A Massively Parallel Line Simplification Algorithm Implemented Using Chapel

Michael Scherger  
Department of Computer Science  
Texas Christian University  
Email: m.scherger@tcu.edu

Huy Tran  
Department of Computing Sciences  
Texas A&M University Corpus Christi  
Email: htran@islander.tamucc.edu

Abstract - Line simplification is a process of reducing the number of line segments 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. An enhanced version of this algorithm was developed in 1992 and has a complexity of $O(n \log n)$. In this paper, we present a parallel line simplification algorithm and discuss the implementation results using only one instruction stream of the parallel Multiple-instruction-stream Associative Computing model (MASC). The parallel algorithm is implemented in Chapel, a parallel programming language developed by Cray Inc., has parallel complexity of $O(n)$ on $n$ processors. The performance of the parallel program was then evaluated on different parallel computers.

Keywords: Parallel algorithms, associative computing, SIMD algorithms, line simplification, vertex elimination, level curve

1. Introduction

2D planar level curves are the polylines where mathematical functions take on constant values. An example of a level curve in AutoCAD is shown in Figure 1. The number of digitized line segments collected is far more than necessary [2]. Due to the high complexity of often-irregular geospatial functions, the number of line segments to represent the planar level curve can be very large, which may cause inefficiencies in visual performance. Therefore, the polyline needs to be represented with fewer segments and vertices. It is necessary to perform a polyline simplification algorithm on a 2D planar level curve.

In this problem, the line segments of polylines are digitized in a raster scan order (left-to-right, top-to-bottom). The raster scan ordering of the line segments requires intensive searching on the remaining set of line segments to reconstruct the 2D planar curve polylines ($O(n^2)$ searches). A much simpler problem is 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 then be straightforward to apply a polyline simplification algorithm.

The Douglas-Peucker line simplification algorithm is considered an effective line simplification algorithm [2, 13]. 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 segments. Furthermore, in 1992 Hershberger and Snoeyink introduced an improvement for Douglas-Peucker algorithm to gain an enhanced $O(n \log n)$ time complexity [4]. The speed up is achieved by using binary search to maintain the path hulls of subchains. Different approaches to this issue have also been discussed in [5, 10, and 12]. 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 a previous paper [12] we presented a polyline simplification algorithm using the Multiple-
instruction-stream Associative Computing model (MASC) [6, 8] to reduce the number of vertices required to represent polylines. MASC is an enhanced SIMD model with associative properties. By using the constant global operations of the MASC model, our algorithm has a parallel complexity of linear time \( O(n) \) in the worst case using \( n \) processing elements.

For this research we present the results of an initial implementation, benchmarking, and performance analysis for the aforementioned algorithm.

This paper is organized as follows. Section 2 will briefly discuss the MASC model of parallel computation that the algorithm is grounded upon. Section 3 will discuss the sequential and parallel polyline simplification algorithms in more detail. Section 4 will briefly discuss the implementation of the algorithm using Chapel. Section 5 will present the results of our implementation and Section 6 will provide the discussions on future work and conclusion.

2. The MASC Model of Parallel Computation

The following is a description of the Multiple Associative Computing (MASC) model of parallel computation. As shown in Figure 2, the MASC model consists of an array of processor-memory pairs called cells and an array of instruction streams.

![Conceptual view of MASC](image)

A MASC machine with \( n \) cells and \( j \) instruction streams is denoted as \( \text{MASC}(n,j) \). It is expected that the number of instruction stream processors be much less than the number of cells.

Cells can receive their next set of instructions to execute from the instruction stream broadcast network. Cells can be instructed from their current 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 streams 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.

MASC 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. While SIMD architectures can execute data parallel programs very efficiently and normally can obtain near linear speedup, data parallel programs in many applications are not completely data parallel and contain several non-trivial regions where significant branching occurs [3]. In these parallel programming regions, only a subset of traditional SIMD processors can be active at the same time. With MASC, control parallelism can be used to execute these different branches simultaneously. Other MASC properties are described in [6, 7, 8, 9, 11].

