# Edge intersection on the hypercube computer

Chandrasekhar Narayanaswami and William Randolph Franklin *

*Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY 12180, USA*

Communicated by S.G. Akl
Received 21 June 1990
Revised 5 December 1991

*Keywords:* Parallel algorithms, computational geometry, performance evaluation

## 1. Introduction

We describe a parallel algorithm for a hypercube computer for determining and reporting the intersections between line segments lying on the plane. This problem occurs in many geometric applications such as interference detection, visible surface determination, and set operations on polygons.

The sequential algorithm [1] is given first and its parallelization is presented next. Results from implementation are also provided. This complements both the optimal sequential algorithms of Chazelle and Edelsbrunner [2] and Mulmuley [5], which seem hard to parallelize, and the parallel algorithm of Goodrich [3], which seems difficult to implement. In the sequential case, experimental studies by Pullar [7] show that in a wide variety of practical situations, the algorithm chosen here for parallelization performs better than other theoretically superior algorithms.

## 2. The sequential algorithm

1. Partition the 2D region of interest into $G \times G$ uniform square cells.
2. For each edge, determine which cells it passes through and write ordered pairs (*cell number, edge number*). The left and bottom boundaries of a cell belong to it whereas the top and right boundaries do not. Appropriate adjustments are made while dealing with the peripheral (border) cells of the grid.
3. Sort the list of ordered pairs by the cell number and collect all the edges that pass through each cell.
4. For each cell, compare all the edges in it, pair by pair, to test for intersections. To determine the intersection between a pair of edges, the endpoints of each edge are first tested against the equation of the other edge to check whether an intersection occurs. Actual intersections are then determined if necessary. Intersections that fall outside the current cell are ignored. This handles the case of some pairs of edges occurring together in more than one cell. Intersections which fall on the cell boundaries are also handled correctly because no boundary belongs to more than one cell.

## 3. Parallelization of the sequential algorithm

The key idea in parallelizing the sequential algorithm is that computation of the cells through which the edges pass and of the intersections in the cells can each be done concurrently.

*Correspondence to:* Professor C. Narayanaswami, IBM Corporation, AWD, Graphics Architecture, 11400 Burnet Road, Austin, TX 78758-9260, USA. Email: chandra@innerdoor.austin.ibm.com, phone: +1 512 838 1105.

* Email: wrf@ecse.rpi.edu, phone: +1 518 276 6077.

The parallelization is discussed in the context of a hypercube machine which is an MIMD (Multiple-Instruction-Multiple-Data) coarse-grained, message-passing parallel computer with the processors (each with its local memory) connected in a hypercube network [4]. In addition to these processors, there is a *host* processor which is used for I/O and to control the node processors.

The second step of the sequential algorithm is parallelized by letting the host distribute the edges among the node processors which then compute the cells through which the edges pass. At the end of the above step, the cell-edge pairs of the edges handled by a processor are stored in its local memory.

Before the intersection computation within the cells can proceed, all the cell-edge pairs belonging to a cell have to be gathered in the processor responsible for it. To do this, every processor may have to communicate with every other processor. This requires an *all-to-all-personalized* (global) communication scheme. A simple (though suboptimal) way of doing such an information exchange is to reduce the operation to an all-to-all communication (not personalized) operation. This type of communication can be done by embedding a ring in the hypercube network (using a Gray code labeling for the processor) and by circulating the cell-edge pairs around it. In the first iteration every processor sends the cell-edge pairs it computed, to the next processor in the ring. In the next $n-2$ iterations, the processors send the message they received in the last iteration, to the next processor in the ring. Communication and computation are overlapped, i.e., while the message is on its way to the next processor, every processor processes the buffer it just sent out, to check whether any relevant cell-edge pairs are present in the buffer, in which case it stores them in its pair list. In order to speed up the retrieval step, the cell-edge pairs are sorted at the beginning so that the pairs for each processor are stored contiguously in the message buffers. After $n-1$ such iterations, every processor would have communicated with all the other processors and will therefore have all the pairs it needs for computing the intersections.

