A MASSIVELY PARALLEL LINE SIMPLIFICATION ALGORITHM
USING AN ASSOCIATIVE COMPUTING MODEL

A Thesis
By
HUY V. TRAN

Submitted in Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE

Texas A&M University – Corpus Christi
Corpus Christi, Texas

Fall 2011

Major Subject: Computer Science
A MASSIVELY PARALLEL LINE SIMPLIFICATION ALGORITHM
USING AN ASSOCIATIVE COMPUTING MODEL

A Thesis
By
HUY V. TRAN

Submitted in Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE

Computer Science Graduate Program
Department of Computing Sciences
Texas A&M University – Corpus Christi

Approved by:

__________________________________________  _______________________________________
Michael Scherger, Committee Chair           Dulal Kar, Committee Member

__________________________________________  _______________________________________
Petru-Aurelian Simionescu, Committee Member  John D. Fernandez, Department Chair

Fall 2011
ABSTRACT

Line simplification is a process of reducing the number of line segments and vertices to represent a polyline. This reduction in the number of line segments and vertices can improve the performance of spatial analysis applications. The classic Douglas-Peucker algorithm developed in 1973 has a complexity of $O(mn)$, where $n$ denotes the number of vertices and $m$ the number of line segments. Its enhanced version proposed in 1992 has a complexity of $O(n \log n)$. In this report, we present a parallel line simplification algorithm using a parallel Multiple-instruction-stream Associative Computing model (MASC). Using one instruction stream of the MASC model, our algorithm has a parallel complexity of $O(n)$ in the worst case using $n$ processing elements. The parallel algorithm is implemented using Chapel, a parallel programming language being developed by Cray Inc. The performance of the parallel program was then evaluated on different parallel computers.
ACKNOWLEDGEMENTS

I would like to thank Dr. Michael Scherger and Dr. Petru-Aurelian Simionescu for their direction, assistance and guidance to this work. Dr. Simionescu has given me an initial topic to start this thesis. Dr. Scherger’s recommendations and guidance have been invaluable for my thesis.

I also wish to thank all faculty members in the Department of Computing Sciences. I have had a wonderful and unforgettable time at Texas A&M University Corpus Christi.

I also wish to thank my fiancée, Tuyet, for her love and support that always encourages me in my work.

A special thank goes to my parents and my brother in Vietnam; I would have not finished the program without their support.
TABLE OF CONTENTS

CHAPTER                                                                 Page

ABSTRACT ................................................................................................................... iii

ACKNOWLEDGEMENTS .................................................................................................. iv

TABLE OF CONTENTS ............................................................................................... v

LIST OF TABLES ......................................................................................................... vi

LIST OF FIGURES ....................................................................................................... vii

1. INTRODUCTION ............................................................................................... 1
   1.1. A Specific Problem ...................................................................................... 3
   1.2. Contribution ................................................................................................. 4

2. LITERATURE REVIEW .................................................................................... 6
   2.1. Line Simplification Algorithms ................................................................. 6
   2.2. Douglas-Peucker Algorithm ........................................................................ 7
   2.3. Enhanced Douglas-Peucker Polyline Simplification Algorithm ................. 10
   2.4. A Naïve Sequential Algorithm ................................................................... 11
   2.5. Summary ...................................................................................................... 13

3. THE MASC MODEL OF PARALLEL COMPUTATION ................................ 15
   3.1. Introduction .................................................................................................. 15
   3.2. Cells, Instruction Streams and Other Properties ........................................ 16
   3.3. Programming Techniques with the MASC Model ...................................... 19
   3.4. Summary ...................................................................................................... 22

4. PARALLEL ALGORITHM DESIGN ................................................................. 23
   4.1. Foster’s Design Methodology .................................................................... 23
4.2. A MASC Polyline Simplification Algorithm .............................................. 25
4.3. Summary ...................................................................................................... 32
5. ALGORITHM IMPLEMENTATION AND ANALYSIS................................. 33
  5.1. Algorithm Implementation Programming Language .................................. 33
  5.2. Algorithm Implementation ....................................................................... 37
  5.3. Result Performance and Analysis .............................................................. 44
6. CONCLUSION ................................................................................................... 54
REFERENCES ........................................................................................................ 55
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Execution time (microseconds) on number of processors (SMP)</td>
<td>45</td>
</tr>
<tr>
<td>2.</td>
<td>Execution time (microseconds) vs. number of points (2 processors)</td>
<td>47</td>
</tr>
<tr>
<td>3.</td>
<td>Execution time (microseconds) vs. number of points (4 processors)</td>
<td>48</td>
</tr>
<tr>
<td>4.</td>
<td>Execution time (microseconds) vs. number of points (8 processors)</td>
<td>49</td>
</tr>
<tr>
<td>5.</td>
<td>Speed up and Efficiency SMP machines</td>
<td>51</td>
</tr>
<tr>
<td>6.</td>
<td>Number of processors and Execution time (microseconds) (clustering)</td>
<td>52</td>
</tr>
<tr>
<td>FIGURE</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>20.</td>
<td>49</td>
<td></td>
</tr>
</tbody>
</table>
21. Speed Up and Efficiency vs. Number of Processors (SMP)............................... 51
22. Execution time (microseconds) vs. Number of Processors (clustering) .......... 52
CHAPTER 1

INTRODUCTION

Two-dimensional planar level curves are the polylines where mathematical functions take on constant values. In application such as AutoCAD or MATLAB, these planar level curves are represented as collections of line segments. An example of a level curve is shown in Figure 1. The number of digitized line segments collected is far more than necessary [2]. Due to the complexity of the geospatial mathematical functions, the number of line segments to represent the planar level curves can be very large, which may cause inefficiencies in visual performance. This problem is not new. In 1973, two authors Douglas and Peucker have mentioned it in paper [2] when they worked with numerically recorded lines from maps and photographs. The big number of collected points and lines has affected the operating speed of many analysis programs and graphic devices [2]. Therefore, the polylines need to be represented with fewer segments and vertices. The elimination of unnecessary points, such as duplicates, can be of significance. It is necessary to perform a polyline simplification algorithm on two-dimensional planar level curves.

The Douglas-Peucker sequential line simplification algorithm is considered an effective line simplification algorithm [2, 19]. The algorithm uses the closeness of a vertex to a segment as a rejection condition. Its worst-case complexity is $O(mn)$, where $n$ denotes the number of vertices and $m$ the number of segment. Furthermore, in 1992 Hershberger and Snoeyink proposed an improvement for the Douglas-Peucker algorithm to gain an enhanced $O(n \log n)$ time complexity [6]. The speeding up is achieved by applying binary searches to maintain the path hulls of sub-chains. Different approaches to this line simplification issue
have also been discussed in the papers [7, 16, 18, 19 and 20]. However, even the worst-case complexities $O(mn)$ and $O(n \log n)$ are considered computationally expensive when it comes to work with significantly large visualizations. In addition, the common thing between the two algorithms mentioned above is that they already know the shape of the working data, or the data of points and segments have been already organized in a specific order.

![Figure 1: An example of a level curve](image)

If the line segments were acquired in a stream order, then the end vertex of one line segment is the beginning vertex of the next line segment in the file. It would be straightforward to apply a polyline simplification algorithm. However, in this problem, the line segments of polyline, which are obtained from the level curves of mathematical functions, are digitized in a raster scan order. The raster scan ordering of the line segments requires intensive searching on the remaining set of line segments to reconstruct the polyline, which ends up in $O(n^2)$ time complexity for searches.
1.1. A specific problem

Consider the line segments shown in Figure 2. Two connected segments, for example AB and CD, can have one of the four following arrangements:

- A is coincident or near-coincident to C.
- A is coincident or near-coincident to D.
- B is coincident or near-coincident to C.
- B is coincident or near-coincident to D.

The coincidence or collinearity of vertices is determined by investigating their coordinates (x, y). Two near-coincident vertices are considered coincident if their distance is less than or equal to an accepted tolerance $\alpha$.

Figure 2: An example of how two connected segments are not in a stream order.

In order to perform the polyline simplification, the digitized line segments in the file needs to be re-arranged. The random nature of the digitized line segments necessitates a massive number of search operations to determine coincident points. This is illustrated by a sequential algorithm which will be discussed in more details in section 2 Literature Review. The sequential algorithm is to re-arrange these line segments into a stream where line
segments having coincident vertices are moved closed to each other. For example, as shown in Figure 3, five line segments have been digitized. After the re-arrangement, the stream order of these five segments would be: B2 A2 A1 B1 B3 A3 A4 B4 A5 B5. Then, each vertex will be investigated with its next vertex for coincidence and eliminated accordingly. In this example, points A1, B1, A3, B4 will be eliminated due to coincidence, and points A4, B3 will be rejected due to collinearity.

