A Parallel Algorithm For Surface Triangulation

Theodore Johnson, Panos E. Livadas, Sunjay E. Talele
Dept. of Computer and Information Science
University of Florida

March 31, 1993

Abstract

In many scientific fields, three dimensional surfaces must be reconstructed from a given collection of its surface points. Applications for surface reconstruction exist in medical research and diagnosis as well as in design intensive disciplines. Fuchs, Kedem, and Uselton and Keppel show that surface reconstruction via triangulation can be reduced to the problem of finding a path in a toroidal graph. This paper presents a parallel algorithm to find the minimum cost acceptable path in an m by n toroidal graph. We then show an implementation of the parallel algorithm on a parallel architecture, using a message passing approach. Results are shown, along with suggestions for future enhancements.

1 Introduction

The problem of surface reconstruction is important in many scientific disciplines, such as medical research and diagnosis, architecture, and in geometric design. For instance, light microscopes capable of high magnifications are monocular, and thus can only generate cross sections of an object. Given a set of these cross sections, the three–dimensional surface must be reconstructed. Similarly, human organs can be reconstructed from cross–sectional X–rays (tomographs), and terrain surfaces can be reconstructed from survey information.

Consider a collection of a surfaces' parallel cross sections, as shown in Figure 1. Each cross section is defined by a finite set of points along the boundary of ia contour. We can approximate the surface by a collection of triangles between any two successive contours as shown in Figure 2. These triangles, which we will call *tiles*, are formed by two vertices on one contour and one vertex on the other contour. Keppel [Kep75] shows that, if two consecutive contours consist of m and n vertices, respectively, then the number of tile arrangements is T, where

$$T(m,n) = \frac{(m+n)!}{(m-1)!(n-1)!}$$

For $n = m = 12$ points, we obtain approximately 10^8 different tile combinations, and therefore different surface shapes. Thus, an exhaustive search is unacceptable.

Previously, Keppel [Kep75] and Fuchs et al. [FKU77] have shown that the problem of finding an optimal triangular arrangement can be reduced to finding a minimum cost path in a directed toroidal graph, which

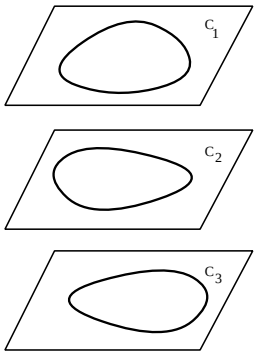


Figure 1: Intersection of surface by three planes.

can be solved by Dijkstra's algorithm. In this paper, we present a dynamic programming solution to the triangular arrangement problem. We map the toroidal graph to a planar graph, whose structure we exploit to form a parallel algorithm, suitable for a message–passing parallel architecture. An implementation is presented, along with optimizations and an analysis of the speedup obtained.

2 Previous And Related Work

Previous techniques for extracting surface geometries from volume data essentially fall into three categories: defining surfaces from contour data (contour stitching), defining surfaces from sampled data (surface construction), and geometrically deformable models. Keppel [Kep75] and Fuchs et al. [FKU77] proposed an optimal approximation of a three–dimensional surface from a set of cross sections using triangulation. The optimal surface was determined by finding a path in a directed toroidal graph. Keppel [Kep75] defined an optimal surface to be an acceptable surface represented by the maximal weighted path, thus maximizing the enclosed volume of the surface. Fuchs et al. [FKU77], on the other hand, sought the minimum weighted path, thus finding the minimum surface area of the polyhedron formed by the triangular mesh. Christiansen and Sederberg [CS78] pointed out that heuristics work best in these methods when contour pairs are similar in size and shape, and are mutually centered. Furthermore, dissimilar contour pairs can be transformed such that they are "normalized", and subsequent heuristics produce good results.

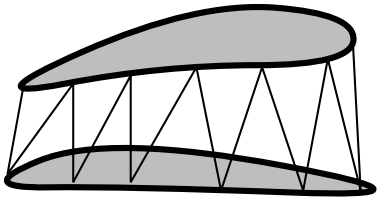


Figure 2: A pair of contours connected with triangles to form a surface.

The above *contour stitching* methods work well for the simple case of a single closed contour on each slice, but not as well in the case of *branching*. Branching occurs when either a single contour splits into several contours, or several contours merge into a single contour. Christiansen and Sederberg [CS78] and Anjyo et al. [AOUK87] proposed transforming each pair of adjacent serial sections into a number of simple cases. In more acomplex cases, human interaction would be required. This may be unacceptable in some cases, such as a clinical environment, where user interaction should be minimal.

Lin, Chen, and Chen [LCC89] connected two–dimensional cross sections using dynamic elastic contour interpolation, spline theory, and quadratic variation–based surface interpolation [LCC89], rather than the contour stitching methods of the above algorithms. A series of intermediate contours are formed between each of the original cross sections using elastic interpolation. Then, contours are mapped to a surface function, to calculate initial surface values. These surface values are then refined by quadratic variation–based surface interpolation to produce the final surface representation.

The second method of surface reconstruction involves creating surface geometries from regularly spaced data values in a three–dimensional grid. Typically, data from MRI (magnnetic resonance imaging), CT (X–ray computed tomography), and SPECT (single photon emission computed tomography) scans are provided in this grid format. A surface may be defined as a set of points with a given value. For instance, data points from a CT scan may have values corresponding to tissue density. Since bone has a different density than the soft tissue around it, the boundary between the densities represents the surface of the bone.

These grid–oriented methods examine a "cube" for which the data value at each corner is known. The corner values are compared with the desired surface value. If a given cube has at least one corner value above the desired value and one corner below the desired value, we know the surface exists in the cube. If we scale the corner values by subtracting the desired surface value, then the value of the desired data set is zero. So, for any cube edge, if the signs of the endpoints are opposite, then the surface must pass through that edge. The precise location of the surface vertex is determined by assuming the values along a cube edge linearly interpolate the cube corners. Furthermore, if the surface intersects edge(s) of a cube face, then the surface must intersect the cube face also.

Lorensen and Cline [LC87], Wyvill et al. [WMW86], and Bloomenthal [Blo88] all use a variation of this method. Bloomenthal [Blo88] uses an algorithmic approach in determining surface vertices, while the other two methods are table–driven. Bloomenthal's method [Blo88] also allows for more sampling around particular areas of a surface, allowing for a better surface fit in some cases. The method of Wyvill et al. [WMW86] considers the cube corner values of each face. There are only seven combinations of cube values for a given face. Lorensen and Cline's [LC87] *marching cubes* algorithm, on the other hand, examines the

values of all the cube corners.

This set of algorithms does not suffer from the branching problem because the entire surface is extracted in the volume. These methods, are however, restricted to generating models in which elements are at most the size of a voxel (volume element), which makes approximating the data impossible. Also, these methods are incapable of generating a closed model of an object which is not necessarily closed, such as the interior of an open bottle. Note that these cube methods are appropriate only when dealing with grid data. In the case of contour data, triangulation methods will me more efficient.

The third method of constructing surface geometries from volume data is geometrically deformable models (GDMs) [M+91]. This method differs from the previous two in that it attempts to approximate, rather than interpolate the surface. The motivation behind GDMs is that a geometric approximation provides more opportunities for analyzing and visualizing the object than interpolative methods.

