and then we proceed with the optimization of the objective function. The data is analyzed and based on the function values, is categorized into predefined number of clusters. Similarity and dissimilarity between the samples is determined using the suitable objective function. Generally, it follows the centroid based clustering function [CSPK09].
This type of clustering requires a high amount of distance computations involving large matrix or vector computations. The computation of the distances between data sample, between data samples and cluster centers, can be carried out in parallel. The comparison of these distances can also be done in parallel. Therefore, a SIMD computer system is suitable for this purpose. One example of a partitional clustering algorithm is "Squared Error" clustering algorithm whose parallel implementation can be found out in the paper by Li and Fang [LF89]. We are going to see the implementation of *K-means* which is another partitional clustering algorithm in later sections.

Hierarchical Clustering consists of successive development of clusters. This follows one of the two approaches. Either we can start with a single cluster, then split it into two clusters until we get clusters with reasonable number of data samples into it or we can start with some predefined number of clusters and then go on merging those clusters until we get a single cluster or a predefined value. The first approach is called *top down* or *divisive approach* and the second one is called *bottom up* or *agglomerative approach*. The splitting and merging of clusters is performed using a distance function. These two approaches in hierarchical clustering are denoted by a *dendogram*. A dendogram is a binary tree which has all the data items at its leaves. An example of a dendogram is shown in Figure 2.4. As we can see in Figure 2.4, we have all the samples of data like in this case A, B, ...F at the leaf node stage of a dendogram. The dotted line denotes the *threshold* or *stopping criteria* for the algorithms. Thus, if we follow bottom up approach and went of merging the clusters, at the threshold line we end up with three clusters $\{a\}, \{b, c, d\} and \{e, f\}$ as indicated in Figure 2.4 [CSPK09]. A lot of research has been done on parallelizing hierarchical clustering algorithms. The paper [Ols95] gives an overview of parallel implementations of some of the algorithms falling in this category. Effectiveness and topology of parallel implementation again depends upon the metric chosen, problem to be solved, algorithm and processor. However, like partitional clustering in this case also SIMD processors are suitable for parallel implementation [Ols95].

Model Based Clustering assumes certain probabilistic model of the data and then estimates its parameters. The data is a result of a mixture of C sources which are treated as C

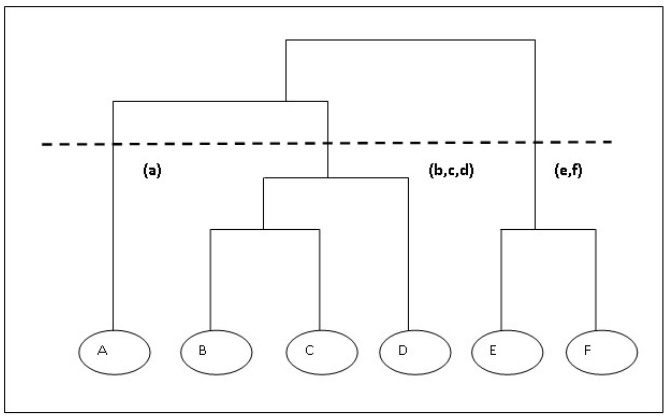


Figure 2.4: An example of Dendogram

potential clusters. This type of clustering is treated under the name of *mixture density*. All the regions in the data are associated with a density function. If in some region, the data is located close to each other, its density is high and therefore a cluster is formed whereas if in some region if the density of the data is low, they are treated as outliers and are then put into other clusters. They are very complex to implement and not a lot of research is found regarding their implementation [CSPK09].

Applications

Clustering algorithms are used in various areas. Some applications of the clustering are:

1. *Image segmentation*
2. *Object and character recognition*
3. *Information retrieval and Data mining*

Image segmentation is used in many computer vision applications. Image segmentation is defined as the partitioning of an image into regions so that each region can be considered as homogeneous with respect to some function like intensity, colour or texture. Clustering can be applied to this application in order to group the pixels to form highly homogenous regions. Use of clustering for image segmentation was performed three decades ago and it is still used in this area. Feature vectors are defined for each pixel in an image and are composed of functions of image intensity and pixel location. Clustering algorithms are then used on these feature vectors, which groups them into homogeneous clusters. These newly formed clusters then correspond to the image segments. Figure 2.5 shows how clustering is used in image segmentation. This basic idea of using clustering in im-

age segmentation can be successfully used for intensity images with or without structure, range images and multispectral images [JMF99]. One parallel algorithm to perform clustering while doing image segmentation is proposed by Khotanzad and Bouarfa [KB90]. The algorithm falls under model driven clustering category and performs parallel clustering using multi-dimensional histogram to represent feature vectors for the image pixels and considering the density of the data. The algorithm was implemented on a SEQUENT parallel computer [KB90].

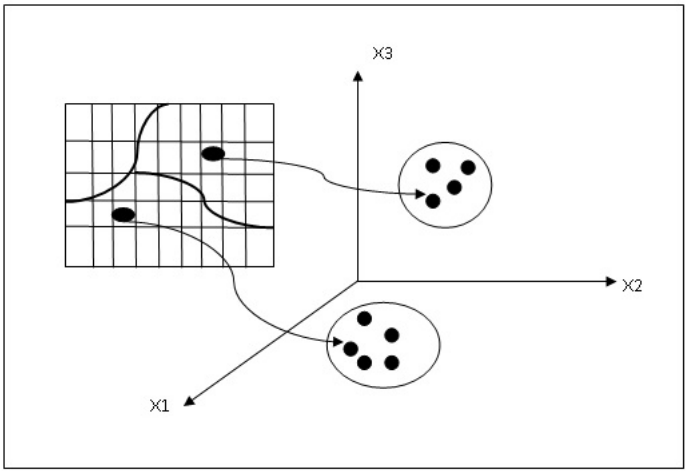


Figure 2.5: Using Clustering in Image Segmentation

Object Recognition refers to the process of recognizing 3D objects in the range data. Objects are represented as views in the database. Objects in 3D have multiple views. View of the object means the range image of the corresponding object. When an unknown view of the object is to be recognized, it is not possible to compare it with all possible views of that object. Hence, a *view class* concept is used which acts like an index. From an unknown view a subset of views of a subset of objects in the database are chosen for further comparison. Thus, in this way a view class is a set of qualitatively similar views of an object. Clustering is used to construct the view classes. Similarity based on shape or spectral features is found out between various views of each object and similar views are grouped together in classes. Thus, clustering is an efficient way to organize the views of an object as well as to speed up its recognition process [JMF99]. Parallel K-means clustering algorithm can be used to recognize objects. Feature space of views of objects can be represented by using colors and positions of the pixels and then a parallel K-means can be used for clustering. A detailed explanation of this is given in [HKRA97].

Character Recognition refers to the process of identification in handwriting systems. Various automatic handwriting systems store different styles of handwritings which overlap in many areas. So, a proper classification of those styles is necessary in order to increase

the efficiency of those systems. Clustering is used to make classes of different styles and to speed up the recognition process [JMF99].

Information retrieval commonly called as IR deals with automatic storage and retrieval of documents. Search engines are the classic example of an IR system. According to the search keywords, relevant documents are returned to the user. However, in order to increase the efficiency, precision and recall of the search engines, clustering is used. Similar documents are clustered together. Whenever a text based search is performed, the distance of search keywords with the clusters is found out and the relevant documents in the cluster with minimum distance are returned. Similarity matrices and distance functions are used for this purpose. Similar approach is followed for *data mining*. The data is clustered before performing the mining. Clustering acts like an initial stage for the data mining process and helps to extract meaningful information from unstructured data efficiently. Thus, use of clustering in the area of data mining and IR increases the quality and efficiency of the process and various parallel implementations of partitional and hierarchical algorithms are effectively used in this area [JMF99].

2.3 K-means Algorithm

Overview

K-means is a well known and commonly used partitional clustering algorithm especially in the area of information retrieval and data mining. It classifies data into a set of predefined clusters by using the distance function. It uses the centroid metric to find the distance between the cluster centers and the data samples. Let k be the number of predefined clusters and D be the dataset having number of samples m and no. of attributes of the samples n. So, D is considered as a two dimensional array having m rows and n columns. Thus,the input to the algorithm is a dataset which can be representated by two dimensional array. Rows indicate number of samples in the dataset and columns indicate the attributes of the samples. Output of the algorithm is the number of predefined clusters containing the categorized elements. K-means algorithm is given as follows.

Listing 2.1: K-means Algorithm

```
1 Find the distance between all sample points of the dataset.
2 Distance is calculated by using Euclidean distance formula.
3 dist(x,y) = sqrt((x_1 - y_1)^2 + (x_2 - y_2)^2 + ....(x_n - y_n)^2)
4 Find the farthest points by considering these distance matrix.
5 Put those points into predefined number of clusters.
6 Make them initial centroids for corresponding clusters.
7 do
8 for each data sample x ∈ D
```

Calculate the distance of those data samples with all the initial clusters
Put the data sample into the cluster with minimum distance.
Recalculate the centroids using formula −
Centroid for cluster c after adding element $x = (c_1 + x_1)/2...(c_n + x_n)/2$
end for
while Centroids are same as the previous centroids of the cluster after completion of sequential scanning.

Stopping Criteria: K-means algorithm can follow two stopping criteria. It can either stop after the definite number of iterations or if the centroids don't move after scanning all the elements sequentially. When all the elements are sequentially scanned once, we can't guarantee that this is the correct solution as we recalculate the centroids when each new element is added to the cluster. So, the sequential scanning process is repeated until the centroids for all the clusters after scanning match with their centroids before scanning. However, as sometimes this process would turn into infinite loop, a definite number of iterations are defined for scanning. The timing complexity of K-means is given by *O(tkn)* where, n is the number of data points, k is the number of clusters and t is the number of iterations. One major advantage of K-means algorithm is its efficiency and ability to work on large datasets as well which do not fit into the memory. However, it can run infinitely if stopping criteria is not controlled properly and is very sensitive to outliers [ZMH09].

Parallelizing K-means

K-means algorithm mainly depends on distance calculations that would require a total of *O(nk)* time for one iteration. So, firstly, the distance computation between all the samples to find farthest samples and secondly, the distance computation between the object and various cluster centers can be parallelized. However, when a new element is added to the cluster, its centers are modified and then in the next iteration next sample is compared with the new centroids. So, the iterations should be executed sequentially. This parallelism can be achieved by using map-reduce pattern. Implementation details regarding this approach are discussed in later sections.

Another approach which uses parallel map-reduce pattern to implement K-means is discussed in the paper [ZMH09]. This approach too parallelizes the distance computations and then serially executes the iterations but in a slightly different way. The map function is used to assign each sample to the closest cluster while the reduce function is used to update the new centroids. The map function accepts the file containing a sequence of a a <key, value> pair each pointing to the record in a database. In each map task, the mapper function can compute the closest cluster for a particular record. The reduction function is then used to sum up the samples stored in each cluster to calculate the new centroids for the cluster. This is illustrated in Figure 2.6. However, the paper doesn't

mention about the serial scanning part. Though the experimental results of this approach show speed, scale up and ability to work on reasonable size of the data, it doesn't clarify on the quality of clusters. As parallel mapping first puts the records to the closest clusters and then their centroids are calculated, we can't guarantee reliability of mapping. This is because while mapping it only considers the distance between original centroids of the clusters and samples while in reality after adding each sample the centroids change and the distance of next sample with new centroid should be considered [ZMH09].

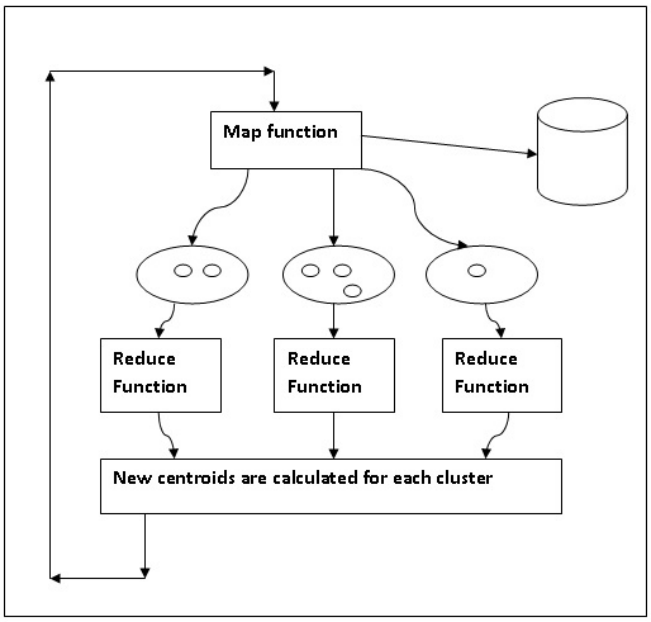


Figure 2.6: An approach to parallelize K-means using map-reduce pattern

2.4 K-means Implementation in Lime

Reduction Pattern

As mentioned earlier, reduction pattern can be used for distance computation in Lime. K-means uses the Euclidean distance measure which is computed using the formula -
dist(x,y) = $\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ...(x_n - y_n)^2}$
Thus, this computation requires addition of all the partial inputs to generate the final summarized distance value. The serial implementation of this computation is given as follows in Lime.

Listing 2.2: Serial Implementation - Reduction Pattern for distance computation

```
1  public void calculateDistancArray(){
2      //Calculating distance between the samples using Euclidean distance formula
3          for (int i : 0:: dataSetRows−1) {
4            for (int j : i+1:: dataSetRows−1) {
5              double sum = 0;
6              for (int k :0:: dataSetColumns−1) {
7                double x = (dataSet[i][k]− dataSet[j][k]);
8                x= x∗x;
9                sum += x; //Serial reduction inside for loop
10             }
11             distanceArray[i][j] = Math.sqrt(sum);
12           }
13         } //End of calculating distance between the points
14   }
```

This serial reduction can be parallelized using parallel reduction operator in Lime. Lime uses (+!) operator to perform reduction in parallel. In the above serial implementation reduction is performed inside the for loop which makes it serial. If it is taken out from the for loop and +! can be used to perform parallel reduction. Parallel implementation can be performed as follows.

Listing 2.3: Parallel Implementation - Reduction Pattern for distance computation

```
1  public void calculateDistanceArray(){
2      //Calculating distance between the samples using Euclidean distance formula
3        double[] partialSumArray = new double[dataSetColumns];
4          for (int i : 0:: dataSetRows) {
5            for (int j: i+1:: dataSetRows−1) {
6              for(int k : 0:: dataSetColumns−1) {
7                double x = (dataSet[i][k]− dataSet[j][k]);
8                x= x∗x;
9                partialSumArray[k] = x;
10             }
11             //Parallel Reduction Pattern
12             distanceArray[i][j] = Math.sqrt(+!partialSumArray);
13           }
14         } //End of calculating distance between the points
15   }
```

This implementation stores the partial values of all the attributes into an array inside the for loop. This makes it ready for the reduction and thus a reduction pattern is applied on that array parallely outside the for loop. This type of parallelization is applied at three places in the implementation. Once distances between all the samples are calculated to create initial clusters, then twice when the distance of a sample is found out with the ex-

isting clusters.

Fusing Map with Reduce:- The distance computation implemented above is an example of map followed by reduction. Inside the for loop, computations between the two elements of two arrays with same index is implemented and then their square is performed. This is a map pattern where all the elements in the array are combined with the value to get a target value and can be performed parallely. Lime allows "@" operator to perform mapping parallely. However, this operator implements map operation using only one value for the array. For example, all the elements in an array will be multiplied with same value, or incremented using same value, but elements of different arrays with same index can't be combined together parallely. This is illustrated in following listing.

Listing 2.4: Map Operation in Lime - Example

```
1 public static void main(String[] args) {
2     int[] row = {1,2,3};
3     int[] row2 = {4,5,6}
4     int[] row1 = row@+#2; //Allowed. Multiplies all the elements of "row" with a constant value
          two.
5     int[] row3 = row@+#row2 //Not allowed.Doesn't perform addition of row[i] with row2[i]
          parallely
6   }
```

Streaming Computation

The implementation performs a streaming computation which isolates the tasks of actual cluster formation and printing the final output. Thus, it works like a pipeline pattern. The output of first task is given to the printing process which prints the final output. It is shown in Listing StreamingComputation.

Listing 2.5: Streaming Computation

```
1 public static void main(String[] args) {
2       KMeansParallel kmeans = new KMeansParallel();
3       var taskkmeans = task kmeans.executeKMeans()
4                       => task kmeans.print(int);
5       taskkmeans.finish();
6   }
```

Practically, some more tasks in the execution like accepting data from user, doing distance computations, making initial clusters and then scanning of all the samples can be isolated and connected to each other which would execute them using pipeline pattern. However, current implementation of Lime allows task to be created only for static local methods. We can't use instance variables of a class inside those methods unless they are

final. In the current implementation of K-means, making instance variables final is not appropriate from algorithm's point of view. So, pipelining of different intermediate tasks of K-means wasn't possible in Lime and that's why I have right now isolated only two tasks [ABCR10].

Source Code

The current implementation of K-means algorithm in Lime reads the dataset from the user. Then it finds the distance metric for all the samples in the dataset and makes two initial clusters. As cluster numbers are predefined in K-means, I have considered two clusters in the current implementation. The main structure of K-means algorithm is as follows:

Listing 2.6: Main Structure of K-means program

```
1  <<Import Statements 2.7 >>
2  public class KMeansParallel
3  {
4      <<Initializing variables 2.8 >>
5      <<Read Dataset 2.9 >>
6      <<Calculate Distance Array 2.3 >>
7      <<Find Farthest Samples 2.10 >>
8      <<Initial Clusters 2.11 >>
9      <<Making Initial Clusters 2.12 >>
10     <<Sequential Scanning 2.13 >>
11     <<Print DataSet 2.14 >>
12     <<Print Distance Array 2.15 >>
13     <<Print Cluster One 2.16 >>
14     <<Print Cluster Two 2.17 >>
15     <<Main Print Method 2.18 >>
16     <<Execute KMeans 2.19 >>
17     <<Main Method 2.5 >>
18 }
```

Import Statements - Listing shows used libraries.

Listing 2.7: Import Statements

```
1  import lime.util.Tasks;
2  import java.io.BufferedReader;
3  import java.io.IOException;
4  import java.io.InputStreamReader;
```

Initializing Variables - Shows the declaration of the instance variables of the class.

Listing 2.8: Initializing Variables

```
int clusterNumber = 2;
int dataSetRows; //Stores number of dataset rows
int dataSetColumns; //Stores number of dataset columns
double[][] dataSet = new double[50][10]; // Stores entire dataset
double[][] distanceArray = new double[50][50]; //Stores distances between the data samples
double[][] centroidClusterOne = new double[1][20]; //Stores centroid of cluster one
double[][] centroidClusterTwo = new double[1][20]; //Stores centroid of cluster two
double[][] previousCentroidClusterOne = new double[1][20]; //Stores centroids of previous clusters
double[][] previousCentroidClusterTwo = new double[1][20]; //Required to determine stopping criteria
int[] clusterOne = new int[50]; // Stores elements of cluster one
int[] clusterTwo = new int[50]; // Stores elements of cluster two
int flag = 0, sampleOne = 0, sampleTwo = 0; //Variables to check conditions
double distWithClusterOne, distWithClusterTwo, maxDistance;
int counterClusterOne = 0, counterClusterTwo = 0;
```

Read Dataset - This is the method which accepts the dataset from the user. I have used standard input and output for this implementation because of technical and timing constraints. However, it can be modified later to use datasets in csv format. The program will first accept number of rows (number of samples) followed by number of columns (number of attributes) in a dataset. It will then accept elements in each row separated by space and terminated by newline. The output of the program will be two clusters containing categorized elements along with the final recalculated centroids.

Listing 2.9: Read Dataset

```
public void acceptData(){
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter Number of Samples in the Dataset (Rows in Dataset): ");
  try {
      dataSetRows = Integer.parseInt(br.readLine());
      System.out.println("Enter the attributes (Columns in the DataSet):");
      dataSetColumns = Integer.parseInt(br.readLine());
      System.out.println("DataSet rows are " + dataSetRows);
      System.out.println("DataSet Columns are " + dataSetColumns);
      for(int i : 0::dataSetRows-1) {
      System.out.println("Enter the elements of "+ (i+1) +" row separated by space: ");
      String inputRow = br.readLine();
      String~[] rowElements = inputRow.'split(" ");
      if(rowElements.length != dataSetColumns) {
      System.out.println("Error in the input.Exiting.. ");
      System.exit(0);
        }
```

```
18          for(int j : 0:: dataSetColumns-1) {
19          dataSet[i][j] = Double.parseDouble(rowElements[j]);
20          }
21        }
22    } //End of reading Dataset
23    catch (IOException e) {
24      e.printStackTrace();
25      }
26  }
```

Calculate Distance Array - This method is used to perform distance computation and implements parallel reduction pattern. Refer Listing 2.3.

Find Farthest Samples - This method finds the farthest points in the given dataset in order to make initial clusters.

Listing 2.10: Find Farthest Samples

```
1  public void findFarthestSamples(){
2    for(int i : 0:: dataSetRows-1) {
3      for(int j :0:: dataSetRows-1) {
4          if (maxDistance < distanceArray[i][j]) {
5              maxDistance = distanceArray[i][j];
6              sampleOne = i;
7           sampleTwo = j;
8          }
9     }
10   }
11   System.out.println("\nMaximum Distance is: " + maxDistance );
12   System.out.println("Farthest Points are: " + sampleOne + " , " + sampleTwo);
13 }
```

Initial Clusters - This method creates two clusters using two farthest points found in the above method and then sets the initial cluster centroids.

Listing 2.11: Initial Clusters

```
1  public void initialClusters (){
2    //Making Initial Clusters
3    clusterOne[0] = sampleOne;
4    counterClusterOne++;
5    clusterTwo[0] = sampleTwo;
6    counterClusterTwo++;
7    //Calculating Centroids for initial clusters
8    for(int i : 0:: dataSetColumns-1) {
9    centroidClusterOne[0][i] = dataSet[sampleOne][i];
```

```
10    previousCentroidClusterOne[0][i] = centroidClusterOne[0][i];
11    centroidClusterTwo[0][i] = dataSet[sampleTwo][i];
12    previousCentroidClusterTwo[0][i] = centroidClusterTwo[0][i];
13      }  //Storing  initial  Centroids  for  two Clusters
14  }
```

Making Initial Clusters - This method is calls the two methods above to create initial clusters and then prints the initial centroids of the clusters.

Listing 2.12: Making Initial Clusters

```
1   public void makingInitialClusters(){
2       maxDistance = distanceArray[0][0];
3       this.findFarthestSamples();
4       this. initialClusters ();
5       //Printing  Initial  Centroids  for  Cluster  One
6       System.out.println(" Initial  Centroids for Cluster One are:  ");
7       for (int j : 0:: dataSetColumns-1) {
8        System.out.print(centroidClusterOne[0][j] + "    ");
9       }
10      System.out.println();
11      //Printing  Initial  Centroids  for  Cluster  Two
12      System.out.println(" Initial  Centroids for Cluster Two are:  ");
13      for (int j: 0:: dataSetColumns-1) {
14          System.out.print(centroidClusterTwo[0][j] + "    ");
15      }
16      System.out.println();
17  }
```

Sequential Scanning of Data Samples - This method is responsible for scanning the data sample in the dataset. It performs the iterative portion of K-means discussed in previous section and therefore, implements iteration sequentially. However, here the method again implements the parallel reduction pattern to compute the distance of a sample from both existing clusters.

Listing 2.13: Sequential Scanning of Data Samples

```
1   public void sequentialScanning(){
2           //Sequential  Checking of  all  the  elements  in  the  dataset
3           //Checking their  distance  with the two cluster  centroids  and
4           //putting  them in  the  appropriate  cluster
5           //Recalculating  centroids
6           //Repeating the loop  until  centroids  for  both  the  clusters  don't  move
7           do {
8                   flag =0;
9                  for(int k :  1:: counterClusterOne-1) {
```

```
10          clusterOne[k] = 0;
11        }
12        for(int k : 1:: counterClusterTwo-1) {
13          clusterTwo[k] = 0;
14        }
15        counterClusterOne = 1;
16        counterClusterTwo = 1;
17        double[] partialSumArrayOne = new double[dataSetColumns];
18        double[] partialSumArrayTwo = new double[dataSetColumns];
19          for(int i : 0:: dataSetRows-1) {
20           distWithClusterOne = 0;
21           distWithClusterTwo = 0;
22           for(int j : 0:: dataSetColumns-1) {
23             double x = (centroidClusterOne[0][j] - dataSet[i][j]);
24             x = x * x;
25             partialSumArrayOne[j] = x;  //Storing partial sums in array for parallel
                   reduction
26
27             double y = (centroidClusterTwo[0][j] - dataSet[i][j]);
28             y = y*y;
29             partialSumArrayTwo[j] = y; //Storing partial sums in array for parallel
                   reduction
30
31           }
32           distWithClusterOne = Math.sqrt(+!partialSumArrayOne); //Parallel Reduction
33           distWithClusterTwo = Math.sqrt(+!partialSumArrayTwo);
34
35           System.out.println("Distance of sample " + (i+1) + " with Cluster 1 : " +
                 distWithClusterOne + " and with Cluster 2 : " + distWithClusterTwo);
36
37           if(distWithClusterOne < distWithClusterTwo) { //Comparing the distance of
                 sample with both the clusters
38             if(i != clusterOne[0]) {
39               System.out.println("Putting Sample " + (i+1) + " in Cluster One");
40               clusterOne[counterClusterOne] = i;
41               counterClusterOne++;
42               for (int j : 0:: dataSetColumns-1) {
43                 centroidClusterOne[0][j] = (centroidClusterOne[0][j] + dataSet[i][j]) /2;
44               } // Recalculating Centroids for Cluster One
45             }
46           }
47           else {
48             if(i != clusterTwo[0]) {
49               System.out.println("Putting Sample " + (i+1) + " in Cluster Two");
50               clusterTwo[counterClusterTwo] = i;
```

```
51                      counterClusterTwo++;
52                      for (int j :0:: dataSetColumns-1) {
53                        centroidClusterTwo[0][j] = (centroidClusterTwo[0][j] + dataSet[i][j]) /2;
54                      } //Recalculating centroids for cluster two
55                    }
56                  }
57
58              }//End of sequential scanning of dataset
59              //Comparing previous and current centroids
60              for(int k : 0:: dataSetColumns-1) {
61                if ((centroidClusterOne[0][k] != previousCentroidClusterOne[0][k]) || (
                      centroidClusterTwo[0][k] != previousCentroidClusterTwo[0][k]))
62                  flag = 1;
63                previousCentroidClusterOne[0][k] = centroidClusterOne[0][k];
64                previousCentroidClusterTwo[0][k] = centroidClusterTwo[0][k];
65              }
66          }while(flag == 1);
67    }
```

Print Dataset - Used to print the original dataset.

Listing 2.14: Printing Original Dataset

```
1   public void printDataSet(){
2           //Print whole dataSet
3           System.out.println("\nPrinting the whole DataSet");
4           for(int i : 0:: dataSetRows-1) {
5             System.out.print("\nSample " + (i+1) +"  -> ");
6             for(int j=0; j< dataSetColumns; j = j+2) {
7                 System.out.print(dataSet[i][j] + "  " + dataSet[i][j+1]);
8
9               }
10          } //End of printing whole dataSet
11    }
```

Print Distance Array - Prints the distance array. It is used like assertions, just to ensure the accuracy of the distance computation.

Listing 2.15: Printing Distance Array

```
1   public void printDistanceArray(){
2           //Printing distance array in indicating distance between the samples
3           System.out.println("\nPrinting distance array: ");
4           for(int i :  0:: dataSetRows-1) {
5             System.out.println();
6             for(int j : 0:: dataSetRows-1) {
7                 System.out.print(distanceArray[i][j] + "  " );
```

```
8             }
9         } //End of printing distance array
10    }
```

Printing Cluster One - Printing the final formed cluster. Though I am again using console output here due to technical and timing constraints, in future it can be modified by writing output to csv files.

Listing 2.16: Printing Cluster One

```
1  public void printClusterOne(){
2          //Printing cluster one samples along with its final centroids
3          System.out.print("\n\nElements in Cluster One with Centroids: ");
4          for (int j : 0::dataSetColumns-1) {
5              System.out.print(centroidClusterOne[0][j] + "    ");
6          }
7          System.out.println();
8          for(int i : 0::counterClusterOne-1) {
9             System.out.print("Sample:  " + (clusterOne[i]+1) + " -> ");
10           for(int j : 0::dataSetColumns-1) {
11             System.out.print(dataSet[clusterOne[i]][j] + "   ");
12           }
13           System.out.println();
14         }//End of printing elements of cluster one
15    }
```

Print Cluster Two - Printing the second cluster.