3. Polyline Simplification Algorithms

In order to perform the polyline simplification, the raster scan digitized line segments in the input stream need to be re-arranged. The random nature of the digitized line segments necessitates a massive number of search operations to determine coincident points. For example, as shown in Figure 3, five line segments have been digitized.

After rearrangement, the stream order would be: [B2 A2 A1 B1 B3 A3 A4 B4 A5 B5]. Then, each vertex will be checked 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 deleted due to collinearity. The procedure for accomplishing these results is mentioned in the next two sub-sections.
2.1. Sequential Line Simplification

A sequential algorithm to simplify line segments is constructing polylines from coincident and collinear vertices. This can be obtained by eliminating vertices whose distances to the prior initial vertex are less than a maximum accepted tolerance $\alpha$. The vertices having further distance to the initial vertex ($> \alpha$) could be considered as part of a different polyline. However, finding the coincident and collinear vertices is expensive in this problem.

**Sequential Line Simplification Algorithm**

**Begin**

1. Set pointer $current$ to the first segment in `segArray (current=0)`
2. While $current$ does not reach the end of `segArray`
   2.1. Set pointer $next$ to the next segment of $current$ segment ($next=current+1$)
   2.2. While $next$ does not reach the end of `segArray`
      a. Check if the segment in $next$ has coincident vertices with $current$ segment
      b. If yes, invert $next$ segment if needed
      c. Move the $next$ segment closed to the $current$ segment in the array
      d. Move pointer $current$ to the next segment ($current+=1$)
      e. Repeat step 2.2

2.3. Move pointer $current$ to the next segment of $current$ segment ($current+=1$)
2.4. Repeat step 2

**End**

The sequential algorithm above is to rearrange line segments into a stream order. The mechanism is similar to the selection sort. The algorithm requires 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. This ineffective searching and sorting can be noticed by the usage of two `while` loops in step 2 and 2.2. Consequently, the complexity of this algorithm is $O(n^2)$, where $n$ is the number of vertices.

### 3.2. Parallel Line Simplification Algorithm

Using the MASC model to counter the inefficiencies of the search and sorting discussed earlier, we adopt global constant time search operations of the MASC model to avoid such inefficiencies.

Consider the simple example with five segments having integer-coordinate vertices as shown in Figure 3. 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. This can be adjusted in the functions to check coincidence and collinearity by adding an accepted tolerance $\alpha$ [12].

Again, using the he input data described as in Figure 3, every line of the input file is a line segment consisting of two vertices. Each vertex has an x-coordinate and a y-coordinate. Using cross products we can determine the vertex’s left or right neighbor, which is the other point in the same segment.

We use a tabular organization similar to the one illustrated in Figure 4 as the data structure in our algorithm. That is, the information about left and right neighbors (left$\$ and right$\$) of the currently investigated vertex and its coincident vertex (coin$\$ - if any) are stored in each PE. Vertex A is called on the left of vertex B if A’s x-coordinate is less than B’s or if A’s y-coordinate is less than B’s when A and B have the same x value. Vertex A is called on the right of vertex B if A’s x-coordinate is greater than B’s or if A’s y-coordinate is greater than B’s when A and B have the same x value. In addition to those location variables, two more variables are defined: visited$\$ for tracking if the vertex has been visited and delete$\$ for showing if the vertex should be eliminated or not. Furthermore, every vertex is
assigned to one PE in the MASC model, which results in a massive number of processing elements.

**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’
   4.1. Find the coincident vertex
   4.2. If there is no responder (no coincident vertex)
      4.2.1. Set visited$ = ‘Yes’
   4.3. Get the vertex from the responder (if any)
   4.4. Set empty left$/right$ of the two coincident vertices to its coincident vertex
   4.5. Check if left$/right$ of the two coincident vertices and themselves are collinear
      4.5.1. If not:
         a) Set the current PE’s del$ = ‘Yes’, del$ = ‘No’
         b) Update field having the deleted vertex as neighbor to its coincident vertex (responders)
         c) Set visited$ of both vertices = ‘Yes’
         d) Clear coin$ of both vertices
      4.5.2. Else if they are collinear:
         a) Set both vertices’ del$ to ‘Yes’
         b) Set the current PE’s visited$ = ‘Yes’
         c) Update fields that have the deleted vertices (responders) as neighbor
            i. If the deleted vertex is in left$, update to left$ of the deleted vertices
            ii. Else if the deleted vertex is in right$, update right$ of the deleted vertices
         d) Clear coin$ of both vertices
**End**