Geometrically deformable models are created by placing a "seed" model inside of a volume data set. The model is deformed by a set of relaxation processes which adheres to a set of constraints that provide a measure of how well the model fits the data. These models can be thought of as semi–permeable balloons placed inside a scanned object. The balloon is expanded until its surface reaches the boundaries of the scanned object. The balloon is, in reality, a polygonal mesh. Volume data are sampled only at the vertices of these polygons. Cost functions associated with the model constraints are placed at these vertices as well.

Geometrically deformable models, like the cube algorithms from the previous section, handle the branching problem explicitly by treating a collection of 2D cross sections as a true 3D data set. Since sampled data is treated by placing geometric relationships on models, GDMs are highly adaptive, thus handling elements of noise favorably, and providing for varying levels of detail.

The appropriate technique for reconstructing an surface from the points on the surface depends on the nature of the input data and the desired form of the output. In this paper, we will examine the first reconstruction method, contour stitching, and explore the possibility of speeding up previous attempts via parallel processing. Contour stitching is appropriate when the input is a set of points on the surface, and the desired output is a tiling of the surface.

3 Definition Of The Problem

Consider an unknown three–dimensional surface intersected by a collection of planes. Each plane contains a simple closed curve, which is defined by a finite sequence of points. Any two consecutive points are connected by a *contour segment*. The collection of contour segments which form the polygonal approximation of the

curve is called a *contour*.

We review the notation used by Fuchs et al. [FKU77]. Let one contour, C_1, be defined by the sequence of m contour points $P_0, P_1, P_2, \ldots, P_{m-1}$, and let another contour, C_2, be defined by the sequence of n contour points $Q_0, Q_1, Q_2, \ldots, Q_{n-1}$. A positive ordering exists for each set of points. That is, P_1 follows P_0, P_2 follows P_1, and, in general, P_{i+1} follows P_i where the $+$ is the addition modulo m operator. Similarly, Q_{j+1} follows Q_j where $+$ is the addition modulo n operator.

As mentioned previously, we form the surface between any two successive contours using triangular tiles. The triangle will have two vertices on one contour and one vertex on the other contour. Thus each tile is of the form $\{Q_i, Q_k, P_j\}$ or $\{P_i, P_k, Q_j\}$. Fuchs, et al. [FKU77] makes the observation that any tile of the form above can be reduced to a sequence of *elementary tiles* without changing the approximating surface. An *elementary tile* is defined to be a triangle whose form is either $\{P_i, P_{(1+1) mod\ m}, Q_j\}$ or $\{Q_j, Q_{(j+1) mod\ n}, P_i\}$ (see figure 3). An edge that connects an end of a contour segment on C_1 with an end of a contour segment on C_2 is called a *span*. Thus, an elementary tile is composed of two spans and a contour segment.

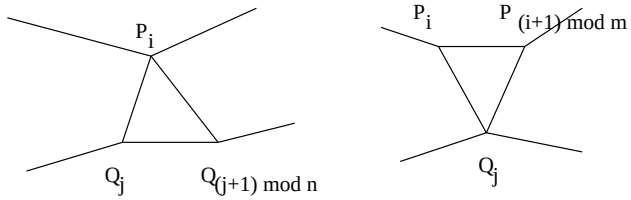


Figure 3: Elementary tiles.

A given set of elementary tiles can be represented by a directed, weighted, toroidal graph, $G = (V, E, \omega)$. Vertices in G correspond to spans, while edges correspond to tiles. We define V and E as follows:

$$V = \{v \in G \ : \ v \ = \ v_{i,j} \ i = 0, 1, 2, \ldots, (m-1); j = 0, 1, 2, \ldots, (n-1)\}$$

$$E = \{e_{\overrightarrow{ij}} \ : \ e_{\overrightarrow{ij}} = v_{i,j} v_{i,(j+1) \bmod n} \} \cup \{e_{i\overrightarrow{j}} \ : \ e_{i\overrightarrow{j}} = v_{i,j} v_{(i+1) \bmod n, j} \}$$

$$\text{for } i = 0, 1, 2, \ldots, (m-1) \text{ and } j = 0, 1, 2, \ldots, (n-1)$$

Furthermore, we associate a weight, ω, with each edge of G, representing the "quality" of each tile. An *acceptable path* is defined to be a closed path with initial vertex $v_{i,0}$ such that the path contains exactly one horizontal edge between any two adjacent columns and exactly one vertical edge between any two adjacent rows. This property ensures that the surface defined by the acceptable path will *fit* together and thus form an *acceptable surface*, as well as a *complete surface*. Now let $\mathcal{P}(i)$ denote the set of all acceptable paths at i.

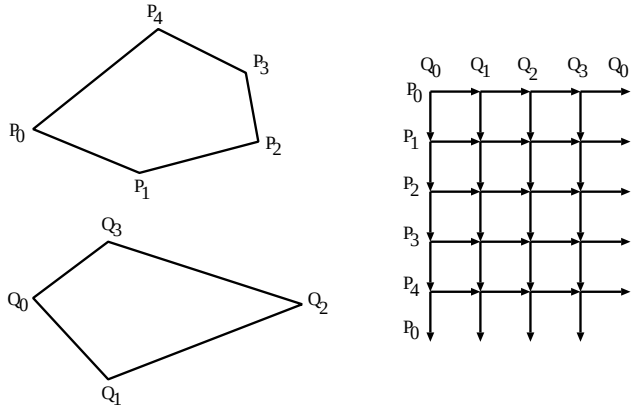


Figure 4: Contours and an associated toroidal directed graph.

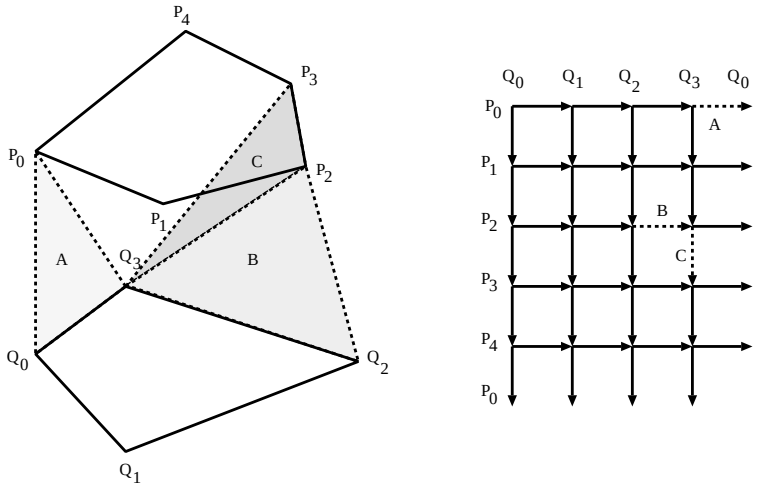


Figure 5: Elementary tiles along with their respective edges in a toroidal graph.

Note that if $\pi \in \mathcal{P}(i)$, the length of π is equal to $(m+n)$. An acceptable path is either a vertex-simple cycle, or consists of two vertex-simple cycles that share one vertex. Figure 6 illustrates a toroidal graph ($m = 5$, $n = 6$) where two acceptable paths at $i = 2$ are highlighted.

Next, we define the *cost* of path, π to be the sum of the weight of the edges in π. We define the *minimum cost path* at i, $\mu(i)$, to be the path among all $\pi \in \mathcal{P}(i)$ such that cost is minimized. Finally, let μ_G be the minimal cost path among all μ's.