Listing 2.17: Printing Cluster Two

```
1  public void printClusterTwo(){
2          //Printing cluster two samples along with its final centroids
3          System.out.print("\nElements in Cluster Two with Centroids: ");
4          for (int j :  0::dataSetColumns-1) {
5              System.out.print(centroidClusterTwo[0][j] + "    ");
6           }
7          for(int i : 0::counterClusterTwo-1) {
8            System.out.println();
9            System.out.print("Sample:  " + (clusterTwo[i]+1) + " -> ");
10           for(int j: 0::dataSetColumns-1) {
11             System.out.print(dataSet[clusterTwo[i]][j] + "   ");
12           }
13         }//End of printing elements of cluster two
14    }
```

Main Print Method - This is the main print method which calls the other print methods. This main print method is used for streaming computation. It works like a sink. It accepts

the signal to print which is in this case a dummy integer from executeKMeans method and then prints the output to the console.

Listing 2.18: Main Print Method

```
1 public void print(int m){
2         this.printDataSet();
3         this.printDistanceArray();
4         this.printClusterOne();
5         this.printClusterTwo();
6   }
```

Executing K-means Method - Acts like a source task. It is connected to the print method. Once this task is completed, it signals the print method and then printing task is executed.

Listing 2.19: Execute K-means Method

```
1 public int executeKMeans(){
2     this.acceptData();
3     this.calculateDistanceArray();
4     this.makingInitialClusters();
5     this.sequentialScanning();
6     return 0;
7   }
```

Main Method - Creates an object of a KmeansParallel class. Then it creates two tasks, one to execute KMeans and second to generate the output. Finally, it connects the two tasks and starts the streaming computation. Refer Listing 2.5.

Output

Listing 2.20 shows the output generated by the print method of algorithm. In this case, the algorithm is run on a small amount of sample data to test its accuracy. The output first prints the original dataset. Then it shows the distance array with the distance between all the seven samples of this dataset. Finally, it prints the two clusters with their final centroids and samples included in the clusters. I have also included print statements in the implementation which show the iterative process of finding the clusters, recalculation of centroids, dataset reading process and the clusters. But, it is not possible to include them here as that makes output too lengthy. So, these statements are used as assertions in the implementation only just to ensure the correct functioning of the algorithm. We can observe from the final two clusters that intra-cluster distance is minimized and inter-cluster distance is maximized which shows the closeness of the elements grouped in the same cluster.

Listing 2.20: Output of above implementation

```
Printing the whole DataSet
Sample 1 -> 1.0  1.0
Sample 2 -> 1.5  2.0
Sample 3 -> 3.0  4.0
Sample 4 -> 5.0  7.0
Sample 5 -> 3.5  5.0
Sample 6 -> 4.5  5.0
Sample 7 -> 3.5  4.5

Printing Distance Array:
0.0  1.118033988749895 3.605551275463989 7.211102550927978 4.716990566028302
    5.315072906367325 4.301162633521313
0.0  0.0  2.5  6.103277807866851 3.605551275463989 4.242640687119285 3.2015621187164243
0.0  0.0  0.0  3.605551275463989 1.118033988749895 1.8027756377319946 0.7071067811865476
0.0  0.0  0.0  0.0  2.5  2.0615528128088303 2.9154759474226504
0.0  0.0  0.0  0.0  0.0  1.0  0.5
0.0  0.0  0.0  0.0  0.0  0.0  1.118033988749895
0.0  0.0  0.0  0.0  0.0  0.0  0.0

Elements in Cluster One with Centroids: 1.5  2.0
Sample:  1 -> 1.0  1.0
Sample:  2 -> 1.5  2.0

Elements in Cluster Two with Centroids: 3.7333333333333334  4.666666666666667
Sample:  4 -> 5.0  7.0
Sample:  3 -> 3.0  4.0
Sample:  5 -> 3.5  5.0
Sample:  6 -> 4.5  5.0
Sample:  7 -> 3.5  4.5
```

Testing

Sr.No.	Test Case	Result
1.	Intra-cluster distance is minimized	Passed
2.	Inter-cluster distance is maximized	Passed
3.	Accuracy of distance computation and parallel reduction	Passed
4.	Stopping Criteria - Making sure that algorithm terminates properly without going into infinite loop	Passed. Checked for multiple datasets. Had safe and accurate termination for all the datasets.
5.	Efficiency and ability to work on reasonable datasets	Yet to be determined. Needs input output method modification. Considered as future work.
6.	Speed Up	Same as serial execution as small input dataset is used

2.5 Conclusion

Machine learning is the process of training the system to handle the future unknown tasks using its experience with the previous tasks. It overlaps with data mining in the area of unsupervised learning. Clustering, which is a classic example of unsupervised learning category of machine learning, mainly refers to the classification of unstructured data into a group of clusters so that all the elements in one cluster are homogeneous. This report discussed various applications of clustering highlighting its importance. Further, it discussed overview of K-means algorithm which very popular and commonly used in the area of data mining before performing data analysis. Two parallel approaches along with one implementation for K-means algorithm are discussed in the paper. Three major tasks can be considered as future work for the above implementation. First one will be reading the data from CSV file format. This would require a DSN connection with Excel database which would help to fetch data into the dataset using a simple SQL select query. Once clustering is performed on this dataset, data in one cluster is again inserted into CSV files representing the clusters again by using insert sql query. This inserting work again can be made parallel for the clusters and would also help to test the algorithm on reasonable size datasets. Further two future tasks include using a parallel map pattern in distance computation and using more modular streaming computation. The current implementation faced some technical issues in Lime and previous sections explain why

these two modifications are not present in the current implementation. Thus, to conclude, K-means is a very well known and widely used algorithm for clustering which is an important problem applicable to various areas and hence approaches to parallelize K-means are absolutely effective to increase its efficiency and ability.

Bibliography

[ABCR10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: The liquid metal programming language – language reference manual. Technical report, IBM Research Division, 2010.

[CFZ09] Bertrand Clarke, Ernest Fokoué, and Hao Helen Zhang. *Principles and Thoery for Data Mining and Machine Learning*. Number 978-0-387-98134-5. Springer, 2009.

[CSPK09] KJ Cios, RW Swiniarski, W Pedrycz, and LA Kurgan. *Data Mining*. Springer, 2009.

[Gha04] Zoubin Ghahramani. Unsupervised learning. In *Advanced Lectures on Machine Learning*. Springer-Verlag, 2004.

[HKRA97] B. Heisele, U. Kreljel, W. Ritter, and Daimler-Benz AG. Tracking non-rigid, moving objects based on color cluster flow. *IEEE Computer Vision and Pattern Recognition*, (1063-6919), June 1997.

[JMF99] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3), September 1999.

[KB90] Alireza Khotanzad and Abdelmajid Bouarfa. Image segmentation by a parallel, nonparametric histogram based clustering algorithm. *Pattern Recognition*, 23(9), 1990.

[LF89] Xiaobo LI and Zhixi FANG. Parallel clustering algorithms. In *Parallel Computing*. North-Holland, 1989.

[Mit97] Tom Mitchell. *Machine Learning*. Number 0-07-042807-7. McGraw Hill, 1997.

[Ols95] Clark F. Olson. Parallel algorithms for hierarchical clustering. Elsevier Science B.V, January 1995.

[WYB+10] Wernick, Yang, Brankov, Yourganov, and Strother. Machine learning in medical imaging. *IEEE Signal Processing Magazine*, 27, 2010.

[ZMH09] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. Number 5931, December 2009.

3 Parallelized Inside Algorithm

Yihao Fang

3.1 Introduction

This project aims to civilize computers, though it is only a small step during their civilization process. To be particular, the target of the project is to guide computers learning natural language grammars given articles in natural languages (such as Chinese and English), and let them learn faster than ever before by parallelizing the learning algorithm. The inside algorithm is part of the Expectation Maximization (EM) learning algorithm [MS99]. The more efficient the algorithm, the faster the learning process. The inside algorithm was optimized via dynamic programming at the time it was invented. However, in a modern language, such as modern English, the number of grammar rules is too large for the serial inside algorithm to process efficiently. Under the premises that more and more cores were built into a single computer today, a faster response time can be achieved by parallelizing the algorithm.

3.2 Formal Grammars

Probabilistic Context-Free Grammar (PCFG) is a grammar to interpret modern languages such as modern English and Chinese. Research with the grammar is still on going, like Richard and his colleagues in Stanford proposed Compositional Vector Grammar (CVG) in combining with PCFG to learn discrete syntax and continuous semantics [SBMN13]. Before describing deeply into the inside algorithm, first we answer the question "What are Context Free Grammar (CFG), and Probabilistic Context Free Grammar (PCFG)?".

Context-Free Grammars

A Context-Free Grammar (CFG) is a formal grammar consisting of the follow properties:

1. A set of terminals, $\{w^k\}, k = 1, ..., V$
2. A set of nonterminals, $\{N^i\}, i = 1, ..., n$
3. A designated start symbol, N^1
4. A set of rules, $\{N^i \rightarrow \zeta^j\}$, where ζ^j is a sequence of terminals and nonterminals

Usually, start symbol N^1 stands for the a sentence, which is the starting point the parser begins with.

Probabilistic Context-Free Grammars

Probabilistic Context-Free Grammar is a Context-Free Grammar with probabilities assigned to its production rules. It is a specialized form a Weighted Context-Free Grammar that has each production assigned a weight to. For PCFG, "if we consider all the possible expansions of a non-terminal, the sum of their probabilities must be 1" [JM08]. And if each production is chosen independently, then the probability of the parsed tree, P(t), is the product of the probabilities of its productions.

Mathematically, a PCFG possesses the following properties [MS99]:

1. A set of terminals, $\{w^k\}, k = 1, ..., V$
2. A set of nonterminals, $\{N^i\}, i = 1, ..., n$
3. A designated start symbol, N^1
4. A set of rules, $\{N^i \rightarrow \zeta^j\}$, where ζ^j is a sequence of terminals and nonterminals
5. A corresponding set of probabilities on rules such that $\forall i \cdot \Sigma_j P(N^i \rightarrow \zeta^j) = 1$

Before parsing sentences with a PCFG, some notations should be established. A sentence is denoted by a sequence of words $w_1 ... w_m$, and the subsequence $w_a ... w_b$ is denoted by w_{ab}. A single rewriting operation of the grammar is denoted by a single arrow $\rightarrow$. If as a result of one or more rewriting operations a nonterminal N^j can be rewritten as a sequence of words $w_a ... w_b$, then N^j dominates the sequence of words $w_a ... w_b$, and written either $N^j \stackrel{*}{\Rightarrow} w_a ... w_b$ or $yield(N^j) = w_a ... w_b$. If nonterminal N^j spans positions a through b in the string, but not specifies what words are actually contained in this subsequence, the symbol N^j_{ab} is written. These notations are summarized in the table below. [MS99]

Notation	Meaning
G	Grammar (PCFG)
t	Parse tree
$\{N^1, ..., N^n\}$	Nonterminal vocabulary (N^1 is start symbol)
$w^1, ..., w^V$	Terminal vocabulary
$w_1 ... w_m$	Sentence to be parsed
N^j_{pq}	Nonterminal N^j spans positions p through q in string

Nonterminal and Terminal are both nodes in the graph and they do share some common properties, such that we can define the Node class in Lime for containing common properties for terminal and nonterminal:

<src/Node.lime 1>

```
1  public value class Node extendedby Terminal, Nonterminal {
2    // elements in the collection are ordered by the sequence number
3    protected int sequence;
4
5    Node(int sequence) {
6      this.sequence = sequence;
7    }
8
9    public int getSequence() {
10     return sequence;
11   }
12 }
```

The keyword extendedby above makes sure value classes we defined are in the closed world for hardware optimization. Nonterminals in the grammar are represented as the class Nonterminal in a Lime file (as below), and it extends Node class as it is a node in the graph.

<src/Nonterminal.lime 2>

```
1  public final value class Nonterminal extends Node {
2    protected CharArray tag;
3    public Nonterminal(CharArray tag, int seq) {
4      super(seq);
5      this.tag = tag;
6    }
7    @Override
8      public boolean equals(Object obj) {
9        return (obj != null) && (obj instanceof Nonterminal) && this.tag.equals(((Nonterminal)
             obj).tag);
10     }
11 }
```

The "equals" method above is to check whether the given object is the same as the current one. Similarly to nonterminals, terminals can be implemented as the class below:

<src/Terminal.lime 3>

```
public final value class Terminal extends Node {
  protected CharArray lexeme;

  public Terminal(CharArray lexeme, int seq) {
    super(seq);
    this.lexeme = lexeme;
  }

  public CharArray getLexeme() {
    return lexeme;
  }
}
```

Each terminal object associates with one lexeme (word) as represented as a string of characters.

For simplifying the implementation, first we transform all production rules to Chomsky Normal Forms (CNFs). Every context free grammar production rules can be transformed in to an equivalent grammar in Chomsky Normal Form (CNF). In CNF, a production is either a binary production $A \rightarrow BC$ or unary production $A \rightarrow \alpha$, where A, B and C are nonterminals, and α is a terminal.

First, because both binary productions and unary productions are production rules, and they inherit common properties, we can define the base class Production in Lime (as below), then general information can be represented and maintained in the base class only.

<src/Production.lime 4>

```
public value class Production extendedby UnaryProduction, BinaryProduction {
  protected Nonterminal parent;
  protected double probability;
  // elements in the collection are ordered by the sequence number
  protected int sequence;
  public Production(Nonterminal parent, double probability, int seq) {
    this.parent = parent;
    this.probability = probability;
    this.sequence = seq;
  }

  public int getSequence() {
    return sequence;
```

```
14    }
15    public Node getParent() {
16      return parent;
17    }
18    public double getProbability() {
19      return probability;
20    }
21  }
```

After that, we can define binary productions in Lime:

<src/BinaryProduction.lime 5>

```
1   public final value class BinaryProduction extends Production {
2     protected Nonterminal lChild;
3     protected Nonterminal rChild;
4     public BinaryProduction(Nonterminal parent, Nonterminal lChild, Nonterminal rChild,
          double probability, int seq) {
5       super(parent, probability, seq);
6       this.lChild = lChild;
7       this.rChild = rChild;
8     }
9     public Nonterminal getLChild() {
10      return lChild;
11    }
12    public Nonterminal getRChild() {
13      return rChild;
14    }
15  }
```

For the binary production, there is one nonterminal on left side of the rule and two children nonterminals on the right side as to conform to the $A \rightarrow BC$ in CNF:

<src/UnaryProduction.lime 6>

```
1   public final value class UnaryProduction extends Production {
2     protected Terminal child;
3     public UnaryProduction(Nonterminal parent, Terminal child, double probability, int seq) {
4       super(parent, probability, seq);
5       this.child = child;
6     }
7     public Terminal getChild() {
8       return child;
9     }
10  }
```

As per the production $A \rightarrow \alpha$ in CNF, we implement it into the class model UnaryProduction as above.

A PCFG contains sets of nonterminals, terminals and production rules. We define collections for the definition of the sets. The size of the collection is not known beforehand. The grammar is mutable, in the other words, new productions, nonterminals or terminals may be added to the grammar iteratively during the learning process. Hence, to simplify, we use the ArrayList class from the java.util library to store elements in collections, as it is the variable length implementation of arrays.

<src/NonterminalCollection.lime 7>

```
1  import java.util.ArrayList;
2  public final class NonterminalCollection {
3    protected ArrayList elements;
4
5    public NonterminalCollection() {
6      elements = new ArrayList();
7    }
8
9    public Nonterminal addNonterminal(CharArray tag) {
10     Nonterminal nt = new Nonterminal(tag, size());
11     elements.add(nt);
12     return nt;
13   }
14
15   public int size() {
16     return elements.size();
17   }
18
19   public Nonterminal get(int index) {
20     return (Nonterminal)elements.get(index);
21   }
22 }
```

<src/TerminalCollection.lime 8>

```
1  import java.util.ArrayList;
2  public final class TerminalCollection {
3    protected ArrayList elements;
4
5    public TerminalCollection() {
6      elements = new ArrayList();
7    }
8
9    public Terminal addTerminal(CharArray lexeme) {
```

```
10      Terminal t = new Terminal(lexeme, size());
11      elements.add(t);
12      return t;
13    }
14
15    public int size() {
16      return elements.size();
17    }
18
19    public Terminal get(int index) {
20      return (Terminal)elements.get(index);
21    }
22  }
```

<src/BinaryProductionCollection.lime 9>

```
1   import java.util.ArrayList;
2   public final class BinaryProductionCollection {
3     protected ArrayList elements;
4
5     public BinaryProductionCollection() {
6       elements = new ArrayList();
7     }
8
9     public BinaryProduction addBinaryProduction(Nonterminal parent, Nonterminal lChild,
          Nonterminal rChild, double probability) {
10      BinaryProduction bp = new BinaryProduction(parent, lChild, rChild, probability, size());
11      elements.add(bp);
12      return bp;
13    }
14
15    public int size() {
16      return elements.size();
17    }
18
19    public BinaryProduction get(int index) {
20      return (BinaryProduction)elements.get(index);
21    }
22
23    public BinaryProduction[[]] toValueArray(){
24      BinaryProduction[] bpa= new BinaryProduction[size()];
25      for (int i : 0 :: size() - 1){
26        bpa[i] = get(i);
27      }
28      return new BinaryProduction[[]](bpa);
```

```
29    }
30  }
```

<src/UnaryProductionCollection.lime 10>

```
1   import java.util.ArrayList;
2   public final class UnaryProductionCollection {
3     protected ArrayList elements;
4
5     public UnaryProductionCollection() {
6       elements = new ArrayList();
7     }
8
9     public UnaryProduction addUnaryProduction(Nonterminal parent, Terminal child, double
          probability) {
10      UnaryProduction up = new UnaryProduction(parent, child, probability, size());
11      elements.add(up);
12      return up;
13    }
14
15    public int size() {
16      return elements.size();
17    }
18
19    public UnaryProduction get(int index) {
20      return (UnaryProduction)elements.get(index);
21    }
22
23    public UnaryProduction[[]] toValueArray(){
24      UnaryProduction[] bpa= new UnaryProduction[size()];
25      for (int i : 0 :: size() - 1){
26        bpa[i] = get(i);
27      }
28      return new UnaryProduction[[]](bpa);
29    }
30  }
```

At this point, we have defined nonterminals, terminals, productions, and collections of them. They are parts of the Probabilistic Context Free Grammar (PCFG), such that we can define the grammar model and those parts as its attributes.

<src/Grammar.lime 11>

```
1   public final class Grammar {
2     // start symbol
3     protected Nonterminal start;
```

```
4    // set of productions
5    protected UnaryProductionCollection unaryProductionCollection;
6    protected BinaryProductionCollection binaryProductionCollection;
7    // set of terminals
8    protected TerminalCollection terminalCollection;
9    // set of nonterminals
10   protected NonterminalCollection nonterminalCollection;
11
12   public Grammar(){
13     unaryProductionCollection = new UnaryProductionCollection();
14     binaryProductionCollection = new BinaryProductionCollection();
15     terminalCollection = new TerminalCollection();
16     nonterminalCollection = new NonterminalCollection();
17      start  = nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'S'}));
18   }
19   public Nonterminal getStart() {
20     return start ;
21   }
22 }
```

In addition to above properties, PCFGs conform to the following model invariance [MS99]:

1. Place invariance: the probability of a subtree does not depend on where in the string the words it dominates, in the other words, for all k such that $P(N^j_{k(k+c)} \rightarrow \zeta)$ is the same.
2. Context-free: the probability of a subtree does not depend on words not dominated by the subtree as such that

   $P(N^j_{kl} \rightarrow \zeta |$anything outside k through $l) = P(N^j_{kl} \rightarrow \zeta)$
3. Ancestor-free: the probability of a subtree does not depend on nodes in the derivation outside the subtree,

   $P(N^j_{kl} \rightarrow \zeta |$any ancestor nodes outside $N^j_{kl}) = P(N^j_{kl} \rightarrow \zeta)$

As to this point, we've described properties and constraints for PCFG, such that we can begin to solve the questions like, [MS99]

1. What is the probability of a sentence w_{1m} given the grammar G: $P(w_{1m}|G)$?
2. How are rule probabilities (weights) [SJ07] of the grammar G estimated as such to maximize the probability of a sentence: $\underset{G}{argmax} P(w_{1m}|G)$?

In general, the first question is addressed by inside probability and inside algorithm, and the second question is answered by the Estimation Maximization (EM) Algorithm, EM

Algorithm is an unsupervised learning algorithm designed on top of Inside Probability and Outside Probability. The difference between supervised learning algorithm and unsupervised learning algorithm is that supervised learning usually is performed in a batch mode, whereas, the unsupervised learning could be conducted incrementally.

Considering the complexity of the EM algorithm, the project scope is narrowed to only the implementation of a part of the learning algorithm – inside probability and inside algorithm.

3.3 Learning Algorithms

Estimation Maximization (EM) algorithm is the algorithm that learns the production rules and rule probabilities in the probabilistic context-free grammar. As inside and outside probabilities are part of the EM algorithm, the complexity and performance of the inside outside algorithm (the algorithm to calculate inside and outside probabilities) do significantly affect the overall performance of EM algorithm.

Inside outside algorithm was introduced by James K. Baker in 1979 as a generalization of the forward backward algorithm on hidden Markov models to stochastic context-free grammars. The outside probability $\alpha_j(p,q)$ is the total probability of beginning with the start symbol N^1 and generating the nonterminal N^j_{pq} and all the words outside $w_p...w_q$, given a grammar G [MS99]:

$$\alpha_j(p,q) = P(w_{1(p-1)}, N^j_{pq}, w_{(q+1)m}|G)$$

The inside probability $\beta_j(p,q)$ is the total probability of generating words $w_p...w_q$, given the root nonterminal N^j and a grammar G:

$$\beta_j(p,q) = P(w_{pq}|N^j_{pq}, G)$$

Inside Probability

The inside probability of a substring is computed bottom up inductively on the length of the string subsequence:

Base case: We want to find $\beta_j(p,q)$ (the probability of a rule $N^j \rightarrow w_k$):

$$\beta_j(k,k) = P(w_k|N^j_{kk}, G) = P(N^j \rightarrow w_k|G)$$

Inductive step: We want to find $\beta_j(p,q)$ for $p < q$. As this is the inductive step using a Chomsky Normal Form grammar, the first rule must be of the form $N^j \rightarrow N^r N^s$, so we

can proceed by induction, dividing the string in two in various places and summing the result:

Then, $\forall j, 1 \leq p < q \leq m$,

$$\begin{aligned}
\beta_j(p,q) &= P(w_{pq}|N^j_{pq}, G) \\
&= \Sigma_{r,s}\Sigma^{q-1}_{d=p} P(w_{pd}, N^r_{pd}, w_{(d+1)q}, N^s_{(d+1)q}|N^j_{pq}, G) \\
&\quad \text{«chain rule»} \\
&= \Sigma_{r,s}\Sigma^{q-1}_{d=p} P(N^r_{pd}, N^s_{(d+1)q}|N^j_{pq}, G) P(w_{pd}|N^j_{pq}, N^r_{pd}, N^s_{(d+1)q}, G) \\
&\quad P(w_{(d+1)q}|N^j_{pq}, N^r_{pd}, N^s_{(d+1)q}, w_{pd}, G) \\
&\quad \text{«PCFG invariance»} \\
&= \Sigma_{r,s}\Sigma^{q-1}_{d=p} P(N^r_{pd}, N^s_{(d+1)q}|N^j_{pq}, G) P(w_{pd}|N^r_{pd}, G) P(w_{(d+1)q}|N^s_{(d+1)q}, G) \\
&\quad \text{«definition of inside probabilities»} \\
&= \Sigma_{r,s}\Sigma^{q-1}_{d=p} P(N^j \rightarrow N^r N^s) \beta_r(p,d) \beta_s(d+1,q)
\end{aligned}$$

Inside Algorithm

The inside algorithm is a dynamic programming algorithm that designed according to the definition of inside probability. The inside algorithm is described as follows:

let the input be a sentence W consisting of m words: w_1 ... w_m

let the grammar contain n nonterminal symbols N_1 ... N_n

let N_1 be the start symbol

let $inp[m, m, n]$ be an array of inside probabilities, where the first and second dimensions stand for the start and end of the span respectively, and the third dimension stands for the index of the nonterminal in the collection

initialize all elements of inp to 0

for each $p = 1$ to m

  for each unary production $N_i \rightarrow w_p$

    $inp[p, p, i] = P(N_i \rightarrow w_p)$

for each $l = 2$ to m, where l is the length of the span

for each $p = 1$ to $m - l + 1$, where p is the start of the span
  $q = p + l - 1$, where q is the end of the span
  for each $r = p$ to $q - 1$, where r is a partition of the span
    for each production $N_i \rightarrow N_j N_k$
      if $inp[p, r, j] > 0$ and $inp[r + 1, q, k] > 0$ **then**
        $inp[p, q, i] += P(N_i \rightarrow N_j N_k) inp[p, r, j] inp[r + 1, q, k]$

In Lime, the implementation of the inside algorithm can be written as below. First a serial implementation is given, then a concurrent implementation is developed on top of the serial one.

<src/Trainer.lime 12>

```
1  public class Trainer {
2    protected CharArray[[]] sentence;
3    protected Grammar grammar;
4    protected double[][][] inp;
5    public Trainer(CharArray[[]] sentence, Grammar grammar) {
6      this.sentence = sentence;
7      this.grammar = grammar;
8    }
9    public void calculateInsideProbabilities () {
10     int m = sentence.length;
11     int n = grammar.nonterminalCollection.size();
12     inp = new double[m][m][n];
13
14     <<Initialize all elements to zero serially 13>>
15     <<Process unary productions serially 14>>
16     <<Process binary productions serially 15>>
17   }
18 }
```

INCLUDED BLOCKS: 13 on page 54, 14 on page 55, 15 on page 55

<Initialize all elements to zero serially 13>

```
1      for (int i = 0; i < m; i++)
2        for (int j = 0; j < m; j++)
3          for (int k = 0; k < n; k++)
4            inp[i][j][k] = 0;
```

USED IN: src/Trainer.lime on page 54

<Process unary productions serially 14>

```
for (int p = 0; p < m; p++){
  for (int j = 0; j < grammar.unaryProductionCollection.size(); j++) {
    UnaryProduction up = grammar.unaryProductionCollection.get(j);
    if (up.getChild().getLexeme().equals(sentence[p])) {
      inp[p][p][up.getParent().getSequence()] = up.getProbability();
    }
  }
}
```

USED IN: src/Trainer.lime on page 54

<Process binary productions serially 15>

```
for (int l = 2; l <= m; l++)
  for (int p = 0; p <= m − l; p++) {
    int q = p + l − 1;
    for (int r = p; r < q; r++){
      for (int j = 0; j < grammar.binaryProductionCollection.size(); j++) {
        BinaryProduction bp = grammar.binaryProductionCollection.get(j);
        if (inp[p][r][bp.getLChild().getSequence()] > 0 && inp[r + 1][q][bp.getRChild().
            getSequence()] > 0) {
          inp[p][q][bp.getParent().getSequence()] = inp[p][q][bp.getParent().getSequence()
              ]
              + (bp.getProbability()
              ∗ inp[p][r][bp.getLChild().getSequence()]
              ∗ inp[r + 1][q][bp.getRChild().getSequence()]);
        }
      }
    }
  }
```

USED IN: src/Trainer.lime on page 54

Outside Probability

The outside probabilities are calculated top down by induction as follows:

Base case: The base case is the probability of the root of the tree being nonterminal N^i with nothing outside it:

$$\alpha_1(1,m) = 1$$
$$\alpha_j(1,m) = 0, \text{ for } j \neq 1$$

Inductive step: In terms of the previous step of the derivation, a node N^j_{pq} with which we are concerned might be on the left or the right branch of the parent node. Thus we can proceed inductively by summing both probabilities:

$$\begin{aligned}
\alpha_j(p,q) &= [\Sigma_{f,g}\Sigma_{e=q+1}^{m} P(w_{1(p-1)}, w_{(q+1)m}, N^f_{pe}, N^j_{pq}, N^g_{(q+1)e})] \\
&\quad + [\Sigma_{f,g}\Sigma_{e=1}^{p-1} P(w_{1(p-1)}, w_{(q+1)m}, N^f_{eq}, N^g_{e(p-1)}, N^j_{pq})] \\
&\quad \text{«chain rule»} \\
&= [\Sigma_{f,g}\Sigma_{e=q+1}^{m} P(w_{1(p-1)}, w_{(e+1)m}, N^f_{pe}) P(N^j_{pq}, N^g_{(q+1)e} | N^f_{pe}) P(w_{(q+1)e} | N^g_{(q+1)e})] \\
&\quad + [\Sigma_{f,g}\Sigma_{e=1}^{p-1} P(w_{1(e-1)}, w_{(q+1)m}, N^f_{eq}) P(N^g_{e(p-1)}, N^j_{pq} | N^f_{eq}) P(w_{e(p-1)} | N^g_{e(p-1)})] \\
&\quad \text{«definitions of inside and outside probabilities»} \\
&= [\Sigma_{f,g}\Sigma_{e=q+1}^{m} \alpha_f(p,e) P(N^f \to N^j N^g) \beta_g(q+1,e)] \\
&\quad + [\Sigma_{f,g}\Sigma_{e=1}^{p-1} \alpha_f(e,q) P(N^f \to N^g N^j) \beta_g(e,p-1)]
\end{aligned}$$

Evaluating the probability of a sentence

Given the grammar, the probability of a sentence can be interpreted as inside probability and calculated by inside algorithm:

$$\begin{aligned}
P(w_{1m}|G) &= P(N^1 \overset{*}{\Rightarrow} w_{1m}|G) \\
&= P(w_{1m}|N^1_{1m}, G) \\
&= \beta_1(1,m)
\end{aligned}$$

The probability of the sentence given the grammar can also be written as the formula of outside probabilities and calculated by outside algorithm:

For any k, $1 \leq k \leq m$,

$$\begin{aligned}
P(w_{1m}|G) &= \Sigma_j P(w_{1(k-1)}, w_k, w_{(k+1)m}, N^j_{kk}|G) \\
&= \Sigma_j P(w_{1(k-1)}, N^j_{kk}, w_{(k+1)m}|G) P(w_k | w_{1(k-1)}, N^j_{kk}, w_{(k+1)m}, G) \\
&= \Sigma_j \alpha_j(k,k) P(N^j \to w_k)
\end{aligned}$$

Estimating the parameters of a grammar

PCFG can be trained in both the supervised and unsupervised ways. The supervised learning algorithm is to count the number of times that a particular rule is used upon the entire training corpus. The formula is written as follows:

$$\hat{P}(N^j \rightarrow \zeta) = C(N^j \rightarrow \zeta)/\Sigma_\gamma C(N^j \rightarrow \gamma), \text{ where } C \text{ stands for counting}$$

The unsupervised learning algorithm refines production probabilities incrementally. Assume that we have a set of training sentences $W = (W_1, \ldots, W_\omega)$, with $W_i = w_1, ..., w_{m_i}$, and assume the sentences in the training corpus are independent, summing the contributions from sentences gives the probability of $\hat{P}(N^j \rightarrow N^r N^s)$ when N^j is nonterminal and $\hat{P}(N^j \rightarrow w^k)$ when N^j is preterminal (left hand side of the unary production).