Next, the cells are evenly distributed among the node-processors and the computation within the cells is done concurrently. For reasons of simplicity of implementation of the communication mechanism, every processor gets a set of cells with consecutive labels. This implies that a processor gets a set of cells that occupy contigu-

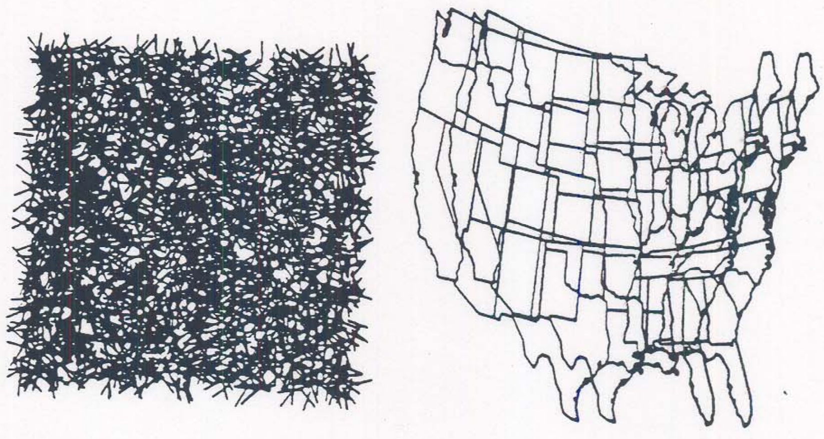
(a) Random edges (b) Overlay of USA maps

Fig. 1. Plots of data sets.

and by
In the
:ll-edge
in the
cessors
t itera-
ımuni-
. while
cessor,
st sent
e pairs
stores
ıp the
at the
or are
After
l have
'S and
ls for

mong
vithin
ns of
ınica-
et of
hat a
ıtigu-

ous regions in the scene. Finally, each node-processor sends its results back to the host-processor.

### 3.1. Implementation and results

The above algorithm was implemented on an Intel IPSC/1 hypercube machine with 32 processors and was tested extensively on three data bases with different geometric characteristics. One of them was a randomly generated data set. The second was a set of edges representing the state boundaries of the USA. For the third data set, the map of USA was shifted by 10% and was overlaid on it. Figure 1 shows a plot of two of the data sets.

Figures 2 and 3 show the times for the main stages of the algorithm for grid resolutions of 5 and 40, respectively. The dotted lines represent the timings for the case where 32 processors were used. A point $(x, y)$ on a dotted line in the graph indicates that the processor whose *id* was $x$ took $y$ seconds for that stage of the algorithm. The solid lines represent the timings for the case where 16 processors were used. A point $(x, y)$ on a solid line in the graph indicates that the processor whose *id* was $x/2$ took $y$ seconds for that stage of the algorithm.

The experiments and a closer study of the graphs reveal the following:

(1) The time to put the edges into the cells scaled inversely with the number of processors used. The time spent in casting the grid with 32 processors was about half the time spent when 16 processors were used. This was true, both for random and real edges.

(2) The work of putting the edges in the grid is distributed evenly among the processors. This is shown by the fairly horizontal shape of the graphs for the grid time. This is the case even though the lengths of the edges vary considerably. The reason for this was that though the independent edge lengths varied significantly, the sum of the lengths of the edges processed by each of the