3.1 Mapping the Toroidal Graph to a Planar Graph

By cutting G open and gluing two copies of G together we obtain a new planar graph $G'(V', E', \omega')$, as illustrated in Figure 7. By transforming the toroidal graph to a planar graph, our original problem has become the following:

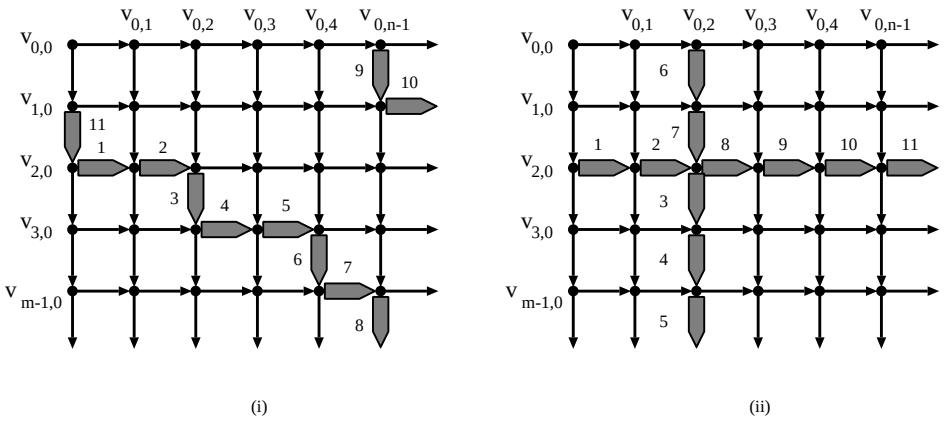


Figure 6: A toroidal graph and two acceptable paths at $i = 2$.

Find the minimum cost path $\pi'(i)$ among all simple paths with initial vertex at $v'_{i,0}$ and terminal vertex at $v'_{(m+i),n}$ $\forall i \in \mathcal{M}$. Then the minimum cost path $\mu'_{G'}$ of G' is the one among all $\pi'(i)$'s ($i \in \mathcal{M}$) of minimum cost.

For the remainder of the paper, we shall deal with the planar graph only.

3.2 Non-crossing Paths

Assume $\mu(i)$ and $\mu(j)$ are two minimal cost acceptable paths at $v_{i,0}$ and $v_{j,0}$, respectively. If $i < j$, then $\mu(i)$ *does not cross* $\mu(j)$. That is, $\mu(i)$ lies entirely "above" $\mu(j)$, but may still have common vertices and edges with $\mu(j)$. This can be seen by observing Figure 8. Paths P_0 and P_1 are shown going through points A and B. The two routes from A to B are ACB and ADB. There are three cases:

- The cost of the path ACB is greater than the cost of ADB. In this case, P_1 should follow ADB.
- The cost of the path ACB is less than the cost of ADB. In this case, P_0 should follow ACB.
- The cost of the paths, ACB and ADB, are equal. In this case, we can replace one of the two paths with the other path.

We can see that the determination of the path μ_G can be done as follows. First, we determine the path $\mu(0)$ (Fig. 9(i)). Now, $\mu(m)$ is a copy of $\mu(0)$ which starts at $v_{m,0}$ and ends at $v_{(m+m),n}$ (Fig. 9(ii)). Since paths do not cross, the calculation of all other paths $\mu(1)$, $\mu(2)$, ..., $\mu(m-1)$, can be restricted to the subgraph of G that is obtained by "chopping off" the portion of the graph G lying above $\mu(0)$ and the portion below $\mu(m)$. The shaded areas in Figure 9(iii) represent the chopped off portions of the graph.

Suppose that instead of calculating $\mu(1)$ after calculating $\mu(0)$, we calculate $\mu(m/2)$. Now, the paths $\mu(i)$, $i = 1 \ldots m/2 - 1$ can be calculated in the subgraph bounded by $\mu(0)$ and $\mu(m/2)$.

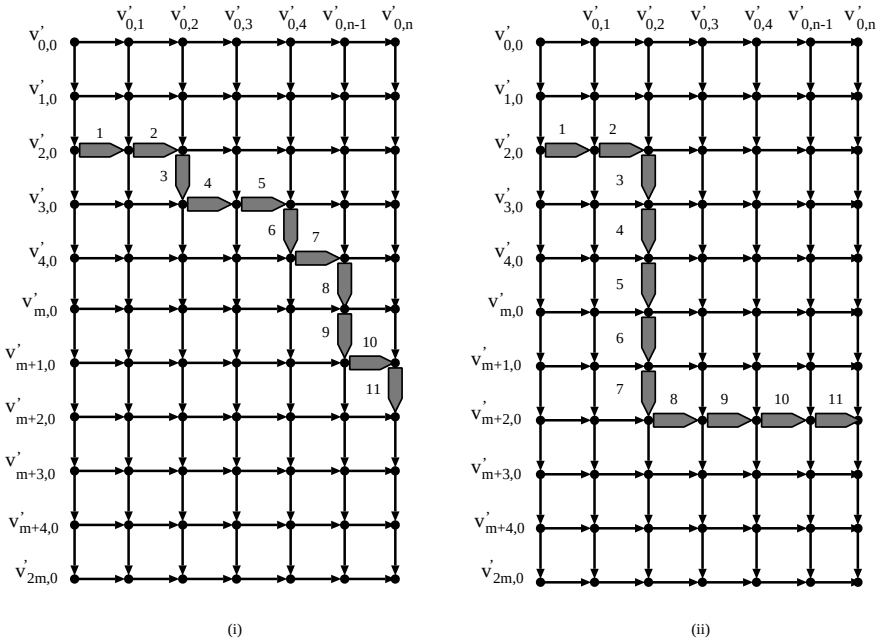


Figure 7: Mapping the toroidal graphs in Figure 2.4 to planar graphs. The images of the two acceptable paths in Figure 2.4 (i) and (ii) are shown in (i) and (ii), respectively.

The regions formed by chopping off subgraphs can also be divided into finer pieces, resulting in a quicksort–type algorithm. Fuchs et al. [FKU77] show that this algorithm reduces the time to compute the minimum cost acceptable path from $mT(m,n)$ to $T(m,n)\log(m)$, where $T(m,n)$ is the time to solve the single source shortest path problem on a toroidal graph. Below is the pseudocode for the serial version of this algorithm.

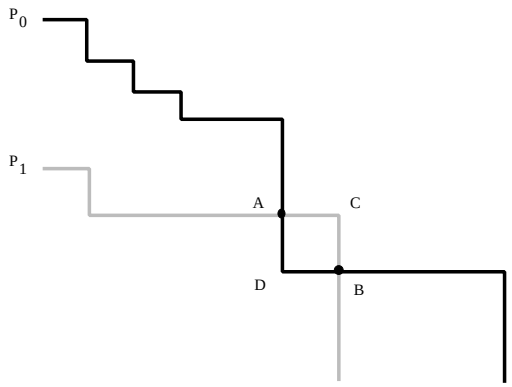


Figure 8: Non-crossing paths.