The process of parameter reestimation is repeated until the change in the estimated probability of the training corpus is small. "If G_i is the grammar (including rule probabilities) in the i^{th} iteration of training, then we are guaranteed that the probability of the corpus according to the model will improve or at least get no worse" [MS99]

$$P(W|G_{i+1}) \geq P(W|G_i)$$

The premise of why this EM approach is deserved as an unsupervised training approach is whenever a new sentence $W_{\omega+1}$ added into the grammar, only the probabilities are calculated and then added onto the previous estimation results, in the other words, the training is performed incrementally.

3.4 Parallelized Inside Algorithm

Re-considering the implementation of the inside algorithm above, the map pattern is being applied to the steps of initialization and processing binary productions. Fork-join is utilized when processing binary productions. Lime provides the range feature that make convenient implementing the map pattern. The split and join keywords in Lime help implement the fork-join pattern. The structure of the parallelized Inside Algorithm can be described as follows:

<src/ConcurrentTrainer.lime 16>

```
1 import lime.util.Tasks;
2 public class ConcurrentTrainer {
3   protected static CharArray[[]] sentence;
4   protected static Grammar grammar;
5   protected static double[][][] inp;
6
```

```
7   public static void init(CharArray[[]] s, Grammar g) {
8     sentence = s;
9     grammar = g;
10    int m = s.length;
11    int n = grammar.nonterminalCollection.size();
12    inp = new double[m][m][n];
13
14    <<Initialize all elements to zero concurrently 17>>
15  }
16
17  public static void calculateInsideProbabilities () {
18    processUnaryProductions();
19    processBinaryProductions();
20  }
21
22  <<Process unary productions concurrently 18>>
23
24  <<Process binary productions concurrently 19>>
25 }
```

INCLUDED BLOCKS: 17 on page 58, 18 on page 58, 19 on page 60

When initializing all elements of the inside probability array to zero, ranges are applied to make all assignments parallel.

<Initialize all elements to zero concurrently 17>

```
1     for (int i : 0 :: m − 1)
2       for (int j : 0 :: m − 1)
3         for (int k : 0 :: n − 1)
4           inp[i][j][k] = 0;
```

USED IN: src/ConcurrentTrainer.lime on page 57

When processing unary productions, the algorithm splits the array of unary productions into n individual productions, where n is the size of the array. It creates n filters (threads) to process the n individual productions respectively. The results are then joined into one single stream. Invalid tuples (-1, -1, 0.0) are discard, and valid tuples are being processed and stored into the array of inside probabilities.

<Process unary productions concurrently 18>

```
1   public static void processUnaryProductions() {
2     UnaryProduction[[]] ups = grammar.unaryProductionCollection.toValueArray();
```

```
3     final int n = ups.length;
4     var filter = processUnaryProductionFilter();
5     // fork-join
6     var t = Tasks.single ('(sentence, ups))
7     => task forkUnaryProductions
8     => #
9     => task split '(CharArray[[]], UnaryProduction)[[n]]
10    => task [(n) filter ]
11    => task join '(int, int, double)[[n][]]
12    => (# '(int, int, double)[[]], size n)
13    => task joinUnaryProductions;
14    t.finish();
15  }
16
17  public static local Filter<'(CharArray[[]], UnaryProduction), '(int, int, double)[[]]>
        processUnaryProductionFilter() {
18    return task processUnaryProduction;
19  }
20
21  public static local '(int, int, double)[[]] processUnaryProduction(CharArray[[]] sentence,
        UnaryProduction up) {
22    int m = sentence.length;
23    '(int, int, double)[] buf = new '(int, int, double)[m];
24    int counter = 0;
25    for (int p : 0 :: m - 1) {
26      if (up.getChild().getLexeme().equals(sentence[p])) {
27        buf[counter++] = '(p, up.getParent().getSequence(), up.getProbability());
28      }
29    }
30    if (counter > 0){
31      '(int, int, double)[] out = new '(int, int, double)[counter];
32      for (int i : 0 :: counter - 1) {
33        out[i] = buf[i];
34      }
35      return new '(int, int, double)[[]] (out);
36    } else {
37      return new '(int, int, double)[[]]{'(-1, -1, 0.0)};
38    }
39  }
40
41  public static local '(CharArray[[]], UnaryProduction)[[]] forkUnaryProductions(CharArray
        [[]] sentence, UnaryProduction[[]] ups) {
42    '(CharArray[[]], UnaryProduction)[] tuples = new '(CharArray[[]], UnaryProduction)[ups.
        length];
43    for (int i : 0 :: ups.length - 1) {
```

```
44        tuples[i] = '(sentence, ups[i]);
45      }
46      return new '(CharArray[[]], UnaryProduction)[[]] (tuples);
47    }
48
49    public static void joinUnaryProductions('(int, int, double)[[]] tuples) {
50      for (int i : 0 :: tuples.length - 1) {
51        if(tuples[i].0 != -1) {
52          inp[tuples[i].0][tuples[i].0][tuples[i].1] = tuples[i].2;
53        }
54      }
55    }
```

USED IN: src/ConcurrentTrainer.lime on page 57

Divide and conquer is the philosophy behind dynamic programming. Dynamic programming is the approach of dividing the problem again and again until that can not be divided. Conquering of the problem relies on conquering of the its subproblems. Smaller subproblems must be calculated before the ones depending on them are calculated. Thus when parallelizing the algorithm, subproblems on the same level can be parallelized, however, the calculations of the problem and its subproblems must be conducted sequentially. In other words, its subproblems must be calculated first the the problem relying on them are conquered. The reason why we process the unary productions first is that they are the smallest subproblems, as only a single word is derived from its preterminal. After that we calculate the probability of the span dominating two words, then three words, etc. Calculation of the same length spans can be parallelized, as they are same level subproblems not relying on each other. That is the reason why the computation below only parallelizes spans that dominate the same number of words.

<Process binary productions concurrently 19>

```
1   public static void processBinaryProductions() {
2     int m = sentence.length;
3     BinaryProduction[[]] bps = grammar.binaryProductionCollection.toValueArray();
4     for (int l = 2; l <= m; l++)
5       // map pattern
6       for (int p : 0 :: m - l) {
7         int q = p + l - 1;
8         for (int r = p; r < q; r++){
9           for (int j = 0; j < bps.length; j++) {
10            BinaryProduction bp = bps[j];
11            if (inp[p][r][bp.getLChild().getSequence()] > 0 && inp[r + 1][q][bp.getRChild().
                  getSequence()] > 0) {
```

```
12                  inp[p][q][bp.getParent().getSequence()] = inp[p][q][bp.getParent().getSequence()
                        ]
13                      + (bp.getProbability()
14                      * inp[p][r][bp.getLChild().getSequence()]
15                      * inp[r + 1][q][bp.getRChild().getSequence()]);
16                }
17              }
18            }
19          }
20    }
```

USED IN: src/ConcurrentTrainer.lime on page 57

Apart from the map pattern as implemented by the range above, value types, closed world definitions and task programming model are also used in PCFG models and algorithms, in order to generate more optimized hardware-related artifacts for parallel computing. One of the important value type is the CharArray as defined below. CharArray is the value type replacement of String, (as String is not a value class in Lime,) such that all the types defined on top are possibly also defined as value types, such as Terminal and Nonterminal.

<src/CharArray.lime 20>

```
1  public final value class CharArray {
2    protected char[[]] e;
3
4    public CharArray(char[[]] e) {
5      this.e = e;
6    }
7
8    @Override
9      public boolean equals(Object obj) {
10     if(obj == null) return false;
11     if (!( obj instanceof CharArray)) return false;
12     CharArray that = (CharArray)obj;
13     if (this.e.length != that.e.length) return false;
14         for(int i: 0 :: this.e.length - 1) {
15           if (this.e[i] != that.e[i]) return false;
16         }
17         return true;
18     }
19
20   public int length() {
21     return e.length;
```

```
22    }
23
24    public CharArray add(CharArray that) {
25      return add(that.e);
26    }
27
28    public CharArray add(char[[]] that) {
29      int length = this.length() + that.length;
30      char[] rte = new char[length];
31      for(int i: 0 :: this.length() - 1) {
32        rte[i] = this.e[i];
33      }
34      for(int j: this.length() :: length - 1) {
35        rte[j] = that[j - this.length()];
36      }
37      return new CharArray(new char[[]](rte));
38    }
39
40    public static CharArray valueOf(int val) {
41      char[] buf = new char[1024];
42      int counter = 0;
43      while(val > 0) {
44        buf[counter++] = (char)(val % 10 + 48);
45        val = val / 10;
46      }
47      char[] rt = new char[counter];
48      for(int i: 0 :: rt.length - 1) {
49        rt[i] = buf[rt.length - i - 1];
50      }
51      return new CharArray(new char[[]](rt));
52    }
53  }
```

3.5 Experiments

Considering the memory limitation of the experiment environment, we simulate around 500 production rules for the grammar model in order to avoid the out-of-memory exception, though the number of rules in a modern language such as English, Chinese, is larger than the experimental number. Another consideration is that the grammar evolves by time, in other words, it is not static. Hence we make the grammar model mutable and convert to value object and value array only for computing, such that the grammar model has the ability to learn new rules and be evolving.

<src/TrainerTest.lime 21>

```
public class TrainerTest {
  public static void main(String[] args) {
    <<Initialize grammar rules 22>>
    <<Construct the sentence 23>>
    <<Test the serial inside algorithm 24>>
    <<Test the parallelized inside algorithm 25>>
  }
}
```

INCLUDED BLOCKS: 22 on page 63, 23 on page 64, 24 on page 64, 25 on page 65

During the testing, Part-of-speech (POS) tags VP, NNS, VBP, VBG and NN are added to the set of nonterminals. Lexeme "Dogs", "like", "eating" and "sausage" are appended to the set of terminals. Then binary productions are constructed S $\rightarrow$ NNS VP [1], VP $\rightarrow$ VBP VP [0.4], VP $\rightarrow$ VBG NN [0.6], and then unary productions NNS $\rightarrow$ Dogs [1], VBP $\rightarrow$ like [1], VBG $\rightarrow$ eating [1], NN $\rightarrow$ sausage [1]. After that, lexeme "sausage1" to "sausage500" and corresponding unary rules are inserted to the grammar model.

<Initialize grammar rules 22>

```
    final Grammar g = new Grammar();
    Nonterminal vp = g.nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'
        V', 'P'}));
    Nonterminal nns = g.nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'
        N', 'N', 'S'}));
    Nonterminal vbp = g.nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'
        V', 'B', 'P'}));
    Nonterminal vbg = g.nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'
        V', 'B', 'G'}));
    Nonterminal nn = g.nonterminalCollection.addNonterminal(new CharArray(new char[[]]{'
        N', 'N'}));

    Terminal dogs = g.terminalCollection.addTerminal(new CharArray(new char[[]]{'D', 'o', 'g',
        's'}));
    Terminal like = g.terminalCollection.addTerminal(new CharArray(new char[[]]{'l', 'i', 'k',
        'e'}));
    Terminal eating = g.terminalCollection.addTerminal(new CharArray(new char[[]]{'e', 'a', 't'
        , 'i', 'n', 'g'}));
    Terminal sausage = g.terminalCollection.addTerminal(new CharArray(new char[[]]{'s', 'a', '
        u', 's', 'a', 'g', 'e'}));

    g.binaryProductionCollection.addBinaryProduction(g.start, nns, vp, 1);
    g.binaryProductionCollection.addBinaryProduction(vp, vbp, vp, 0.4);
```

```
15    g.binaryProductionCollection.addBinaryProduction(vp, vbg, nn, 0.6);
16
17    g.unaryProductionCollection.addUnaryProduction(nns, dogs, 1);
18    g.unaryProductionCollection.addUnaryProduction(vbp, like, 1);
19    g.unaryProductionCollection.addUnaryProduction(vbg, eating, 1);
20    g.unaryProductionCollection.addUnaryProduction(nn, sausage, 1);
21
22    for (int i = 1; i <= 500; i++) {
23      Terminal sausagei = g.terminalCollection.addTerminal(new CharArray(new char[[]]{'s', 'a'
            , 'u', 's', 'a', 'g', 'e'}).add(CharArray.valueOf(i)));
24      g.unaryProductionCollection.addUnaryProduction(nn, sausagei, 1);
25    }
```

USED IN: src/TrainerTest.lime on page 63

Then we create the sentence "Dogs like eating sausage" as below.

<Construct the sentence 23>

```
1     // sentence "Dogs like eating sausage"
2     CharArray[[]] sentence = new CharArray[[]]{new CharArray(new char[[]]{'D', 'o', 'g', 's'}),
3         new CharArray(new char[[]]{'l', 'i', 'k', 'e'}),
4         new CharArray(new char[[]]{'e', 'a', 't', 'i', 'n', 'g'}),
5         new CharArray(new char[[]]{'s', 'a', 'u', 's', 'a', 'g', 'e'})};
```

USED IN: src/TrainerTest.lime on page 63

We initialize the serial Trainer and invoke method calculateInsideProbabilities for testing the serial inside algorithm.

<Test the serial inside algorithm 24>

```
1     long start, end, diff;
2     double rt = 0;
3     Trainer trainer = new Trainer(sentence, g);
4     start = System.nanoTime();
5     trainer.calculateInsideProbabilities();
6     rt = trainer.inp[0][sentence.length - 1][0];
7     end = System.nanoTime();
8     diff = end - start;
9     System.out.print("When calculating the inside probability serially, the probability of the
          sentence given the grammar is " + rt + ", ");
10    System.out.println("and it takes " + diff + " nanoseconds,");
```

USED IN: src/TrainerTest.lime on page 63

Then we initialize the ConcurrentTrainner and call calculateInsideProbabilities for test parallelized inside algorithm.

<Test the parallelized inside algorithm 25>

```
1  System.out.print("When calculating the inside probability concurrently, the probability of
       the sentence given the grammar is ");
2  start = System.nanoTime();
3  ConcurrentTrainer.init(sentence, g);
4  ConcurrentTrainer.calculateInsideProbabilities();
5  rt = ConcurrentTrainer.inp[0][sentence.length − 1][0];
6  end = System.nanoTime();
7  diff = end − start;
8  System.out.print("" + rt + ", ");
9  System.out.println("and it takes " + diff + " nanoseconds,");
```

USED IN: src/TrainerTest.lime on page 63

The experiment of the parallelized inside algorithm is compared against its serial counterpart. Response time are recorded for each run of both algorithms. Test results are compared under the same production rules and the same input sentence ("Dogs like eating sausage"). The results are measured on average of 10 executions of both algorithms. When computing concurrently on an 8-core computer(**Appendix A**), it takes 1.732 seconds on average for the parallelized algorithm to complete, while its serial counterpart spends around 0.002 seconds on execution.

3.6 Conclusion

With the help of Lime language, we observed that Parallelized Inside Algorithm utilized more cores in the local computer in contrast that its serial version only consumes CPU time from a single core. As more cores are working concurrently, computation resources are supposed to be consumed in the more optimized mode. However, we observed that the response time of the parallelized algorithm is longer, while being executed in bytecode. More work are to be done for processing large volume of articles on real FPGAs, in order to obtain a better measure for parallelized algorithm.

Bibliography

[JM08] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. Prentice-Hall, second edition, 2008.

[MS99] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[SBMN13] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing With Compositional Vector Grammars. In *ACL*. 2013.

[SJ07] Noah A. Smith and Mark Johnson. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics*, 33(4), 2007.

Appendix A: Processor Information

processor	0
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	1
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	2
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	3
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	4
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	5
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	6
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB
processor	7
model name	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
cpu MHz	3392.093
cache size	8192 KB

4 A SAT Solver

Robert C. G. Fuller

In this chapter, an implementation of the Davis-Putnam algorithm is presented. The Lime platform by IBM is used for the implementation.

4.1 Boolean Satisfiability

The determination of the satisfiability of a Boolean formula is an important problem. The question of this problem is regarding whether there exists an assignment of truth values to a Boolean formula such that the evaluation of the formula with respect to this assignment yields **true**. If such an assignment exists, the formula is said to be *satisfiable*; otherwise, it is said to be *unsatisfiable*.

The SAT solver implemented was done so to expect input in *conjunctive normal form* (See Definition 1) for simplicity. Since any propositional formula can be converted to another formula in CNF, a solver that works with such formulæ is sufficient.
Definition 1. *A formula ϕ is said to be in Conjunctive Normal Form ("CNF"), if it is a conjunction of finitely many disjunctions, each containing finitely many Boolean literals.*

If a formula contains clauses that are not concurrently satisfiable[1], then it is clear that the formula (in CNF) is not satisfiable.

An adaptation of the Davis-Putnam algorithm (§ 4.2) exploits this fact to derive for a given formula in CNF an *equi-satisfiable* formula.

[1] That is, the two clauses cannot both under the same assignment σ be satisfied.

4.2 The Davis-Putnam Algorithm

The Davis-Putnam algorithm [DP60] is an approach to the determination of the satisfiability of a Boolean formulas. Within the investigation at hand, formulas are restricted to those in CNF (this will be assumed henceforth). In this context, DP involves the repeated application of *resolution* (Definition 2) between the clauses of its input formula, and those derived at a previous application (if applicable) of resolution within the algorithm.
Definition 2. *Given two clauses $x_1 \vee \cdots \vee x_n$ and $y_1 \vee \cdots \vee y_m$, the* **resolution rule** *is defined such that*

$$\frac{x_1 \vee \cdots \vee x_n, y_1 \vee \cdots \vee y_m}{x_1 \vee \ldots x_{k-1} \vee x_{k+1} \vee \cdots \vee x_n \vee y_1 \cdots \vee y_{l-1} \vee y_{l+1} \vee \cdots \vee y_m}$$

where x_k and y_l are complements (that is, one is the negation of the other) of one another.
Definition 3. *We will call a clause that has been derived through the application of the resolution rule to two clauses the* **resolvant** *of those two clauses.*
Example. *If $\phi = p \vee q$ and $\phi_2 = \neg q$, then resolution is applied as*

$$\frac{(p \vee q), (\neg q)}{(p \vee q) \vee \neg q}$$

yielding the formula

$$p$$

Then, p is the resolvant of $p \vee q$ and $\neg q$.

The process involving the application of the resolution rule to pairs of clauses within an input formula (the pairs chosen such that each member contains the complement of a literal in the other) yields a formula that is *equisatisfiable* with the original input.

4.3 Implementation

The implementation of the Davis-Putnam algorithm was done through the conversion of input to (long integer) bit vectors and their manipulation through the repeated application of a resolution procedure. Each bit vector represents a clause of the original input formula. The clauses are stored as an array of long integers, which in turn represents the input formula.

Each bit (in order) represents the presence or absence of a given literal. For example, bit 0 being set indicates the presence of the literal '`A`' in a given clause, and bit 26 that of the literal '`Z`,' with '`a`' being represented by bit 27, '`z`' by bit The gap between the ASCII values of `'Z'` and `'a'` is accounted for in the implementation.

The grammar of the input language to the program is given in the Figure below.

```
FORMULA  ::=  ε | CLAUSE | FORMULA '*' CLAUSE
CLAUSE   ::=  LITERAL | CLAUSE ('+' | '_' ) LITERAL
LITERAL  ::=  'A' | ... | 'Z' | 'a' | ... | 'z'
```

Input grammar

The plus symbol (+) is used as the disjunctive symbol, while star (∗) is conjunctive. This was done (as opposed to using the usual ∧ and ∨ symbols) to allow both for ASCII input, and the use of the letter v as a variable name. A Boolean variable consists of a single lower-case Roman letter, and its negation by the corresponding upper-case letter.

The overall program is represented through a single class, which follows in this document.

The solver was implemented using a single Lime class. The class contained an array of long integers to represent the collection of clauses that appear in a formula input to the program that the implementation generates. This array is called disjunctions. A similar array, res is to hold the final resolvant that is calculated by the program. This is directly used to determine the satisfiability of the original input formula.

The number of disjunctions is used a number of times in the implementation, so this value is computed from the input and stored as numDisj. Another value, resNum gives a number of resolvants present at different places in the overall computation.

Whether a formula is satisfiable is indicated by the final state of isSAT. This value is necessary within the implementation due to way in which stream computation was utilized; the value is set by a stream computation sink, which instigates the calling of a method within the class to set the value of isSAT.

<VariableDeclarations 1>

```
1   string originalFormulaString;
2
3   public int numVars = 0;
4   // public int numDis;
5
6   boolean debug = false;
7   boolean debug2 = false;
8   boolean debug3 = false;
9   boolean debug4 = true;
10  boolean testOutput = false;
11  boolean testOutput1 = true;
12
13  public long [] variables;
14  public int numVarsUB;
```

```
15
16    public long [] disjunctions;
17    public long [] res;
18
19    public int numDisj;
20    public int resNum;
21
22    public boolean isSAT = false;
23
24    public final int shiftBy = 'a' - 'A' - 6;   // 6 places between Z and a.
25
26    string [] resolveFormulas;
27    int numResForms = 0;
```

USED IN: code/SAT.lime on page 72

The class that encapsulates the program is called SAT, given below.

<code/SAT.lime 2>

```
1  import lime.util.Tasks;
2  import java.util.BitSet;
3  import java.util.Random;
4  import java.io.*;
5  import java.util.Date;
6  import java.lang.Math;
7
8  class SAT{
9
10 <<VariableDeclarations 1>>
11
12    public SAT(String formula){
13       // TODO: - Clean up clause input with more than one
14       //         instance of one literal.
15
16       originalFormulaString = formula;
17       numDisj = countDisjunctions(formula);
18       numVarsUB = formula.length();
19
20       disjunctions = new long[numDisj];
21       for(int i = 0; i < numDisj; i++)
22          disjunctions[i] = 0;
23
24
25       int thisDisj = 0;
```

```
26      for (int i = 0; i < formula.length(); i++){
27        if (formula.charAt(i) == '*')
28          thisDisj++;
29        else if (formula.charAt(i) != '+' && formula.charAt(i) != ' '){
30          if (formula.charAt(i) > 'Z')
31            set(thisDisj, formula.charAt(i) - 'A' - 6);
32          else
33            set(thisDisj, formula.charAt(i) - 'A');
34
35        }
36      }
37    }
38
39  <<SupportingMethods 12>>
40
41  <<ShowDisjunctions 4>>
42
43  <<ShowDisjunctionsLocal 5>>
44
45  <<CleanupMethod 10>>
46
47  <<ResolveMethod 9>>
48
49  <<MathMethods 11>>
50
51  <<CheckSATMethod 13>>
52
53  <<DoCheckMethod 14>>
54
55  <<TestingMethods 15>>
56
57  <<MainMethod 16>>
58
59  }
```

INCLUDED BLOCKS: 1 on page 71, 12 on page 81, 4 on page 74, 5 on page 75, 10 on page 79, 9 on page 77, 11 on page 80, 13 on page 83, 14 on page 83, 15 on page 86, 16 on page 89

Several methods are used in conjunction with disjunctions. The first, countDisjunctions() returns a count of the number of clauses that are encountered within a formula that is passed to it (as a string).

The countDisjunctions() method operates simply by assuming that formula[] is well formatted (according to the solver grammar) and then counting the number of conjunction

symbols (that is, the symbol '*') that it encounters. The method is declared as static and it works on supplied arguments rather than class instance fields so that it is submissive as code for FPGA acceleration. This method is necessary in order to determine the size of the array that is to contain the (bitvector-represented) clauses of the input formula.

<CountDisjunctions 3>

```
1  public static int countDisjunctions(String formula){
2    int ret = 0;
3
4    if(formula.length() > 0)
5      ret = 1;
6
7    for(int i = 0; i < formula.length(); i++)
8      if(formula.charAt(i) == '*')
9        ret++;
10
11   return ret;
12 }
```

The method showDisjunctions() is used in debugging and for demonstration. Its purpose is to output a representation of stored disjunctions of either the original or derived (through resolution) formula related to a given instance of the SAT class.

The showDisjunctions() method takes as arguments a specifier that indicates which formula (original input or resolved) is to be displayed, and also a string (binOrChar) that is an indication of how the formula's clauses are to be displayed (using either binary strings or characters to display the literals of a formula).

<ShowDisjunctions 4>

```
1
2  public void showDisjunctions(string which, string binOrChar){
3    if (which.equals("original"))
4      for (int i = 0; i < numDisj; i++){
5        if (binOrChar.equals("bin"))
6          System.out.println(Long.toBinaryString(disjunctions[i]));
7        else
8          showChars(disjunctions[i]);
9
10       if ((i + 1) < numDisj)
11         System.out.print(" * ");
12     }
13   else
14     for (int i = 0; i < resNum; i++){
15       if (binOrChar.equals("bin"))
16         System.out.println(Long.toBinaryString(res[i]));
```

```
17          else
18            showChars(res[i]);
19
20          if ((i + 1) < resNum)
21            System.out.print(" * ");
22        }
23
24    }
```

USED IN: code/SAT.lime on page 72

The method showDisjunctions(long [], int) does much the same as the method above, except that it works with an array of disjuctions (and outputs in the format of a formula) that is passed as an argument, along with the size of that array.

The showDisjunctions(long [], int) method is used for debugging, and is not part of the main implementation.

<ShowDisjunctionsLocal 5>

```
1   public static void showDisjunctions(long [] disjs, int num){
2     for (int i = 0; i < num; i++){
3       showChars(disjs[i]);
4
5       if (i + 1 < num)
6         System.out.print(" * ");
7     }
8
9   }
```

USED IN: code/SAT.lime on page 72

The variables and their initial values utilized by the resolve() method are listed below. A temporary array of long bit-vectors that represent the running state of the resolved formula is kept in tmp[]. The flag changed is used to indicate whether an application of the resolution rule has caused a change in the state of the collection of clauses being worked with. If not, then the method's work is done.

A BitSet is used to keep track of the clauses that have been used to form resolvants within a given stage of resolution.

<ResolveLocalVariables 6>

```
1     res = new long [numDisj * 4](disjunctions);
```

```
2     long [] tmp;
3     resNum = numDisj;
4     int tmpNum = 0;
5     boolean changed = true;
6     BitSet touched = new BitSet();
```

USED IN: ResolveMethod on page 77

The resolution step checks to see whether resolution is applicable to the currently examined clauses. If so, code is run that adds a new resolvant to the growing array of resolved clauses, stored as the array tmp[]. If not enough space remains after this for another clause to be added to tmp[], more space is allocated for the array. Once a clause has been added, changed flag is set to true, indicating that another level of resolution may be necessary.