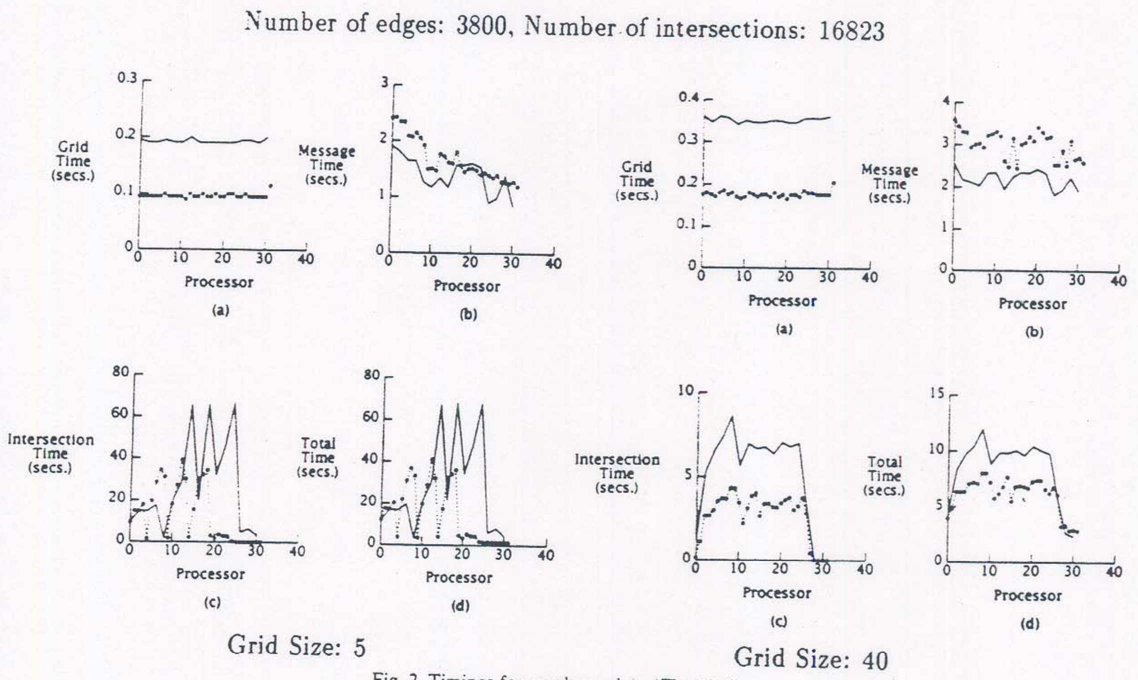

Fig. 2. Timings for random edges (Fig. 1(a)).

processors was quite uniform. If this is not the case, more sophisticated schemes for division of work among the processors have to be developed for this step. However, this does not appear to be necessary while processing real scenes.

(3) The intersection time scaled inversely with the number of processors when the data was random. The time taken by 16 processors was approximately twice the time taken by 32 processors. For certain grid sizes, when real data was used, the intersection time did not change much when 16 or 32 processors were used because a few contiguous cells had many edges at all grid resolutions. The naive work sub-division scheme allocated contiguous cells from a dense region to the same processor. Thus, the time taken by the processor responsible for this dense region dominated the computation. For random edges, the naive allocation of cells to the processors is not a problem. A solution to this problem is suggested in the following section.

(4) For both random edges and real edges the intersection time decreased as the number of cells was increased from 25 to 1600. This is due to the reduction in the total number of cell-edge pairs tested for intersections.

(5) The communication (message) time became significant (when compared with the total time) when the number of cells was increased and also when the number of edges was small. When the number of cells was high this was due to the rapid rise in the number of cell-edge pairs, whereas while dealing with a small number of edges, the actual intersection computation took very little time compared to the communication time. In the first case a coarser grid resolution will give better results. In the second case it is not worthwhile to use many processors to solve the problem.

The main observation is that, while in the sequential algorithm a broad range of finer grids usually produce faster results, in the parallel al-

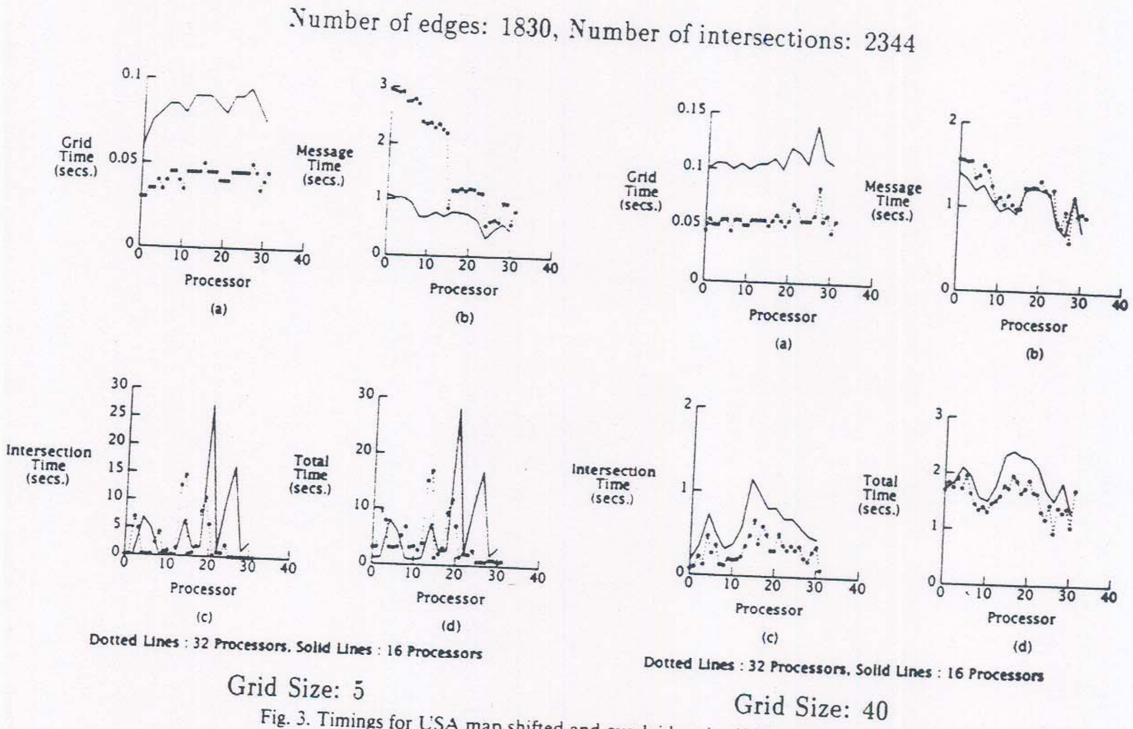

Fig. 3. Timings for USA map shifted and overlaid on itself (Fig. 1(b)).

gorithm the number of cells has to be reasonably small so that the computation to communication ratio is high.

