Finding Smallest Paths in Rectilinear Polygons on a Hypercube Multiprocessor

Afonso Ferreira*
CNRS - LIP
Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07, France

Joseph G. Peters†
School of Computing Science
Simon Fraser University
Burnaby, B.C., V5A 1S6, Canada

Abstract

A *smallest path* between two points in a polygon is a rectilinear path that simultaneously minimizes distance and the number of line segments in the path. In this abstract, we show how to find a smallest path between any two points in a simple rectilinear polygon with n vertices on a hypercube multiprocessor with $max(n,p)$ processors in time $O(t + \log n(\log\log n)^2)$ where $p = n\log n$ and $t = O(\log^2 n)$ are the current best bounds for finding trapezoidal decompositions.

1 Introduction

The design and analysis of geometric algorithms for parallel architectures is a young research area that has only received serious attention during the last five or six years. Even so, there are several powerful general techniques for designing geometric algorithms for the (shared-memory) PRAM model of parallel computation (e.g., [1, 2]), which, together with the large collection of other results for PRAM's (see [7] for a survey), can be used to design quite sophisticated algorithms. In contrast, the development of geometric algorithms for distributed-memory parallel architectures is in a fairly primitive state, although a few general techniques have been developed recently [5, 6]. The concentration of researchers on the PRAM model may seem surprising at first since the vast majority of existing parallel architectures are of the distributed-memory type, but the attention is not difficult to explain. The shared memory of the PRAM effectively eliminates the data routing and collision avoidance problems that dominate computations in distributed-memory architectures and permits the development of the sophisticated data structures needed to solve many geometry problems. Furthermore, the fundamental *recursive doubling* (*pointer jumping, shortcutting*) technique, which is used in almost all PRAM algorithms has no efficient analog on most distributed-memory parallel architectures. In this abstract, we describe an efficient parallel algorithm to find *smallest paths* in simple rectilinear polygons on hypercube multiprocessors, currently one of the most popular types of distributed-memory parallel architectures.

A *smallest path* is a rectilinear path that is simultaneously a *shortest path* with respect to the L_1 metric, and a *straightest path* (i.e., *minimum link path*). Smallest paths have applications in VLSI design (minimizing *vias*), robot motion planning, and the design of rush-hour traffic routes. McDonald and Peters [8] showed that there is a smallest path between any pair of points in any simple rectilinear polygon and developed an optimal $O(n)$ time sequential algorithm for finding smallest paths. They also presented an $O(\log n)$ time, $O(n\log n)$ space parallel algorithm for an n processor CREW PRAM. The most expensive steps in the PRAM algorithm are four trapezoidal decompositions using an algorithm from [2]. The remainder of the algorithm runs in $O(\log n)$ time on the weaker EREW PRAM model with only $n/\log n$ processors and $O(n)$ space.

A direct simulation of the PRAM algorithm from [8] on a hypercube takes $O((\log n \log\log n)^2)$ time with $n\log n$ processors. The algorithm presented in this paper requires only $O(\log^2 n)$ time and $n\log n$ processors. However, the cost of our hypercube algorithm is dominated by several invocations of a trapezoidal decomposition algorithm from [6]. The remainder of our algorithm uses $O(\log n(\log\log n)^2)$ time and only n processors. As a by-product of our algorithm, we have developed a new hypercube technique for finding and eliminating nested pairs in parenthesis systems, and a modified broadcasting technique for eliminating staircases. Both techniques can be

*On leave from the University of Sao Paulo (Brazil), project BID/USP.

†Supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. A-0322 and by the Centre National de la Recherche Scientifique.

implemented in O(log n) time on a hypercube with n processors. It may be possible to improve our processor bound to n by replacing the trapezoidal decompositions with simpler and more problem-specific computations. The algorithm in [6] uses *m-way search trees* which are quite general and powerful.

Our hypercube algorithm is based on the sequential algorithm from [8] which is outlined in the next section. The hypercube algorithm, presented in the third section, uses standard hypercube data movement operations which are described in [6]. For details of the hypercube trapezoidal decomposition algorithm, also see [6]. The proof of correctness of our hypercube algorithm differs from the analysis in [8] in that several new technical lemmas are needed to establish the correctness of our corner location procedure, the nested pair elimination method, and the staircase elimination procedure. To keep our abstract short, we have eliminated these technical lemmas and their proofs (which are long, but not difficult). Most implementation details of the algorithm in Section 3 have been omitted for the same reason.