<ResolutionStep 7>

```
1           if (i != j && res[i] != (long)'@' && res[j] != (long)'@'){
2             for (long k = pow2(26); k < pow2(51); k *= 2){
3               if ( (k & res[i] & (res[j] << shiftBy) ) != 0){
4                 changed = true;
5                 tmp[tmpNum++] = subRes(res[i], res[j], k);
6                 if (tmp[tmpNum-1] == 0){
7                   return false;
8                 }
9
10                touched.set(i); touched.set(j);
11
12                if (tmpNum + 1 > tmpSize){
13                  long [] tmp2 = new long[(int)tmpSize * 2];
14
15                  for (int l = 0; l < tmpSize; l++)
16                    tmp2[l] = tmp[l];
17
18                  tmp = tmp2;
19                  tmpSize *= 2;
20                }
21
22              }
23            }
24          }
```

USED IN: ResolveMethod on page 77

Resolution over a collection of clauses is gone about through the use of shifting (since clauses are represented as bit vectors) and comparison of the upper and lower portions of a given clause bit vector. Literals are represented using both lower and upper case Roman letters, with a lower-case literal being a variable and the corresponding upper-case letter representing its negation. After the gap between the representative values of `'Z'` and `'a'` in the ASCII representation is accounted for, the application of the resolution rule between two clauses is straight forward. The process of this application is gone about in the following code. It is accomplished (when two clauses are found from which a resolvant can be derived) by shifting the bits of one by the difference between the ASCII representation values of `'A'` and `'a'`. This aligns the two parts of the clause bit vector that contain respectively variables and their negations. Then, depending on the direction of the shift and to which clause shifting was applied, the result of the bitwise `AND` operation between the two is subtracted from one or the other. The process is repeated twice for each clause bit vector (once to eliminate variables whose negations are present in the compared clause, and once to eliminate its negations whose variables appear in the compared clause). The code below illustrates how this is gone about in application, within the subRes() method.

<SubResMethod 8>

```
1  public static long subRes(long res1, long res2, long k){
2    return (res1 − k) | (res2 − (k >>> 26));
3  }
```

The comparison of clauses is done in the following way. For clause res[i], comparison is done between it and clauses res[i+1::resNum], where resNum is the number of clauses stored within res. The array res is initially set to reflect the values of the clauses of the input formula and at each iteration of resolution is updated to contain a current collection of resolvant clauses.

<ResolveMethod 9>

```
1  public boolean resolve(){
2  <<ResolveLocalVariables 6>>
3
4      touched.clear();
5
6      cleanup();  // Remove tautologies from formula.
7
8      if (numDisj == 0)
9        return true;
10
11     int tmpSize = 42;
12     tmp = new long[42];
13     int iterations = 0;
14     while (changed){
```

```
15          iterations++;
16          int internalIterations = 0;
17          if (iterations > 100000){
18            System.out.println("\n1: "+originalFormulaString);
19            return false;
20          }
21
22          //System.out.println("Iterations = " + iterations);
23
24          changed = false;
25          touched.clear();
26
27          tmp = null;
28          tmp = new long[numDisj * 1500];
29          tmpSize = numDisj * 1500;
30          tmpNum = 0;
31
32          for(int i = 0; i < numDisj * 1500; i++)
33            tmp[i] = 0;
34
35          for (int i = 0; i < resNum; i++){
36            for (int j = 0; j < resNum; j++, internalIterations++){
37              if (internalIterations > 1000000){
38                System.out.println("\n2: "+originalFormulaString);
39                return false;
40              }
41  <<ResolutionStep 7>>
42            }
43          }
44          if (changed){
45            for (int i = 0; i < resNum; i++)
46              if (!touched.get(i) && (res[i] != 0)){
47                tmp[tmpNum++] = res[i];
48                if (this.debug2){
49                  System.out.println("Adding \n");
50                  showChars(res[i]);
51                  System.out.println(" (" + i + ")");
52                }
53              }
54
55
56            res = new long[tmpNum];
57
58            for (int i = 0; i < tmpNum; i++)
59              res[i] = 0;
```

```
60
61          for (int i = 0; i < tmpNum; i++)
62            res[i] = tmp[i];
63          resNum = tmpNum;
64
65          if (this.debug3){
66            //System.out.println("ITER = " + (iterations - 1));
67            showDisjunctions("res", "char");
68            System.out.println();
69          }
70        }
71      }
72
73      if (this.debug4)
74        System.out.println("OK-SAT");
75      return true;
76    }
77  @
78
79  The {\small cleanup()} method checks a formula's clauses for the presence of
80   literals along with their compliments. If a clause is found to have both,
81  it is deleted.
```

USED IN: code/SAT.lime on page 72 INCLUDED BLOCKS: 6 on page 75, 7 on page 76

<CleanupMethod 10>

```
1   public void cleanup(){
2     long [] tmp = new long[numDisj];
3     int j;
4
5     for (int i = 0; i < numDisj; i++)
6       tmp[i] = 0;
7
8     for (int i = 0; i < numDisj; i++)
9       if ((disjunctions[i] & (disjunctions[i] << shiftBy)) != 0
10          || (disjunctions[i] & (disjunctions[i] >>> shiftBy)) != 0)
11        disjunctions[i] = (long)'@';
12    j = 0;
13    for (int i = 0; i < numDisj; i++)
14      if (disjunctions[i] != '@')
15        tmp[j++] = disjunctions[i];
16
17    disjunctions = null;
```

```
18      disjunctions = tmp;
19
20      //System.out.println(j + " of " + numDisj + " clauses kept.");
21
22      numDisj = j;
23
24    }
```

USED IN: code/SAT.lime on page 72

Several methods are used as simplified versions of others that are available through usual Java / Lime libraries. Implemented are pow2() and log2() that find a particular power or logarithm of the base 2. The method threeOr() is used to combine the results of three-way parallelized code whose output is boolean. The method resolve() is the primary functional component of the implemented solver. It is the implementation of the resolution rule (see Definition 2), and goes about the application of this rule in stages, as one would by hand. In the first stage, the input clauses are examined, comparing one to another for applicability of the resolution rule between pairs of clauses (determined by whether one contains at least one complement of a literal within the other in the case of each examined pair). Once applicability is determined, the method subRes() (shown later) is applied to find the resolvant of the two clauses. This is stored in a temporary array called tmp.

<MathMethods 11>

```
1   public static long pow2(long expon){
2     long ret = 1;
3
4     for (int i = 0; i < expon; i++)
5       ret *= 2;
6
7     return ret;
8   }
9
10  public static long log2(long val){
11    int ret = 0;
12
13    while (val >= 2){
14      val /= 2;
15      ret++;
16    }
17
18    return ret;
19  }
```

USED IN: code/SAT.lime on page 72

A number of utility methods are used by the solver (either directly or in debugging / testing).

Methods beginning with set are used by the solver directly.

Since method of computation is through streams, variable states must be set by methods. This is the purpose for the method setSAT(). This method simply takes as input the value that indicates the result of a run of the satisfiability checking portion of code, and sets the corresponding value within the class that implements the solver.

The set() method is to set a bit within a bit vector that correspond to a given literal.

<SupportingMethods 12>

```
1  public void set(int disj, int bit){
2    //System.out.println(numDisj + ": " + disj);
3    disjunctions[disj] |= (long)Math.pow((double)2, (double)bit);
4  }
5
6  public void show(int disj){
7    System.out.println(Long.toBinaryString(disjunctions[disj]));
8  }
9
10 public static long binOrd(char toRep){
11   if (toRep > 'Z')
12     return((long)Math.pow(2, toRep - 'A' - 6));
13   else
14     return((long)Math.pow(2, toRep - 'A'));
15 }
16
17 public static void showBin(long val){
18   System.out.println(Long.toBinaryString(val));
19 }
20
21 public void setSAT(boolean SATres){
22   if (!this.isSAT)
23     this.isSAT = SATres;
24 }
25
26 local static boolean threeOr(boolean x1, boolean x2, boolean x3){
27   return (x1 || x2 || x3);
28 }
29
30 public static void showChars(long val){
```

```
31
32      if (val == (long)'@'){
33        System.out.print('@');
34        return;
35      }
36
37      for (long i = 1, j = 1; i <= binOrd('z'); i *= 2, j++)
38        if ((val & i) != 0){
39
40          char toPrint = (char)('A' + j - 1);
41
42          if (i > binOrd('Z'))
43            toPrint += 6;
44
45
46          System.out.print(toPrint);
47          if (val >= i * 2)
48            System.out.print(" + ");
49        }
50    }
```

USED IN: code/SAT.lime on page 72

In order to represent an input disjunction as a bit vector, the input formula is examined and each of its literals is assigned an integer value. The value assigned to an input literal is related directly to its ASCII order. Literals are represented by the Roman alphabet, with lowercase letters corresponding to variables and upper case to their negations (eg., $\neg a$ is represented by A).

Formulas (in CNF) then are represented by arrays of bit vectors, each element representing a disjunction.

4.4 Concurrency

Several areas exist in which parallelization could be used to potentially improve running time performance (albeit at the cost of memory, in some cases). A number of tasks exist that need to be filled that may run in parts independently of one another. However, in some cases, their operation is informed by the entire formula being processed, rather than just one part. For this reason, extra memory is needed to hold sometimes multiple copies of the formula being examined.

Tasks that are directly amenable to parallelization include:

1. Parsing of input and translation of the formula into bit vectors for storage and manipulation formula string
2. Counting of disjunctions within the formula
3. Examination of a resolvant formula to determine satisiability of its original formula.

Task (3) was implemented with concurrency with marginal success. It is currently unknown the source of an issue that produces incorrect results, but the result is that some satisfiable formulæ are reported as unsatisfiable.

It would be desirable to go about the resolution process in a way that utilized concurrency.

A direct approach similar to (1) – (3) would not be completely possible, since the process of resolution is an iterative one, subsequent steps relying on the results of previous ones.

Below can be seen the implementation of the checkSAT() method. This method is now depreciated because of a reformulated way in which resolve() is performed, but is left in this document to show prior methods for use of concurrency. Within the method itself there is not direct use of concurrency, rather its functionality is used as a unit of concurrent execution. Multiple copies of it are optionally executed concurrently within the doCheck() method.

<CheckSATMethod 13>

```
1  local static boolean checkSAT(long [[]] formula, int num){
2    boolean ret = false;
3    for (int i = 0; i < num; i++)
4      if(formula[i] != 0)
5        ret = true;
6
7
8    return ret;
9  }
```

USED IN: code/SAT.lime on page 72

The doCheck() method (depreciated) was responsible for parallelizing the checkSAT() method. More accurately, it is able to run multiple instances of the checkSAT() method concurrently, with optional FPGA acceleration.

In the presented code, only 3-way concurrency is used, but further subdivision of work is possible with only technical modifications to the code.

<DoCheckMethod 14>

```
1  public boolean doCheck(string accel, string concurr){
2    long [[]] temp = new long[[resNum]](res);
3    long [[]] resImm = temp[0::resNum − 1];
4
5    long [] resPadded;
6
7
8    int N = 3;
9    int formSize = numDisj;
10   int chunkSize = (int)Math.ceil(formSize / N);
11   if (formSize < N)
12     chunkSize = 1;
13
14   int paddedSize = chunkSize ∗ N;
15
16   final LFilter<long,boolean> chSAT = (long # long [[]], size resNum) =>
17                       task checkSAT(long [[]], resNum);
18
19   final LFilter<long,boolean> doChSATPar = (long # '(long,long,long))
20                     => task split '(long,long,long)
21                     => task [chSAT, chSAT, chSAT]
22                     => task join '(boolean,boolean,boolean)
23                     => task threeOr(boolean,boolean,boolean);
24
25   resPadded = new long[paddedSize];
26
27   for (int i = 0; i < paddedSize; i++)
28     if (i < resNum){
29       resPadded[i] = res[i];
30     }
31     else{
32       resPadded[i] = 0;
33     }
34
35
36
37
38   long [[]] temp2 = new long[[paddedSize]](resPadded);
39   long [[]] resPaddedImm = temp2[0::paddedSize−1];
40
41   if (concurr.equals("concurrent"))
42     if(accel.equals("accelerated")){
43       var cS = Tasks.source(res)
44         => ([ doChSATPar ])
```

```
45              => task setSAT(boolean);
46
47            cS.finish();
48          }
49          else{
50            var cS = Tasks.source(res)
51              => doChSATPar
52              => task setSAT(boolean);
53
54            cS.finish();
55          }
56        else if(accel.equals("accelerated")){
57            var cS = Tasks.source(res)
58              =>  ([ chSAT ])
59              => task setSAT(boolean);
60
61            cS.finish();
62          }
63
64          else{
65            var cS = Tasks.source(res)
66              => chSAT
67              => task setSAT(boolean);
68
69            cS.finish();
70          }
71
72        return this.isSAT;
73
74    }
```

USED IN: code/SAT.lime on page 72

Two methods, runTests() and buildTests() have been written to test the functionality of the SAT solver code. The former runs tests based upon input parameters related to the number of test cases to be run, and constraints on them. The constraints are to the kind of testing to be done (between running parallel or serial, FPGA-accelerated or non-FPGA-accelerated code), and the maximum length of each test case.

The code of the testing methods is not essential to the implementation, and so will not be discussed in detail. However, what it does is to run the tests constrained as mentioned above, and output the results (along with test strings, since they are generated at random) to text files. The results are based on wall-clock readings implemented through the Date

class.

<TestingMethods 15>

```
1  public void runTests(int NT, int ML, string accel, string conc, string tk){
2    SAT sat1 = new SAT("p");
3    Date time1 = new Date();
4    Date time2 = new Date();
5
6
7    string [] testFormulas = new string[NT];
8
9    long [] testTimes = new long[NT];
10   boolean [] isSat = new boolean[NT];
11   int [] numDisjunctions = new int[NT];
12   int [] maxClauseLength = new int[NT];
13
14   if (this.testOutput1)
15     System.out.println("Generating test cases.");
16
17   testFormulas = SAT.buildTestCases(NT, ML, tk);
18
19   if (this.testOutput1)
20     System.out.println("Starting tests.");
21
22   for(int i = 0; i < NT; i++){
23
24     sat1 = new SAT(testFormulas[i]);
25
26     time1 = new Date();
27     isSat[i] = sat1.resolve();
28     // isSat[i] = sat1.doCheck(accel, conc);
29     time2 = new Date();
30
31     testTimes[i] = time2.getTime() − time1.getTime();
32     numDisjunctions[i] = sat1.numDisj;
33
34   }
35   if (this.testOutput){
36     System.out.println("Tests completed.");
37
38     System.out.println("Writing result data.");
39   }
40   try{
41     string filename = new string();
42     filename = Integer.toString(NT)
```

```
43              +"-tests_"
44              +Integer.toString(ML)
45              +"-maxliterals--"
46              +concurr
47              +"-"+accel+
48              ".input.txt";
49          FileWriter  fstream = new FileWriter(filename);
50          BufferedWriter out1 = new BufferedWriter(fstream);
51
52          for (int i = 0; i < NT; i++)
53            out1.write("i=" + i + ",FORMULA=" + testFormulas[i] + "\n");
54
55          out1.close();
56
57        }catch (Exception e){
58          System.err.println("Error: " + e.getMessage());
59        }
60
61        try{
62          string filename = new string();
63          filename = Integer.toString(NT)
64              +"-tests_"
65              +Integer.toString(ML)
66              +"-maxliterals--"
67              +concurr
68              +"-"+accel+
69              ".results.txt";
70
71          FileWriter  fstream = new FileWriter(filename);
72          BufferedWriter out1 = new BufferedWriter(fstream);
73
74          for (int i = 0; i < NT; i++){
75            string isSatString;
76            if (isSat[i])
77              isSatString = new string("SAT");
78            else
79              isSatString = new String("UNSAT");
80
81            out1.write("TEST=" + i + ",SAT=" +
82                    isSatString + ",LENGTH=" +
83                    testFormulas[i].length() +
84                    ",NUMCLAUSES="+numDisjunctions[i] +
85                    ",TIME=" + testTimes[i] + "\n");
86          }
87
```

```
88        out1.close();
89
90      }catch (Exception e){
91        System.err.println("Error: " + e.getMessage());
92      }
93      if (this.testOutput){
94        System.out.println("Done writing result data.");
95
96        System.out.println("Done.");
97      }
98    }
99
100   static public string[] buildTestCases(int nt, int mtl, string tk){ // maxTestLength is
          actually # of literals in clause.
101     string [] ret = new string[numTests];
102     BitSet used = new BitSet(100);
103     Random gen = new Random();
104
105     char [] literals = {'a','A','b','B','c','C','d','D','e','E','f','F',
106                 'g','G','h','H','i','I','j','J','k','K','l','L',
107                 'm','M','n','N','o','O','p','P','q','Q','r','R',
108                 's','S','t','T','u','U','v','V','w','W','x','X',
109                 'y','Y','z','Z'};
110
111     int nl = 52;
112
113     char [] connectives = {'*','+'};
114     int nc = 2; // Number of connectives
115
116     if ((mtl / 2) * 2 != mtl)
117       maxTestLength--;
118
119     if (tk.equals("random")){
120       for (int i = 0; i < nt; i++){
121         int thisLength = 0;
122         while (thisLength < 2)
123           thisLength = Math.abs(gen.nextInt()) % mtl;
124
125         ret[i] = new string("");
126
127         for (int j = 0; j < thisLength; j++){
128           int litInd = Math.abs(gen.nextInt()) % nl;
129
130           while(used.get( literals [ litInd ]) )
131             litInd = Math.abs(gen.nextInt()) % nl;
```

```
132
133             ret[i] += literals[litInd];
134             used.set(literals[litInd]);
135
136             // ret[i] += literals[Math.abs(gen.nextInt()) % numLiterals];
137             if (j+1 < thisLength){
138               int cInd = Math.abs(gen.nextInt()) % nc;
139               ret[i] += connectives[cInd];
140               if(connectives[cInd] == '*')
141                 used.clear();
142             }
143           }
144         }
145       } else if (testKind.equals("test")){
146
147
148         ret[0] = "pq*pQ*Pq";
149         ret[1] = ret[0]+"*PQ";
150       }
151       return ret;
152     }
```

USED IN: code/SAT.lime on page 72

The main method was employed here strictly for testing purposes and is included here for completeness. Its presence does not impact the overall implementation strictly of the SAT solver.

Within the main method are examples of the execution of the basic testing suite included with the implementation.

<MainMethod 16>

```
1   public static void main(String[] args){
2     boolean isSatRes;
3
4     int NUMCASES = 500;
5     int MAXLITERALS = 50;
6
7     SAT ubs1 = new SAT("p + q * P + q * p + Q");
8
9   <<TestExecutions 17>>
10
11  }
```

USED IN: code/SAT.lime on page 72 INCLUDED BLOCKS: 17 on page 90

Below are the executions that were done toward testing of the SAT solver implementation. In the first two cases, concurrency is requested, in one case with acceleration (by FPGA) and in the other without. In both cases, a random set of data is requested for testing.

In the second pair of calls to SAT.runTests(), the same kind of testing is requested as above, but without concurrency. Currently, this is the only reliable testing that is available, as concurrency is not functional due to what probably is an issue with the coding of the implementation.

<TestExecutions 17>

```
1  ubs1.runTests(NUMCASES, MAXLITERALS, "accell", "concurrent", "random");
2  ubs1.runTests(NUMCASES, MAXLITERALS, "noaccel", "concurrent", "random");
3
4  ubs1.runTests(NUMCASES, MAXLITERALS, "accell", "serial", "random");
5  ubs1.runTests(NUMCASES, MAXLITERALS, "noaccel", "serial", "random");
```

USED IN: MainMethod on page 89

4.5 Acceleration

Several opportunities existed for the use of FPGA acceleration. Most notably are those that also are amenable to parallelization.

The checkSAT() method can be used by doCheck() to create a computing stream that uses acceleration. Whether this is done is optional, and based upon parameters passed while instantiating the SAT class.

Further tasks that could use acceleration include those gone about by the math methods (log2() and pow2(), and those involving simple manipulation of bit vectors (*eg.*, subRes(), and possibly some sub-tasks of resolve() that could be separated by further modularization of that method).

4.6 Performance

Theoretically, the performance of the DP algorithm is known to be related directly to the cardinality of the collection of distinctive clauses generated [Gal77]. However, some formulæ can cause the algorithm to require exponential time (see again [Gal77]).

The performance of the implementation might appear slightly better than a version whose literals are not constrained to only single letters of the Roman alphabet. However, this would only be with respect to the length of a given formula, a full comparison not being possible with an implementation whose input language included a larger set of literals. The complexity of the formulæ that are processed by the implemented solver is much less potentially complex than those whose literals are not constrained as indicated above.

It is most desirable to at least obtain a simple wall clock-based running time testing of the implementation. However, because of a persistent issue involving formulæ whose resolvants are obtainable only in more than one step, this was not possible. The issue is outlined below.

4.7 Known Issues

Two major issues exist with the solver implementation. Mentioned already (see § 4.4) is the non-functionality of the concurrent branch of the implementation.

4.8 Other Implementations

Practical SAT solvers would likely implement the Davis-Putnam-Logemann-Loveland algorithm, rather than the earlier Davis-Putnam algorithm as has this implementation. As such, comparable solvers would likely not be found among the practical ones in use at the time of this report's writing.

Implementations of DP, even concurrent DP do exist, however. Of these there is included PaSAT [SBK01] by Sinz, Blochinger, and Küchlin, and algorithm by Chen and Liu [CL87], and one by Fang and Chen [FC92].

The Chen and Liu algorithm takes a straight-forward approach to the application of concurrency to the problem. This procedure divides the work done by the serial DP algorithm into independent subtasks, each of which is an instance of DP on specifically chosen subformulæ of the input formula. At each subtask division, a literal is chosen from the formula. One subtask examines the satisfiability of the formula without that literal, and another without its complement. These are done concurrently, and as mentioned are independent.

Another algorithm that claims implementation of a parallelized version of DP is by Fang and Chen, and utilizes vector machines. The algorithm operates through the application of splitting the original problem along the line of several variables at once, as opposed

to the earlier Chen and Liu approach of one-at- a-time variable subdivision of the problem.

The PaSAT implementation employs more optimization than the others examined, but still going about the basic DP procedure. Some literal selection heuristics are utilized, and parallelization is gone about by dynamically splitting the search space of the problem. Further, a type of backjumping is utilized that is informed by the progression of the solution. This differs from the other implementations examined which do not explicitly use information about the progression of the problem to inform future decisions about the search for a solution.

The choice of the methods employed for the implementation this chapter described in terms of parallelization were more or less intuitive and meant to explore the concurrency platform offered by Lime. As such, no particular pattern was followed beyond that of the most obvious avenues of parallelization for the algorithm it implements.

Further, the parallelization that is employed (since the work is unfinished) only offers some potential speedup to supporting functions to the main work of the algorithm (in reading input and checking results).

4.9 Future Work

It would be desirable to implement further parallelizatoin and acceleration, as disucssed in §§ 4.4, 4.5.

Performance testing could be done to compare between serial and parallel branches of the implementation.

Further, testing could be augmented to include correctness testing with the use of proven SAT solvers, such as minisat, PaSAT, or ManySAT. Performance testing could be contemplated as well in this way, but most certainly existing SAT solver implementations would outperform the implemented solver.

It would be advantageous and open further avenues of use of the SAT solver if it were to accept more than formulæ in CNF.

The solver could be expanded to accept formulæ constructed over a broader base of literals than the Roman alphabet.

The algorithm that was chosen (Davis-Putnam) is not a standard for commercial and otherwise serious SAT solvers. It would therefore be desirable to implement the Davis-Putnam-Logemann-Loveland procedure, which is a more popular choice.

Bibliography

[CL87] Wen-Tsuen Chen and Lung-Lung Liu. A parallel approach for theorem proving in propositional logic. *Inf. Sci.*, 41(1):61–76, February 1987.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *JACM*, 7(3), July 1960.

[FC92] Ming-Yi Fang and Wen-Tsuen Chen. Vectorization of a generalized procedure for theorem proving in propositional logic on vector computers. *IEEE Trans. Knowl. Data Eng.*, 4(5):475–486, 1992.

[Gal77] Zvi Galil. On the complexity of regular resolution and the davis-putnam procedure. *Theoretical Computer Science*, 4(1):23 – 46, 1977.

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. Pasat — parallel satchecking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9(0):205 – 216, 2001. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

5 Valley Flooding Simulation in Cellular Automata

Apurva Kumar

5.1 Introduction

The power in cellular automata comes from being able to replicate complicated behaviour using simple rules and methods. Cellular automata (cellular automaton (*sing.*)) have been used successfully in modelling and solving real-world problems.

Their applications include cryptography, image processing and even music. This report looks at a particular application of cellular automata in simulating flooding of a valley. The implementation uses simple rules and demonstrates emergent complex behaviour from them. A complicated calculation of flow rates and soil erosion has been modelled in cellular automata already [7]. However, the intent of this report is to study various parallel algorithms, implement them in Lime and observe any possible speedup.

An implementation to this problem is presented using the Liquid Metal (Lime) Language which is being developed by IBM. This language is an extension to the Java programming language and provides extra support and speedup for parallel tasks specifically to be executed on an FPGA device. For this particular program, the FPGA will be simulated virtually.

The representation of cellular automata is particularly convenient for use in parallel programming. A commonly used parallel algorithm [5] involves dividing up the cellular automata into separate parts that can all be executed concurrently. This will be replicated in Lime.

Dividing up the cellular automata into different sizes or shapes affects time and space complexity of the algorithm. As with most parallel algorithms, real timing of the al-

gorithm depends on the available hardware. In this case, the FPGA hardware is being simulated so actual time speedup will not be observed. Instead a theoretical comparison of the algorithmic complexity will be used.

The aim is to maximise the parallel computation the most, but there is a balance between space complexity and parallelisation, and this is clearly seen in the implementation of cellular automata [5] and in this report. If the parallel pieces of the cellular automata are small, more parallelisation is achieved while using a lot of memory for the computation. If the sizes are large, the space complexity is reasonable but possible parallelisation of the algorithm is sacrificed.

Breaking up the cellular automata into the smallest possible independent pieces would mean that the most parallelisation can be hoped for. However, as the input size is increased (in a realistic hardware system), less parallelisation would be possible. The hardware is not expected to increase in efficiency with the size of the input so each thread would do more work in serial for a large input making its complexity depend on the size of the input.

5.2 Cellular Automata

A Cellular automaton is represented by an nxm grid of cells (n can equal m) each of which have a set of properties that can achieve a range of values. Each particular value that the cell can take on is called the state of the cell. Then state of the cellular automata is defined as the values of the properties of all cells at any one time. The cells are also referred to as finite state machines [1].

The state of a cell depends on the states of the cells around it so as to produce some relational behaviour. Cells react to their neighbours, changing their state depending on what the state of those cells around them are. The main observation is that the entire cellular automaton produces some overall intelligent or complicated behaviour by following some simple rules at the cellular level. The cells are oblivious to the patterns produced at the higher level.

What many designs vary on is the exact definition of "neighbours" of cells [8]. The cells are generally considered to be squares, but need not be. In fact, the cells can be any shape that is able to tessellate. Some examples can be seen in Figure 7.2.

For the implementation, the first drawing in Figure 7.1 is used to calculate neighbours of a cell, although this can be swapped out for any of the other representations discussed above. These decisions are tied to the phenomenon being simulated and even personal choice. As mentioned before, most cellular automata are used for their modelling powers and a model can be as simplistic or complicated as the designer chooses.

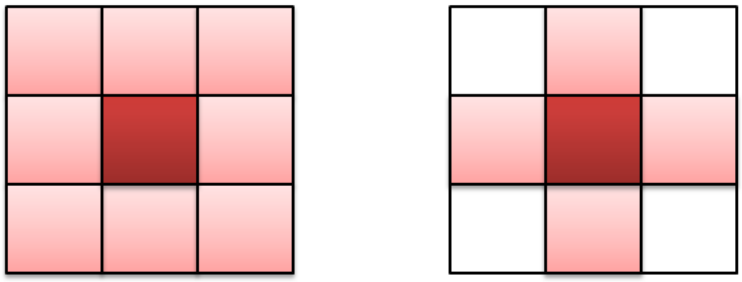


Figure 5.1: Two possible calculations of neighbours on square cells

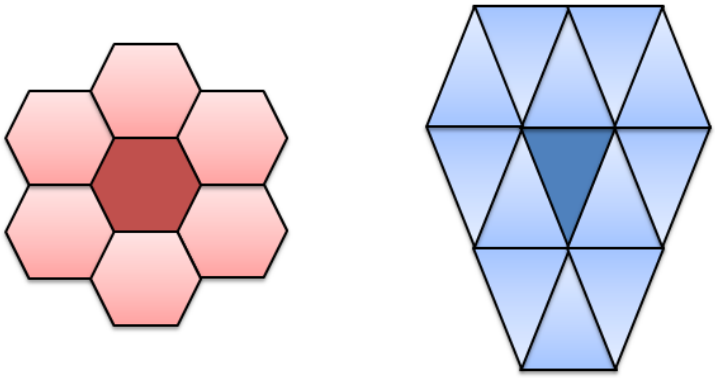


Figure 5.2: Hexagonal and triangular tessellation of cells

5.3 The Problem – Simulating a Valley Flooding

As stated before, there are many applications to cellular automata. One prominent use of cellular automata is in the modelling of physical phenomena as they provide a new and intuitive approach to the problem. One problem we will look at is the modelling of the flooding of a valley, in which cellular automata are commonly used as an assessment tool [4,7].