![Segments Example](image)

**Figure 3:** An example of five segments with integer coordinates.

1.2. Contribution

The overall contribution of this work is the design of a polyline simplification algorithm using the Multiple-instruction-stream Associative Computing Model (MASC) [15, 16] to reduce the number of vertices required to represent polylines. The MASC model is an enhanced SIMD model with associative properties. The algorithm has to work with data sets as described above, where points and segments are obtained from a raster scan on three-dimensional graphs. By using the constant global operations of the MASC model, the
massively parallel algorithm theoretically has a parallel complexity of linear time $O(n)$ in the worst case.

This work also consists of an implementation and analysis of the algorithm. For the implementation, the parallel algorithm is employed using the parallel language Chapel. The Chapel program would then be tested on different available devices on campus such as a normal workstation, penguin server and thresher cluster. The analysis on performance of the algorithm would be discussed from those tests.

The remainder of the paper is structured as follows. Chapter 2 provides a review of previous research and background information fields relating to the Douglas-Peucker algorithm, its enhanced version, and a simple sequential algorithm to re-arrange segments in the data set. Chapter 3 introduces the MASC model of computation and its properties along with a touch on the language of choice Chapel. Chapter 4 contains the design and details of the massively parallel polyline simplification algorithm contributed by this work. Chapter 5 discusses on its implementation and benchmarking. Finally, Chapter 6 is the conclusion of this work and considers some topics that may be considered for future research in the area.
CHAPTER 2

LITERATURE REVIEW

This chapter is an overview of what other researchers have done to solve the polyline simplification problem. The first section is about the four categories that a polyline simplification algorithm could be classified. The second section is a summary on the Douglas-Peucker algorithm, which has an algorithmic complexity $O(mn)$ with $n$ vertices and $m$ segments and has a worst-case running time $O(n^2)$. The third section is an introduction to the enhanced Douglas-Peucker algorithm, which has a complexity of $O(n \log n)$. The fourth section is a discussion on a naïve algorithm to re-arrange the segments in a data file. Finally, the chapter would be concluded by a summary in the fifth section.

2.1. Line Simplification Algorithms

An algorithm to simplify line segments is constructing polylines from coincident and collinear vertices. Simplification algorithms can be classified into four categories: independent point routines, localized processing routines, extended local processing routines (with constraints), and extended local processing routines (without constraints) [2].

Algorithms in the first category which concern about the independent point routines are very simple. They do not take into account any mathematical relationship between points. An example for this algorithm is that for a set of $n$ points, only the even-order points are reserved while all of odd-order points are deleted. These approaches are very computationally efficient. However, they are unacceptable due to losing plots’ accuracy.
The second category exploits the characteristics of the neighboring points to decide whether or not to reject points. They use custom criteria as rejection conditions. An example for this algorithm is to determine the minimum Euclidian distance between two points. If the distance is less than a predefined minimum, one point will be rejected.

The third and the forth categories exploit the characteristics beyond the neighboring points and evaluate sections of the line. Jenks in his paper [8] used perpendicular method to produce an acceptable result. A perpendicular is calculated as a distance from a point to a segment. If the perpendicular is less than a predefined tolerance, three points (including two points of the segment) can be considered collinear and the investigated point can be deleted. The Douglas Peucker algorithm [2] is also in this category.

2.2. Douglas-Peucker Algorithm

The Douglas-Peucker algorithm relies on the distance between points and lines. It uses this Euclidian distance as a rejection condition. The algorithm preserves the shape of the polyline using a tolerance factor which may be varied according to the level of simplification the application requires.

The algorithm is generalized into the following recursive procedure. Given an array of vertices \( V \), the algorithm \( D-P (V, i, j) \) simplifies the polyline from \( V_i \) to \( V_j \).
Algorithm D-P (V, i, j)

1. $V_f$ farthest from the line $V_iV_j$.
2. Let dist be distance from $V_f$ to the segment.
3. If dist > TD then
4.   D-P ($V, i, f$) /*Split at $V_f$ and approximate*/
5.   D-P ($V, f, i$) /*recursively*/
6. Else
7.   Output $V_iV_j$ /* Use $V_iV_j$ in the approximation*/

Given a sequence of points as depicted in Figure 4, the Douglas Peucker algorithm is described as follows.

Figure 4: Douglas-Peucker algorithm – a description example.

The first point A is selected as the initial anchor point for the polyline. The last point B is chosen as the initial floating point. Among two points D and C, C has a greater perpendicular distance to AB. CF is considered the maximum perpendicular distance from the point set to the line AB. This distance CF would then be compared to a tolerance distance TD. If CF is greater than TD, then C would become the new floating point for the polyline.

After this phase, the new line constructing from the anchor point A and the new floating point C is investigated. DE is the maximum perpendicular distance to the line AC. If
DE is greater than TD, D would become the new floating point, C is saved in the first position of a stack. If DE is less than TD, D is selected as the new anchor point and all points between A and D would be discarded. In this example, DE is greater than TD. This would end the iteration for anchor point A.

The next iteration is executed in the following fashion. The last floating point now becomes the anchor point, and the first point taken from the stack becomes the new floating point. The process described in the previous two paragraphs is repeated. This recursive method keeps going until the last point is taken from the stack. Figure 5 shows an example of simplified polylines.

![Figure 5: Line simplification](image)

(a) original polylines (b) simplified polylines (c)

As first mentioned in Chapter 1, this Douglas-Peucker algorithm has a algorithmic $O(mn)$ complexity time, where $n$ is the number of input vertices and $m$ the number of segments. A mathematical analysis reported by Douglas and Peucker [2] shows that its worst case-running time is quadratic $O(n^2)$. 
2.3. Enhanced Douglas-Peucker Polyline Simplification Algorithm

The enhanced Douglas-Peucker polyline simplification algorithm was given in 1992 by Hershberger and Snoeyink [6]. It can be called the path hull algorithm. The two authors recognized that splitting vertices in a polyline must be found on its convex hull. Convex set is a set of points containing all line segments between each pair of its points. A convex hull of a set of points is the smallest convex set containing that set point of points. The path hull algorithm uses two main operations.

Build(V, i, j, PH) builds the path hull PH of point set V by choosing the middle vertex as the tag and computing the convex hulls of the two sub-polylines.

Split (PH, v, k) splits the polyline at V_k and returns a path hull of the sub-polyline consisting of the tag vertex.

The path hull algorithm is proposed as follows. Given an array of vertices V, the algorithm D-P-hull (V, i, j, PH) simplifies the polyline from V_i to V_j.

Algorithm D-P-hull (V, i, j, PH)

1. V_f farthest from the line V_iV_j.
2. Let dist be distance from V_f to the segment.
3. If dist < TD then
4. Output (V, i, j)
5. Else
6. If V_f is less than the tag vertex (distance) then
7. Split (PH, V, f)
8. D-P-hull (V, f, j, PH)
9. Build (V, i, f, PH)
The two authors claimed that the work done by each line of the algorithm can be bounded by the complexity of $O(n \log n)$, where $n$ denotes the number of vertices.

2.4. A Naïve Sequential Algorithm

If we notice from the two algorithms discussed above, they both have a priori knowledge about the polylines they work. They can pick the farthest point from a segment without going through all of the data. However, the working data on this parallel algorithm design doesn’t have such information. As being introduced in Chapter 1, the problems which this work deals with are both simplification and massive searching in the set of points for coincident points.

The simplification algorithm obtains polyline simplification by eliminating vertices whose distances to the prior initial vertex are less than a maximum accepted tolerance $\alpha$ (Figure 2). The vertices having further distance to the initial vertex (greater than $\alpha$) could be considered as part of a different polyline. However, searching for the coincident and collinear vertices is computationally expensive.