Recently, de Berg [4] has solved a (sequential) query version of the smallest paths problem with O(log n) query time and O(n log n) preprocessing time. Sack [9] has also studied several problems involving smallest paths. We will assume familiarity with standard computational geometry terminology.

2 Sequential Algorithm

To find a smallest path between two points s and t in a polygon Q, start by drawing a line down from s until the boundary of Q is encountered, turn left, follow the boundary of Q until directly under t, and draw a line up to t. (See Figure 1.) The simple path produced by this process is called a ***starting path***.

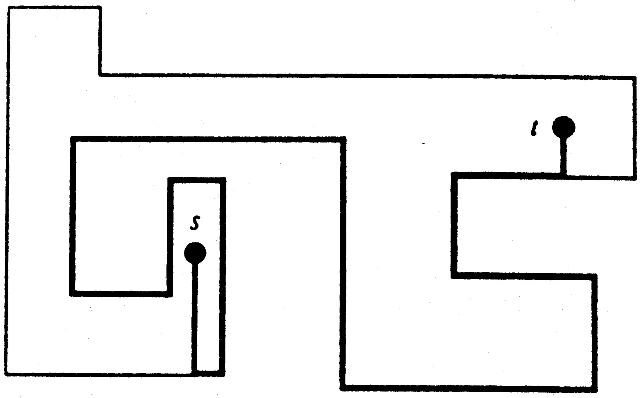

Figure 1: A *starting path* from s to t.

Figure 2 shows five different types of transformations that can be used to improve a starting path. In the diagrams, thin lines indicate the boundary of a polygon, thick lines indicate paths, striped regions indicate interior regions of a polygon, and (solid) grey regions are exterior to a polygon. Transformations veS and vvS are "shortcut" transformations which reduce both distance and number of segments by replacing an arbitrary simple path with a chord between a ***visible pair***. Visible pairs can be obtained from a trapezoidal decomposition of Q. Transformations U and Z reduce distance and the number of segments respectively, and Transformation C eliminates unnecessary collinear points.

An ***ending path*** is a simple path that is obtained from a starting path by repeatedly applying transformations from Figure 2, in such a way that simplicity of the path is maintained, until no further transformations can be made. McDonald and Peters [8] proved that an ending path is a smallest path.

3 Parallel Algorithm

An m-dimensional hypercube has $n = 2^m$ processors (PE's) labelled with the integers from 0 to $n-1$. Two PE's are neighbours iff the binary representations of their labels differ in exactly one bit position. The input consists of a polygon Q and two points s and t within Q. (We consider the boundary of Q to be "within" Q.) Q is presented as a sequence of vertices and edges in counterclockwise traversal order of its boundary. The vertices of Q are stored in traversal order in consecutively numbered PE's. Each edge is stored with its first (in traversal order) endpoint. The coordinates of s and t are known to all PE's. The output of the algorithm is a smallest path between s and t, stored as a sequence of vertices and edges in consecutively numbered PE's starting with s and its incident edge in PE$_0$. In the following description, steps of the smallest paths algorithm are shown in italic font and details of the hypercube implementation are in roman font. Standard hypercube operations are shown in bold font. All of these standard operations can be implemented in time O(log n) with n PE's except sort which takes O(log n(log log n)^2) time. (See [6, 3] for details.) The current best time and processor bounds for finding a trapezoidal decomposition are O(log^2 n) and n log n respectively [6], and these costs dominate our algorithm.

1. *Find a starting path from s to t by modifying the polygon Q as follows so that s and t occur at corners of the modified polygon. First find the horizontal and vertical chords through s. This divides Q into 4 subpolygons, each of which has s as one corner. One of the subpolygons contains t and the other*