A particular cell on the grid is considered the source of the water (it could be a crack in a dam wall or a spring). It has a height that is the maximum height the water level can reach. All other cells have water level value of 0. This is the initial state of the cellular automaton. The end state is reached when all the cells have reached this maximum height of water. The speed with which the water spreads is determined by the rules applied to each cell. These are as follows:

1. A cell that has any one of its neighbours with twice the height of its own water, takes half that height as its own (rounding up).
2. Otherwise: A cell that has a majority of neighbours with heights higher than itself increases its own height by one.

Although we approach the problem in a very simplistic manner this can easily be made quite complicated by the addition of other simple rules. The cells are not limited to one property. In most cases, each cell has a number of properties. Each cell can, for example, have two heights: a land height and a water height, where the water height will always be above the land height. Also, after a certain amount of water height has built up on a specific land height, the land can be made to "give way" by reducing its height if it is a certain type land. In this manner, many complicated scenarios can be modelled easily using cellular automata with simple rules applied at a basic level. In Lime, the acceleration can only operate on simple types so objects cannot be used for this purpose. Thus, in the interest of a speedy solution, we will model each cell as a single integer which represents the height of the water level at a cell.

5.4 Parallel Algorithms for Cellular Automata

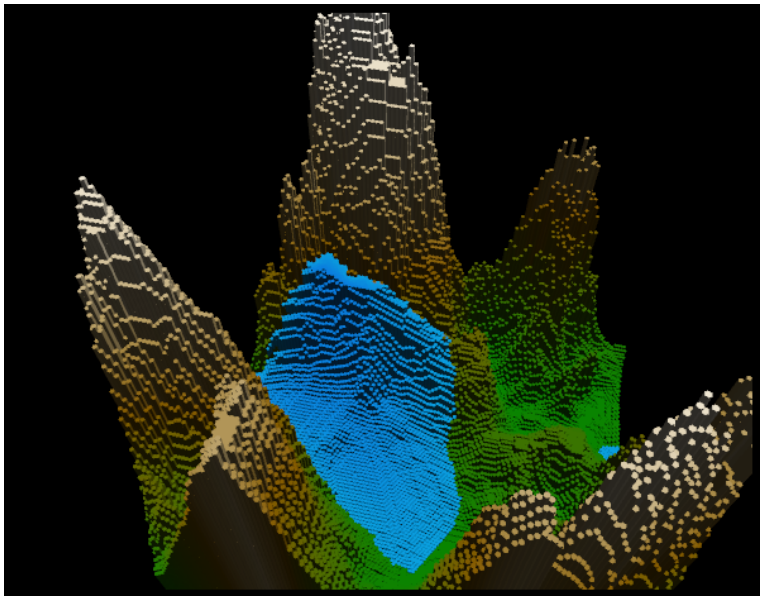


Figure 5.3: A 3D implementation of cellular automata, modelling the flooding of a valley using hexagonal cells and following the rules mentioned in Section 7.3 [3].

As discussed in Structured Parallel Programming [6], there are many patterns one can use to design solutions in parallel programming. Cellular automata are particularly suited to parallel algorithms due to their structure and algorithm for computation. The current generation relies only on the previous generation and therefore has no need for mutual exclusion in a particular generation. This makes a parallel solution much more simpler and easier to design. The following solutions model a cell as a single entity or process, but this does not necessarily need to be the case. A single process can contain a strip or a

block of cells in the cellular automata since they can be processed in any order [1].

Because cellular automata computation can be broken down into smaller subproblems of the same nature which are independent of each other, this structure naturally lends itself to the fork-join pattern for parallel programs where control flow can be broken up into parallel parts that execute together and later combine for the final result. Then simple iteration can be implemented over the generations to complete the computation.

Several other patterns can be used such as stencil and pipeline patterns depending on which part of the solution is focused on. As a map pattern, the same set of rules are applied to each input. A single input can be a single cell in the cellular automaton. Also, it produces an output of the same shape as the input because the number of cells does not change. Therefore cellular automata can also be modelled as a map pattern in relation to the computation for a single cell.

Since the computation of the next value of a cell depends on the cells nearest to it, this part would be suited to a stencil pattern. The stencil pattern is a generalisation of the map pattern where each subproblem can access not only the input (cell) but a number of its neighbours as well using a set of fixed offsets. This means that the order and computation of the neighbours method also stays constant which is another characteristic of cellular automata [6].

Cellular automata can also be modelled as a partially synchronised parallel pipeline too if you look at the movement from one generation to another. The structure of the pipeline would be a sequence of parallel and sequential stages, each pair of which represent a single generation. The sequential sections are synchronised only which represents the motion from one generation to the next. Since the operation that the parallel and sequential parts do are the same, a second way that the this can be structured is by having only one parallel section and one sequential section and the output of the sequential section feeds back to the parallel section as shown in Figure 7.4.

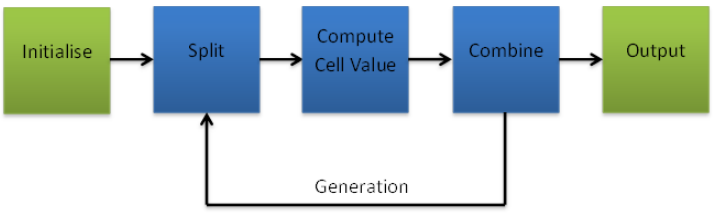


Figure 5.4: The different sections of a pipeline for a cellular automata problem

The implementation uses these patterns for different sections of the solution.

5.5 Development

When designing this solution, there are several ways to approach the problem. The biggest factor turned out to be the limitations of the language in designing a solution. The Lime language can only do parallel acceleration on simple types. This was an important factor to consider in order to make best use of the possible acceleration [2]. For this reason, the cells have been simplified to a single integer, which removes the needs for any abstraction on that level.

We start with a Grid class that defines the basic methods that a cellular automaton would need. A Grid is just a two-dimensional array of integers (cells) that can perform various actions. It also has two other variables that define the length and width of the array.

The important method here is the getNeighbours method, that defines how the neighbours are retrieved for each cell in the Grid. This varies depending on what the shape of the cell is as well what problem you are trying to solve.

<code/CAValley/Grid.lime 1>

```
1  /*
2   * Grid
3   * Version 1.0
4   *
5   * Contains the structure of the cellular automata.
6   */
7  public class Grid {
8    private int grid [][];
9    private int length;
10   private int width;
11
12   public Grid() {
13     Grid(5,5);
14   }
15   public Grid(int length, int width) {
16     this.grid = new int[length][width];
17     this.length = length;
18     this.width = width;
19     for (int i: 0::length-1) {
20       for (int j: 0::width-1) {
21         this.grid[i][j] = 0;
22       }
23     }
24   }
25
26   public int getCellValue(int x, int y) {
```

```
27      return grid[x][y];
28    }
29    public void setCellValue(int x, int y, int val) {
30      this.grid[x][y] = val;
31    }
32
33    public int getLength() {
34      return this.length;
35    }
36    public int getWidth() {
37      return this.width;
38    }
39
40    public int[] getNeighbours(int x, int y) {
41      <<getNeighbours 4>>
42    }
43    public void printGrid() {
44      for (int i = 0; i < this.length; i++) {
45        for (int j = 0; j < this.width; j++) {
46          System.out.print(grid[i][j]);
47        }
48        System.out.println();
49      }
50      System.out.println();
51    }
52  }
```

INCLUDED BLOCKS: 4 on page 103

Next we have a Valley class which holds the logic for the simulation of the flooding of a valley. This contains a Grid object that can be manipulated as well as a tuple array called info. Both these variables contain the same information, but the info array is the best representation for the parallelisation using Fork-Join, as can be seen later.

The Valley class has three major methods. The first one is the computeCell method that evaluates the current value of a cell given the state of its neighbours. The flood method is responsible for carrying out the computation for each generation in the cellular automaton. This will be discussed further on.

The third method is the outputGen method that is implemented below. It is responsible for writing the result of the parallel computation back to the original grid. And also to print it out so that intermediate checks can be performed.

<code/CAValley/Valley.lime 2>

```
1  /*
2   * Valley
3   * Version 1.0
4   *
5   * Contains the logic for how a valley is flooded.
6   */
7
8  import lime.util.Tasks;
9
10 public class Valley {
11   private static '(int,int,int,int,int,int,int,int,int)[] info;
12   private Grid world;
13
14   public Valley () {
15     Valley(5,5);
16   }
17   public Valley (int len, int wid) {
18     world = new Grid(len, wid);
19     info = new '(int,int,int,int,int,int,int,int,int)
20                   [world.getLength()*world.getWidth()];
21   }
22
23   static void outputGen(int[[]] gen) {
24     int [][] temp = new int[world.getWidth()][world.getLength()];
25     int x = 0;
26
27     for (int i = 0; i < 5; i++) {
28       for (int j = 0; j < 5; j++) {
29         temp[i][j] = gen[i];
30         x++;
31       }
32     }
33     world.setGrid(temp);
34     for (int i = 0; i < gen.length; i++) {
35       System.out.print(gen[i] + "/");
36     }
37     System.out.println();
38   }
39   static local int computeCell(
40               '(int,int,int,int,int,int,int,int,int)neighbours){
41     int cell = max = higher = 0;
42     int []neighbour = {neighbours.0, neighbours.1, neighbours.2,
43         neighbours.3, neighbours.4, neighbours.5, neighbours.6,
44             neighbours.7, neighbours.8};
45     <<computeCell 5>>
```

```
46      return cell;
47    }
48    static void flood() {
49      <<flood 6>>
50    }
51  }
```

INCLUDED BLOCKS: 5 on page 104, 6 on page 105

The following is an example main class that can be used to run and time the program. It is included here for completeness.

<code/CAValley/Main.lime 3>

```
1  /*
2   * Main
3   * Version 1.0
4   *
5   * Executes the simulation. Times the execution.
6   */
7  public class Main {
8    public static void main(String[] args) {
9      Valley valley = new Valley();
10
11     final long startTime = System.currentTimeMillis();
12     valley.flood();
13     final long endTime = System.currentTimeMillis();
14
15     System.out.println("Total execution time: " + (endTime - startTime));
16   }
17 }
```

The getNeighbours method returns a list of values for the eight neighbours of a cell in the following order.

<getNeighbours 4>

```
1      int[] temp = new int[8];
2
3      int xminusone = (x + length - 1) % length;
4      int xplusone = (x + 1) % length;
5      int yminusone = (y + width - 1) % width;
6      int yplusone = (y + 1) % width;
7
8      temp[0] = grid[xminusone][yminusone];
```

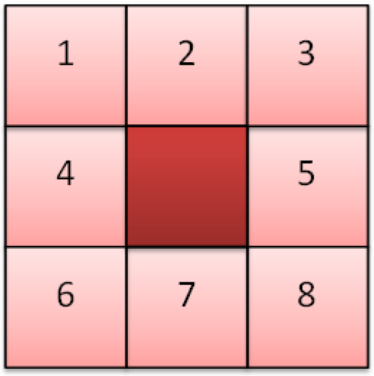


Figure 5.5: The order in which neighbours of a cell are calculated in the implementation

```
9     temp[1] = grid[xminusone][y];
10    temp[2] = grid[xminusone][yplusone];
11    temp[3] = grid[x][yminusone];
12    temp[4] = grid[x][yplusone];
13    temp[5] = grid[xplusone][yminusone];
14    temp[6] = grid[xplusone][y];
15    temp[7] = grid[xplusone][yplusone];
16
17    return temp;
```

USED IN: code/src/Grid.lime on page 100

The computeCell method in the Valley class holds the simple rules required to calculate the final value of a single cell in the current generation.

In this case, the method calculates the maximum level of water of the cells neighbours and then counts how many neighbours have heights more than the cell itself. With this information, it assigns the new value of the cell accordingly.

<computeCell 5>

```
1     for (int i = 1 i < 9; i++) {
2       if (neighbour[i] > max) {
3         max = neighbour[i];
4       }
5       if (neighbour[i] > neighbour[0]) {
6         higher++;
7       }
8     }
9     if ((neighbour[0] * 2) <= max ) {
10      cell = max%2 ? max/2 + 1 : max/2;
11    }
```

USED IN: code/src/Valley.lime on page 101

The flood method in the Valley class carries out the parallel computation. All the computation is done in a simple for-loop where each iteration represents a single generation.We first populate the tuple array info with each cell's value and a list of its neighbours values. In the tuple, this is just a list of nine integers, the first one being the value of the cell in question and the other eight are the values of its neighbours.

Next we define a constant version of the same tuple list to be fed to the parallel stream. This is a requirement of Lime. The split method takes in this constant list of tuples and splits it into 25 tasks (one for each cell because its a 5x5 grid). In the pattern this matches the fork method that splits the control flow into the specified number of pieces.

Each of these 25 tasks are then given the method computeCell to execute on its information. These are placed in a combination of braces: ([task]). This instructs Lime to accelerate the tasks happening inside these braces. The join method then takes the output of these tasks and combines them into a single constant output, in this case its a constant array of 25 integers. A last task is needed so that the output of the current generation is placed back into the cellular automaton and printed out so that intermediate checks can be implemented. The implementation of this method can be seen earlier. The instruction called finish waits for all the parallel computation to finish and the stream to close before continuing.

Several points need to be noted here. Even though the size of the grid can vary according the implementation, the number of splits and consequently the number of joins and output values need to be a compile time constant for Lime to be able to accelerate this process. This cannot work for a varying size of grid, so in fact the flexibility of the class cannot be used. I have used a 5x5 grid here as an example, but larger grids were tested and the results are included in a later section.

In this implementation, the 25 cells are divided into 25 tasks. The reason that it has been implemented in this way here is that the Lime implementation on an FPGA handles all the acceleration itself. By decomposing a problem into the simplest parts, the scheduler is given as much flexibility as possible. It is not necessary that these 25 tasks map to 25 separate threads on a hardware level (thereby making the program slower). Rather than guess the appropriate size of a subproblem, these details have been left to the acceleration provided by the language itself, giving it the freedom to arrange the tasks in the most efficient way possible. Further reasons for this choice are discussed under the Efficiency section of this report.

<flood 6>

```
1    for (int i = 0; i < 10; i++) {
```

```
2         //lengths and widths
3         int l = world.getLength();
4         int w = world.getWidth();
5         int count = 0;
6         //for each iteration
7         //get cell value and its neighbours
8         for (int x = 0 ;x < l; x++) {
9           for (int y = 0; y < w; y++) {
10            int[]temp = world.getNeighbours(x,y);
11            info[count] = `(world.getCellValue(x,y),
12                  temp[0],temp[1], temp[2], temp[3],
13                        temp[4], temp[5], temp[6], temp[7]);
14            count++;
15          }
16        }
17        //PARALLELISATION
18        `(int,int,int,int,int,int,int,int,int) [[]] constInfo = new
19            `(int,int,int,int,int,int,int,int,int) [[]] (info);
20        var t1 = Tasks.source(constInfo)
21            => #
22            => task split `(int,int,int,int,int,int,int,int,int) [[25]]
23            => ([ task [ (25) task computeCell ] ])
24            => task join int [[25]]
25            => task outputGen(int[[25]]);
26        t1.finish();
27    }
```

USED IN: code/src/Valley.lime on page 101

5.6 Alternate Implementations

Because of the requirements of the parallel section of fork-join, a second implementation where instead of a two dimensional array, a simple static final one-dimensional array can be used in order reduce the amount of computation required to convert the grid into tuples. The following implementation does not make a copy of the grid whereas the previous implementation copies it twice. The parallelisation part is omitted since it is a replication of the previous implementation. This reduces the space complexity without giving up any parallelisation possibility. However, it makes the input extremely rigid as the grid has to be immutable and very tedious to initialise. In addition to this, one cannot store the final result of the computation back into the same grid. Either the result would have to be printed out or it would have to be stored in a new array. Another disadvantage

of this implementation is that the grid is statically available to all programs. This makes it vulnerable to be changed by any external program or be corrupted.

Thus, this is by no means a more elegant solution to the problem.

<code/CAValley/World.lime 7>

```
/*
 * World
 * Version 1.0
 */
public class World
{
  static final int length = 5;
  static final int width = 5;
  static final int [[25]] world =     {0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,
                          0,0,1,1};

  static local int getPosition(int x, int y) {
    int pos = (x * width) + y;
    return pos;
  }

  static local int [] getNeighbours(int x,int y) {
    int [] neighbours = new int[9];

    int xminusone = (x + length - 1) % length;
    int xplusone = (x + 1) % length;
    int yminusone = (y + width - 1) % width;
    int yplusone = (y + 1) % width;

    neighbours[0] = world[getPosition(xminusone,yminusone)];
    neighbours[1] = world[getPosition(xminusone,y)];
    neighbours[2] = world[getPosition(xminusone,yplusone)];
    neighbours[3] = world[getPosition(x,yminusone)];
    neighbours[4] = world[getPosition(x,yplusone)];
    neighbours[5] = world[getPosition(xplusone,yminusone)];
    neighbours[6] = world[getPosition(xplusone,y)];
    neighbours[7] = world[getPosition(xplusone,yplusone)];
    neighbours[8] = world[getPosition(x,y)];

    return neighbours;
  }

  static local int calculateCellValue(int [] neighbour) {
    int cell = max = higher = 0;
```

```
40      for (int i = 1 i < 9; i++) {
41        if (neighbour[i] > max) {
42          max = neighbour[i];
43        }
44        if (neighbour[i] > neighbour[0]) {
45          higher++;
46        }
47      }
48      if ((neighbour[0] * 2) <= max ) {
49        cell = max%2 ? max/2 + 1 : max/2;
50      }
51      return cell ;
52    }
53  }
```

5.7 Test Cases and Intermediate Results

In order to appropriately identify intermediate results, the grid values were printed out after each generation and checked against pre-computed values. The test was performed on a 5x10 grid, represented as a matrix below and the initial values were:

$$\begin{pmatrix} 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The next generations were computed based on applying the computeCell method to each cell. The outputs were:

$$\begin{pmatrix} 10 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 10 & 6 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 6 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 10 & 7 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 7 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and so on until the final state which is:

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \end{pmatrix}$$

The algorithm here was just used to model the flow of water, but it could be used to calculate the number of generations that were required to flood the space. This could be a valley, but could equally be a tank or a basin. The generations tend to correspond to time steps so to calculate the amount of time required to fill a space could also be calculated from this implementation.

These intermediate results were taken at each generation to ensure that the correct computation was being done by the parallel algorithm.

5.8 Efficiency

For a problem with an nxn grid of cells and m generations for a solution, a naive linear implementation would have each cell being computed one after the other: $O(n^2)$. For m iterations the complexity of the final solution would be $O(mn^2)$.

Usually the grid is much larger than the number of generations. So when we try to make this algorithm efficient, we are going to concentrate on as much parallelisation as possible on the individual cells.

In the fork-join implementation, the grid is broken down into an appropriate number of subtasks. In the absolute best-case scenario (considering you have a processor for each cell in the grid) the complexity for one generation is $O(1)$. For m generations, this would turn out to be $O(m)$. But since this is probably impractical, consider a more reasonable case.

Suppose that the amount of work that ended up being done by a particular thread was a single column of the grid. The complexity in this case would be $O(n)$ and for m generations, this could be $O(mn)$. And since m is usually much lower than n, we could assume $O(n)$. Comparing this to the linear implementation, the parallel algorithm runs twice as fast as the serial one. This is quite a substantial speedup, given the reasonableness of our assumptions.

Another comparison is space complexity. Already, it is clear that the parallel implementation will take much more space than the linear implementation. Linearly, the implementation requires $O(n^2)$ space which is the size of the grid. Each cell has eight neighbours,

therefore, by dividing the problem up into single-cell subproblems would increase the space complexity eight-fold.

Considering a row of cells would mean a little more than $3n$ would be required to include the neighbours as well. This is already a great improvement and this shows that choosing the sub-problem wisely can reduce the space complexity greatly. This would give a stronger argument of a block of cells to be considered as a subproblem as some of the neighbours are the cells that need to be computed themselves.

This analysis shows that there is a tradeoff between space and time complexity on this problem. The greater the number of subproblems, the more likely it is that more parallelisation occurs, but the larger space is required for the computation. The larger the subproblem, the less space is needed in relation to the amount of cells. But there is some obvious parallelisation that cannot be taken advantage of.

In this implementation the subproblem size has been chosen to be 1. This would mean that the number of tasks produced would be n^2 and the potential acceleration can only be limited by the hardware itself. However, the space complexity would be eight times that of a sequential implementation: $O(8n^2)$.

Firstly, in this particular solution, the size of the grid was taken to be extremely small, and therefore did not directly influence the efficiency of the algorithm

Secondly, in the interest of parallelisation and relying on the optimised implementation of the Lime language on an FPGA, it was intended to give the compiler as much freedom as possible in terms of paralellisation and computation power.

Finally, due to the basic nature of the Lime language, it was not possible to pass in an array of tuples to the split method in order to break the problem size into larger chunks. Since the split method in Lime can only accept basic types and tuples to work with, the designer would only be left with breaking down the problem completely into single-celled subproblems or have a very long tuple of integers that would require strong rules to be adhered to when identifying neighbours of a cell. Therefore, it is definitely one of the ways that the current solution can be improved, and with a more general language can be easily achieved with more readable code.

5.9 Conclusion

Cellular automata are extremely useful for modelling physical problems and use simple rules to emulate seemingly complicated behaviour of a whole system. They are almost ideal for parallel processing as cells in a particular generation do not depend on each other for their value computation. This means that they can be computed independently without the need for mutual exclusion and locks. Several parallel programming patterns can

be applied to cellular automata including pipelining, fork-join and stencil patterns.

In this report, a Fork-Join implementation of cellular automata computation was discussed with the Lime (Liquid Metal) Language developed by IBM, which is an extension applied to the Java language. Given the simulated resources of an FPGA board, rather than the real hardware, real-time acceleration of the implementation was not found. Lime is a language still under construction and very much in its infancy. Therefore, due to its limitations as a language for the moment, certain generalised implementations were also not possible.

However, a solution was obtained for which, in principle, strong arguments can be made, both in efficiency of the algorithm used and the resources at hand. By dividing the solution in to as small pieces as possible, greater advantage of parallelisation was obtained. However as soon as the problem size decreased, the space complexity of the solution increased greatly. Thus, it was seen that a correct selection of base subproblem size can allow for a tradeoff between space and time complexity to make a more efficient algorithm in all areas.

Bibliography

[1] Gregory R Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley, 1999.

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: The liquid metal programming language – language reference manual. Technical report, IBM Research Division, 2010.

[3] Divan Burger, Karl Zoller, Megan Duncan, and Apurva Kumar. Hexcore: Cellular automata simulator and editor, 2011.

[4] Johnny Douvinet, Daniel Delahaye, and Patrice Langlois. Application of cellular automata modelling to analyse the dynamics of hyper-concentrated stream flows on loamy plateaux. In *The 7th International Conference on Hydroinformatics*. IWA Publishing, 2006.

[5] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. 1993c.

[6] Michael McCool. *Structured parallel programming: patterns for efficient computation*. Elsevier / Morgan Kaufmann, Amsterdam, 2012.

[7] Peter C. Nwilo, D. Nihinlola Olayinka, and Ayila E. Adzandeh. Flood modelling and vulnerability assessment of settlements in the Adamawa state floodplain using GIS and cellular framework approach. *Global Journal of Human Social Science*, 12(3), 2012.

[8] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Champaign, IL, 2002.

6 Primality Testing Using the Miller-Rabin Algorithm

D. Adam Lazzarato

6.1 Introduction

For my project, I chose the subject of primality testing. Primality testing is the process in determining if an integer has no other integral factors other than 1 and itself. If this is the case, the number is said to be *prime*. If other integral factors are found, the number is said to be *composite*. The prospect of quickly being able to determine the primality of an arbitrarily long number is very intriguing. Primes of the form $2^n - 1$ are called *Mersenne primes*. There is a lot of research effort going into the search for Mersenne primes [Aro]. This area is also interesting to me because testing for primality is essential in fields like cryptography. As communication systems in the world become more advanced, and the requirement of secure communication lines grows, strong cryptographic algorithms are needed to make it difficult or altogether infeasible for adversaries to decrypt an intercepted secret message.

6.2 The RSA Public Key Cryptosystem

A good example of prime numbers in a cryptography system is in the RSA public key cryptosystem. Public key cryptography enables secure communications between parties. Each of the two participants in the communication generate a *public key P* and a *secret key S*. Public keys can be made public and given to anyone. Secret keys are known only by the one party who generated it. These keys are used to enable communication between the two parties. They will be used to *encrypt* and *decrypt* secret *messages*. If the secret key

is compromised, then the security of the communication channel is compromised as well. These keys act as methods that encode a message M. Both P and S are the methodal inverses of each other. That is, encoding and decoding M with P and S in any order will reproduce the message M.

$$M = S(P(M))M = P(S(M))$$

The strength of the RSA public key cryptosystem comes from the fact that it is very hard to factor the product of two large prime integers. This is due to how the public and secret keys are created: [Mit]

- Select two large prime numbers p and q at random with p is not equal to q. Large is considered to be around 1024 bits for both p and q.
- Calculate $n = pq$.
- Calculate $\phi(n) = (p-1)(q-1)$.
- Choose e such that $1 < e < \phi(n)$ and $\gcd(e, n) = 1$.
- Calculate d, the multiplicative inverse of e: $(d \text{ x } e) \pmod{\phi(n)} = 1$.
- $P = (e, n)$ is the *RSA public key*.
- $S = (d, n)$ is the *RSA secret key*.

The larger the p and q values are, the larger n will be, and the more difficult it will be for adversaries to factor.

For example (using small numbers):

- $p = 3, q = 11$
- $n = 33$.
- $\phi(n) = 2x10 = 20$.
- $e = 7$
- $d = 3$.
- $P = (7, 33)$ is the *RSA public key*.
- $S = (3, 33)$ is the *RSA secret key*.

This example will not produce a very large universe for the messages to exist in, since n is very small. That is another reason to use large p and q values, so their product n is large.

The standard method of sending an encrypted message using RSA encryption from Bob to Alice proceeds like so:

- Bob finds Alice's public key P_A.
- Bob sends Alice the *ciphertext* $C = P_A(M)$.
- Alice decrypts the cyphertext using her secret key by computing $S_A(C) = S_A(P_A(M)) = M$.

Since Alice is the only one who can decrypt the message using S_A, the message from Bob to Alice is secure. If Bob wants to send a message to Alice, Bob must choose a secret message M to encrypt. Let $M = 13$:

- Bob encrypts M using the Alice's public key: $C = P_A(M) = M^e \bmod n = 13^7 \bmod 33 = 7$
- Now, Alice is the only one who can decrypt the ciphertext. They can decrypt $C = 7$ like so: $M = S_A(C) = C^d \bmod n = 7^3 \bmod 33 = 13$.
- Alice now has the original $M = 13$.

If an adversary intercepts the ciphertext of a communication, they will be tasked with finding the factors of n to determine the public key and secret key belonging to the receiver. This is a very hard problem [Gal]. If the adversary found the factors p and q of n, the remaining tasks the to reconstruct the public and secret keys are relatively easy. In multiple decades of research, the problem of factoring large prime numbers has still proven to be extremely difficult. A major breakthrough in number theory algorithms is needed to make this problem easier, and to eventually crack RSA. In the meantime, it is advisable to use integers with hundreds or even thousands of bits for p and q to combat even the strongest and most resource-wealthy adversary [Kar].

6.3 Primality Testing

Primality testing is the process of finding large primes. There are many large prime numbers, so it is feasible to randomly test large numbers until a prime is found. The *Prime number theorem* approximates the density of prime numbers up to a given number n to be $n/\log n$. Randomly selecting an integer n will yield a $1/\log n$ probability that it is prime.

A *pseudoprime* number is a number that passes a primality test for a some input but is actually composite. A *base-a pseudoprime* is a number n that is composite and $a^{n-1} \equiv 1 \bmod n$. Fermat's theorem shows that if a given number n is prime, then n satisfies $a^{n-1} \equiv 1 \bmod n$ for all $1 < a < n$. Finding an a that satisfies the equality implies the compositeness of n. Using $a = 2$ gives good results, but is it still possible to be incorrect.

If $a = 2$ satisfies the equation, then n is either prime or a base-2 pseudoprime. As a result, the following procedure is generated to test for pseudoprimality:

<code/Pseudoprime.txt 1>

```
1 Pseudoprime(n)
2   if (Modular−Exponentiation(2,n−1,n) != 1) return composite
3   else return prime
```

The *Modular-Exponentiation(a, b, n)* method is implemented later, in the Miller-Rabin algorithm, and is a highly optimized way to calculate $a^b \bmod n$. If the method returns composite, then n is definitely composite. However, if the method returns prime, there is a slight chance that n can still be composite. There is a possiblity that n is a base-2 pseudoprime, but that probability is very small. There are composite numbers called *Carmichael numbers* [Cor09] [Gal] that can fool the above method for all a such that $0 < a < n$. Here lies the need for the Miller-Rabin algorithm.