We call segArray the array of line segments. The sequential algorithm to arrange the segments back into a stream order is described as follows:
Begin

1. Set pointer current to the first segment in segArray (current=0)

2. While current does not reach the end of segArray

3. Set pointer next to the next segment of current segment (next=current+1)

4. While next does not reach the end of segArray

5. Check if the segment in next has coincident vertices with current segment (the four arrangements discussed in Chapter 1 Figure 2)

6. If yes, invert next segment if needed (in case of the second and forth arrangements in Chapter 1 Figure 2)

7. Move the next segment closed to the current segment in the array

8. Move pointer current to the next segment (current+=1)

9. Repeat step 4

10. Move pointer current to the next segment of current segment (current+=1)

11. Repeat step 2

End

Figure 6 is an example output for this algorithm on the example in Chapter 1 Figure 3 for five segments.
The sequential algorithm above is to re-arrange line segments into a stream order, which means the end of a segment in line is the beginning point of the segment next line. The mechanism of this algorithm is similar to the selection sort algorithm, where we search for a value and put it into the correct position in the list. The algorithm requires massive searching all line segments for every investigated line segment to look for the line segment having coincident vertex and move it to the right place. The ineffective searching and sorting can be noticed by the usage of two while loops in step 2 and step 4. Consequently, the complexity of this algorithm is $O(n^2)$, where $n$ denotes the number of vertices. In addition, after the line segments are re-arranged, we also need to implement a simplification algorithm to obtain the simplified polylines. This work makes the simplification task is even more expensive.

### Figure 6: Output example of segments arranging algorithm.

| B2 (1, 2) | A2 (0, 3) |
| A1 (0, 3) | B1 (-1, 3) |
| B3 (-1, 3) | A3 (-3, 3) |
| A4 (-3, 3) | B4 (-4, 3) |
| A5 (-4, 3) | B5 (-5, 2) |

2.5. **Summary**

Chapter 2 has discussed about different categories in solving line simplification problem. Among the available algorithms, Douglas-Peucker algorithm and its enhanced version are considered most visually effective line simplification algorithm [20]. However, their complexity $O(mn)$ and $O(n \log n)$ are still expensive when we have to deal with massive number of points and segments. Moreover, the problems this work has to deal with are more complex due to the expensive searching the data file. Chapter 2 also discussed a
simple segments-arranging algorithm and revealed that the most expensive operation in the algorithm is the searching operations for coincident points. The next chapter will be an introduction about the Multiple-instruction-stream Associative Computing (MASC) model to see how it uses the associative operations in searching problem.
CHAPTER 3

THE MASC MODEL OF PARALLEL COMPUTATION

3.1. Introduction

The first concept about the Multiple Associative Computing (MASC) model was introduced in [15]. In the MASC model, the major characteristic is that the model enables accessing data by associative searching rather than sequential addresses. In another word, the MASC model uses data parallelism as a basis operation for associative searching. The following figure is a conceptual description of the MASC model of parallel computation. As shown in the Figure 7, the MASC model consists of an array of processor-memory pairs called cells and an array of instructions streams.

![Figure 7: Conceptual view of MASC model.](image-url)
3.2. Cells, Instruction Streams and Other Properties

A MASC machine with n cells and j instruction streams is denoted as MASC(n, j). It is expected that the number of instruction stream processors is much less than the number of cells. Otherwise, the communication cost for those instruction streams would be very expensive. The model also includes three virtual networks:

i. A cell network used for cell-to-cell communication. This network is used for the parallel movement of data between cells. This network could be a linear array, mesh, hypercube, or a dynamic interconnection network. In addition, this network could also be a shared memory accessed by multiple processors in a Symmetric Multiprocessing (SMP) model.

ii. A broadcast/reduction network used for communication between an instruction stream and a set of cells. This network is also capable of performing common reduction or broadcasting operations.

iii. An instruction stream network used for inter-instruction stream communication.

a) Cells Properties:

Cells consist of a processing element (PE) and a local memory. The accumulated memory of the MASC model consists of an array of cells. There is no shared memory between cells. Therefore, each PE can only access to the memory of its own cell. According to [15], related data items are usually grouped together into records and typically stored one per cell. The authors assume that there are more cells than actual data.
Cells can receive their next set of instructions to execute from the instruction stream broadcast network. Cells can be instructed from their current assigned instruction stream to send and receive messages to other cells in the same partition using some communication pattern via the cell network. Each instruction stream processor is also connected to two interconnection networks. An instruction stream processor broadcasts instructions to the cells using the instruction stream broadcast network. The instruction stream also may need to communicate and may do so using the instruction stream network. Any of these networks may be virtual and be simulated by whatever network is present. The algorithm in this report is implemented using both SMP architecture and interconnected clusters.

b) Instruction Stream (IS) Properties:

The MASC model provides one or more instruction streams. Each active instruction stream is assigned to a unique dynamic partition of cells. This allows a task that is being executed in a data parallel fashion to be partitioned into two or more data parallel tasks using control parallelism. The multiple IS’s supported by the MASC model allows for greater efficiency, flexibility, and re-configurability than is possible with only one instruction stream. However, the one instruction stream version provides an straight-forward approach to a problem.

While SIMD (Single Instruction Multiple Data) architectures can execute data parallel programs very efficiently and normally can obtain linear speedup, data parallel programs in many applications are not completely data parallel and contain several non-trivial regions where significant branching occurs [5]. Moreover, the communication overhead between instruction streams and cells in the
broadcast/reduction network could also prevent the application reaching a near linear speedup. In these parallel programming regions, only a subset of traditional SIMD processors can be active at the same time. With the MASC model, control parallelism can be used to execute these different branches simultaneously.

An instruction stream can send a command to an active cell for the command to be executed at the cell. Meanwhile, an inactive cell receives a command from its instruction stream but does not execute it. Each instruction stream has an ability to unconditionally (or preemptively) activate all cells listening to it.

c) Other Properties of the MASC Model

Each instruction stream is a processor with a bus or broadcast/reduction network to all cells. Each cell listens to only one instruction stream and initially, all cells listen to the same instruction stream. The cells can switch to another instruction stream in response to commands from the current instruction stream.

Cells without further work are called idle cells and are assigned to a specified instruction stream, which among other tasks manages the idle cells.

The average time for a cell to send a message through the cell network to another cell is characterized by the parameter $t_{\text{route}}$. Each cell also can read or write a word to an I/O channel. The maximum for a cell to execute a command is given by the parameter $t_{\text{local}}$. The time to perform a broadcast of either data or instructions is given by the predictability parameter $t_{\text{broadcast}}$. The time to perform a reduction operation is given by the predictability parameter $t_{\text{reduce}}$. The time for a cell to perform this I/O transfer is characterized by the parameter $t_{\text{I/O}}$. The time to perform instruction stream synchronization is characterized by the parameter $t_{\text{sync}}$. 
An instruction stream can instruct its active cells to perform an associative search in time $t_{\text{bcast}} + t_{\text{local}} + t_{\text{reduce}}$. Successful cells are called responders, while unsuccessful cells are called non-responders.

The instruction stream can activate either the set of responders or the set of non-responders. It can also restore the previous set of active cells in $t_{\text{bcast}} + t_{\text{local}}$ time. Moreover, each instruction stream has the ability to select an arbitrary responder from the set of active cells in $t_{\text{bcast}} + t_{\text{local}}$ time.

An active instruction stream can compute the associative operations OR, AND, Greatest Lower Bound, or Least Upper Bound on a set of values in all active cells in $t_{\text{reduce}}$ time. An idle cell can be dynamically allocated to an instruction stream in $t_{\text{synch}} + t_{\text{bcast}}$ time.

These predictability parameters were identified using an object oriented description of the MASC model in [16]. They were developed to identify the performance costs using different architecture classes of parallel computing equipment. When the MASC model is implemented using a traditional SIMD computer such as STARAN or Wavetracer DTC or Zephyr, the MASC model is highly deterministic and the predictability costs can often be calculated and are often “best possible” [10]. Many of the predictability parameters for MASC operations become fixed or operate in one step [10].

3.3. Programming Techniques with the MASC Model

The difference in programming techniques between conventional parallel model and the MASC model is the dynamic memory allocation. In C-based data-parallel memory
allocation, additional fields (addresses), not cells, are allocated to the active processors. On the other hand, the MASC model uses cells’ contents to allocate the memory.

Taking advantage of this dynamic memory allocation, associative searching is the fundamental operation when working with the MASC model. Furthermore, constant-time operations, tabular representation of abstract data structures, responder processing and control parallelism are applications of associative searching that we need to concern when designing algorithms or programs on the MASC model.

Theoretically, without taking communication and latency costs into account, data parallelism is done in constant time in the MASC model. Therefore, MASC programs make extensive use of the constant time functions based on associative searching (max, min, boundaries).