(6) For a fixed grid resolution, the communication time went up as the number of processors increased because of the bigger size of the network, but was still less than half the total time.

### *3.2. Extensions for larger hypercubes*

Based on our experiments with 16 and 32 processors hypercubes, the following changes are recommended to improve the performance of the algorithm in the context of massive hypercubes.

#### *3.2.1. Communication-related improvements*

(1) In the current global communication scheme, all cell-edge pairs travel a distance of $(n-1)$, even though the diameter of the hypercube network is only $\log n$. While this is reasonable in the context of 16–32 node hypercubes, it is not practical for larger hypercubes. A better approach is to sort the cell-edge pairs based on the intended destination and use more sophisticated hypercube communication schemes such as the ones discussed by Johnsson and Ho in [4].

(2) If the preliminary task of putting the edges in the cells takes only a small fraction of the total time, as experiments with 16 to 32 processors indicate, it might be more efficient to let the host processor compute and distribute the cell-edge pairs to the nodes. The nodes then compute the intersection in the cells assigned to them. If this is done, only host-to-node and node-to-host communication are used and there is no need for all-to-all personalized communication.

(3) If a static assignment of cells to the processors is used, the location and orientation of the edges can be used to distribute the edges to the processors in the first step. This will reduce the amount of communication needed to organize the pairs. However, long edges which pass through many cells could require still global distribution.

#### *3.2.2. Load-balancing improvements*

(1) By assigning contiguous cells to processors in a cyclic fashion, it can be ensured that different processors get assigned to contiguous cells from a region where the density of the edges is high. This will help achieve better load-balancing while computing intersections.

(2) Instead of partitioning the work among the processors by distributing the cells equally while computing intersections, the work can be partitioned more evenly by distributing the cells among the processors such that the sum of the squares of the pairs in each cell over all the cells handled by every processor is approximately the same. This can be done by building a table containing the cumulative squares of the pairs in the cells. This procedure will take care of uneven distribution of edges.

## 4. Conclusion

Even with static, non-randomized, allocation of work to the processors, and a simple communication scheme, message-passing time was less than half the total time. In addition the slowest processor took less than 50% over the average processor time. This result, when combined with timing results of various sequential algorithms reported in [7], suggests that the parallel algorithm presented will compare favorably with efficient parallelizations of other practical sequential algorithms. Thus, using uniform grids is an effective method of finding, in parallel, all the intersections among large set of small edges on a hypercube with up to 32 processors, at least. Moreover, since newer hypercube machines have faster communication paths, our results are expected to improve.

As shown in [6], other related geometric problems are similarly parallelizable.

## Acknowledgment

This work was supported by NSF Presidential Young Investigator grant CCR-8351942. Partial support for this work was provided by the Directorate for Computer and Information Science and Engineering, NSF Grant No. CDA-8805910. We also used equipment at the Computer Science Department in Rensselaer Polytechnic Institute. Part of the research reported here was made possible through the support of the New Jersey Commission on Science and Technology and the Rutgers University CAIP Center's Industrial Members.

## References

[1] V. Akman, W.R. Franklin, M. Kankanhalli and C. Narayanaswami, Geometric computing and the uniform grid data structure, *Comput. Aided Design* **21** (7) (1989) 410–420.

[2] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, in: *Proc. 29th Ann. Symp. on Foundations of Computer Science*, White Plains (1988) 590–600.

[3] M.T. Goodrich, Intersecting line segments in parallel with an output-sensitive number of processors, *SIAM J. Comput.* **20** (4) (1991) 737–755.

[4] L.S. Johnsson and C.-T. Ho, Optimal broadcasting and personalized communication in hypercubes, *IEEE Trans. Comput.* **38** (9) (1989) 1249–1268.

[5] K. Mulmuley, A fast planar partition algorithm, 1, in: *Proc. 29th Symp. on Foundations on Computer Science*, White Plains (1988) 580–589.

[6] C. Narayanaswami, Parallel processing for geometric applications, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, December 1990.

[7] D. Pullar, Comparative study of algorithms for reporting geometrical intersections, In: *Proc. Internat. Symp. on Spatial Data Handling*, Zurich (1990) 66–75.