6.4 Miller-Rabin Algorithm

The Miller-Rabin primality test is an algorithm that tests for the primality of a number n. It can be assumed that this number is odd and greater than 2, or else the result is trivial. The Miller-Rabin primality test makes two improvements on the previous *Pseudoprime* test:

- Tries multiple, random values for a. Using multiple values for a will evaluate the possiblity that n is base-2 pseudoprime, a possiblility that would not have been caught using the previous method.
- During the modular exponentiation computation, the algorithm looks for a *nontrivial square root of 1, mod n*. A number x is a nontrivial square root of 1 if x satisfies $x^2 \equiv 1 \bmod n$ and x is not a trivial square root, 1 or -1. A nontrivial square root of 1, mod n can only exist if the number is composite. This is an optimization in finding composite numbers over the previous method.

The input is an odd number n to be tested for primality. Let s be a predetermined number of randomly chosen base values a to be tried. Let a be randomized using a random number generator that works in the interval 1 to $n - 1$. The Miller-Rabin test uses a method *witness(a,n)* to determine if a proves the compositeness of n. The *witness(a,n)* method is an improvement of the *Pseudoprime* test, $a^{n-1} \not\equiv 1 \pmod{n}$.

The Miller-Rabin test is a very effective primality test. The main variables in the algorithm are the predefined number of iterations of the test, s and the randomization of a. If a sufficient value for s is chosen, there is a very slight chance that the algorithm is incorrect.

As for how to determine s, if n is an odd composite number there exist at least $(n-1)/2$ witnesses to n's compositeness. This relation is a good guideline on how to set s because it is provides a very good probability that a witness will be found.

The error rate the the Miller-Rabin algorithm is extremely low. For any odd number n, and valid s assignment, the probability that the Miller-Rabin test using n and s as parameters makes error is at most 2^{-s}. This is another way to determine your assignment for s. Since many random values for a are being tested, every iteration of the test reduces the probability that the test will make an error [Cor09].

6.5 Serial Implementation in Java

For my serial implementation of the Miller-Rabin algorithm, I chose to use the *BigInteger* class instead of using *int* for storing integers. This is because *int* can only store up to 32 bits. This amount of bits does not create a long enough integer to be useful for cryptosystems. The *BigInteger* class, on the other hand, does not have an upper bound for the amount of digits it can store. For an example of how the *BigInteger* class is used, the serial version of the *modularExponentiation* method is shown below.

A parallel implementation of the Miller-Rabin algorithm using the Lime language is presented later.

The s variable is predetermined by the user, and is the number of iterations of the Miller-Rabin test. There are no strict rules on how to assign s, but there is certainly no reason to make it greater than n. Any additional iterations after n iterations will raise the possibility of duplicating calculations, which produce the same results. But, as per the equation above, the number of witnesses in an odd composite number is $(n-1)/2$. This provides a good guess at how to set s.

The final method to be called in all paths of execution is the *finish* method. This method calculates the execution time in nanoseconds and the primality result. The method also exits with a status code based on the primality result. The status codes are: 0 if n is prime, 1 if n is composite, -1 if there is an error or if $n = 1$. (1 is neither prime nor composite).

The main method collects the input and does some overhead before proceeding to call the Miller-Rabin test. First, the start time of the program is taken. The variable n is intialized and then assigned a value from the input arguments. If $n = 1$, then the program terminates with the result "NEITHER.". If $n = 2$, "PRIME." is returned for correctness of primality testing. The case where $n = 2$ is a case that the Miller-Rabin algorithm does not catch, though 2 is a prime number. Finally, if n is even, (or, if $n \bmod 2 = 0$) there is no point to proceed because the number is definitely composite. Once the overhead is taken care of, the Miller-Rabin algorithm is called with n and s as parameters.

<code/MillerRabin.java 2>

```
import java.math.*;
import java.util.*;

public class MillerRabin {
    private static final Random rnd = new Random();
    public static final int s = 40;

    // LEP links to implementations below:
    <<Random 4>>
    <<MillerRabinTestSerial 3>>
    <<WitnessSerial 5>>
    <<ModularExponentiationSerial 6>>

    public static void finish(long start, int code, String result){
      long end = System.nanoTime();
      System.out.println((end - start));
      System.out.println(result);
      System.exit(code);
    }

  public static void main(String[] args) {
    long start = System.nanoTime();

    BigInteger n = BigInteger.ZERO;

    try{
      n = new BigInteger(args[0]);
    } catch (ArrayIndexOutOfBoundsException e){
      System.out.println("Please provide an odd number to test for primality "
          + "as a command line argument");
      System.exit(-1);
    }

    if (n.compareTo(BigInteger.ONE) == 0) {
      finish(start, -1, "NEITHER.");
    } else if (n.compareTo(new BigInteger("2")) == 0) {
      finish(start, 0, "PRIME.");
    } else if (n.mod(new BigInteger("2")) == BigInteger.ZERO){
      finish(start, 1, "COMPOSITE.");
    } else {
      if (millerRabinTest(n, s)) {
        finish(start, 0, "PRIME.");
      } else {
        finish(start, 1, "COMPOSITE.");
```

```
45          }
46        }
47      }
48  }
```

INCLUDED BLOCKS: 4 on page 119, 3 on page 119, 5 on page 120, 6 on page 121

The Miller-Rabin test takes n, the number being tested for primality, and s, the predefined maximum number of iterations of the Miller-Rabin test. In each iteration, a new random number between 1 and $n-1$ is generated. Then, *witness* determines if a is a witness to the compositeness of n. If so, n is declared composite. If none of these iterations produce a witness, then the algorithm returns that n is prime.

<MillerRabinTestSerial 3>

```
1  public static boolean millerRabinTest(BigInteger n, int s){
2    BigInteger a;
3    for (int j=1; j<=s; j++) {
4      a = random(BigInteger.ONE, n.subtract(BigInteger.ONE));
5      if (witness(a, n)){
6        return false; // composite
7      }
8    }
9    return true; // prime
10 }
```

USED IN: code/MillerRabin.java on page 118

The serial implementation has the added benefit of being able to use the *BigInteger* java class. An example of how the *BigInteger* class is used is in the random number generator that generates a, the test values that *witness* uses. Since a random number generator for ranges doesn't exist, one had to be written.

The random number generator used by the Miller-Rabin test to randomize a needs to work in the interval 1 to $n-1$. Since the *BigInteger* package is being used, the only randomization is in a constructor that uses the number of bits as an argument. First, subtract the *min* from the *max* to get the *range* of the interval. Then, take the bit-length of the *range*. Create new *BigInteger* variables using the bit length and the random number generator *rnd*. Wait until a value is found that is less than the *range* value. Now, the method adds *min* back to the result. The result is now a random value between *min* and *max*.

<Random 4>

```
1  // Generates a random integer in [min, max]
```

```
2  public static BigInteger random(BigInteger min, BigInteger max) {
3      BigInteger range = max.subtract(min);
4      int length = range.bitLength();
5      BigInteger result = new BigInteger(length, rnd);
6      while (result.compareTo(range) >= 0) {
7          result = new BigInteger(length, rnd);
8      }
9      return result.add(min);
10 }
```

USED IN: code/MillerRabin.java on page 118

The *witness* method first generates a random number a to be tested as a witness for n. The reason the line is commented out is because calls to Math.random are considered global and forbidden in Lime. The implementation requires a random number generator that can generate random numbers in a range 1 to $n-1$ using no global methods. Next, *witness* determines a representation for $n-1$ of the form $n-1 = 2^t - u$. The procedure then calculates $a^{n-1} \bmod n$ by calculating $a^u \bmod n$ and squaring the result t times. These squarings satisfy $x_i \equiv a^{2^i u} \bmod n$ for $0 \leq i \leq t$. If a nontrivial square root of 1 has been discovered, the algorithm has found a witness to the compositeness of n [Cor09]. By Fermat's theorem, if the algorithm has found $x_t = a^{n-1} \bmod n \neq 1$, then n is definitely composite.

<WitnessSerial 5>

```
1  public static boolean witness(BigInteger a, BigInteger n){
2    // let t, u be s.t. t>= 1, u is odd and n-1 = 2^t.u
3    int t=0;
4    BigInteger u = n.subtract(BigInteger.ONE);
5    BigInteger two = new BigInteger("2");
6    while (u.mod(two).equals(BigInteger.ZERO)) {
7      u = u.divide(two);
8      t = t + 1;
9    }
10
11   BigInteger x = BigInteger.ZERO;
12   BigInteger x_m1 = modularExponentiation(a, u, n); // x_0
13
14   for (int i=1; i<=t; i++) {
15     x = (x_m1.pow(2)).mod(n);
16
17     if (x.compareTo(BigInteger.ONE) == 0 &&
18         x_m1.compareTo(BigInteger.ONE) != 0 &&
19       x_m1.compareTo(n.subtract(BigInteger.ONE)) != 0) {
```

```
20          return true;
21        }
22        else {
23          x_m1 = x;
24        }
25      }
26      return x.compareTo(BigInteger.ONE) != 0;
27    }
```

USED IN: code/MillerRabin.java on page 118

The *modularExponentiation* method computes $a^b \bmod n$ using the binary representation of b. The method of doing this calculation using the binary representation of the exponent is called *repeated squaring* [Cor09]. The *while* loop does some necessary overhead calculations: it computes the k, the length of the binary representation of b in bits.

<ModularExponentiationSerial 6>

```
1   public static BigInteger modularExponentiation(BigInteger a, BigInteger b, BigInteger n){
2     BigInteger d = BigInteger.ONE;
3     String b_binary = b.toString(2);
4     int k = b_binary.length();
5
6     for (int i=0; i<k; i++){ // iterate from leftmost bit
7       d = (d.multiply(d)).mod(n);
8       if (b_binary.charAt(i) == '1'){
9         d = (d.multiply(a)).mod(n);
10      }
11    }
12    return d;
13  }
```

USED IN: code/MillerRabin.java on page 118

6.6 Parallelization

There are two major points in which the Miller-Rabin algorithm could be parallelized:

- There are s calls to *witness*. Each call is independent of the others, which makes the order of the calls irrelevant. These calls can be parallelized to occur simultaneously.

- There are random values for a generated. Since n can potentially be very large, and s random numbers must be generated, the task of generating these random numbers can take time.

The parallelization scheme of my implementation is a MapReduce scheme [MRR12]. Calls to *witness* are executed in parallel, as a "map" step. The results of the calls are all "reduced" to one single answer using the *endit* method. This parallelization method has been used before to execute the Miller-Rabin test in parallel [Aba]. Also, random number generation for a occurs inside *witness*. Therefore, the first item is taken care of.

Another possibility for parallelization for this problem is to have workers that are responsible for executing methods rather than following input through their entire life span. For example, there are workers that only execute *witness* when there is input available. Other workers take care of the random number generation, and so on. This method would create a large and unnecessary amount of overhead and communication dependency between workers. This method is not favourable and would be much more difficult to implement.

6.7 Parallel Implementation

The major difference between my serial and parallel implementations is that my serial program calls external libraries. Their library equivalents could be used. Also, very importantly: a random number generator could be called to generate random numbers for a. None of this is true for the parallel implementation. Since global methods can not be called, methods like *myPow* and *intToBinary* must be implemented. Also, as a result of global methods not being available, the *BigInteger* library can no longer be used. So, the parallel implementation of the algorithm must use the standard integer library.

My implementation of the Miller-Rabin algorithm in Lime is inside a single .lime file. FPGA acceleration can also be enabled, making it possible to run this algorithm on hardware. A value is assigned for s, and all of the methods from the parallel implementation are declared, plus *myPow* and *intToBinary*.

<code/MillerRabinLime.lime 7>

```
1 import java.math.*;
2 import java.util.*;
3
4 public class MillerRabinLime {
5     public static final int s = 15;
6     <<Main 8>>
7     <<MillerRabinTest 9>>
8     <<Witness 10>>
9     <<MyPow 11>>
```

```
10      <<ModularExponentiation 12>>
11      <<EndIt 14>>
12      <<IntToBinary 13>>
13  }
```

INCLUDED BLOCKS: 8 on page 123, 9 on page 124, 10 on page 124, 11 on page 125, 12 on page 126, 14 on page 127, 13 on page 126

The *main* method is implemented exactly the same as the serial implementation. However, the parallel implementation is using the *integer* library.

<Main 8>

```
1  public static void main(String[] args) {
2    long start = System.nanoTime();
3    int n = 0;
4
5    try{
6      n = Integer.parseInt(args[0]);
7    }
8    catch (ArrayIndexOutOfBoundsException e){
9      System.out.println("Please provide an odd number to test for primality "
10         + "as a command line argument");
11     System.exit(-1);
12   }
13
14   if (n == 1) {
15     finish_exit(start, -1, "NEITHER.");
16   } else if (n == 2) {
17     finish_exit(start, 0, "PRIME.");
18   } else if (n % 2 == 0){
19     finish_exit(start, 1, "COMPOSITE.");
20   } else {
21     millerRabinTest(n);
22   }
23 }
```

USED IN: code/MillerRabinLime.lime on page 122

This *millerRabinTest* method is where the parallelism occurs. First, the input number n is copied s times into an array r is used as the source for the Lime task. Then, the task is created, using r as a source. The task is to call *witness*, and then redirect the output of *witness* into another method, *endit*. Calling *witness* is the "map" stage. Input is copied to

s different data blocks, and the *witness* function is called on all of them. What remains is a set of results *r* that is of the same type and size as the input. Then, *endit* is the "reduce" stage. The results array *r* is scanned for the results of *witness*. We are specifically interested in if any witnesses to the compositeness of *n* were found. Together, these method calls execute the MapReduce parallelization scheme. Finally, the Lime task is finished.

<MillerRabinTest 9>

```
1  public static void millerRabinTest(int n){
2    int[] r = new int[]{n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n};
3
4    var wit = Tasks.source(r)
5      => ([ task witness ])
6      => task endit;
7
8      wit.finish();
9  }
```

USED IN: code/MillerRabinLime.lime on page 122

The *witness* method acts as the "map" step in the MapReduce parallelization scheme. The biggest difference to the *witness* method is that it is being called as a part of a Lime task. Also, random number generation is now the responsibility of this method. Other than that, the implementation details and calculations are very similar. Again, note that the *integer* library is being used since Lime recognizes all external libraries as global and therefore are unable to be called. The biggest problem with this method is that there is no working random number generator for a in the range of 1 to $n-1$. The serial implementation had one, but that was because the *Math* library was able to be called. Because of this, the implementation isn't quite complete.

<Witness 10>

```
1  public static local int witness(int n) {
2    // int a = 1 + (int)(Math.random() * ((n - 1) + 1));
3
4    // let t, u be s.t. t>= 1, u is odd and n-1 = 2^t.u
5    int t=0;
6    int u = n - 1;
7    while(true){
8      u = u / 2;
9      t = t + 1;
10     if(u % 2 != 0) {
11       break; // u is odd
12     }
13   }
```

```
14
15    int x = 0;
16    int x_m1 = modularExponentiation(a, u, n); // x_0
17
18    for (int i=1; i<=t; i++) {
19      x = mypow(x_m1, 2) % n; //(x_m1.pow(2)).mod(n);
20
21      if (x == 1 &&
22          x_m1 != 1 &&
23        x_m1 != (n-1)) {
24        return 1; // composite 1
25      }
26      else {
27        x_m1 = x;
28      }
29    }
30    if (x != 0){
31      return 1; // composite 1
32    }
33    return 0; // prime 0
34  }
```

USED IN: code/MillerRabinLime.lime on page 122

MyPow is a method that calculates $base^{exp}$. This method had to be implemented because calls to the Math library are not considered as local using Lime. The result is calculated by using the shift operator, as provided by Java. Shifting the exp variable is an improvement over the more primitive approach of multiplying $base * result$, (exp number of times). This method reduces the amount of multiplications necessary to compute the result [Mic]. The result is returned as an integer. This value is used in the *witness* method.

<MyPow 11>

```
1   public static local int mypow(int base, int exp) {
2     int result = 1;
3       while (exp != 0) {
4           if ((exp & 1) == 1) {
5               result *= base;
6           }
7           exp >>= 1;
8           base *= base;
9       }
10      return result;
11  }
```

USED IN: code/MillerRabinLime.lime on page 122

The *modularExponentation* method is showed for completeness. Note that the method *Integer.toBinaryString(int a)* is unavailable in Lime, so one is presented below.

<ModularExponentiation 12>

```
1  public static local int modularExponentiation(int a, int b, int n){
2    int d = 1;
3    String b_binary = intToBinary(b);
4
5    int k=0;
6    char tmp;
7    while(true){
8      try{
9        tmp=b_binary.charAt(k);
10       k++;
11     }
12     catch (StringIndexOutOfBoundsException e){
13       break;
14     }
15   }
16
17   for (int i=0; i<k; i++){ // iterate from leftmost bit
18     d = (d * d) % n;
19     if (b_binary.charAt(i) == '1'){
20       d = (d * a) % n;
21     }
22   }
23   return d;
24 }
```

USED IN: code/MillerRabinLime.lime on page 122

IntToBinary is a method that takes and integer *a* as input and returns a String of the integer's binary representation. This method had to be implemented because calling *Integer.toBinaryString(int a)* is considered global by default. The method can handle up to 32 bit integers. The algorithm stops execution when the necessary binary digits have been generated.

<IntToBinary 13>

```
1  public static local String intToBinary(int a) {
2      String binary = "";
```

```
3       int mask = 1;
4       for(int i = 0; i < 31; i++){
5           if ((mask&a) != 0) {
6               binary = "1" + binary;
7           }
8           else {
9               if(mask>a){
10                  break;
11              }
12              binary = "0" + binary;
13          }
14          mask = mask * 2;
15      }
16      return binary;
17  }
```

USED IN: code/MillerRabinLime.lime on page 122

The final method to be called in all paths of execution is the *endit* method. This is the "reduce" step in the MapReduce parallelization scheme that is being exploited in the parallel implementation. This method scans the results array *r* of the calls to *witness* for any witnesses to the compositeness of *n*. If one is found, *n* is declared composite and the program is halted immediately.

<EndIt 14>

```
1   static void endit(int n){
2     if(n==0){
3       System.out.println("PRIME.");
4     }
5     else if(n==1){
6       System.out.println("COMPOSITE.");
7       System.exit(0);
8     }
9     else{
10      System.out.println("ERROR.");
11      System.exit(0);
12    }
13
14  }
```

USED IN: code/MillerRabinLime.lime on page 122

6.8 Possible Improvements to Parallel Implementation in Lime

There are improvements that can be made to the parallel implementation in Lime. Many of these improvements deal with the limitations of Lime. Firstly, I was unable to use the *BigInteger* class to handle arbitrarily long integers to test for primality. This would have increased the possible usage of the program up to even the extreme standards of modern cryptographic algorithms such as RSA [Kar]. For example, prime numbers for p and q in the RSA algorithm could be generated. With the standard integer library, this is not possible. This also forced me to implement my own power and integer to binary string methods. These methods are not as optimized or thoroughly tested as the standard library methods.

Next, using an array in the *millerRabinTest* function (one element for each iteration of the test) to store the data and results to facilitate the MapReduce parallelization scheme in Lime is not ideal, either. This adds overhead in space by populating the array initially. When testing large numbers for primality, and s is large, this array will take up an unnecessarily large and unnecessary amount of space in memory. Also, the results of the calls to *witness* must be searched in the array after the tasks are done. The System.exit(0); that is done in my parallel implementation is there to exit as soon as n is declared composite. This is done to achieve minimum runtime, and to avoid the searching of the results array.

As for the parallelization scheme, I believe I have found the best method of splitting the Miller-Rabin algorithm into parallelizable tasks. It is a method that has been demonstrated before [Aba]. It is a very simple and easy to see parallelization scheme: There are s iterations of the Miller-Rabin test that need to be executed. Each one computes a new random value a and executes the Miller-Rabin test. Other parallelization schemes such as a functional parallelization would not be as effective. For example, in a functional parallelization, different workers are started and their only job is to execute functions such as the witness function and the modular exponentiation. This method would not work as well because it would add to the amount of overhead being done by communicating the intermediate results of the test.

6.9 Results

Since there are so many differences between my implementations, comparing the running times isn't fair as it could be. The serial version has a better structure and uses library methods, as explained in the previous section. These improvements include the use of the *BigInteger* class. Additionally, the serial implementation is not used on the FPGA.

Therefore, the data types and environments that the different implementations are running in are completely different. The timing values also depend on the assigned value for s.

Testing the first 1000 primes [Alf] and the first 1000 composite numbers [Nat] with my serial implementation of the Miller-Rabin algorithm, the running times increased approximately linearly with s. For example, using the 2000 test cases mentioned before, the average running time for the test was 1.46ms for $s = 20$ and 11.76ms for $s = 200$, an unnecessarily large value for s. The serial implementation was also able to handle large prime numbers. I tested all of the primes on Chris Caldwell's page [Cal] successfully.

The ideal running time for the parallel implementation would exhibit a speedup of around s, if s tasks were created to call *witness* and a different worker executed each one. When implemented using Lime, I expect that the running time would be sublinear, as in, there would not be a speedup of s for a parallelization with s workers. This is because of the fact that there is overhead in splitting the tasks and creating the workers. Finally due to the MapReduce parallelization scheme [MRR12], once all the calls to *witness* are executed in the Miller-Rabin test, Lime must go through the results to determine if any witnesses have been found. In the serial implementation, this is not the case: any execution of the Miller-Rabin test can halt execution with a result of compositeness for a given call to *witness*.

6.10 Conclusion

In conclusion, there are several limiting factors in Lime that make a parallel implementation of the Miller-Rabin primality test very difficult and creates sub-optimal program control flow and limited input size (of no practical use). I have outlined several implementation details that would be greatly beneficial to the users of the parallel implementation and the MapReduce parallelization scheme that is exploited in the program.

Bibliography

[Aba] Cristina Abad. An Introduction to Parallel Programming with MapReduce. `http://cnx.org/content/m20644/1.10/`.

[Alf] Peter Alfeld. The 1,000 smallest prime numbers. `http://www.math.utah.edu/~pa/math/primelist.html`.

[Aro] Jacob Aron. New 17-million-digit monster is largest known prime. `http://www.newscientist.com/article/dn23138-new-17milliondigit-monster-is-largest-known-prime.html`.

[Cal] Chris Caldwell. Random Small Primes. http://primes.utm.edu/lists/small/small3.html.

[Cor09] Thomas H. Cormen. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2009.

[Gal] Jean Gallier. Notes on Public Key Cryptography And Primality Testing Part 1: Randomized Algorithms Miller-Rabin and Solovay-Strassen Tests. http://www.cis.upenn.edu/~jean/RSA-primality-testing.pdf.

[Kar] Enis Karaarslan. Primality Testing Techniques and the Importance of Prime Numbers in Security Protocols. http://www.karaarslan.net/bildiri/PrimeTestingAndSecurity.pdf.

[Mic] Sun Microsystems. Java BigInteger source code. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/java/math/BigInteger.java#BigInteger.modPow%28java.math.BigInteger,java.math.BigInteger%29.

[Mit] Shyamal Mitra. RSA Algorithm Example. http://www.cs.utexas.edu/~mitra/honors/soln.html.

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier/Morgan Kaufmann, Amsterdam, 2012.

[Nat] NaturalNumbers.org. The first 1000 composite numbers. http://www.naturalnumbers.org/C-1000.txt.

7 Sequence Alignment Using Smith-Waterman Algorithm

Yue Sun

7.1 Introduction

Sequence alignment is a method to determine similar regions between two strings. In bioinformatics, sequence alignment is a way to identify the similarity between sequences of DNA, RNA, or protein. The differences between sequences may caused by insertions, deletions and mutations of individual elements in the sequences [1].

There are several algorithms designed to solve the sequence alignment problem, such as Smith-Waterman algorithm, Hirschberg's algorithm, and Needleman-Wunsch algorithm. Both Smith-Waterman algorithm and Needleman-Wunsch algorithms find an optimal alignment in $O(nm)$ time, using $O(nm)$ space. The main difference between them is the way they score the matrix cells. Hirschberg's algorithm is better in performance, which still takes $O(nm)$ time, but needs only $O(min\{n, m\})$ space [2]. All these three algorithms return the match part of the sequences, and the match found is the best match between the sequences.

The reason I choose Smith-Waterman algorithm is that this algorithm can achieve a great speedup on a platform based on FPGA chips. FPGA-based version of the Smith-Waterman algorithm shows FPGA (Virtex-4) speedups up to 100x over a 2.2 GHz Opteron processor [6]. The Smith-Waterman algorithm is suitable for parallelization, because the speedup can be achieved when calculating elements from a same anti-diagonal concurrently [3].

My approach to this problem is to use the classic parallel Smith-Waterman algorithm which works on anti-diagonals. Here the fork-join pattern is used when dealing with

each diagonal, and update the results from each diagonal to the matrix serially.

7.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm is one way of achieving sequence alignment between two sequences. Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure, instead of looking at the whole sequence [5].

Steps of Smith-Waterman algorithm:

Let a be sequence A, b be sequence B. Let $m =$ length of a, $n =$ length of b. D is the matrix, and $D(i, j)$ stands for the element(i, j) in the matrix.

- Build a matrix D size of $(m + 1) \times (n + 1)$.
- Initialize the matrix: $D(i, 0) = 0, 0 \leq i \leq m$; $D(0, j) = 0, 0 \leq j \leq n$.
- Update each $D(i, j)$ where $1 \leq i \leq m, 1 \leq j \leq n$ using the following rules.

Here the function: $w(a_{\mathrm{i}}, b_{\mathrm{j}})$ is used to compare characters from a, b.

$$w(a_{\mathrm{i}}, b_{\mathrm{j}}) = \begin{cases} 2 & a_{\mathrm{i}} = b_{\mathrm{j}} \\ -1 & a_{\mathrm{i}} \neq b_{\mathrm{j}} \end{cases} \tag{7.1}$$

$$D(i, j) = max \begin{cases} D(i-1, j-1) + w(a_{\mathrm{i}}, b_{\mathrm{j}}) & \text{Match/Mutataion} \\ D(i-1, j) + w(a_{\mathrm{i}}, -) & \text{Deletion} \\ D(i, j-1) + w(+, b_{\mathrm{j}}) & \text{Insertion} \end{cases} \tag{7.2}$$

Because the elements depend on each other, we must start the updating from $D(1, 1)$ and continue it horizontally, vertically, or diagonally. They are all feasible for serial version of the program but only when updating the matrix diagonally will achieve parallelism.

- Track back from the last element in the matrix according to the direction of movement used to construct the matrix. A diagonal jump means there is an alignment when $w = 2$ or a mutation when $w = -1$. A top-down jump implies there is a deletion. A left-right jump implies there is an insertion. This step will return two reversed sequence with the parts where deletion, insertion or mutation happens.
- Reverse the result from track back and print it.

Example of matrix updating:
Sequence a = ACACACTA, Sequence b = AGCACACA

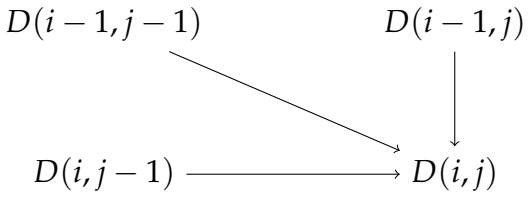
$$D = \begin{bmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{bmatrix}$$

The matrix is built according to updating part of the algorithm. The track back path is shown by the numbers in blue. A mutation will add a'()' in the result string. A deletion adds '-'. An insertion adds a '+'.

In this case, the result of track back is ATCACAC-A, A-CACACGA. The final result is A-CACACTA, AGCACAC-A.

7.3 Smith-Waterman Algorithm in Parallel

Each element $D(i,j)$ in the matrix has a dependency with $D(i-1,j)$, $D(i,j-1)$ and $D(i,j)$, to get $D(i,j)$ we need those three previous elements first.



In this situation, the parallelization doesn't simply exist between rows or columns. The later element in a row can only be calculated after the previous element, and the later row can only be calculated after the previous row; The later element in a column can only be calculated after the previous one, and the later column can only be calculated after the previous column.

Now, let's have a look at if the parallelization exist when traverse the matrix diagonally. As it is shown in the example below, we can easily find out the dependency still exist between diagonals, but there is no dependency between elements in a same diagonal when the previous two diagonal is calculated. Therefore, my solution is to reach each diagonal

serially but calculate the elements in a same diagonal concurrently. In this project, I use fork-join pattern [4] to achieve it.

$$D = \begin{bmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{bmatrix}$$

This Smith-Waterman algorithm is like Dynamic Programming. The previous elements are the subproblem of current problem. To solve the current problem, subproblem need to be solve first. All element have direct or indirect dependency on the first element in the first diagonal, which is the smallest subproblem. The last element is like the root in Dynamic Programming, to solve it all subproblems need to be solved. Problems on a same level (diagonal) can be solve concurrently. This property guarantees finding of the optimal local alignment with respect to the scoring system being used.

There is a draw back of this approach when it needs to copy part of the matrix to the parallel region when processing each diagonal. The space complexity is increased to $O(3mn)$.

7.4 Selection and Decisions of Algorithm

In this topic, only building the matrix can be done concurrently. As described in the last section, to traverse the matrix diagonally is the key to realise the parallisation.