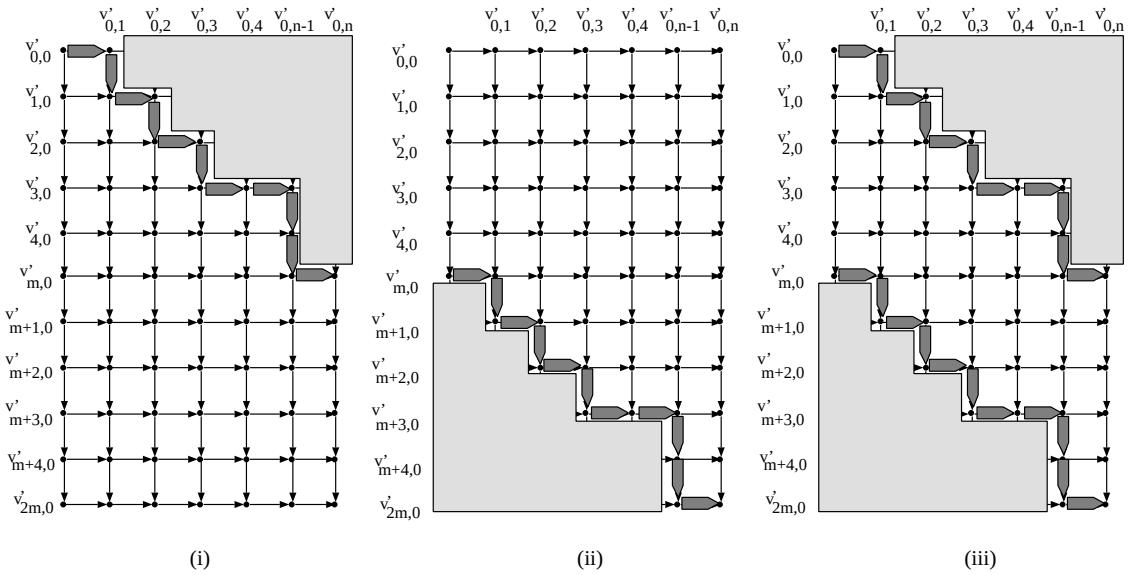


Figure 9: Restricting G to the subgraph obtained by removing the portions above $\mu(0)$ and below $\mu(m)$.

```
Serial Algorithm

shortestpath(G) {
   P0 = findonepath(G,0)
   G'=restrict(G,P0)
   {P1,..,Pm}=findallpaths(G',1,m)
   return(min(P0,..,Pm))
}


findallpaths(G,pathlo,pathhi) {
   m=(pathlo+pathhi)/2
   if(pathlo<m) {
      Pm=findonepath(G,m)
      Plo=findallpaths(restrict(G above Pm),pathlo,m-1)
      Phi=findallpaths(restrict(G below Pm)m+1,pathhi)
   }
   return(union(Plo,Pm,Phi)
}
```

4 Parallelizing The Solution

In the serial algorithm, after path $\mu((i+j)/2)$ is found, G can be divided into two smaller subgraphs. These are independent subtasks, so after finding the path $\mu((i+j)/2)$, we can find the paths $\mu(i+1), \ldots, \mu((i+j)/2-1)$ and $\mu((i+j)/2+1), \ldots, \mu(j-1)$ in parallel. This leads to a divide and conquer type algorithm, as is illustrated in Figure 10.

As with a parallel quicksort algorithm, this algorithm will be efficient only if we can find a single path

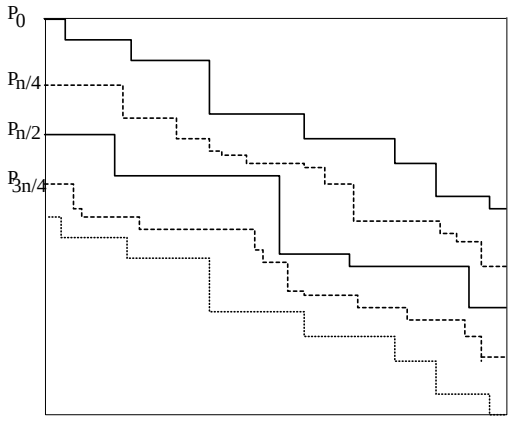


Figure 10: Divide and conquer method of computing paths.

efficiently in parallel. Consider the problem of finding the shortest path from a node s to another node t in the graph G. Let us refer to the nodes by their grid coordinates (x, y). Assume that s is at $(0, 0)$ and t is at (i, j). The edges in G only go down or to the right, never up or left. Therefore, all paths to (i, j) must travel through either $(i-1, j)$ or $(i, j-1)$, as shown in Figure 11. If we let $P(i, j)$ be the minimum weight path from $(0, 0)$ to (i, j) and let $C(i, j)$ be the cost of that path, then

$$C(i,j) = min(C(i-1,j) + w((i-1,j),(i,j)), C(i,j-1) + w((i,j-1),(i,j)))$$

Furthermore, $P(i, j)$ is either $P(i-1, j)|((i-1, j), (i, j))$ or $P(i, j-1)|((i, j-1), (i, j))$, depending on which path has lower cost.

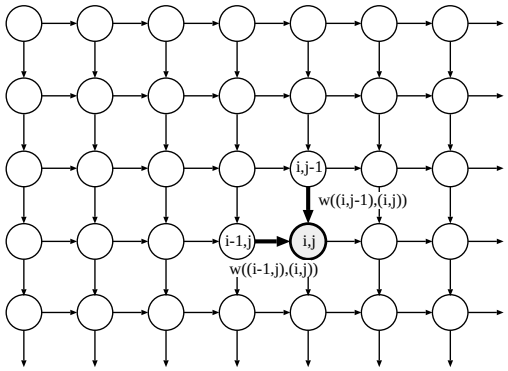


Figure 11: Dynamic programming calculation of the shortest path.

Calculating the minimum cost path thus reduces to a dynamic programming problem, iterating across all (i, j) such that $i + j = k$, for all k between 0 and $2n$, as shown in Figure 12. The parallelization is to split up the iteration for each diagonal in the graph represented by a particular value of k.

For load balancing, each processor should be responsible for the same number of nodes along a diagonal. Also, to minimize messages between nodes, the nodes for which a processor is responsible should be

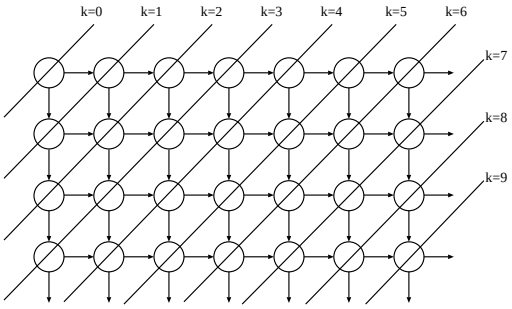


Figure 12: Dynamic programming - iterating across all k.

contiguous. Figure 13 shows the distribution of nodes based on these considerations. These considerations lead to the algorithm given below. Note the algorithm does not include the actual computation for finding the shortest path. It is merely a control structure to allocate computations to the appropriate processors.