“Tabular data structures” (data tables) are the main data structures in most MASC applications. The reason is that the concept of processing a whole column of a table (associative operation) is easy to comprehend and develop. In conventional programming practice, popular abstract data structures such as lists, queues, stacks or trees are applied using address manipulation via pointers and indexes. In an associative computing model, the real physical address relationship between data does not exist.
The responders of an associative search are cells which successfully match the associative search query. Using responder selection can help achieve constant-time memory allocation in the MASC model. A single instruction stream is assigned to all idle cells. When there is an instruction stream that requires one or more new cells, that instruction stream will ask the idle-cell managing instruction stream. Then, the cells will be chosen from the idle-cell pool and allocated to the requesting instruction stream.

Control parallelism can help the MASC model efficiently use all its data cells. Its mechanism depends on splitting responders into mutually exclusively subsets. One simple example is that the conditional expression if consists of two mutually exclusive partitions: one containing cells that return true and the other containing cells that return false.

According to the authors of the MASC model, with an associative computer with a constant number $k$ of instruction streams, we can possibly gain a significant speedup of up to
k in runtime. In addition, if the number of instruction streams is not restricted to be constant, we can feasibly design new algorithms with lower complexities using the MASC model.

3.4. Summary

The MASC model has formed new programming techniques for parallel computing by taking advantage of its associative ability. It makes use of today’s inexpensive computing power to enable parallel programming. The MASC model uses tabular data structures, massively parallel searching and simple syntax. The MASC model is suitable for all levels of computing, from PCs/workstations to multiple instruction streams SIMDs or heterogeneous networks.
CHAPTER 4

PARALLEL ALGORITHM DESIGN

Chapter 4 is about the design of parallel polyline simplification algorithm. The parallel algorithm uses Foster’s design methodology, which will be introduced in the chapter’s first section, as a foundation to build the parallel algorithm. In addition, in order to cope with the expensive searching process of the problem, the parallel simplification algorithm takes advantage of associative look-up operation of the MASC model. The details of the algorithm will be discussed in the second section along with explanations and discussions. The summary at the end will conclude the chapter.

4.1. Foster’s Design Methodology

Ian Foster proposed a four-step process for designing parallel algorithms [4]. The four design steps are partitioning, communication, agglomeration, and mapping (Figure 9).

Figure 9: Foster’s parallel algorithm design methodology.
Partitioning is to conceptually split computation and data into pieces. There are two kinds of partitioning: domain decomposition and functional decomposition. Domain decomposition relates to data splitting. Data is divided into smaller pieces in order to associate computations with the data. In other words, we try to break up the data structure into smaller data structure. For example, polylines can be broken into segments, and segments can be broken into vertices or points. Functional decomposition relates to computation splitting. Computation tasks are divided into smaller pieces in order to associate data with the computational tasks. In other words, it is essential to recognize the necessary functions in a computation and what relationships between those functions are. In addition, it is also critical to determine how data runs through those functions. From those perspectives, we can decide whether there are computations that can be implemented in parallel.

Communication is to determine what kind of data structure and values are passed among the tasks. There are two kinds of communication: local communication and global communication. Local communication is applied when a task needs values from a small number of other tasks in order to perform its own computation. In contrast, global communication happens when a significant number of primitive tasks must contribute data in order to perform a computation. An example of global communication is reduction operations where all of the values from processes are collected to perform a reduction (sum, max, min, or, and). Furthermore, communication is part of the overhead. Minimizing communication is a vital goal to improve parallel algorithm performance.

Agglomeration is the process of grouping small tasks into larger tasks in order to improve performance or simplify programming. The number of combined tasks can be expected to be greater than the number of processors. Agglomeration benefits the parallel
algorithm design many aspects. Agglomeration improves the performance by lowering communication overhead. It also enhances the scalability. Moreover, agglomeration helps reduce software engineering by taking advantage of the existing sequential code.

Mapping is the process of assigning tasks to available processors. Mapping could be done by the Operating System (centralized multiprocessor), or manually done by users (distributed memory system). The goals of mapping are to maximize processor utilization and minimize inter-processor communication. Processor utilization is the average percentage of time when the processors are actively executing tasks necessary for the solution of the problem. Processor utilization is optimum when the computations are evenly balanced between processors. The balanced load allows all processors to begin and end execution at the same time. On the contrary, the processor utilization will dramatically drop when one or more processors are idle while the remaining processors are still busy. However, finding an optimal mapping of tasks is an NP-hard problem.

4.2. A MASC Polyline Simplification Algorithm

Realizing the inefficiency of searching and sorting when coping with the problem as discussed in Chapter 1 and Chapter 2, we adopt the Foster’s design methodology and global constant time operations of the MASC model to avoid such inefficiency.

4.2.1. Applying Foster’s design methodology

The working data set of the simplification algorithm is a collection of line segments consisting of two points. An example of this working data file has been introduced in Figure 1. These geometric structures points and segments give a clear perspective on how to partition or decompose the problem. We use domain decomposition on this working data set.
Each point in the collection is associated with one primitive task (processing element). The task on each point is to communicate with other tasks to find the coincident and collinear vertices on the polyline. Therefore, it is expected that number of processing elements involved in the parallel algorithm could be very large.

The polyline simplification problem requires each point to search for its coincident/collinear counterparts (if any). In the partitioning step, each task (processing element) is associated with one point. Therefore, the communication for the parallel polyline simplification algorithm is global communication. The behavior of this communication is a reduction operation. All processing elements will participate in the global communication and will return the coincident point to the processing element that calls the reduction. We can expect to have a vast amount of communication in this algorithm. Without an appropriate architecture, this communication overhead can be very expensive.

The polyline simplification algorithm should be highly agglomerated. In partitioning step, we assume that each point would be associated with a task (processing element). With a big number of points, it could be impossible to implement in reality. Therefore, in fact each task will have a subset of points from the original collection. The coincident points and collinear points will then be searched in the local memory before the point-holding task communicates with other tasks. This way the algorithm can reduce the amount of communication to do the work. However, if we have too little processing elements, the performance of the parallel algorithm could be affected. An optimum number of points for each processing element could be an interesting issue to be investigated.
4.2.2. The MASC Algorithm and Description

The MASC line simplification algorithm is described as follows. All of the variables and the tabular data structures used in the algorithm will be explained in more details after.

MASC_LINE_SIMPLIFICATION Algorithm

Begin

1. Set all PEs to active
2. Set del$ = ‘No’, visited$ = ‘No’
3. Set left$/right$ to the other vertex of the segment
4. For all PEs, repeat until no visited$ = ‘No’
5. Find the coincident vertex
6. If there is no responder (no coincident vertex)
7. Set visited$ = ‘Yes’
8. Get the vertex from the responder (if any)
9. Set empty left$/right$ of the two coincident vertices to its coincident vertex
10. Check if left$/right$ of the two coincident vertices and themselves are collinear
11. If not:
12. Set the current PE’s del$ = ‘Yes’, del$ = ‘No’
13. Update field having the deleted vertex as neighbor to its coincident vertex (responders)
14. Set visited$ of both vertices = ‘Yes’
15. Clear coin$ of both vertices
16. Else if they are collinear:
17. Set both vertices’ del$ to ‘Yes’
Set the current PE’s visited$ = ‘Yes’

Update fields that have the deleted vertices (responders) as neighbor

If the deleted vertex is in left$, update to left$ of the deleted vertices

Else if the deleted vertex is in right$, update right$ of the deleted vertices

Clear coin$ of both vertices

Back to step 4

End

Consider the simple example with five segments having integer-coordinate vertices as shown in Figure 3 in Chapter 1. In the example, coincident vertices have the same value of coordinates, and three vertices are called collinear if the triangle composed by them has an area value of zero (or two lines each of which is constructed from two different points in the three points have the same slope). This can be adjusted in the functions to check coincidence and collinearity by adding an accepted tolerance $\alpha$. Coincidence and collinearity will be discussed more in Chapter 5.

The input data for the algorithm are described as Figure 3 as well. Every line of the input file is a line segment consisting of two vertices. Each vertex has an x-coordinate and a y-coordinate. Therefore, at the first state of the algorithm, we can determine the vertex’s left and right neighbor, which is the other point in the same segment.

We use a tabular organization similar to the one illustrated in Figure 10 as the data structure in the parallel algorithm. That is, the information about left and right neighbors
(left$ and right$) of the currently investigated vertex and its coincident vertex (coin$ - if any) are stored in each processing element (PE).