Using the MASC model, our algorithm does not have to re-arrange the line segments because it takes advantage of associative searching. The operations “Find its coincident vertex” in step 4.1 and “Find vertices that have it as neighbor” in step 4.5.1b and 4.5.2c return values in constant time. After the program finishes (all visited$ 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 remaining vertices are maintained with their left$ and right$ neighbors.

Figure 4 illustrates the initial table representing the original digitized vertices. During 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 will be visited after the program finishes.

Figure 5 demonstrates the table after one iteration. The associative searching capabilities of the MASC model helps each round of iteration take constant time. Figure 5 shows the final state of the table after all vertices are visited. Figure 8 is the resultant polyline constructed by fewer segments and vertices. The running time of the algorithm is $O(n)$ in the worst case when there is no coincidence between vertices.

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE B1</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE A2</td>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE B2</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE A3</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE B3</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE A4</td>
<td>B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE B4</td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE A5</td>
<td>B5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE B5</td>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4: The initial table for the parallel algorithm**

<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>A1</td>
<td></td>
<td>Yes</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>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A3</td>
<td>B3</td>
<td></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>B4</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B4</td>
<td>A4</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE A5</td>
<td>B5</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>PE B5</td>
<td>A5</td>
<td></td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6: The table after one iteration.**

<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>A5</td>
<td></td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7: The table after all nodes are visited**
4. Implementation Using Chapel

While many parallel programming tools exist that could be used to implement the parallel algorithm as described in Section 3, the language Chapel was selected because it can implement many (if not all) of the features of the MASC model of parallel computation efficiently using SMP’s or clusters. Chapel is a parallel programming language that has been in development (and currently being used) by Cray Inc. since 2007. Chapel was selected as the language of choice because of its language features in data parallelism, task parallelism, concurrency and nested parallelism via high-level abstractions. The major features of Chapel used in this research are parallel domains and arrays and parallel iteration.

4.1. Parallel 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 that 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 += “Mike”;
Age(“Mike”) = 21;
```

4.2. 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. Results and Analysis

The parallel program was tested on various parallel computing machinery (SMPs and clusters) that supported Chapel. Test data from the level curves in Figure 1 was used for benchmarking this algorithm. 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.

5.1. Results: Symmetric Multiprocessors

The first test was conducted on a Dell workstation in 8 processors at 3.0 GHz and has 16 GB of RAM. The following graph (Figure 9) is the execution times collected from the test run. The unit in the number of processors column is microseconds.

![Figure 9: Execution time (microseconds) vs. number of data points (SMP).](image)
5.2. Results: Clusters

The cluster machines used in this research include 8 computing nodes, each of which has 2 processors running at 3.0 GHz with 4 GB of RAM. Therefore, there are 16 processors in total. A 1 Gbps network connects these computing nodes. 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 graph (figure 10) shows the execution times vs. the number of processor cores for a fixed (maximum) number of points.

![Figure 10: Execution time (ms) vs. number of processors (cluster).](image)

5.3. Analysis

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 2) in 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. From this observation, we can conclude that clustering architecture is not suitable for this research problem MASC model until we can have a really fast network (close to the speed of SMP machine’s bus).

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.

A parallel polyline simplification algorithm was implemented using Chapel and the program was tested on different parallel architectures. 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 adversely affected the performance of the parallel program.

One interesting study to extend this work would be to consider using hardware accelerators such as CUDA or Intel Phi. The ability to efficiently manage a massive number of threads provides a potential capability on the parallel polyline simplification algorithm when each thread in hardware accelerator can be considered a processing element in the conceptual MASC model. The performance would be increased significantly.

References