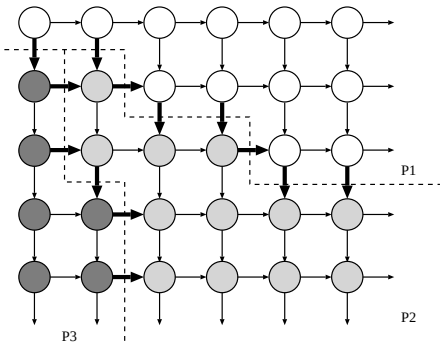


Figure 13: Allocation of nodes to processors.

```
Parallel Algorithm

shortestpath(G,m,n,p){
   /* find the shortest path in 2m x n G using p processors */
   P0 = findonepath(G;1,p;0)
   G'=restrict(G,P0)
   findallpaths(G;1,p;1,m)
   return(min(P0,..,Pm))
}


findallpaths(G,prlo,prhi,pathlo,pathhi){
   m=(pathlo+pathhi)/2
   if(pathlo<m){
      Pm=findonepath(G; prlo,prhi; m)
      if(prlo<prhi) then in parallel {
            /* Next line is graph splitting criteria */
            /* n_pr_lo = number of processors to allocate to lower subgraph */
            n_pr_lo= ⌈(prlo + prhi)/2⌉ − 1
            findallpaths(restrict(G below Pm),prlo,prlo+n_pr_lo,m+1,pathhi)
            findallpaths(restrict(G above Pm),prlo+n_pr_lo+1,prhi,pathlo,m-1)
```

```
        }
      else{
        findallpaths(restrict(G above Pm),prlo,prlo,pathlo,m-1)
        findallpaths(restrict(G below Pm),prlo,prlo,m+1,pathhi)
      }
   }
}
```

5 Implementation

We have implemented the above parallel algorithm on a BBN TC2000 parallel computer. Each node in the TC2000 consists of a Motorola 88100 RISC CPU and 88200 cache/memory management units. Interprocessor communication was accomplished via message queues, which had to be simulated on the shared memory machine.

Various issues needed to be considered in the implementation of the above algorithm. Data distribution, computational responsibilties, coordination, and synchronization all needed to be addressed. We used a master–slave paradigm to implement the above parallel algorithm. The master runs the above parallel control structure algorithm, while the slaves accept compute requests sent to them by the master. We begin, then, by downloading the compute task to each of the slave nodes. Next, we allocate globally visible message queues, so that the nodes may communicate with each other as well as the master (coordinator) node.

We start a computation by distributing the contour points $P_0, P_1, P_2, \ldots, P_{m-1}$, and $Q_0, Q_1, Q_2, \ldots, Q_{n-1}$ for two successive contours. We use the *area* of the elementary tile represented by an edge as the weight for that edge. Note that from this information, the weight of any edge e_{ij} in the $2m \times n$ graph G may be calculated.

Each processor must know the part of the graph for which it is responsible as illustrated in Figure 13. Processors split (as equally as possible) the nodes along a diagonal. Each processor uses the same algorithm to determine which graph nodes it is responsible for. Hence, the node responsibility information is implicit, and does not require any interprocessor communication or intervention from the master processor. All communication in Figure 13 takes place across the dotted lines. Recall that the edges in the graph are directed only to the right and down. A processor responsible for the node directly to one side of a dotted line will have to wait for another processor to send cost information for the node directly to the other side of the dotted line. Similarly, a processor determines when it needs to send information about a node it is calculating to another processor.

To determine which processor is responsible for a particular node, we must first calculate the number of

nodes, n_k, that lie at the same manhattan distance k as the node in question. We define the *position* of a node to be the order of the node on the diagonal. The node positioned at the upper–left of the diagonal is at position 1, while the node at the bottom right of the diagonal is at position n_k. Denote the total number of processors by p, and the processor number by i. Then processors $P_1, P_1, \ldots P_p$ are responsible for nodes $[N_{i-1} + 1..N_i]$, where $N_0 = 0$ and $N_i = \frac{n_k - N_{i-1}}{p-i+1} + N_{i-1}$.

Another issue is that of recovering the shortest path after the cost of the shortest path has been found. At each node, we tag the node with the entering edge, which represents the last edge of the shortest path to the node. Calculating the shortest path now consists of backtracking from the terminal node to the start node. Figure 14 shows an example of recovering a path. Note that backtracking (shown as arrows) may involve crossing processor boundaries, so all slave nodes involved in the computation of the path must be prepared to cooperate in the backtracking process as well.

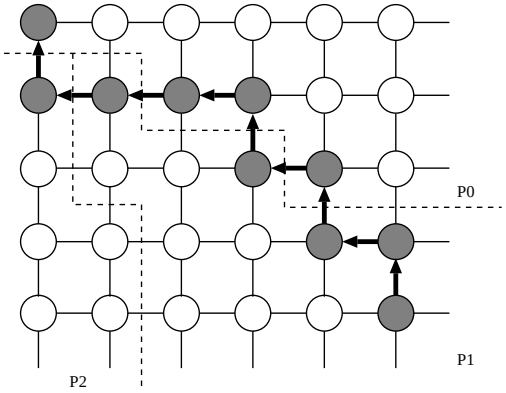


Figure 14: Backtracking to recover shortest path.

After each slave processor finishes processing the graph nodes it is responsible for, the slave waits for backtracking messages from other slaves. The slave node which is responsible for the terminal node of the path initiates the backtracking process. As each processor backtracks within its own subgraph boundaries, it sends the portion of the path it is responsible for to the master. Once a node is reached that is not the processor's responsibility, control is passed to the appropriate processor. The processor then goes back to waiting for a backtrack messages since, the path may cross into a processor's responsibility region more than once, as shown in Figure 14.

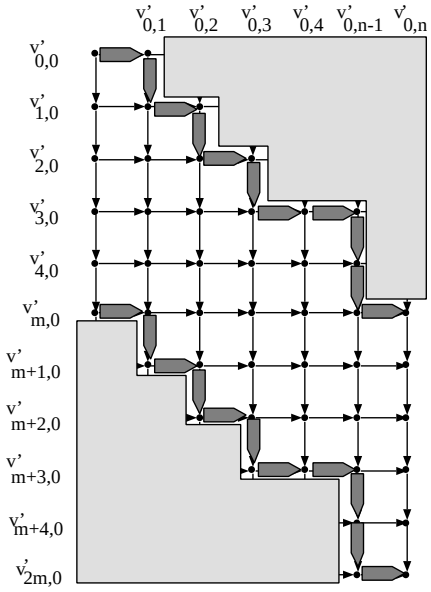
The master knows when it has received all nodes in a given path when it receives the start node of the path. Upon receiving the last node, the master sends terminate messages to all processors involved in computing the path, so that the slave processors can stop waiting for any more backtrack messages. The slaves can then return to soliciting new compute requests.

Upon receiving a computed path from the slave nodes, the master node must determine what part of the graph to "chop off". The computed path will replace one of the old bounding paths for a slave node, depending on whether the slave will work on paths above or below the path just computed. Thus, the slaves should be informed of the new bounding path by the master.

The slave nodes know the dimensions of the graph G to be $2m \times n$, from the initial distribution of contour points. To determine what part of a subgraph is to be worked on, we only need to store the top and bottom boundaries of G. We denote by M the set of non-negative integers bounded above by $2m$. We define T and B to be the top and bottom boundaries, respectively, of graph G, that is

$$T = \{t_i \in M; 0 \leq i \leq n\}, \qquad B = \{b_i \in M; 0 \leq i \leq n\}$$

Thus, to update the bounding paths of a calculation, the master only need send T or B to the slave nodes, not the entire graph. Figure 15 shows a sample subgraph, and the corresponding boundaries.



m=5

n=6

T = [0 0 1 2 3 3 5]

B = [5 6 7 8 8 10 10]

Figure 15: Graph boundaries.