<table>
<thead>
<tr>
<th>PE</th>
<th>vertex</th>
<th>left$</th>
<th>right$</th>
<th>coin$</th>
<th>visited$</th>
<th>del$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE A1</td>
<td>B1</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B1</td>
<td>A1</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A2</td>
<td>B2</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B2</td>
<td>A2</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A3</td>
<td>B3</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B3</td>
<td>A3</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A4</td>
<td>B4</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B4</td>
<td>A4</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A5</td>
<td>B5</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B5</td>
<td>A5</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: The initial table of the parallel algorithm.

Vertex A is called on the left of vertex B if A’s x-coordinate is less than B’s or if A’s y-coordinate is less than B’s when A and B have the same x value. Vertex A is called on the right of vertex B if A’s x-coordinate is greater than B’s or if A’s y-coordinate is less than B’s when A and B have the same x value.

In addition to those location variables, two more variables are defined: visited$ for tracking if the vertex has been visited by the algorithm and delete$ for showing whether the vertex would be eliminated or reserved. Furthermore, every vertex is theoretically assigned to one processing element (PE) in the MASC mode, which results in a massive number of processing elements.

Using the MASC model, our algorithm does not have to re-arrange the line segments because it takes advantage of the associative searching in the model. The operations “Find its coincident vertex” in step 5 and “Find vertices that have it as neighbor” in step 13 and 19 return values in constant time. After the program finishes (all variables visited$ in all
processing elements are ‘Yes’), there would be vertices whose del$ is ‘No’. Those remaining vertices belong to the simplified polylines of the level curve’s visual representation. The directions of the remaining vertices are maintained with their left$ and right$ neighbors.

### 4.2.3. Example

We use the example mentioned in Chapter 1 to illustrate this parallel polyline simplification algorithm. The following figure is repeated after Figure 3.

![Figure 3 (repeated): An example of five segments with integer coordinates.](image)

Figure 10 introduced in section 4.2.2 illustrate the initial table representing the original digitized vertices of the example. In each iteration a vertex is used in an associative search for its coincident vertex. Then, it checks their neighbors if they are collinear points. Appropriate actions are executed to guarantee that after every round of iteration, there is no deleted vertex in the table, and all vertices would be visited after the algorithm finishes. Figure 11 demonstrates the tabular data structure after one iteration. The associative searching capabilities of the MASC model helps each round of iteration take constant time.
<table>
<thead>
<tr>
<th>vertex</th>
<th>left$</th>
<th>right$</th>
<th>coin$</th>
<th>visited$</th>
<th>del$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE A1</td>
<td>B1</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE B1</td>
<td></td>
<td>A2</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A2</td>
<td>A2</td>
<td>B2</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B2</td>
<td>A2</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A3</td>
<td></td>
<td>B3</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B3</td>
<td>A3</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A4</td>
<td></td>
<td>B4</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B4</td>
<td></td>
<td>A4</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A5</td>
<td></td>
<td>B5</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B5</td>
<td></td>
<td>A5</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11: The table after one iteration of the parallel algorithm.

Figure 12 shows the final state of the table after all vertices are visited. We can notice that all the points in the left$ and right$ columns are the remaining points. From these left and right points, we can reconstruct the polylines with the remaining points.

<table>
<thead>
<tr>
<th>vertex</th>
<th>left$</th>
<th>right$</th>
<th>coin$</th>
<th>visited$</th>
<th>del$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE A1</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE B1</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE A2</td>
<td>A2</td>
<td>B2</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B2</td>
<td>A2</td>
<td></td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A3</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE B3</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE A4</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE B4</td>
<td>A5</td>
<td>A2</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PE A5</td>
<td>B5</td>
<td>A2</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B5</td>
<td></td>
<td>A5</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: The final table of the parallel algorithm.

Figure 13 is the result polyline constructed by fewer segments and vertices. The algorithmic complexity of the algorithm is $O(n)$ in the worst case when there is no coincidence between vertices.
4.3. Summary

The Floyd’s methodology and the MASC conceptual model have provided a tool to design the parallel algorithm. The parallel polyline simplification algorithm follows steps in the methodology to come up with how the problem should be partitioned and how the data points should mapped to different processing elements. Each point is assigned to a processing element. If we don’t have enough processing elements, the point collection can be divided into p parts where p is the number of processing elements used. Each part is located in its processing element’s memory. The MASC model offers the associative search operations to theoretically find the coincident points in constant time. The parallel polyline simplification algorithm has the algorithmic parallel complexity of $O(n)$. 

Figure 13: An example of the polyline result.
CHAPTER 5

PARALLEL ALGORITHM IMPLEMENTATION AND ANALYSIS

Chapter 5 is a discussion about how the parallel polyline simplification algorithm is implemented, and gives the performance result collected from the implementation along with the analysis. The first section in Chapter 5 is an introduction of the parallel language Chapel, which was used as the programming language for the parallel implementation. The language features that we used in the implementation will mentioned in this section. The second section is about the parallel algorithm implementation. The coincidence, collinearity, main components of the program and the output files will be investigated. The third section is the performance result when the program is tested on different machines. The chapter will then be concluded by a summary.

5.1. Algorithm Implementation Programming Language

5.1.1. Language of Choice

The most significant factor of deciding the parallel language for the parallel polyline simplification algorithm is data parallelism. Message Passing Interface (MPI) is currently the most popular tool to deploy a parallel program. However, it is very tedious when we have to work with MPI, and the tool does not provide a high-level of parallel design as long as does not support data parallelism. Therefore, MPI would not be a perfect option to implementing the parallel polyline simplification algorithm.

Chapel is a parallel programming language being developed by Cray Inc. since 2007. Even though the language is still an incomplete work, it introduces very potential ability in the high performance computing industry. Chapel is the language of choice because of its
language features in data parallelism, task parallelism, concurrency and nested parallelism via high-level abstractions. This section is a touch to Chapel to introduce the tool this work used to implement the parallel polyline simplification algorithm. The Chapel version this work used is Chapel 1.3.0, the new version Chapel 1.4.0 was released on November 2011.

5.1.2. **Base Language Concepts in Chapel**

a) **Basic types and variables**

Chapel supports a number of built-in types such as floating point and integer types, complex numbers, Boolean values, and strings. Variable declaration in Chapel is different from C or FORTRAN. It follows the general form:

```
var <name> [ : <definition> ] [ = <initializer> ] ;
```

where name is the variable name, definition denotes its types or structural properties, and initializer provides the initial value of the variable. For example, the declaration of an integer variable could be

```
var an_integer_number : int = 0;
```

b) **Locales**

In Chapel, the term locale is a reference to the unit of a particular parallel architecture which has access to memory. For example, on cluster architecture, each node and its local memory could be called a locale, or on Symmetric Multi-Processing (SMP) architecture, each processor in the machine could be considered a locale. To run the program on multiple locales in the parallel architecture, users need to specify the name and address of the locales in an environment variable before compiling the Chapel program; more information could be found in the README-multi_locale coming with the Chapel language implementation.

Control Flows and Functions
Chapel supports a variety of standard control flows statements such as for loops, while loops, if conditionals, select statements, and returns. Chapel also supports function definitions that have default argument values and argument matching by name. For example, to define a function in Chapel, users can follow this general form

```chapel
proc Foo_Function (int argument1, string argument2, list argument3)
{
    Bar_Function();
}
```

c) **Parallelism in Chapel**

Chapel is designed based on a multi-threaded execution model, which means parallelism is not defined using task-based model but independent computations implemented using threads. The language provides high-level abstractions for parallelism using threads that are implemented by the compiler and the runtime system. This ability releases users from thread/process management and synchronization. Users can really “think parallel”.

### 5.1.3. Data Parallelism

a) **Domains and Arrays**

A domain is a language construct to define the size and shape of arrays. Domains support parallel iteration. Chapel has two main classes of domains: arithmetic domains and indefinite domains.