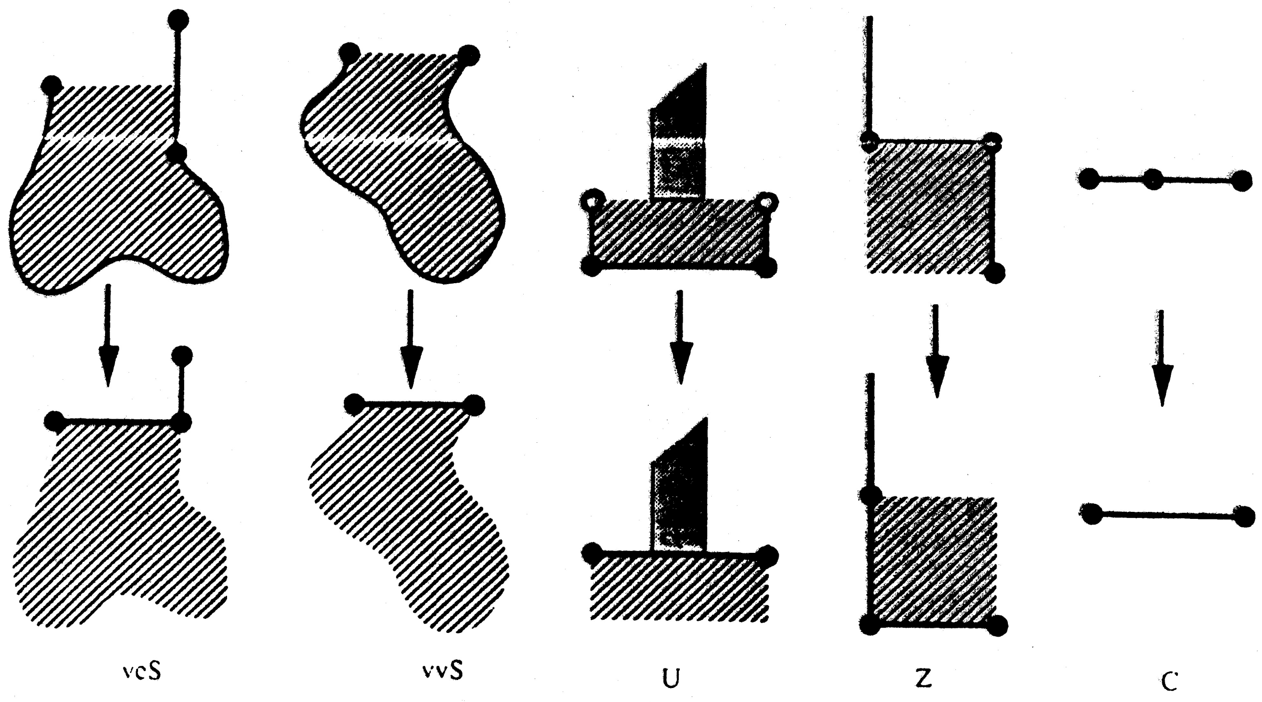

Figure 2: Path Transformations.

three are discarded. (We will ignore the degenerate cases where s is on the boundary of Q.) The chords through t are handled similarly. Now, s and t divide the boundary of the modified polygon (which we will also call Q in subsequent steps) into two paths, P and Q \ P, from s to t. The starting path P is the counterclockwise path starting at s.

- Each PE determines if the vertical or horizontal line through s intersects its line segment. Use find-min operations to select the segments immediately below, above, left, and right of s. Repeat for t. Do a cyclic shift of Q so that the segment below s is stored in PE_0.

- Let a_0, a_1, a_2, and a_3 be the (indices of the) four selected boundary segments with respect to s. Define b_0, b_1, b_2, and b_3 similarly for t. Locate b_0 (the segment immediately below t) in the sorted list a_0, a_1, a_2, a_3. If $a_i < b_0 < a_{(i+1)\bmod 4}$, then the parts of the chords from s to the boundary segments a_i and $a_{(i+1)\bmod 4}$ form the corner at s of the subpolygon that should be retained. Similarly, find $b_j < a_0 < b_{(j+1)\bmod 4}$ to determine the corner containing t.

- Mark all vertices with indices between a_i and $a_{(i+1)\bmod 4}$ and mark all vertices with indices between b_j and $b_{(j+1)\bmod 4}$. Concentrate all vertices with two marks. These are the vertices on the starting path. Create the new corners at s and t. Perform a cyclic shift so that s is stored in PE_0.

2. *Find all locations where "horizontal" shortcut transformations can be applied. Each visible pair has an associated pair of integers (corresponding to the PE's that hold the elements). These pairs of integers are "properly nested" in the sense that no two pairs (x_1, x_2) and (y_1, y_2) have $x_1 < y_1 < x_2 < y_2$. Furthermore, elimination of a pair also eliminates all pairs nested within it [8]. Therefore, it is only necessary to consider non-nested pairs.*

- Use the trapezoidal decomposition algorithm from [6] to find the horizontal trapezoidal edges of Q. Use a sorted list of horizontal edges of Q with a modified bitonic merge to extract the horizontal visible pairs.