The shortest path among P_i must be determined at the end of the computations. In the case where more than one processor is working on a given path, the path will have to be sent to the master for the purposes of graph splitting. Thus, after the master splits the graph, it can compare the path to previous paths found and discard the more expensive one.

In the case where only one processor is working on a path, the path need not be sent back to the master. At this point, the single processor is working in a region which is solely its responsibility. Here, the single processor compares the calculated path with the others it has found so far. After the single processor has

finished calculating the paths it is responsible for, it sends the least cost path to the master.

6 Results

Figure 16 shows the execution speedups for the parallel algorithm using a pair of contours, consisting of 100, 200, 300, 400, and 500 points, respectively. Since calculation of pairwise contours are independent of each other, examining a single pair is sufficient for measuring the performance of our algorithm. For simplicity, we have used the same number of contour points for each contour, ($m = n$). The contours were generated in a somewhat random fashion, to produce more realistic results.

Note the slight performance degradation between one and two processors. In the one processor case, a master and slave were on the same CPU, while in the two processor case, they were on different CPUs. The degradation is due to the message overhead between the CPUs. In all other processor configurations, one processor was reserved for use as a master.

The large drop in exection times from 2 to 3 processors is expected, since two slave nodes are participating in the computation rather than one. Note the small change between using 3 and 4 processors, as well as between 5 and 6 processors. Currently, we allocate *half* of the processors to each subgraph. Therefore, in the case of say, 3 slave processors, processor 1 will still compute half of the graph, while processors 2 and 3 compute the other half. In this case, processor 1 is responsible for roughly the same amount of computation as in the 2 slave processor case, thus increasing the 3 processor case's execution time to roughly that of the 2 processor case. This suggests a better heuristic is needed to allocate processors to subgraphs.

Also, note that the speedup begins to diminish as you increase the number of processors, and actually begins to degrade at about 12 processors. This is mostly due to the shared memory architecture on which the algorithm was implemented. A large amount of message overhead was incurred by the excessive traffic created by sending messages between processors in the shared memory architecture. The algorithm is better suited to a true memory–passing architecture, which, in our case, had to be simulated.

Figure 6 shows an example of the input and output of our algorithm. On the left is the set of contours which represents a glass and is input to the algorithm. On the right the derived surface by the algorithm is displayed.

7 Optimizations

The results given in the previous chapter are from using the parallel algorithm. Several optimizations can be made to both reduce message passing and storage.

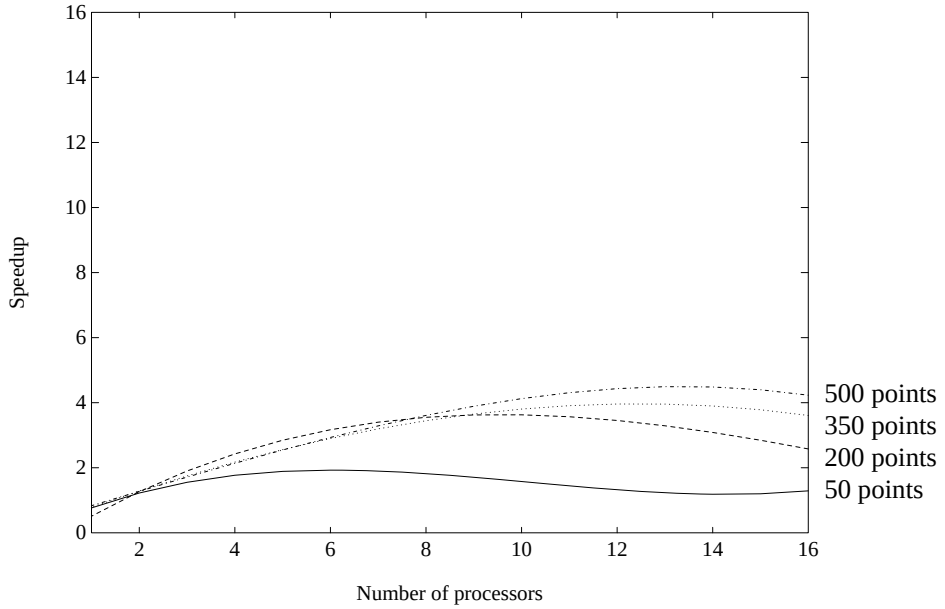


Figure 16: Speedup using parallel algorithm (without optimizations).

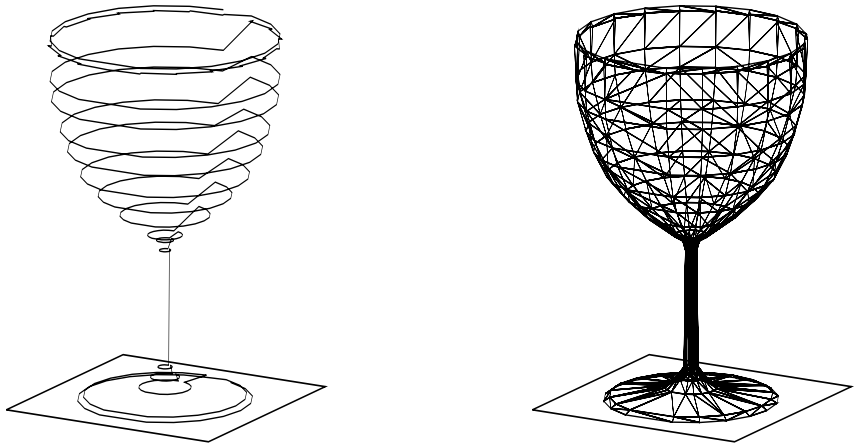
7.1 Processor allocation heuristic

In the original description of the algorithm, the processor allocation criteria is based solely on the number of paths in the computation. That is, we define `n_pr_lo` in the parallel algorithm given above to be $\lceil (prlo + prhi)/2 \rceil - 1$. As mentioned previously, the divide-and-conquer nature of our algorithm will lead to an irregularly shaped grid as the one shown in Figure 17. In this case, the number of paths is not a true indication of the work to be done in a particular subgraph. For better load balancing, we split the graph based on the number of nodes to be calculated in a subgraph. Namely, the number of processors assigned to calculate paths above P_m should be proportional to the number of graph nodes above P_m.

We define W_{lo} to be the subgraph in which the paths $\mu(\texttt{pathlo})$ through $\mu(\texttt{m} - \texttt{1})$ will be found, and we define W_{hi} similarly. Our load balancing mechanism is based on the sizes of W_{lo} and W_{hi}, respectively. We now change the graph splitting criteria in the parallel algorithm to:

`n_pr_lo = (prhi-prlo+1) *` $|W_{lo}|$ `/ (`$|W_{lo}|$`+`$|W_{hi}|$`)`

Figure 18 shows the speedups obtained with by applying this heuristic, using the same input data sets as in Figure 16.



(a) Glass contours

(b) Glass surface

Figure 17: Input and output of the algorithm.

7.2 Minimizing messages along a diagonal

Another consideration of our algorithm is the tradeoff between parallelism and message passing. For instance, if we have a diagonal consisting of n nodes and we have n processors, will the parallelism of assigning one node to each processor overcome the message passing overhead? For this purpose, it may be better to require that a processor be responsible for at least d nodes along a diagonal in order to reduce the number of messages passed.