The steps are as follow:

1. Keep track of index of elements and the value of previous elements that are directly related to each current element in a diagonal in the matrix.
2. After a whole diagonal is traversed, pass the tracked information to the Lime parallel region.
3. Split the whole information according to the number of element in the diagonal and solve them concurrently.

4. Join the results into an array and update the results to the original matrix serially according to the index from the input.
5. Do the same thing to the rest of the matrix.
6. After the whole matrix is built, do the back trace and print the final result.

Before next diagonal is reached, the current diagonal must be calculated, because the later one depends on it. The reason for keeping track of the index and neighbours' value is due to the restriction of Lime that data used in the parallel region must be local or constant. If we try to read or modify the original matrix in parallel region, the original matrix must be constant which cannot be modified. By inputing the tracked values into the parallel region, we achieve the same calculation without reaching the original matrix and by using the index of target element, we can update the matrix (successfully) in the later serial part. Although the update is still in serial, the potential speedup can still be achieved by doing the calculation concurrently.

7.5 Development and Explanation

Main.lime is the main file of the whole project. It executes each step of the Smith-Waterman algorithm.

<src/code/Main.lime 1>

```
1  import lime.util.Tasks;
2
3  public class Main {
4    public static void main(String [] args) {
5      SmithWaterman sw = new SmithWaterman();
6      long endTime;
7      long startTime = System.nanoTime();
8      sw.process();
9      endTime = System.nanoTime();
10     sw.backtrack();
11     sw.printMatrix();
12     sw.printScoreAndAlignments();
13     System.out.println("Time: " + (endTime − startTime) + "ns");
14   }
15 }
```

The sequences to be compared is final static array in the SmithWaterman class, because they have to be accessed in the parallel section, therefore they have to be constant.

The matrix is initialized in the constructor of SmithWaterman. Method process() builds the matrix concurrently according to the rules introduced in the first section. Method

backtrack() tracks from the highest score in the matrix back to the first element using the previously calculated path. The print method prints out the final matrix and the best match of the two sequences.

Here we use the difference between two system times to show how much time the parallel part takes.

SmithWaterman.lime realises the whole process of the SmithWaterman algorithm. The initialization section and backtrack section are serial and the matrix building section is parallel.

<src/code/SmithWaterman.lime 2>

```
1  import lime.util.Tasks;
2
3  public class SmithWaterman {
4    static final char[[]] seqA = {'A','C','A','G','T','A','C'};
5    static final char[[]] seqB = {'A','C','G','T','A','C','T'};
6    public static int[][] D;
7    public int mScore;
8    public String mAlignmentSeqA = "";
9    public String mAlignmentSeqB = "";
10
11   public SmithWaterman() {
12     D = new int[seqA.length + 1][seqB.length + 1];
13     for (int i = 0; i <= seqA.length; i++) {
14       D[i][0] = 0;
15     }
16     for (int j = 0; j <= seqB.length; j++) {
17       D[0][j] = 0;
18     }
19   }
20
21   <<Weight 3>>
22
23   <<Process 7>>
24
25   <<ProcessMatrix 6>>
26
27   <<update 5>>
28
29   <<Calculate 4>>
30
31   <<Backtrack 8>>
32
33     public void printMatrix() {
34     System.out.print("D =          ");
```

```
35      for (int i = 0; i < seqB.length; i++) {
36        System.out.print(String.format("%4c ", seqB[i]));
37      }
38      System.out.println();
39      for (int i = 0; i < seqA.length + 1; i++) {
40        if (i > 0) {
41          System.out.print(String.format("%4c ", seqA[i-1]));
42        } else {
43          System.out.print("      ");
44        }
45        for (int j = 0; j < seqB.length + 1; j++) {
46          System.out.print(String.format("%4d ", D[i][j]));
47        }
48        System.out.println();
49      }
50      System.out.println();
51    }
52
53      public void printScoreAndAlignments() {
54      System.out.println("Score: " + mScore);
55      System.out.println("Sequence A: " + mAlignmentSeqA);
56      System.out.println("Sequence B: " + mAlignmentSeqB);
57      System.out.println();
58    }
59  }
```

INCLUDED BLOCKS: 3 on page 137, 7 on page 139, 6 on page 138, 5 on page 138, 4 on page 138, 8 on page 140

Weight() method is used for comparing the ith element of Sequence a to jth element of Sequence b. If they are the same it returns 2, otherwise it returns -1.

<Weight 3>

```
1  static local int weight(int i, int j) {
2      return (seqA[i - 1] == seqB[j - 1] ? 2 : -1);
3  }
```

USED IN: src/code/SmithWaterman.lime on page 136

Method calculate() receives five pieces of information of the matrix. They are index i, index j, the value of left element, the value of diagonal element and the value of up el-

ement. It applies the basic rules of the Smith-Waterman algorithm on those parameters, and returns the value that needs to be assigned to $D(i, j)$.

<Calculate 4>

```
1 static local `(int,int,int) calculate (`(int,int,int,int,int) p){
2   int scoreDiag = p.3 + weight(p.0, p.1);
3   int scoreLeft = p.2 − 1;
4   int scoreUp = p.4 − 1;
5   `(int,int,int) result= `(p.0,p.1,Math.max(Math.max(Math.max(scoreDiag, scoreLeft),
        scoreUp), 0));
6   return result;
7 }
```

USED IN: src/code/SmithWaterman.lime on page 136

Method update() receives a list of parameters about the elements that need to be updated. The list of parameter is the result of the parallel region; a serial task execute the update method. The information saved in a list of tuple, the first integer in the tuple is the index i, the second is index j, and the third is the value that needs to be assigned to $D(i, j)$.

<update 5>

```
1 static void update(`(int,int,int) [[]] points){
2     for(int i=0; i<points.length; i++){
3       D[points[i].0][ points[i].1]= points[i].2;
4     }
5 }
```

USED IN: src/code/SmithWaterman.lime on page 136

Method ProcessMatrix() is the parallel region of the program. Every time the program finishes reading a diagonal, it will pass a list of parameter about each elements in the diagonal to this method. The method splits the whole data and uses the calculate method on them concurrently and gathers the results in a list and assigns that result to the matrix serially. For the moment the program has a problem with the number of tasks that can be executed concurrently. The number should be different according to the number of elements in the diagonal. However, Lime has a restriction that the number of tasks has to be defined at compile time and can not be modified. Therefore the number can not be changed according to the size of the diagonal. For the moment, I use 1 as the number to ensure the correctness of result.

<ProcessMatrix 6>

```
1  public void processMatrix('(int,int,int,int,int)[] point, int n){
2      var process = Tasks.source(point)
3        => #
4        => task split '(int,int,int,int,int)[[1]]
5          => ([task[(1) task calculate ]])
6          => task join '(int,int,int)[[1]]
7          => task update('(int,int,int)[[]]) ;
8          process.finish();
9  }
```

USED IN: src/code/SmithWaterman.lime on page 136

Method process() traverses the initialized matrix diagonal by diagonal. Every time it finished reading a diagonal, it passes the information of the diagonal to the parallel region. Each diagonal will be processed one after another serially.

<Process 7>

```
1  public void process(){
2    int i = 1, j = 1;
3    int ii = 1, jj = 1;
4    int length = 1;
5    int k = 0;
6    '(int,int,int,int,int)[] arrays = new '(int,int,int,int,int)[length];
7    while (i <= seqA.length && j <= seqB.length) {
8      arrays[k] = '(i,j,D[i][j-1],D[i-1][j-1],D[i-1][j]);
9      k++;
10     if (i == seqA.length) {
11       processMatrix(arrays,length);
12       if ((jj+1) <= seqB.length) {
13         jj += 1;
14         j = jj;
15         i = 1;
16       } else {
17         ii += 1;
18         i = ii;
19         j = seqB.length;
20         length--;
21       }
22       arrays = new '(int,int,int,int,int)[length];
23       k = 0;
24     } else if (j == 1) {
25       processMatrix(arrays,length);
26       k = 0;
```

```
27          if (( jj + 1) <= seqB.length) {
28            length++;
29            jj += 1;
30            j = jj ;
31            i = 1;
32          } else {
33            ii += 1;
34            j = seqB.length;
35            i = ii ;
36          }
37          arrays = new ‘(int,int,int,int,int)[length];
38        } else {
39              i++; j−−;
40        }
41      }
42  }
```

USED IN: src/code/SmithWaterman.lime on page 136

Method backtrack() traces back from the last updated element to the first. It uses the previously calculated path to determine if match, mutation, insertion or deletion happens: () means a mutation, '-' means a deletion and '+' means an insertion.

<Backtrack 8>

```
1       public void backtrack() {
2       int i = 1;
3       int j = 1;
4       int max = D[i][j ];
5
6       for (int k = 1; k <= seqA.length; k++) {
7         for (int l = 1; l <= seqB.length; l++) {
8           if (D[k][l] > max) {
9             i = k;
10            j = l ;
11            max = D[k][l];
12          }
13        }
14      }
15      mScore = max;
16      int k = seqA.length;
17      int l = seqB.length;
18
19      while (k > i) {
```

```
20        mAlignmentSeqB += "−";
21        mAlignmentSeqA += seqA[k − 1];
22        k−−;
23      }
24      while (l > j) {
25        mAlignmentSeqA += "+";
26        mAlignmentSeqB += seqB[l − 1];
27        l−−;
28      }
29      while (D[i][j] != 0) {
30        if (D[i][j] == D[i−1][j−1] + weight(i, j)) {
31          if (weight(i, j)==2) {
32            mAlignmentSeqA += seqA[i−1];
33            mAlignmentSeqB += seqB[j−1];
34          } else {
35            mAlignmentSeqA += ")";
36            mAlignmentSeqA += seqA[i−1];
37            mAlignmentSeqA += "(";
38            mAlignmentSeqB += ")";
39            mAlignmentSeqB += seqB[j−1];
40            mAlignmentSeqB += "(";
41          }
42          i−−;
43          j−−;
44          continue;
45        } else if (D[i][j] == D[i][j−1] − 1) {
46          mAlignmentSeqA += "+";
47          mAlignmentSeqB += seqB[j−1];
48          j−−;
49          continue;
50        } else {
51          mAlignmentSeqA += seqA[i−1];
52          mAlignmentSeqB += "−";
53          i−−;
54          continue;
55        }
56      }
57      while (i > 0) {
58        mAlignmentSeqB += "−";
59        mAlignmentSeqA += seqA[i − 1];
60        i−−;
61      }
62      while (j > 0) {
63        mAlignmentSeqA += "+";
64        mAlignmentSeqB += seqB[j − 1];
```

```
65          j−−;
66      }
67      mAlignmentSeqA = new StringBuffer(mAlignmentSeqA).reverse().toString();
68      mAlignmentSeqB = new StringBuffer(mAlignmentSeqB).reverse().toString();
69  }
```

USED IN: src/code/SmithWaterman.lime on page 136

7.6 Test and Result

To test the results of the parallel version of the program, we compared them with the results from the serial version. we did several test with different input of sequences, and the result from the testing shows the result from the parallel version is correct.

Input: SeqA = ACAGTAC, SeqB = AAGTCAC.

$$D = \begin{bmatrix} & - & A & A & G & T & C & A & C \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 2 & 1 & 0 & 0 & 2 & 1 \\ C & 0 & 1 & 1 & 1 & 0 & 2 & 1 & 4 \\ A & 0 & 2 & 3 & 2 & 1 & 1 & 4 & 3 \\ G & 0 & 1 & 2 & 5 & 4 & 3 & 3 & 3 \\ T & 0 & 0 & 1 & 4 & 7 & 6 & 5 & 4 \\ A & 0 & 2 & 2 & 3 & 6 & 6 & 8 & 7 \\ C & 0 & 1 & 1 & 2 & 5 & 8 & 7 & 10 \end{bmatrix}$$

Sequence A: ACAGT+AC
Sequence B: A-AGTCAC

Input: SeqA = GCAGTAC, SeqB = AAGTCACT.

$$D = \begin{bmatrix} & - & A & A & G & T & C & A & C & T \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ G & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ C & 0 & 0 & 0 & 1 & 1 & 3 & 2 & 2 & 1 \\ A & 0 & 2 & 2 & 1 & 0 & 2 & 5 & 4 & 3 \\ G & 0 & 1 & 1 & 4 & 3 & 2 & 4 & 4 & 3 \\ T & 0 & 0 & 0 & 3 & 6 & 5 & 4 & 3 & 6 \\ A & 0 & 2 & 2 & 2 & 5 & 5 & 7 & 6 & 5 \\ C & 0 & 1 & 1 & 1 & 4 & 7 & 6 & 9 & 8 \end{bmatrix}$$

Sequence A: +GCAGT+AC+
Sequence B: A–AGTCACT

7.7 Conclusion

In this project, I used Lime and Smith-Waterman algorithm to compute sequence alignment. To make parallelization possible for the Smith-Waterman algorithm, the program goes through the matrix one diagonal at a time. Because elements from same diagonal have no dependence on each other, I use fork-join pattern to split the whole diagonal, and calculate the elements concurrently, therefore a potential speedup can be achieved. Otherwise, in the serial version, only one element can be calculated at a time. However, because of the copying of data before the parallel region, the space complexity may become bigger.

The program still has some disadvantages now. The major problem is that due to the Lime restriction, the number of subtasks needs to be fixed, so that it is not very flexible when splitting the problem. Another possible disadvantage is there is too much copying of data in the program, which can increase the space complexity to $O(3mn)$. This is because Lime doesn't allow modification of the original matrix during the parallel region. Improving this is left as future work.

Bibliography

[1] Mount DM. *Bioinformatics: Sequence and Genome Analysis (2nd ed.)*. Cold Spring Harbor Laboratory Press: Cold Spring Harbor, NY., 2004.

[2] Hirschberg and D. S. A linear space algorithm for computing maximal common subsequences. `http://www.akira.ruc.dk/~keld/teaching/algoritmedesign_f03/Artikler/05/Hirschberg75.pdf`, 1975.

[3] Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot. Acceleration of the smith–waterman algorithm using single and multiple graphics processors. `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/Acceleration_of_the_SmithWaterman_algorithm_using_single.pdf`, 2010.

[4] Michael McCool, Arch D. Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier/Morgan Kaufmann, Amsterdam, 2012.

[5] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. `http://gel.ym.edu.tw/~chc/AB_Papers/03.pdf`, 1981.

[6] Olaf O. Storaasli. Exploring accelerating science applications with fpgas. `http://rssi.ncsa.illinois.edu/2007/proceedings/papers/rssi07_08_paper.pdf`, 2007.

8 Genetic Algorithm for Set Partitioning

Misha Virk

8.1 Introduction

The set partitioning problem is defined in terms of mathematical notation as [7]:

$$\text{Minimize } z = \sum_{j=1}^{n} c_j \, x_j$$

$$\text{Subject to } \sum_{j=1}^{n} a_{ij} \, x_j = 1, \quad i = 1 \ldots m$$

$$x_j \in \{0, 1\}, \quad j = 1 \ldots n$$

where a_{ij} is binary for all i and j, and $c_j > 0$. The goal is to determine values for the binary variables x_j that minimize the objective function z.

Set partitioning has many applications like crew scheduling, tanker routing, switching circuit design and assembly line balancing. Crew scheduling problem appears in many mass transport systems like airline, bus and railway industry [2, 6]. Set partitioning is well-known for airline crew scheduling as airline industry is growing very fast with approximate 25,000 flights are flying all over the world everyday [8]. Thus, the crew expenses are considered to be second largest for an airline (gasoline being the first), crew scheduling is really important for the aviation industry to have the potential cost savings as defined in the literature [1, 7].

The airline crew scheduling problem is [3, 9] finding a best assignment of flight crew to a group of scheduled flights or flight legs on aircraft. The duty period usually starts and ends at different airports, thus the crew cannot always return home at the end of the duty period and often must layover until the next day's duty begins. This sequence of duties and layovers is defined as crew pairing (means crew will stay together for all the duties within a pairing). In other words, we consider crew pairing, which is a sequence of flight legs beginning and ending at the same crew base in such a way that the arrival city of the flight leg in the sequence coincides with the departure city of the next flight leg. This sequence is named as trip or rotation. Each crew pairing has a certain cost associated with it. The aim is to find a subset of these crew pairing that covers all the flight legs in the schedule (list consisting of the flights that have to be flown) with minimal cost. This problem continues to be challenging as the number of pairings is really high for some airlines.

In this articulation as given in literature [1, 7], a flight leg that must aviate (a takeoff and landing) is represented by each row ($i = 1...m$). The feasible round trip or rotation for a crew (pairing) that might fly is represented by columns ($j = 1...n$). c_j is the cost associated with the allotment of a crew (pairing) to a specific flight leg and the goal is to determine crew (pairing) that covers all the flight legs in the schedule with minimal c_j.

Set partitioning consists of two binary factors [7]; A-matrix and the solution vector. The binary decision values x_j are designated to resolve the set partitioning issue. In the solution, if the column j is included or not is defined by value one or zero. The binary vector x is used to display the solution with the understanding that $x_j = 1(0)$ if bit j is 1(0) in the binary vector.

The elements of matrix a_{ij} are defined by

$$a_{ij} = \begin{cases} 1, & : \quad if\ flight\ leg\ i\ is\ on\ rotation\ j, \\ 0, & : \quad otherwise. \end{cases}$$

As set partitioning problem is an important concept and widely used, several algorithms have been introduced mainly divided into two categories [7]: *exact algorithms* (tend to have the best solution for set partitioning problem) and *heuristic or approximate algorithms* (tend to have a quick good solution).

Another class of algorithms used for set partitioning problem is genetic algorithms. Genetic algorithms are search algorithms developed by [4] based on the principles of natural selection and genetics. One of the main characteristics of a genetic algorithm is that it can reach an approximate solution for hard optimization problems, and thus it can be used for the set partitioning problem.

Genetic algorithm represents set partitioning solution in very forthright and usual manner. Each column j collaborates with a bit in a genetic algorithm string. The bit value depends on column j i.e., one for included in the solution and zero for not included in the solution. Each column is shown with a bit in a computer world for effective memory usage [7].

Exact algorithms are simple to implement and find a global optimal solution. The running time for these algorithms is really large and they also takes lots of memory. The main drawback of these algorithms is that they are only efficient for small set of problem. On the other hand, the genetic algorithm may find an approximate best solution but running time is short and also occupies less memory and are efficient for large set of data [5].

8.2 Genetic Algorithms

In general, a genetic algorithm reaches an optimal solution of the problem through the manipulation of the population of the candidate solutions where other algorithms have only one candidate solution. A genetic algorithm generally reaches the optimal solution in rational time period. As the size and complexity of the problem grows, the time period increases with it.

A genetic algorithm works on simple mechanism as given in literature [7], eliminating the bad traits (appears in the population which does not survive in the selection process) and using good traits (appears in the population which got selected) for mating and reproducing to form the next generation with better traits. This stage is also considered as "survival of the fittest". With the production of more generations, domination of good traits over population can be seen resulting in increasing quality of the solutions.

A simple genetic algorithm is shown below. $P(t)$ is population of individuals (also known as strings) at generation (also known as iteration) t. To decide when to stop a genetic algorithm, several methods are being used like stopping when a specific number of generations are being produced, when a certain or predefined quality range is being reached by the population, or when quality of population stop improving after some number of reproductions.

$t \leftarrow 0$
initailize $P(t)$
evaluate $P(t)$
foreach *generation*
    $t \leftarrow t + 1$
    select $P(t+1)$ *from* $P(t)$

recombine $P(t+1)$
evaluate $P(t+1)$
endfor

Genetic algorithm use two main operators [2]: *mutation* and *crossover* to delve the search space. Crossover acts as a primary tool. It selects two individuals randomly from the selected population to reproduce by exchanging substrings among them arbitrarily. For example, consider two strings S_1 and S_2 (parent strings) of length 12.

$$S_1 = 0\,1\,0\,1\,0\,1 \mid 0\,0\,0\,1\,1\,1$$
$$S_2 = 0\,1\,0\,0\,0\,1 \mid 1\,0\,0\,1\,0\,0$$

Let us consider the crossover point as 6 marked by '|'. A string with length *l*, *l - 1* crossover points are available. Crossover can be 1-point, 2-point, *n*-point or uniform-crossover. Two new substrings (offsprings) will produce after exchanging strings around crossover point S_1' and S_2'.

$$S_1' = 0\,1\,0\,1\,0\,1 \mid 1\,0\,0\,1\,0\,0$$
$$S_2' = 0\,1\,0\,0\,0\,1 \mid 0\,0\,0\,1\,1\,1$$

Mutation acts as a secondary tools for genetic algorithm to ensure the existence of diversity in the population. For example in string S_1 (parent string), mutation is done randomly at position 6 resulting in new string S_1' (offspring).

$$S_1 = 0\,1\,0\,1\,0\,1\,0\,0\,0\,1\,1\,1$$
$$S_1' = 0\,1\,0\,1\,0\,0\,0\,0\,0\,1\,1\,1$$

The basic mechanism used for parallel programs are to divide a large problem into small sets and then work on those sets concurrently using multiple processors.

The term parallel genetic algorithm can either refer to a particular *model* of the genetic algorithm or a means of *implementing* a sequential or parallel model of genetic algorithm. There are many successful parallel implementations of genetic algorithms: some parallelize a single population but each member only interacts within limited range i.e., the set of neighbors, and some divided the population into sub-populations and parallelize sub-populations.

Based on [2], parallel genetic algorithms (model) can be divided into three categories- a single-population master slave genetic algorithm, a single-population fine-grained genetic algorithm and a multiple-population coarse-grained genetic algorithm.

Figure 8.1: Master Slave Genetic Algorithm

A master slave genetic algorithm uses a single population and the multiprocessors are used to evaluate the fitness of the individual to parallelize the population as shown in Figure 5.1.

A fine-grained genetic algorithm also uses a single population and divides it such that a single string is assigned to each processor as shown in Figure 5.2. Selection and crossover operations are restricted to a small neighborhood, but the neighborhoods overlap which give individuals a chance to interact (the individuals with good traits can migrate to other neighborhoods).

A coarse-grained genetic algorithm uses multiple populations which are gained by dividing the single population into sub-population as shown in Figure 5.3. Each sub-population runs like a sequential genetic algorithm and in parallel with other sub-populations. From time to time the individuals with good traits migrate (fit string) to other sub-populations. In many implementations, the individual is able to migrate to its nearest sub-population only.

8.3 Selection of the Algorithm

The coarse-based parallel genetic algorithm (CPGA) is used to solve the real-world set partitioning problem. The algorithm developed by Levine [7] is also known as the island model genetic algorithm (IMGA).

The island model genetic algorithm can be executed on both sequential computers (where all sub-populations using the same processor for computation on the basis of time-sharing) and distributed-memory parallel computer (where each sub-population is having its own processor). The island model genetic algorithm is based on the single-program and multiple data (SPMD) model. This means that every processor is executing the same program but on a different sub-population. When sub-populations exchange the strings, only then synchronization is required. There is a possibility that after a

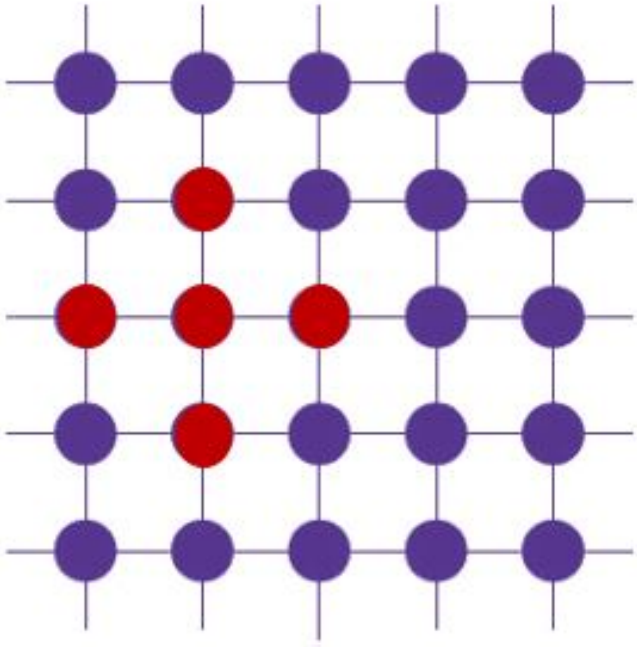


Figure 8.2: Fine-Grained Genetic Algorithm

specific number of migrations, all sub-populations hold the fittest string in the entire population. This means that the fittest string is migrating only among sub-populations, so to avoid that conflict, the arriving string is being checked if its duplicate copy already exists or not.

For the sequential genetic algorithm, the island model genetic algorithm is using a steady-state genetic algorithm with connection to row-oriented view heuristic (both a steady-state genetic algorithm and row-oriented view heuristic are designed by Levine to solve large set partitioning problems). In this sequential algorithm, at each generation, one new string is added to the population.

First, a random string x_{random} is selected and local search heuristic is applied to it. Local search heuristic is used to enhance the solution by moving to a better neighbor solution. The current solution is replaced by the neighbouring solution whenever the neighbouring solution is better. The search ends when no better optimal solution is available. The local search heuristic used here is row-oriented view heuristic.

In this heuristic [7], for a specific number of iterations one row out of total of m rows is selected (in matrix a_{ij}, rows are represented by $i = \{1 \ldots m\}$). There are three possible cases for every row $r : |r_i| = 0, |r_i| = 1, |r_i| > 1$. The result of this heuristic search varies according to these three possible cases and also vary on selection of strategy whether

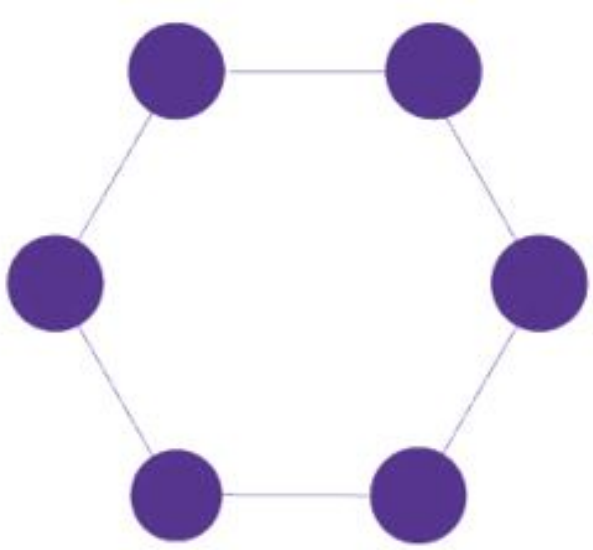


Figure 8.3: Coarse-Grained Genetic Algorithm

it is best-improving or first-improving strategy. In best-improving strategy, the entire neighborhood is evaluated and the current solution is replaced by the best solution. In first-improving strategy, the search terminates when the first better solution is found in the neighborhood and it then replaces the current solution. This algorithm applies row-oriented view heuristic to one constraint which has been chosen randomly by using first improving strategy.

foreach $number\_of\_iterations$
$i = chose\_row(random\_or\_max)$
$improving\_strategy(i, \mid r_i \mid, best\_or\_first)$
endfor

The row-oriented view heuristic and steady state genetic algorithm are found to be almost equally efficient for determining appropriate solutions for set partitioning. In various cases ROW was found to be more efficient with "work quicker, not harder approach".

Second, two parent strings x_1 and x_2 are selected and random number $r \in [0,1]$ is generated. If r is less than p_c (crossover probability, p_c = 0.6) then from two offspring, one new offspring will be created through crossover operation (uniform) and one will be picked randomly, x_{new} and will be added to the population. If r is greater than p_c then one of the parent strings is selected randomly, a new string is created by using the mutation operation and it is added to the population. In both cases, the new string will

be checked whether it's duplicate already exists in the population or not. If yes, the string will goes under further mutation to get a unique string so that it can be added to the population. The individual with worst traits is deleted and x_{new} is added to the population and population will be re-evaluated.

$t \leftarrow 0$
$initialize\ P(t)$
$evaluate\ P(t)$
foreach $generation$
  $local\_search(x_{random} \in P(t))$
  $select(x_1, x_2)\ from\ P(t)$
  if$(r < p_c)$ **then**
    $x_{new} = crossover(x_1, x_2)$
  else
    $x_{new} = mutate(x_1, x_2)$
  endif
  $delete(x_{worst} \in P(t))$
  while $(x_{new} \in P(t))$
    $mutate\ (x_{new})$
  $P(t+1) \leftarrow P(t) \cup\ x_{new}$
  $evaluate\ P(t+1)$
  $t \leftarrow t+1$
endfor

8.4 Genetic Algorithm Components

The arrangement of set partitioning matrix into block "staircase" form is a useful primary step. Building blocks match a short portion of individual and acts as a unit to influence the fitness of individuals and the algorithm works by propagating the building blocks using selection and crossover. The set of columns having their first one in row i is block B_i. In general B_i is defined for each row but it might be null for some. The arrangement of columns in B_i is in the increasing order of the c_j.