Arithmetic domains are represented using multidimensional integer coordinates. They are similar to traditional arrays. Arithmetic domains could be dynamic allocated. An example to create a simple 2D arithmetic domain and array is as follows:

```chapel
var D: domain(2) = [1..m, 1..n];
var A: [D] float; // an m x n array of floating point values
```
Indefinite domains represent a set of special type which is specified by users. Indefinite domains are mostly used to implement associative arrays. The following example creates an array of integers indexed using strings:

```chapel
var People: domain(string);
var Age: [People] int;

People += "Tran";
Age("Tran") = 26;
```

b) Parallel Iteration

Parallel iteration is specified in Chapel using forall loops. The forall loops iterate over domains and arrays. The forall loops provide a high-level of data parallelism or associative searching to users. If there are enough number of processors, all of the elements in the domains/arrays could be accessed in parallel. An example of forall loops is as follows:

```chapel
forall point in Points:
{
    NeighborOf(point) = ...;
}
```

5.1.4. Portability

According to Chapel’s creators, the language achieve the best improvement in productivity if being deployed in an idealized, productive architecture such as a global shared address space, hardware support for multithreading, a high-bandwidth, low-latency network, and latency tolerant processors. It is to say that some features are likely not to work or to perform poorly in a “less-ideal” architecture. For example, on an architecture with a lesser network or not appropriate network architecture, the impact of communication on the program’s performance would be much greater.
5.1.5. Chapel Language Summary

The section provides a brief introduction to the language’s features which will be used in the implementation of the parallel polyline simplification algorithm in this work. Chapel is a very promising language by providing high-level abstractions to users. It helps us focus more on “parallel thinking”. The remaining sections of the chapter are the discussions on the implementation of the parallel algorithm and the analysis.

5.2. Algorithm Implementation

This section will review and briefly discuss the parallel algorithm implementation in Chapel. How the tabular organization of the algorithm is represented. The definitions of coincidence and collinearity will also be evaluated. In addition, the output of the algorithm will be explained as well.

5.2.1. Program Components

![Figure 14: Main components of the parallel program.]

As first mentioned in Chapter 1, the working input for the parallel program is a collection of segments. Each segment occupies one line in the input file. Two points with x and y coordinates form a segment, which means there would be 4 values in a line: $x_A, y_A, x_B, y_B$. The parallel program will then read the file in a sequential manner, for Chapel does not support parallel I/O. The values from the file are used to build the initial tabular organization, which is the main working data structure of the program. After the table of points is created,
the parallel program will run the parallel polyline simplification algorithm on that table. The result polylines and points will be written to the output files. Figure 14 illustrates main components of the parallel program. Note that reading the input file, building the initial table and writing the output files are overhead steps of the parallel program.

5.2.2. Data Structure

Because Chapel provides a very efficient mechanism for data parallelism in domains and arrays (the forall loops), the tabular organization in the parallel algorithm is represented as an array of points in Chapel. Each point in the array is characterized as a class containing variables as described below. Conceptually each element in the array will be assigned to a processing element, and the time it takes to search for a value in the array would be constant. However, since we do not have enough processors or number of threads to do so, the compiler and runtime system of Chapel will use as many available processors as possible to watch over the values in the array.

```chapel
var pointList : [1..number_of_points] Point;
class Point
{
    var point_id: int;
    var x_coord: real(64);
    var y_coord: real(64);
    var coin_point_id: int;
    var left_point_id: int;
    var right_point_id: int;
    var visited : bool;
    var del: bool;
}
```

The point_id variable in the class Point is the unique id of the point. Each point in the 2-dimensional collection set will have two coordinates values x and y, which are denoted as two real variables x_coord and y_coord. The variable coin_point_id is to store the unique id of the coincident point of the current point. This coincident id is just for debugging the
program. It doesn’t play much role in the output of the algorithm. Because the algorithm works with two-dimensional planar level curves, there would be no intersection between the polylines. A point could only have maximum two neighbors – the left and right neighbors. The left and right definition was mentioned in Chapter 4. Two integer variables left_point_id and right_point_id are the unique ids of the point’s two neighbors. If there is no neighbor of that side, the variable would have the value of zero.

5.2.3. Coincidence and Collinearity

Coincidence and collinearity are the two vital factors in the polyline simplification algorithm. Coincidence between two points occurs when their Euclidian distance is zero. Collinearity of three points happens when the two lines constructed from three points have the same slope or the same tan value. Moreover, the simplification also focuses on coincidence threshold and collinearity threshold which can be defined by users. Two points are only called two different points if their Euclidian distance is greater than the coincidence threshold $\alpha$ as described in Figure 2 Chapter 1. The collinearity relationship of three points is determined by area of the triangle formed by those points. Three points are called collinear if the area value is less than or equal to a defined tolerance value. The area is calculated using the cross product of the three points. Figure 15 demonstrates the collinearity relationship between three points A, B, and C and how to figure out the cross product of three points.
Figure 15: Collinearity definition of three points.

The two following Chapel functions are part of the program to check coincidence and collinearity. They are returning true if the points are coincident (a)/ collinear (b).

(a) proc IsCoincidence(source_point, target_points)
{
    var distance_between_points : real (64);
    distance_between_points =
        sqrt ((target_points.x Coord - source_point.x Coord)**2 + (target_points.y Coord - source_point.y Coord)**2);
    if (distance_between_points <= COINCIDENCE_DEFINED_VALUE)
        return true;
    else
        return false;
}
(b) proc IsColinear(first_point, second_point, third_point)
{
    var checking_area: real(64);
    var x_difference, y_difference: real(64);

    y_difference = second_point.y_coord – first_point.y_coord;
    x_difference = third_point.x_coord – first_point.x_coord;

    checking_area = 0.5 * abs(y_difference) * abs(x_difference);

    if (checking_area <= COLINEAR_DEFINED_VALUE)
        return true;
    else
        return false;
}

5.2.4. Data Parallelism

Taking advantage of Chapel’s data parallelism, the parallel program can really follow the algorithm’s design. As mentioned in Chapter 1 and 4, the most expensive operation in the polyline simplification problem is the search operation. The MASC model (Chapter 3) conceptually offers a tool to do associative search with n processing elements assigned to all points. Chapel parallel language really implements the associative operations by using the forall loops. Hence, the coincident point of the investigated point can be accessed in an efficient amount of time with enough number of locales (processors) in a preferable parallel architecture. The following Chapel function is one of the examples in the parallel program using forall operation. It returns the coincident point.
proc GetCoincidentPoint(source_point, pointList)
{
    var coincident_point : Point ();
    coincident_point = new Point ();

    forall target_points in pointList
    {
        if (IsCoincidence(source_point, target_points)
            && (source_point.point_id != target_points.point_id))
        {
            coincident_point = target_points;
        }
    }

    return coincident_point;
}

5.2.5. Output

The program writes the results from the algorithm into three output files: "filename_result.txt", "filename_result_points.txt", and "filename_result_segments.txt". The file "filename_result.txt" is the complete final points table consisting all of the points' statuses such as point ID, x coordinate, y coordinate, left neighbor ID, right neighbor ID, and whether the point should be rejected or not. The file "filename_result_points.txt" contains the running state of the algorithm and the table of remaining points after running the algorithm. Figure 16 is an example of the output file "filename_result_points.txt". Finally, the output file "filename_result_segments.txt" has the same format with the input file but only contains the remaining points of the polylines. Another plotting program will take this output file and visualize the polylines to the screen.
5.2.6. Run the parallel program

The algorithm is implemented using the parallel language Chapel 1.3.0. This section had briefly gone over the important parts of the parallel polyline simplification program. All details and the source code of the parallel program can be found in Appendix A of this report. The program can be compiled using normal compile command of Chapel.

```
chpl -o point_simplification point_simplification.chpl
```

To run the Chapel parallel program on one machine without using the clustering, we use the following command.

```
./point_simplification
```

If we want to run the program on different clustering machines, firstly Chapel’s CHPL_COMM variable environment must be set to gasnet. Names of the clustering machines should be set before running the parallel program. More details and information can be found in the file README-multi-locale, which is located in folder chapel/doc. To run the Chapel parallel program on the clustering machines, the following command should be used:

```
./point_simplification -nl number_of_machines
```
Then the program will prompt users to enter the name of the input file they want to use. After the parallel program finishes, result files would be saved in the same folder.

The next section in this Chapter is the result performance when we test the parallel program on different number of processors and different number of points. Discussions and analysis on the results are also provided.

5.3. Performance Result and Analysis

In order to evaluate the performance of the parallel simplification algorithm, the parallel program is tested on different parallel machines with different number of processors and different size of data. At Texas A&M University Corpus Christi, there are two parallel architectures available to run the program: symmetric multiprocessing (SMP) and clustering. The symmetric multiprocessing machines are single-node machines with multi-core processors, sharing memory, high speed bus, and fast bandwidth. Clustering architecture has different computing nodes connected by a high-speed network and having their own local memory.