If p processors are participating the computation for a given diagonal calculated in the graph, then exactly $2(p-1)$ messages will be needed. Two messages will be needed at each processor boundary (one sent, one received), as illustrated in Figure 19. If we denote m to be the cost of each message, c to be the cost of calculating a given node, and d to be the number of nodes along the diagonal, then the following holds:

- $\frac{cd}{p}$ is the amount of time for p processors to calculate the d nodes along the diagonal.
- $2m(p-1)$ is the message cost involved in calculating the diagonal with p processors

Thus, the total time to calculate the diagonal is $\frac{cd}{p} + 2m(p-1)$. To find the optimal number of processor,

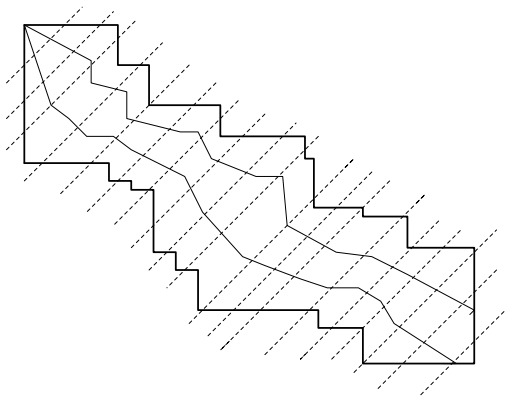


Figure 18: Irregularly shaped graph created by graph splitting.

p, with parameters d, m, and c, we minimize the time with respect to p, and find the optimal number of processors to be $\sqrt{\frac{cd}{2m}}$. Note that this is a pessimistic approach, and assumes that only one message can be sent in the system at one time.

Figure 20 shows execution times for processing two contours, each consisting of 500 contour points. Execution times are shown for $d = 20, 40, 60, 80, 100$. The best performance is obtained when $d = 80$, since requiring that each processor calculate at least 80 nodes on each diagonal achieves the best balance between minimizing message passing and maximizing parallelism.

7.3 Storage Optimizations

In computing a shortest cost path in the graph, we must store the costs of the nodes and the incoming edges for each node. As mentioned previously, we iterate across all diagonals in the graph. Since edges in the graph are directed to the right and below, a diagonal at distance k only needs cost information from the diagonal at distance $k - 1$, as shown in Figure 21. Thus, we can limit our storage to the cost of a single diagonal and the preceeding diagonal. Furthermore, we can deallocate memory used by preceeding diagonals as their information becomes no longer needed. This storage technique is significant in computing large data sets. For example, if we store costs as 4 byte floating points, and have a 500×500 subgraph, $1,000,000$ bytes would be needed to store the cost of all nodes in the graph. By storing only what we need, a single diagonal, we reduce storage to 500×4 or 2000 bytes.

For storing incoming edges to graph nodes, we cannot use the diagonal approach, since information about the whole graph is needed to recover the shortest path. Since the incoming edge can only have two possible values, "up" or "right", a single bit can be used to store each incoming edge.

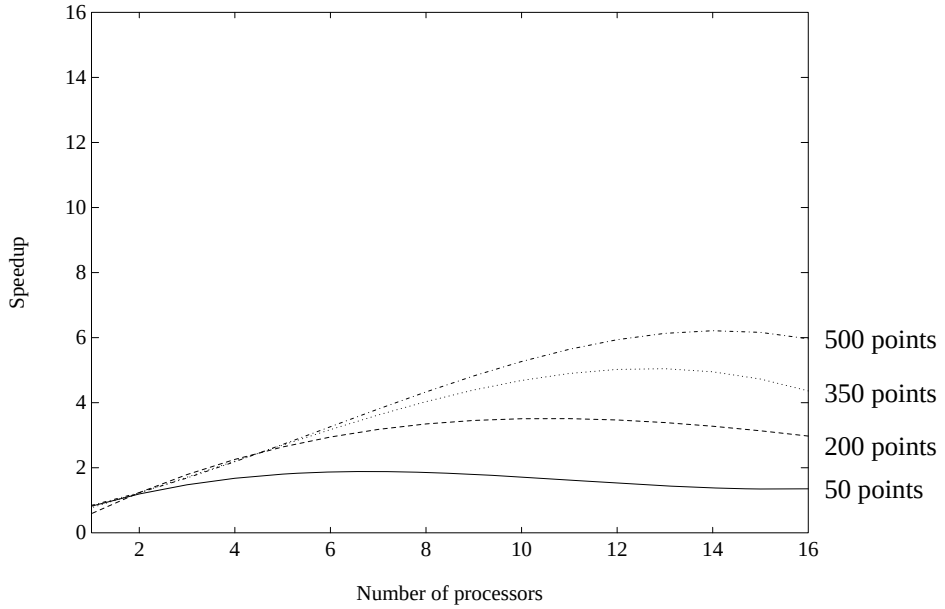


Figure 19: Speedup using parallel algorithm and size of subgraph for processor allocation.

8 Future Work and Optimizations

As discussed in the previous section, it is better to allocate processors based on the size of a subgraph, rather than the number of paths in that subgraph. There is one problem, namely that one of W_{hi} and W_{lo} might be very small, and contain sections where all $\mu(i)$ that are found in the subgraph must contain the same path. In this case, the remaining work contained in the subgraph might be somewhat larger than is indicated by the size of the subgraph. To account for this problem, whenever there is a cut edge in the subgraph, its edges are joined together (see Figure 22). This condition occurs whenever there are two adjacent d in the subgraph such that $|S_d| = 1$, and can be detected and corrected for when the paths are split and distributed with only a constant time penalty[JL92]. Johnson and Livadas [JL92] show that this consideration, along with processor allocation based on graph size, leads to a parallel algorithm with an optimal speedup. The complexity is shown to be $O(mn\log(m)/p)$, if $p = O(mn/((m+n)\log(mn/(m+n))))$, where p is the number of processors, and the toroidal graph is of dimension m by n.

The parallelism that our algorithm achieved was limited by the amount of communication required between processors. The algorithm can have very local communication characteristics, since the processors can be embedded in a line and perform nearest neighbor communications only. However, we had little control over processor allocation on the machine we used for the implementation. Since we use generic message passing for inter-processor communication, our algorithm could be implemented in any distributed or parallel

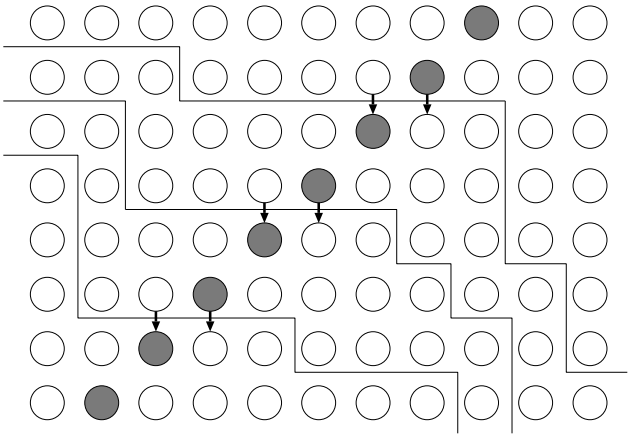


Figure 20: Messages required along a diagonal.

environment. In particular, our algorithm will be more efficient on a message passing multiprocessor, since we simulate message passing on the shared memory of the BBN. We plan to investigate the improvements possible with a message passing multiprocessor that gives us finer control over process placement.