- To eliminate nested pairs, first concentrate all vertices. Then do a modified partial sum in which, at each dimension exchange, a PE in the higher subcube is marked if the pair received from the other subcube encloses the pair it currently stores. Now send all unmarked visible pairs back to their original PE's (i.e., according to the first components of the pairs) and generalize according to the second components, marking all PE's in between the two components. Finally, concentrate all unmarked PE identifiers to obtain the list of non-nested pairs.

3. *In parallel, perform the shortcut transformations associated with all non-nested pairs. Also apply C transformations, if possible, after each shortcut.*

- Each PE examines the information in adjacent PE's to identify locations where new edges (chords) and vertices must be created. Do a **partial sum** to determine the number of new vertices to be created followed by a **distribute** to actually create the positions. The information to fill in each position now comes from vertices that are a constant number of positions away. C transformations are now performed in a similar way.

4. Repeat steps 2 and 3 in the vertical direction.
5. *Any remaining visible pairs of Q will have one endpoint on P and one endpoint on $Q \setminus P$, so only U, Z, and C transformations are possible.*

- Find all horizontal visible pairs and store them so that each vertex of $Q \setminus P$ knows its visible pair. **Concentrate** the visible pairs by first (i.e., $Q \setminus P$) coordinate. Identify locations for U transformations and mark the corresponding PE's at the P end. The marked PE's now store a list of locations on P at which U transformations should be made in reverse order of second (i.e., P) coordinate. **Concentrate**, **invert**, and **distribute** the list to the marked PE's. Perform the U transformations and any required C's using the method of Step 3.
- Repeat for vertical U's and C's.

6. *The only possible remaining transformations are Z's and the C's that follow them. Z's can form staircases, and some staircases (or parts of staircases) can be eliminated in two incompatible ways which we will call "forward" and "backward".*

- Find all horizontal and vertical visible pairs. Staircases will be identified and eliminated by the creation of cluster points in one pass of an "ascend" algorithm, where exchanges within a hypercube dimension take place at each step. C transformations should also be performed, when necessary. The cluster points are potential locations for the new vertices that are created by Z transformations. The implementation is based on the following properties which are true after every step.

    1. Every processor in a subcube knows the visible pair of the only candidate (vertex or cluster point) that can be Z-transformed forwards.
    2. Every processor in a subcube knows the visible pair of the only candidate (vertex or cluster point) that can be Z-transformed backwards.
    3. Every processor knows which candidates are in the upper dimensional part of its subcube and which are in the lower dimensional part.

    In each step, all PE's exchange information in the current dimension and then compare the current backward candidate from the upper subcube with the forward candidate from the lower subcube to determine whether a new cluster is to be created. If a new cluster is created, then all processors in the subcube must update their candidate information.

- The path that results when all Z (and C) transformations have been performed can be recovered by creating the new vertices and edges associated with cluster points and deleting vertices "covered" by cluster points. The method is similar to Step 3.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, vol. 3, pp. 293-327, 1988.

[2] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM J. Comput.*, vol. 18, pp. 499-532, 1989.

[3] R. Cypher and C.G. Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on a Hypercube and Related Computers," *ACM Symposium on Theory of Computing*, 1990.

[4] M. de Berg, "On Rectilinear Link Distance," *Tech. Rep. RUU-CS-89-13*, Dept. of Computer Science, Univ. of Utrecht, 1989.

[5] F. Dehne, A. Ferreira, and A. Rau-Chaplin, "Parallel Fractional Cascading on a Hypercube Multiprocessor," *Proceedings of the Allerton Conference on Communication, Control and Computing*, Monticello, Ill., 1989.

[6] F. Dehne and A. Rau-Chaplin, "Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry," *Jour. of Parallel and Distributed Computing*, vol. 8, pp. 367-375, 1990.

[7] R.M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, ed. J. van Leeuwen, MIT Press, 1990.

[8] K.M. McDonald and J.G. Peters, "Smallest Paths in Simple Rectilinear Polygons," *Tech. Rep. TR 89-4*, School of Computing Science, Simon Fraser Univ., 1989.

[9] J. Sack, "Rectilinear Computational Geometry," *Ph.D. Thesis*, School of Computer Science, Carleton Univ., 1984.