The working data is gotten from the level curves in Figure 1, which was introduced in Chapter 1. There are 20 level curves in the figure with a total of 15,530 points. The parallel program will gradually take all of 20 level curves as input and will measure the time it takes to get the results. The results are compared to the same parallel program but only use one processing element.

5.3.1. Testing on Symmetric Multiprocessing machines

The first test was conducted on a Dell workstation in ST-209 with 8 processors 2.99GHz and has 16GB of RAM. The parallel program is tested using a simulation environment of Chapel. Instead of different clustering machines, the parallel program uses
different number of processors of the workstation to do the work. In order to specify Chapel
to do that, the environment CHPL\_SPAWN\_FN must be set to the value L. The following
tables and graph are the execution time collected from the test run. The unit in the number of
processors column is microseconds.

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>1 processor</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>213518</td>
<td>176071</td>
<td>142798</td>
<td>138358</td>
</tr>
<tr>
<td>1444</td>
<td>612824</td>
<td>612201</td>
<td>514728</td>
<td>408716</td>
</tr>
<tr>
<td>1804</td>
<td>896365</td>
<td>763738</td>
<td>741746</td>
<td>677056</td>
</tr>
<tr>
<td>2268</td>
<td>1791870</td>
<td>1724750</td>
<td>1316680</td>
<td>1305340</td>
</tr>
<tr>
<td>2876</td>
<td>1974220</td>
<td>1637780</td>
<td>1579230</td>
<td>1480580</td>
</tr>
<tr>
<td>3644</td>
<td>4741580</td>
<td>4034270</td>
<td>3380980</td>
<td>2918070</td>
</tr>
<tr>
<td>4528</td>
<td>6236660</td>
<td>4671930</td>
<td>4472010</td>
<td>3906710</td>
</tr>
<tr>
<td>5556</td>
<td>8368920</td>
<td>6711090</td>
<td>6205260</td>
<td>5882410</td>
</tr>
<tr>
<td>6804</td>
<td>10353200</td>
<td>9277240</td>
<td>9104840</td>
<td>7969610</td>
</tr>
<tr>
<td>8264</td>
<td>13534700</td>
<td>12966500</td>
<td>11359400</td>
<td>11119600</td>
</tr>
<tr>
<td>9792</td>
<td>23144900</td>
<td>19091200</td>
<td>18967480</td>
<td>16373500</td>
</tr>
<tr>
<td>11104</td>
<td>30098900</td>
<td>23684900</td>
<td>23434100</td>
<td>20070800</td>
</tr>
<tr>
<td>12444</td>
<td>39248900</td>
<td>28051000</td>
<td>25077600</td>
<td>23155400</td>
</tr>
<tr>
<td>13848</td>
<td>51019100</td>
<td>46195100</td>
<td>37204200</td>
<td>31175300</td>
</tr>
<tr>
<td>15530</td>
<td>70872700</td>
<td>58741100</td>
<td>55981000</td>
<td>53278000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>5 processors</th>
<th>6 processors</th>
<th>7 processors</th>
<th>8 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>146812</td>
<td>84686</td>
<td>83251</td>
<td>86911</td>
</tr>
<tr>
<td>1444</td>
<td>370910</td>
<td>299682</td>
<td>289641</td>
<td>250245</td>
</tr>
<tr>
<td>1804</td>
<td>527546</td>
<td>421749</td>
<td>337691</td>
<td>257963</td>
</tr>
<tr>
<td>2268</td>
<td>1045920</td>
<td>664435</td>
<td>633087</td>
<td>359443</td>
</tr>
<tr>
<td>2876</td>
<td>1470700</td>
<td>1177180</td>
<td>1038650</td>
<td>559711</td>
</tr>
<tr>
<td>3644</td>
<td>2405630</td>
<td>2007080</td>
<td>1492880</td>
<td>938958</td>
</tr>
<tr>
<td>4528</td>
<td>3103030</td>
<td>2895040</td>
<td>1725040</td>
<td>131860</td>
</tr>
<tr>
<td>5556</td>
<td>4828070</td>
<td>4505820</td>
<td>3211970</td>
<td>2169710</td>
</tr>
<tr>
<td>6804</td>
<td>7957100</td>
<td>5262360</td>
<td>4054280</td>
<td>2846400</td>
</tr>
<tr>
<td>8264</td>
<td>9897110</td>
<td>7841110</td>
<td>5586430</td>
<td>4692560</td>
</tr>
<tr>
<td>9792</td>
<td>13066700</td>
<td>9866520</td>
<td>7246440</td>
<td>6268340</td>
</tr>
<tr>
<td>11104</td>
<td>13332900</td>
<td>12160100</td>
<td>10833400</td>
<td>7833770</td>
</tr>
<tr>
<td>12444</td>
<td>19182000</td>
<td>17324900</td>
<td>12324200</td>
<td>9298160</td>
</tr>
<tr>
<td>13848</td>
<td>30316000</td>
<td>21774100</td>
<td>14826400</td>
<td>13066400</td>
</tr>
<tr>
<td>15530</td>
<td>47260100</td>
<td>32414000</td>
<td>26229400</td>
<td>22718600</td>
</tr>
</tbody>
</table>

Table I: Execution time (microseconds) on number of processors (SMP).
For clarity purpose, only the execution time when running the parallel polyline simplification program using 1, 4, and 8 processors are chosen to plot on the following graph.

![Execution Time (ms) vs. Number of Points](image)

**Figure 17: Execution time (microseconds) vs. Number of Points (SMP).**

After using the simulation environment to measure the parallel program’s behavior, we tested the program on different other SMP machines without the simulation environment but use all available processors of the machines to re-confirm the behavior of the parallel algorithm. The execution time can be noticed less than the one tested in the Chapel simulation environment due to the overhead from process management and synchronization of Chapel.

The first SMP testing machine is an Asus laptop. The machine has two processors running at 2.53 GHz with 4.0 GB of RAM. The following table and graph are collected when running the parallel program on this laptop.
<table>
<thead>
<tr>
<th>Number of Points</th>
<th>Sequential Execution Time</th>
<th>Two processor Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>27159</td>
<td>39583</td>
</tr>
<tr>
<td>1444</td>
<td>105431</td>
<td>85651</td>
</tr>
<tr>
<td>1804</td>
<td>152745</td>
<td>124780</td>
</tr>
<tr>
<td>2268</td>
<td>227570</td>
<td>186274</td>
</tr>
<tr>
<td>2876</td>
<td>356401</td>
<td>287244</td>
</tr>
<tr>
<td>3644</td>
<td>566568</td>
<td>459639</td>
</tr>
<tr>
<td>4528</td>
<td>865500</td>
<td>698330</td>
</tr>
<tr>
<td>5556</td>
<td>1294010</td>
<td>1043620</td>
</tr>
<tr>
<td>6804</td>
<td>1925510</td>
<td>1635750</td>
</tr>
<tr>
<td>8264</td>
<td>2820460</td>
<td>2555600</td>
</tr>
<tr>
<td>9792</td>
<td>3938800</td>
<td>3248470</td>
</tr>
<tr>
<td>11104</td>
<td>5055650</td>
<td>4228300</td>
</tr>
<tr>
<td>12444</td>
<td>6380440</td>
<td>5202810</td>
</tr>
<tr>
<td>13848</td>
<td>7875840</td>
<td>6346610</td>
</tr>
<tr>
<td>15014</td>
<td>9435630</td>
<td>7742490</td>
</tr>
<tr>
<td>15530</td>
<td>10528100</td>
<td>8254260</td>
</tr>
</tbody>
</table>

Table II: Execution time (microseconds) vs. number of points (2 processors).

Figure 18: Execution time (microseconds) vs. Number of Points (2 processors).
The second testing SMP machine is a Dell workstation in ST-209 with 4 four processors running at 2.4 GHz and of 4GB RAM. The following table and graph are collected when running the program on this workstation.