9 Conclusions

By exploiting the structure of a directed graph, we have developed a parallel solution to finding a shortest path. The problem has applications in surface reconstruction, where we represent contours of a surface as graphs. Finding the shortest path in these graphs corresponds to finding a "best fit surface" over the contours. By parallelizing the solution, we have obtained a significant speedup to a computationally intensive problem.

By using a different criteria for processor allocation, we obtained better load balancing. Providing a lower bound on computation size as well as a better heuristic for the initial splitting of a graph can reduce message passing, and thus improve the speedup obtained.

References

[AFH81] Ehud Artzy, Gideon Frieder, and Gabor T. Herman. The theory, design, implementation, and evaluation of a three–dimensional surface detection algorithm. *Computer Graphics and Image Processing*, 15:1–24, 1981.

[AOUK87] Ken-Ichi Anjyo, Toshio Ochi, Yoshiaki Usami, and Yasumasa Kawashima. A practical method of constructing surfaces in three– dimensional digitized space. *The Visual Computer*, 3:4–12, 1987.

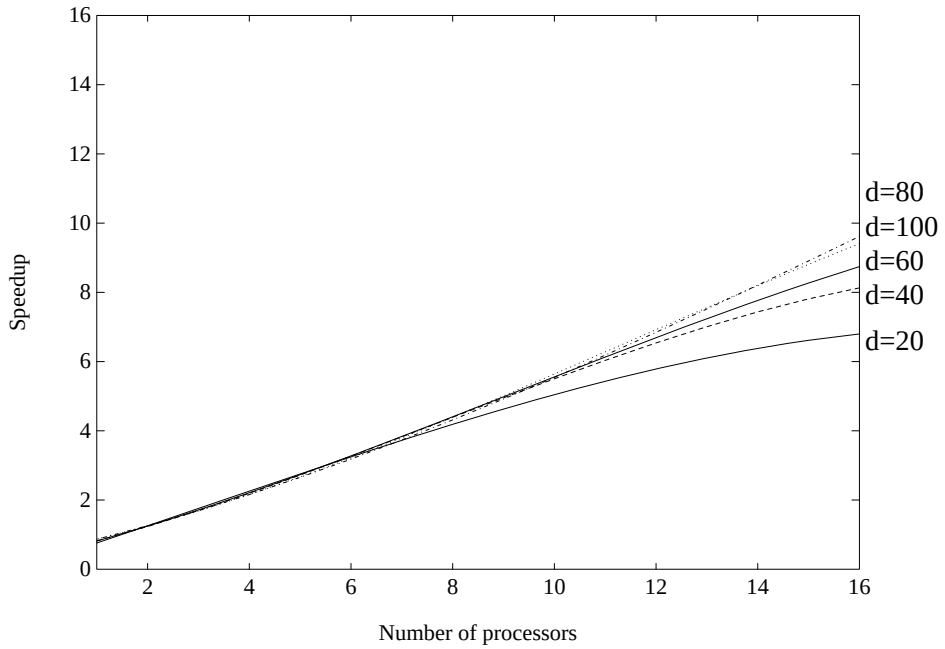


Figure 21: Speedup using parallel algorithm and requiring d nodes to be calculated along a diagonal.

[Blo88] J. Bloomenthal. Polygonalization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[CHRU85] Lih-Shyang Chen, Gabor T. Herman, R. Anthony Reynolds, and Jayaram K. Udupa. Surface shading in the cuberille environment. *IEEE Computer Graphics and Applications*, 5(12):33–43, Dec 1985.

[CS78] H. N. Christiansen and T. W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. *Computer Graphics*, 12(3):187–192, Aug 1978.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, Aug 1988.

[EPO91] A. B. Ekoule, F. C. Peyrin, and C. L. Odet. A triangulation algorithm from abritrary shaped muliple planar contours. *ACM Transactions on Graphics*, 10(2):182–199, Apr 1991.

[FKU77] H. Fuchs, Z. M. Kedem, and S.P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, Oct 1977.

[GD82] S. Ganapathy and T. G. Dennehy. A new general triangulation method for planar contours. *Computer Graphics*, 16(3):69–74, Jul 1982.

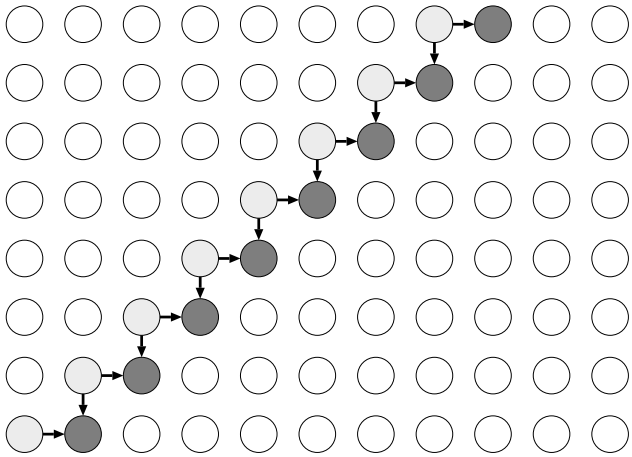


Figure 22: Storage of graph costs diagonal.

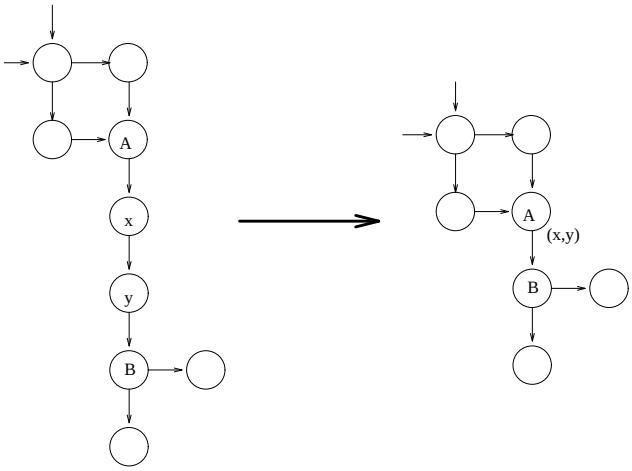


Figure 23: Removing cut edges from further processing.

[JL92] Theodore Johnson and Panos E. Livadas. Finding minimum cost acceptable paths in parallel. To appear in the Journal of Mathematical Imaging and Vision. Also available at cis.ufl.edu:cis/tech-reports/tr92-tr92-010.ps.Z

[Kep75] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM J. Res Develop.*, 19:2–11, 1975.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, Jul 1987.

[LCC89] Wei-Chung Lin, Shiuh-Yung Chen, and Chin-Tu Chen. A new surface interpolation technique for reconstructing 3d objects from serial cross–sections. *Computer Vision, Graphics, and Image Processing*, 48(1):124–143, Oct 1989.

[Liv88] Panos E. Livadas. A reconstruction of an unknown 3-d surface from a collection of its cross sections: An implementation. *International Journal of Computer Math*, 26:143–160, 1988.

[M+91] James V. Miller et al. Geometrically deformed models: A method for extracting closed geometric models from volume data. *Computer Graphics*, 25(4):217–226, Jul 1991.

[MSS92] David Meyers, Shelley Skinner, and Kenneth Sloan. Surfaces from contours. *ACM Transaction on Graphics*, 11(3):228–258, Jul 1992.

[TJL92] Sunjay Talele, Theodore Johnson, and Panos E. Livadas. Surface reconstruction in parallel. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, 1992.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structures for soft objects. *The Visual Computer*, 2:227–234, 1986.