The examples of set partitioning matrix for both the cases i.e., before and after arranging in the block staircase format.The column costs, c_j and the column indices are the numbers at the top and bottom of the matrix. The column costs used here are C_1,C_2,C_3,C_4 till C_{10}.It can be any number depending upon the associated cost with respect to the column. $I = \{1, \ldots, 9\}$ and B_1 = {2,9} ,. . . , B_6 = {10,4}, and B_9 = {1} in the set partitioning matrix here.

The feasibility can be determined by arranging the matrix in this order. In every block, there is an x_j whose value can be set to one. This ordering benefits our algorithm in the following manner. First, an initialization scheme (randomly) sets an x_j which can be set to one for only one block and second, the block column structure is beneficially used by the block crossover operator.

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	1	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	1	1	0	0
1	0	0	1	1	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

SPP matrix before arranging

C_2	C_9	C_3	C_7	C_6	C_5	C_{10}	C_4	C_8	C_1
1	**1**	0	0	0	0	0	0	0	0
0	0	**1**	**1**	0	0	0	0	0	0
0	0	0	0	**1**	0	0	0	0	0
0	0	0	0	**1**	**1**	0	0	0	0
0	0	1	0	0	0	**1**	0	0	0
0	0	0	0	0	0	**1**	**1**	0	0
0	1	0	0	0	0	0	0	**1**	0
0	0	0	1	0	0	0	0	**1**	**1**
0	0	0	0	0	1	0	0	0	**1**
2	9	3	7	6	5	10	4	8	1

SPP matrix after arranging

The set partitioning objective function, the evaluation function and the fitness function are of main focus. The genetic algorithm is intended to minimize the objective function, z. However, using z directly is difficult as it does not take into account the feasibility of the string. Thus, an evaluation function is introduced for merging a cost term and a penalty

term.

The evaluation function measures "how good" a string is in terms of a solution to the set partitioning problem. Genetic algorithm operators often produces infeasible solutions. Thus, the evaluation function is needed to determine the feasibility of a string as a solution to the set partitioning problem which must include the cost of columns in the solution (the set partitioning objective function value) and the degree of (in) feasibility of the string. Thus, a genetic algorithm is not selecting a highly infeasible string for reproduction and is concentrating on feasible or near feasible solutions.

The generic form used for the evaluation function [7] is

$$f = c(x) + p(x),$$

where f = evaluation function,
$c(x)$ = cost function (is Set Partitioning objective function z),
$p(x)$ = penalty function

The penalty function used in this algorithm is countinfz penalty term which represents only the infeasibility of row i, not the magnitude of the infeasibility. The countinfz penalty term is:

$$\sum_{i=1}^{m} \lambda_i \, \phi_i \, (x)$$

where,

$$\phi_i(x) = \begin{cases} 1, & : \quad \textit{if row } i \textit{ is infeasible,} \\ 0, & : \quad \textit{otherwise.} \end{cases}$$

λ_i = scalar weight that penalizes the violation of row i

The aim of countinfz penalty term is to favor solutions which are near a feasible solution over the solutions which are highly fit but far away from any feasible solution.

For assigning to a string, the expected number of reproductive trials are determined by fitness function which is used during the selection phase. This fitness function is needed to be non-negative for the genetic algorithm, so that it is more appropriate for a string and have comparatively large fitness function value. In set partitioning, the fitness function has to be mapped from the evaluation function for this. Fitness function is defined as:

$$u(x) = g(f(x))$$

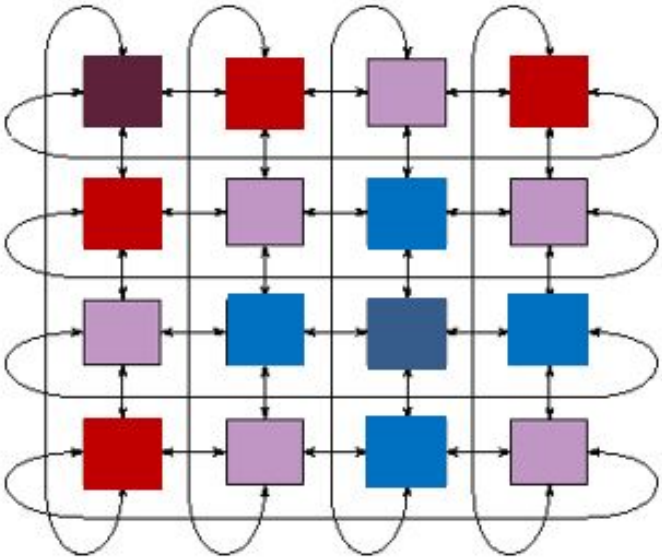


Figure 8.4: Two-dimensional Torodial Mesh

Selection is done through binary tournament in which two strings are arbitrarily selected from the community. Then a reproductive trial is assigned to a more appropriated string. Two binary tournaments are conducted and each lead to generation of one parent string. This results in a new offspring by the combination of these two parent strings.

8.5 Island Model Genetic Algorithm

The island model genetic algorithm developed by Levine [7] will provide different results depending on various characteristics. Like the choice of sequential genetic algorithm used, number of strings selected to migrate and how frequently they are migrated, the bases on which the string(s) will be selected for migration and replacement, the topology used to arrange the sub-population and which sub-population will interact while migration of string(s).

In island model genetic algorithm [7], a pre-fixed number of strings will migrate to other population. The string is migrated to sub-population for two main reasons: first, by sending a fitter string to increase the sub-population's fitness and secondly, to preserve the diversity in the sub-population.

The string is deleted based on two strategies: one, delete the worst-fit string and second, have a probabilistic binary tournament with parameter p_b = 0.4 and the winner will be deleted.

The logical topology selected is two-dimensional toroidal mesh as shown in Figure 5.4. This topology allows each processor to interact with four neighbors only for migration where neighbors are aligned in directions i.e., north, west, east, south and altering them in a same loop (north, west, east, south, north, west, and continued).

This topology is being selected for two main reasons. First, to have a migrant string that arrives from the other sub-population with above average fitness level. Secondly, to reduce the size of search space being explored for that migrant string.

Levine [7] research find that the optimal solutions are find either with very frequent migration or with no migration at all within the sub-population.

$t \leftarrow 0$
$initialize\, P(t)$
$evaluate\, P(t)$
foreach $generation$
    $local\_search(x_{random} \in P(t))$
    $select(x_1, x_2)\ from\, P(t)$
    if$(r < p_c)$ **then**
        $x_{new} = crossover(x_1, x_2)$
    else
        $x_{new} = mutate(x_1, x_2)$
    endif
    $delete(x_{worst} \in P(t))$
    while $(x_{new} \in P(t))$
        $mutate\ (x_{new})$
    $P(t+1) \leftarrow P(t) \cup x_{new}$
    if$(migrationgeneration)$ **then**
        $to = neighbor(myid, gen)$
        $x_{migrate}$ =$string\_to\_migrate(P(t+1))$
        $send\_string(to, x_{migrate})$
        $x_{recv} = recv\_string()$
        $x_{delete} = string\_to\_delete(P(t+1))$
        $replace\_string(\ x_{delete}, x_{recv}, P(t+1))$
    endif
    $evaluate\, P(t+1)$
    $t \leftarrow t+1$
endfor

8.6 Genetic Algorithm Solution for Set Partitioning

We solve a simplified version of the set partition problem using the steady state genetic algorithm [7]. Steady-State Genetic Algorithm is employed as it may be refactored for parallel processing via the island model genetic algorithm [7].

Set partitioning lends itself to a genetic algorithm solution since each set may be represented as an individual within a generation. Each element within the set may be represented as a chromosome that makes up an individual.

To outline the major module of our solution, we introduce the outline of class SPPSolver in which we encapsulate the steady-state genetic algorithm along with other supporting chunks as follows:

<code/SPPSolver.lime 1>

```
1  <<Declare Imports 25>>
2  public class SPPSolver {
3    <<Declare Instance Variables 21>>
4
5    <<SSGA Implementation 6>>
6
7    <<Other Support Methods 20>>
8
9    <<Test Main Driver 2>>
10
11   <<Console Prompt 3>>
12 }
```

INCLUDED BLOCKS: 25 on page 167, 21 on page 166, 6 on page 160, 20 on page 165, 2 on page 157, 3 on page 158

To facilitate testing the class from the console, we require a main method. Its sole responsibility is to instantiate an instance of SPPSolver and invoke its prompt() and process() method:

<Test Main Driver 2>

```
1    public static void main(String[] args) {
2      SPPSolver spp=new SPPSolver();
3      spp.prompt();
4      spp.process();
5    }
```

USED IN: code/SPPSolver.lime on page 157

The console prompt section prompts the user for input, configures the first population and the number of generations we are required to process. These responsibilities are divided up as follows:

<Console Prompt 3>

```
1  public void prompt() {
2    <<Get Initial  Population 4>>
3    <<Get Number of New Generations 5>>
4  }
```

USED IN: code/SPPSolver.lime on page 157 INCLUDED BLOCKS: 4 on page 158, 5 on page 160

Query the user for the initial generation/population i.e. generation 1. Recall each partition is encoded as an individual within our population. Each partition/individual is comprised of multiple digits/Chromosomes. Within a single partition/individual, each digit/chromosome may only be used once. A digit/chromosome may be used multiple times across partitions/individuals e.g.

{ {1}, {1,2}, {2,3}, {1,3,4}, {1,4}, {4}} represent a set of multiple partitions/individuals.

Individual 1 has chromosome 1 activated.

Individual 2 has Chromosomes 1 and 2 activated.

Individual 3 has Chromosomes 2 and 3 activated.

Individual 4 has Chromosomes 1, 3 and 4 activated.

Individual 5 has Chromosomes 1 and 4 activated.

Individual 6 has chromosome 4 activated.

As we encode the population, we keep track of the cost of each individual via an extra bucket in the chromosome array (e.g. for individuals defined by 4 Chromosomes, we maintain buckets 1 to 4 represent the presence of a chromosome and an additional fifth bucket in which we store the cost of the individual).

<Get Initial Population 4>

```
1    try {
2      String userInput = "";
3      BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
4
5      // Prompt for user input.
```

```
6         System.out.println("************************************************");
7         System.out.println("** Welcome to the SPP solver                   **");
8         System.out.println("************************************************");
9
10        System.out.println("* Please enter the number of individuals/sets in each generation: ");
11        userInput       = r.readLine();
12        maxIndividuals = Integer.parseInt(userInput);
13
14        System.out.println("* Please enter the number of Chromosomes in each individual: ");
15        userInput       = r.readLine();
16        maxChromosomes = Integer.parseInt(userInput);
17
18        currentPopulation = new int[maxIndividuals][maxChromosomes+1];
19        offSpring = new int[maxChromosomes+1];
20
21        boolean bnextdest=true;
22
23        for (int i :0:: maxIndividuals-1)
24        {
25          userInput = "";
26          System.out.println("* Please enter a comma separated partition: ");
27          userInput   = r.readLine();
28          userInput = userInput + ",";
29
30          int index=userInput.indexOf(",");
31
32          while (index>0) {
33            String sTemp = userInput.substring(0,index);
34            userInput     = userInput.substring(index+1);
35
36            int arrayIndex = Integer.parseInt(sTemp);
37            arrayIndex--;
38
39            if (arrayIndex>(maxChromosomes-1)) arrayIndex=(maxChromosomes-1);
40            if (arrayIndex<0) arrayIndex=0;
41
42            currentPopulation[i][arrayIndex]=1;
43
44            index=userInput.indexOf(",");
45
46            currentPopulation[i][maxChromosomes] = currentPopulation[i][maxChromosomes] +
                  (arrayIndex+1);
47          }
48
49        }
```

```
50
51      printCurrentPopulation();
```

USED IN: Console Prompt on page 158

Once we have established the initial population i.e. our first generation, we move on to determine how many new generations should be processed. We prompt the user for this as well:

<Get Number of New Generations 5>

```
1       System.out.println("* Please enter the number of new generations to simulate: ");
2       userInput       = r.readLine();
3       maxGenerations = Integer.parseInt(userInput);
4
5       System.out.println("generation Cnt: " + maxGenerations);
6       }
7       catch(Exception ex) {System.out.println("Prompt exception: "+ex.getMessage());}
```

USED IN: Console Prompt on page 158

The SSGA is contained within the process method:

<SSGA Implementation 6>

```
1   public void process() {
2     try {
3       <<Step 1. Evaluate cost of current population 7>>
4       <<Step 2. Loop and process each new generation 9>>
5     }
6     catch(Exception ex){System.out.println("Exception caught: "+ex.getMessage());}
7
8     return;
9   }
```

USED IN: code/SPPSolver.lime on page 157 INCLUDED BLOCKS: 7 on page 160, 9 on page 161

We now delve into each of these steps of the SSGA algorithm.

Prior to processing the new generations, we calculate the cost of the first generation/initial population.

<Step 1. Evaluate cost of current population 7>

```
1    evaluateCost();
```

USED IN: SSGA Implementation on page 160

We are using reduction pattern for calculating cost in parallel.Since we keep each individual's cost via an extra bucket in the currentPopulation array, we merely sum the last bucket for all individuals:

<Evaluate Cost 8>

```
1    currentCost=0;
2    int[] PartialSum = new int[maxIndividuals];
3    for (int j :0:: maxIndividuals−1){
4    PartialSum[j]=currentPopulation[j][maxChromosomes];
5    }
6    currentCost=(+!PartialSum);
7    System.out.println("generation cost: "+currentCost);
```

USED IN: Other Support Methods on page 165

We now crossover and mutate repeatedly to create as many generations as stipulated by the user's input.

<Step 2. Loop and process each new generation 9>

```
1    for (int genCnt:1::maxGenerations−1) {
2      <<Step 3. Initialize OffSpring Array 10>>
3      <<Step 4. Select Random Parents 11>>
4      <<Step 5. Apply heuristic 12>>
5
6      System.out.println();
7      System.out.println("Gen: " + genCnt);
8      System.out.println("Parent Index: " + rdmParent1 + "|" + rdmParent2);
9      System.out.println("Heuristic Value: " + myHeuristicVal);
10     System.out.println();
11
12     if (myHeuristicVal<crossoverProb) {
13       System.out.println("Crossing Over");
14       <<Step 6. Crossover 13>>
15     }
16     else {
17       System.out.println("Mutating");
18       <<Step 7. Mutate 14>>
```

```
19          }
20
21          printOffSpring();
22
23          <<Step 8. Mutate OffSpring if 0 cost 18>>
24          <<Step 9. Replace least fit individual 19>>
25
26          printCurrentPopulation();
27          evaluateCost();
28        }
```

USED IN: SSGA Implementation on page 160 INCLUDED BLOCKS: 10 on page 162, 11 on page 162, 12 on page 162, 13 on page 163, 14 on page 163, 18 on page 165, 19 on page 165

Prior to embarking on the creation of a new generation we perform some house-cleaning. Ensure the offSpring array in which we will store the new child is initialized.

<Step 3. Initialize OffSpring Array 10>

```
1           for (int k :0:: maxChromosomes) offSpring[k]=0;
```

USED IN: Step 2. Loop and process each new generation on page 161

We randomly select a pair of parents from the current generation. We choose two random parents by randomly generating an int between 0 and (maxIndividuals-1) inclusively.

<Step 4. Select Random Parents 11>

```
1           Random myRandomizer = new Random();
2           int rdmParent1 = myRandomizer.nextInt(maxIndividuals);
3           int rdmParent2 = myRandomizer.nextInt(maxIndividuals);
4
5           while (rdmParent1 == rdmParent2) rdmParent2 = myRandomizer.nextInt(
                maxIndividuals);
```

USED IN: Step 2. Loop and process each new generation on page 161

Once we have parents established we may crossover or mutate. This choice is based upon a heuristic value which we also randomly generate.

<Step 5. Apply heuristic 12>

```
1           int myHeuristicVal = (myRandomizer.nextInt(maxChromosomes)%2);
```

USED IN: Step 2. Loop and process each new generation on page 161

Crossover typically comes in three flavors:

- Single Point - choose a single point at which we exchange chromsomes
- Double Point - choose two points at which we exchange Chromosomes
- Uniform - we randomly choose a chromosome from either parent to be the child's

For our implementation we pursue random crossover. Parent 1 or parent 2 is randomly chosen. The selected parent's kth chromosome is then utilized for the offspring.

We keep track of the offpsring's cost in the extra/last bucket of the offSpring[] array.

<Step 6. Crossover 13>

```
1        for (int kross :0:: maxChromosomes−1) {
2          int useParent1 = (myRandomizer.nextInt(maxChromosomes)%2);
3
4          if (useParent1 == 0) offSpring[kross]=currentPopulation[rdmParent1][kross];
5          else offSpring[kross]=currentPopulation[rdmParent2][kross];
6
7          offSpring[maxChromosomes] = offSpring[maxChromosomes] + (kross+1)∗offSpring[
               kross];
8        }
```

USED IN: Step 2. Loop and process each new generation on page 161

Mutation is accomplished by randomly choosing a parent and randomly flipping one of its bits. As we build the child array we maintain its cost. As such, we are also required to adjust the cost upon mutating.

<Step 7. Mutate 14>

```
1        <<Step 7a. Randomly Select a Parent 15>>
2        <<Step 7b. Randomly Flip Bit 16>>
```

USED IN: Step 2. Loop and process each new generation on page 161 INCLUDED BLOCKS: 15 on page 164, 16 on page 164

The first step in mutation is to randomly select a parent. We do this and seed the offSpring with the parent's chromosome's.

<Step 7a. Randomly Select a Parent 15>

```
1   int mutateParent1 = (myRandomizer.nextInt(maxChromosomes)%2);
2
3   if (mutateParent1 == 0) {
4     for (int mtt :0:: maxChromosomes−1) {
5       offSpring[mtt] = currentPopulation[rdmParent1][mtt];
6       offSpring[maxChromosomes] = offSpring[maxChromosomes] + ((mtt+1)∗offSpring[
            mtt]);
7     }
8   }
9   else {
10    for (int mtt :0:: maxChromosomes−1) {
11      offSpring[mtt] = currentPopulation[rdmParent2][mtt];
12
13      offSpring[maxChromosomes] = offSpring[maxChromosomes] + ((mtt+1)∗offSpring[
            mtt]);
14    }
15  }
```

USED IN: Step 7. Mutate on page 163

The second step in mutation is then to randomly flip one of the offSpring's bits. Since we may be required to flip another bit randomly later on we encapsulate this into its own function which we invoke here.

<Step 7b. Randomly Flip Bit 16>

```
1   flipOffSpringBit();
```

USED IN: Step 7. Mutate on page 163, Step 8. Mutate OffSpring if 0 cost on page 165

The logic for flipping an offspring's bit is listed below - we randomly choose a bit and flip it. Since we are dealing with binary Chromosomes only, we add 1 and take the modulus 2. The function's signature is defined as part of the other support methods below:

<Flip OffSpring's Bit 17>

```
1   Random myRandomizer = new Random();
2   int bitToFlip = myRandomizer.nextInt(maxChromosomes);
3   offSpring[maxChromosomes]=offSpring[maxChromosomes]−offSpring[bitToFlip];
4   offSpring[bitToFlip]=((offSpring[bitToFlip]+1)%2);
5   offSpring[maxChromosomes]=offSpring[maxChromosomes]+((bitToFlip+1)∗offSpring[
        bitToFlip]);
6   return;
```

USED IN: Other Support Methods on page 165

After having crossed over and mutated, if the offspring's cost is 0 i.e. no Chromosomes are activated, we mutate until one bit is activated.

<Step 8. Mutate OffSpring if 0 cost 18>

```
1        while (offSpring[maxChromosomes]==0) {
2          System.out.println("Re−mutating − offspring cost was 0");
3          <<Step 7b. Randomly Flip Bit 16>>
4        }
```

USED IN: Step 2. Loop and process each new generation on page 161 INCLUDED BLOCKS: 16 on page 164

We then replace the least fit individual from the population with the child.

<Step 9. Replace least fit individual 19>

```
1        int leastFitIndivIdx=0;
2        for (int lfi :0:: maxIndividuals−1)
3          if (currentPopulation[lfi ][ maxChromosomes] > currentPopulation[leastFitIndivIdx][
               maxChromosomes]) leastFitIndivIdx=lfi;
4
5        for (int childChromo:0::maxChromosomes) currentPopulation[leastFitIndivIdx][
             childChromo] = offSpring[childChromo];
```

USED IN: Step 2. Loop and process each new generation on page 161

Several utility functions are utilized above to print the current generation as well as the next generation.

<Other Support Methods 20>

```
1   public void evaluateCost() {
2     <<Evaluate Cost 8>>
3   }
4
5   public void flipOffSpringBit() {
6     <<Flip OffSpring's Bit 17>>
7   }
8
9
10  public void printCurrentPopulation() {
```

```
11      System.out.println("Current Population Array:");
12      for (int i :0:: maxChromosomes) {
13        if (i==maxChromosomes) System.out.println("Costs for each individual are:");
14        for (int j :0:: maxIndividuals−1) System.out.print(currentPopulation[j][i] + "\t");
15        System.out.println();
16      }
17    }
18
19    public void printOffSpring() {
20      System.out.println("OffSpring Array");
21      for (int i :0:: maxChromosomes) {
22        if (i==maxChromosomes) System.out.println("Costs for offspring:");
23        System.out.println(offSpring[i] + "\t");
24      }
25      System.out.println();
26    }
```

USED IN: code/SPPSolver.lime on page 157 INCLUDED BLOCKS: 8 on page 161, 17 on page 164

Declare variables used to hold current/next population arrays. Instance variables may be considered one of 3 categories as shown below: We also store the number of generations which should be processed:

<Declare Instance Variables 21>

```
1   <<Constants 22>>
2   <<WorkingVariables 23>>
3   <<PopulationAndOffspring 24>>
```

USED IN: code/SPPSolver.lime on page 157 INCLUDED BLOCKS: 22 on page 166, 23 on page 167, 24 on page 167

We only have one constant defined, the crossover probability. We assume a value of 0.6:

<Constants 22>

```
1   final double crossoverProb = 0.6;
```

USED IN: Declare Instance Variables on page 166

Once we evaluate the cost of the current generation, we persist its cost in this variable so it is readily available for comparison:

<WorkingVariables 23>

```
1  int currentCost;
```

USED IN: Declare Instance Variables on page 166

The current population and its offspring as well as the number of generations we intend to process are maintained via the following instance variables:

<PopulationAndOffspring 24>

```
1  int maxGenerations;
2  int maxIndividuals;
3  int maxChromosomes;
4  int [][] currentPopulation;
5  int[] offSpring;
```

USED IN: Declare Instance Variables on page 166

Finally we declare all imports used:

<Declare Imports 25>

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.util.Random;
```

USED IN: code/SPPSolver.lime on page 157

For output, we ask the user to enter the number of individuals in each generation, the number of chromosomes in each individual, value for each individual as a set, cost for each individual and number of generations. The program will terminate after the given number of generations. For every generation, generation count and generation cost is shown. Parent index shows the two randomly selected parent strings. Heuristic value shows value of random number (0,1). If it is 0 crossover operator is used to reproduce offspring and if it is one mutation is used. The cost for the off-spring is calculated and least fit string in the population is replaced by this offspring.

<Output 26>

```
1  ***********************************************
2  ** Welcome to the SPP solver                  **
3  ***********************************************
4  * Please enter the number of individuals/sets in each generation:3
```

```
* Please enter the number of chromosomes in each individual:3
* Please enter a comma separated partition:1,2
* Please enter a comma separated partition:2,3
* Please enter a comma separated partition:1,2,3
Current Population Array:
1    0    1
1    1    1
0    1    1
Costs for each individual are:
3    5    6
* Please enter the number of new generations to simulate:3
generation Cnt: 3
generation cost: 14

Gen: 1
Parent Index: 1|2
Heuristic Value: 1    Mutating
OffSpring Array
0
0
1
Costs for offspring:3
Current Population Array:
1    0    0
1    1    0
0    1    1
Costs for each individual are:
3    5    3
generation cost: 11

Gen: 2
Parent Index: 0|1
Heuristic Value: 0  Crossing Over
OffSpring Array
1
1
1
Costs for offspring:6
Current Population Array:
1    1    0
1    1    0
0    1    1
Costs for each individual are:
3    6    3
generation cost: 12
```

```
50
51 Gen: 3
52 Parent Index: 2|1
53 Heuristic Value: 0   Crossing Over
54 OffSpring Array
55 1
56 0
57 1
58 Costs for offspring:4
59 Current Population Array:
60 1    1    0
61 1    0    0
62 0    1    1
63 Costs for each individual are:
64 3    4    3
65 generation cost: 10
```

8.7 Comparison

The parallel implementation of the steady state genetic algorithm is done in Lime language. There are two ways to achieve parallelization in this case. First, using the steady state genetic algorithm in an island model where all the islands run as steady state genetic algorithm separately and parallel to each other. Second, the parts of steady state genetic algorithm can be parallelized.

In the given implementation, the second option is used for parallelization. Like the evaluation function for calculating cost is parallelized with help of reduction pattern. In general, the algorithm is running sequentially but few parts of the algorithm are running in parallel. It seems to be the most practical way to parallelize the algorithm without using the island model.

The serial implementation of the steady state genetic algorithm is done in Java. It was simple to implement. The program was compiled 35 times to see the consistency in results. The final generation has less cost in comparison to the first one in 90% of the cases and in the rest 10%, the cost is same as the first generation (means no improvement). The results for parallel implementation are almost same as serial implementation in this scenario.

In terms of speed, if the dataset is small the serial implementation is faster for this algorithm. On the other hand, if the dataset is large then the parallel implementation is faster. The smaller dataset just adds overhead in case of parallel implementation. Like while

using join and fork, few seconds are added to join and fork operation to the runtime. If the dataset is really small these extra seconds will acts as an overhead, but if the dataset is really large then these seconds will have no effect and parallelization with reducing the runtime in general.

As the data set increases, the speed decreases. So the size of the dataset and processing speed are inversely proportional to each other. In other words, the size of the dataset and processing time are directly proportional to each other. In general, the dataset used to test this implementation is not really large but the steady state genetic algorithm is only useful for a large dataset for practical implementations [7].

8.8 Conclusion

Set partitioning is an important combinatorial optimization problem mainly used by the airline industry as a mathematical model for airline crew scheduling. The objective of crew scheduling is to find a subset of crew pairing that covers all the flight legs with minimal cost. The recent approach is using genetic algorithms to solve this problem. Genetic algorithms are search algorithms based on the principles of natural selection and genetics and can be parallel or sequential. It uses two main operators: crossover and mutation. The genetic algorithm used to solve a set partitioning problem is an island model genetic algorithm which is a coarse-grained based parallel genetic algorithm. In island model, multiple population is gained by dividing the single population into sub-population (islands). Sub-populations exchange strings occasionally using migration. Every sub-population acts as a steady state genetic algorithm (sequential algorithm) and all sub-populations runs in parallel to each other.

For set partitioning- objective function z, the evaluation function and the fitness function are in the main focus. The genetic algorithm is intended to minimize the objective function, z. Using z directly does not indicate about the feasibility of the string. Thus, an evaluation function is introduced. The evaluation function determines the feasibility of a string as a solution to the set partitioning. The fitness function is used during the selection phase. In case of set partitioning, the fitness function has to be mapped from the evaluation function. Building blocks are used to arrange set partitioning matrix into block "staircase". Genetic algorithm works by propagating the building blocks using selection and crossover. The logical topology selected by Levine is two-dimensional toroidal mesh. It is considered as best topology for parallelization for the island model genetic algorithm.

The parallel implementation of the steady state genetic algorithm is done in Lime language and the serial in Java. The algorithm runs sequentially but few parts of the algo-

rithm are running in parallel. The results for parallel implementation are almost same as serial implementation in terms of consistency. For smaller dataset serial implementation is faster and for larger dataset parallel implementation is faster. The size of the dataset and processing speed are inversely proportional to each other.

Bibliography

[1] P. c. Chu and J. e. Beasley. A Genetic Algorithm for the Set Partitioning Problem. In *Imperial College Workshops*, 1995.

[2] Erick Cantú-Paz. A Survey of Parallel Genetic Algorithms. 1998.

[3] Balaji Gopalakrishnan and Ellis. L. Johnson. Airline Crew Scheduling: State-of-the-Art. *Annals of Operations Research*, 140:305–337, 2005.

[4] J. H. Holland. Adaption in natural and artificial systems. 1975.

[5] Gaofeng Huang and Andrew Lim. *Designing A Hybrid Genetic Algorithm for the Linear Ordering Problem*. 2003.

[6] Ketan Kotecha, Gopi Sanghani, and Nilesh Gambhava. *Genetic Algorithm for Airline Crew Scheduling Problem Using Cost-Based Uniform Crossover*. 2004.

[7] Levine. A parallel genetic algorithm for the set partitioning problem. 1996.

[8] Roy E. Marsten. An Algorithm for Large Set Partitioning Problems. *Management Science*, 20:774–787, 1974.

[9] Akira Tajima and Shinji Misono. *Airline Crew-Scheduling Problem with Many Irregular Flights*. 1997.