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>Sequential Execution Time</th>
<th>Four Processor Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>27177</td>
<td>37106</td>
</tr>
<tr>
<td>1444</td>
<td>97436</td>
<td>89552</td>
</tr>
<tr>
<td>1804</td>
<td>150299</td>
<td>140327</td>
</tr>
<tr>
<td>2268</td>
<td>234774</td>
<td>203175</td>
</tr>
<tr>
<td>2876</td>
<td>371600</td>
<td>280926</td>
</tr>
<tr>
<td>3644</td>
<td>660373</td>
<td>406850</td>
</tr>
<tr>
<td>4528</td>
<td>1013370</td>
<td>624735</td>
</tr>
<tr>
<td>5556</td>
<td>1359280</td>
<td>882225</td>
</tr>
<tr>
<td>6804</td>
<td>2029180</td>
<td>1287760</td>
</tr>
<tr>
<td>8264</td>
<td>2983010</td>
<td>1887990</td>
</tr>
<tr>
<td>9792</td>
<td>4175080</td>
<td>2996340</td>
</tr>
<tr>
<td>11104</td>
<td>5366120</td>
<td>3911670</td>
</tr>
<tr>
<td>12444</td>
<td>6724770</td>
<td>4668890</td>
</tr>
<tr>
<td>13848</td>
<td>8578790</td>
<td>5601690</td>
</tr>
<tr>
<td>15014</td>
<td>9745970</td>
<td>6399350</td>
</tr>
<tr>
<td>15530</td>
<td>10422300</td>
<td>6910470</td>
</tr>
</tbody>
</table>

Table III: Execution time (microseconds) vs. Number of points (4 processors).

![Execution Time (ms) vs. Number of Points](image_url)

Figure 19: Execution time (ms) vs. Number of Points (4 processors).
The third testing machine is a host station called thresher. Thresher is located in ST-208. It has 8 processors running at 2.99 GHz and has 16GB of RAM. The following table and figure are the data collected when we run the parallel program on thresher.

<table>
<thead>
<tr>
<th>Number of Points</th>
<th>Sequential Execution Time</th>
<th>Eight Processors Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>18206</td>
<td>33864</td>
</tr>
<tr>
<td>1444</td>
<td>65652</td>
<td>77737</td>
</tr>
<tr>
<td>1804</td>
<td>100488</td>
<td>101759</td>
</tr>
<tr>
<td>2268</td>
<td>159949</td>
<td>138787</td>
</tr>
<tr>
<td>2876</td>
<td>248211</td>
<td>197416</td>
</tr>
<tr>
<td>3644</td>
<td>395081</td>
<td>265661</td>
</tr>
<tr>
<td>4528</td>
<td>603055</td>
<td>360689</td>
</tr>
<tr>
<td>5556</td>
<td>902196</td>
<td>486256</td>
</tr>
<tr>
<td>6804</td>
<td>1595480</td>
<td>645363</td>
</tr>
<tr>
<td>8264</td>
<td>1977850</td>
<td>903710</td>
</tr>
<tr>
<td>9792</td>
<td>2768000</td>
<td>1184520</td>
</tr>
<tr>
<td>11104</td>
<td>3758770</td>
<td>1464760</td>
</tr>
<tr>
<td>12444</td>
<td>4449880</td>
<td>1792660</td>
</tr>
<tr>
<td>13848</td>
<td>5500960</td>
<td>2165990</td>
</tr>
<tr>
<td>15014</td>
<td>6558240</td>
<td>2467790</td>
</tr>
<tr>
<td>15530</td>
<td>6944440</td>
<td>2600670</td>
</tr>
</tbody>
</table>

Table IV: Execution time (microseconds) vs. Number of points (8 processors).

Figure 20: Execution time (ms) vs. Number of Points (8 processors).
After the execution time is measured on different machines, the speed up of the parallel program is then investigated. Speed up is the ratio between sequential execution time and parallel execution time:

\[
\text{Speed Up} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}
\]

Moreover, the efficiency of the parallel program is calculated as well. The efficiency is a measure of processor utilization. It is defined to be speed up divided by the number of processors used:

\[
\text{Efficiency} = \frac{\text{Speed Up}}{\text{Number Used Processors}}
\]

The following table and figure are the speed up and efficiency of the parallel program when being run on different number of processors with the total 15,530 number of points.

<table>
<thead>
<tr>
<th>Number of Processors Used</th>
<th>Speed Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.27</td>
<td>0.64</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>0.375</td>
</tr>
<tr>
<td>8</td>
<td>2.67</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Table IV: Speed up and Efficiency SMP machines.
Looking at the figures of the execution time when running on different number of processors in the Symmetric Multiprocessing machines, we can see how the performance of the parallel program increases. With 2 processors, there is not much difference between its execution time and the sequential execution time. With 4 processors, the execution time is slightly better than the sequential execution time. However, both 2 and 4 processors yield the behavior $O(n^2)$, which is similar to the time complexity of the sequential algorithm. The speed up with 4 processors is only 1.5.

There is a more interesting story when we run the program on the SMP machine with 8 processors. Observing the graph of the execution time with 8 processors, we can see a dramatic improvement. As the number of points increases, the behavior of execution time is getting closer to linear $O(n)$. The speedup is now 2.67 in this situation. This can prove very potential and appropriate for the MASC model discussed in Chapter 3. If we have enough processing elements for the algorithm introduced in this report, we could achieve a linear speed up comparing to the sequential execution time.
5.3.2. Testing on Clustering Architecture

The clustering machines used in this report are located in ST-208. They include 8 computing nodes, each of which has 2 processors running at 2.99GHz with 4GB of RAM. Therefore, there are 16 processors in total. These computing nodes are connected by a 1 Gbps network. The parallel program is tested from using 1 computing node to using all 8 computing nodes with the data size of 15530 points. The following table is the execution time measured in this test run.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>19269500</td>
</tr>
<tr>
<td>4</td>
<td>9795730</td>
</tr>
<tr>
<td>6</td>
<td>9715270</td>
</tr>
<tr>
<td>8</td>
<td>9785870</td>
</tr>
<tr>
<td>10</td>
<td>9788800</td>
</tr>
<tr>
<td>12</td>
<td>9691860</td>
</tr>
<tr>
<td>14</td>
<td>9698210</td>
</tr>
<tr>
<td>16</td>
<td>9779510</td>
</tr>
</tbody>
</table>

Table VI: Number of processors and Execution time (microseconds) (clustering).

![Execution Time (ms) in Clustering Architecture](image)

**Figure 23: Execution time (ms) vs. Number of Processors (clustering).**
The communication between the computing nodes on the physical network really affects the performance of the parallel algorithm. Reviewing the conceptual model of the MASC model (Figure 7) in Chapter 3, we can clearly see that the broadcast/reduction network is an important factor deciding the performance of the parallel program. Since each subset of points is located on different computing nodes, the parallel program has to go through the physical network in order to get the coincident points. This can be observed on the graph of execution time when the number of points is increasing in the clustering architecture. Running with 4 processors on two computing nodes can reduce the amount of execution nearly 50%. Nevertheless, when more computing nodes are added, the program’s execution time is not improved; it doesn’t change at all. From this observation, we can conclude that clustering architecture is not suitable for the MASC model until we can have a really fast network (closed to the speed of SMP machine’s bus).
CHAPTER 6

CONCLUSION and FUTURE WORK

This report has reviewed the importance of polyline simplification process on geometric applications such as visualizations of level curves or geographic map boundaries. The reduction in the number of points and segments can help improve the efficiency of these applications but still maintain the important geometric characteristics of the visualizations.

Douglas-Peucker’s algorithm has a time complexity of $O(mn)$, and its enhanced version has the time complexity of $O(n \log n)$. After investigating the sequential algorithms, we have developed a massively parallel algorithm on this polyline simplification problem. The developed parallel algorithm takes advantage of the associative operations of the Multiple-instruction-stream Associative Computing Model. The theoretical parallel complexity of the parallel algorithm is $O(n)$.

The MASC model’s architecture and properties were also studied in this report. A significant aspect in the MASC model is data parallelism. Chapel, a parallel language developed by Cray Inc., was chosen as the language to implement the parallel polyline simplification algorithm because of its support for data parallelism.

After the parallel polyline simplification algorithm is implementing using Chapel, the parallel program is then tested on different parallel architectures available at Texas A&M University Corpus Christ: symmetric multiprocessing and clustering. The evaluations have shown that the symmetric multiprocessing architecture is appropriate to support the parallel polyline simplification algorithm. On the other hand, the communication over the network of the clustering architecture really affected the performance of the parallel program.
One interesting study to extend this work would be the consideration of the parallel polyline simplification algorithm to be implemented using CUDA. CUDA is a parallel computing platform and programming model invented by NVIDIA. It is able to dramatically enhance the computing performance by the power of graphical processing unit (GPU). The ability to efficiently manage a massive number of threads provides a potential capability on the parallel polyline simplification algorithm when each thread in CUDA can be considered a processing element in the conceptual MASC model. The performance can be expected to be increased significantly.
REFERENCES


