Solving Visibility Problems on MCCs of Smaller Size*

MI LU

Department of Electrical Engineering, Texas A&M University, College Station, Texas 77843

ABSTRACT

The paper presents mesh-connected computer (MCC) algorithms to solve the visibility problem for a set of disjoint simple objects such as line segments, circles, and simple polygons in the plane. Given a set of \( n \) such objects, the algorithms compute, on a \( \sqrt{n} \times \sqrt{n} \) MCC, a view of these objects in \( O(\sqrt{n}) \) time. Both parallel and perspective views are considered. Earlier published algorithms for computing the views are sequential and have \( O(n \log n) \) time complexity.

Consider the solutions to solve problems of size \( n \) on MCCs with \( p \) processors, where \( p < n \). An analysis is presented of the time performance and the limitations imposed by the computational and communication requirements, and the trade-offs between the size of the MCC and the time complexity that can be achieved.

1. INTRODUCTION

A mesh-connected computer (MCC) operates as a single instruction stream, multiple data stream (SIMD) computer in which processing elements (PEs) are arranged to be a two-dimensional, usually rectangular, array. Each PE can directly communicate with at most four adjacent neighbors, which are connected to it via horizontal or vertical communication paths. Each PE has a constant number of storage registers, and each can perform standard arithmetic or boolean operations in \( O(1) \) time. MCCs have been widely used in image processing and pattern analysis, and new MCC algorithms have been further designed for matrix computation and for solving graph problems and string-matching problems [1–6]. Future applications can be addressed to image understanding, text search, and knowledge bases.

An important and fundamental algorithmic problem in computer graphics is as follows. Given a set of objects in three-dimensional space, compute the

*This is a revised and expanded version of two earlier papers that appeared in the Proceedings of the 1988 International Conference on Parallel Processing and the Proceedings of the 1988 International Conference on Computer Graphics.
view from a specific direction or point. The main issue is to eliminate all parts of the objects that cannot be seen (i.e., those that lie behind other objects). It is a generalization of the hidden-line problem in which objects have straight-line edges. The problem also has numerous applications in the motion planning of robotics and VLSI layout, which have attracted considerable attention in recent years. In our consideration, we simplify the problem conceptually to two-dimensional space, as is often the case in other geometric problems, for example, the shortest-path problem. The solution for the two-dimensional problem is the main part of the three-dimensional solution and will indicate the direction for further research on the corresponding problems in three and higher-dimensional space.

In Section 2 I present the MCC algorithms for solving visibility problems, and in Section 3 the problem solution on MCCs of smaller size than the problem. Section 4 will give the concluding remarks.

2. SOLVING THE VISIBILITY PROBLEM ON AN MCC

The approach we use to solve the visibility problem is to divide and conquer. We recursively divide the mesh (short for mesh-connected computer) into two submeshes of equal size, called left and right (or upper and lower, respectively) submeshes. We solve the two subproblems on two submeshes in parallel and then combine the subsolutions in a binary tree fashion to obtain the final result. Elegant data movements must be designed for the merge step to allow the concurrent operations.

2.1. THE VISIBILITY PROBLEM

A view is a picture one sees when looking from a direction or a point. A view from a point is a perspective view. In this case the view consists of a circle on which the parts of the objects one can see from the given point are projected. A view from a direction is a parallel view. In this case, the view consists of a line on which the parts of the objects that are visible from that direction are projected. Perspective views correspond to our natural way of viewing from a place close to the object set, and parallel views correspond to the viewing from a place far from the object set.

A simple object is a bounded convex object with the following properties:

1. Any parallel or perspective view of it can be computed in constant time.
2. If the views of two simple objects overlap, then constant time suffices to decide which one of the two objects can be seen entirely.
3. Up to four common tangents of two simple objects can be computed in constant time.
Two objects that touch each other (i.e., objects whose boundaries overlap but do not cross) are by definition nonintersecting. Typical examples of simple objects are line segments, disks, and convex polygons with a bounded number of edges. A set of simple objects is shown in Figure 1. Figure 2 indicates the perspective view of them, and Figure 3 indicates the parallel view of them.

The visibility problem can be formally stated as follows: Given a set of $n$ disjoint simple objects, and a point or a direction, report in order the parts of the objects that are visible from the given point or the given direction. A point $p$ is visible from a point $q$ if the line segment $pq$ intersects no objects in the set.

The research work done on visibility problems is included in the following papers. For the case of a single polygon, El Gindy and Avis [7] presented $O(n)$ algorithms. Asanto et al. [8] gave an $O(n + h \log h)$ time algorithm for the case where the $h$ disjoint polygons are convex, and an $O(n \log h)$ time algorithm for the general problem. It has been proved that for $h$ disjoint polygons with $n$ edges, the optimal time complexity to find the visibility polygon using $O(n)$ space is bounded by $O(n + h \log n)$. In [9], Edelsbrunner et al. used $O(n)$ search time, $O(n^2 \log n)$ preprocessing time, and $O(n^2 \log n)$ space to solve the visibility polygon problem. The visibility graph of disjoint polygons with $n$ edges can be found by solving the visibility problem from each vertex of those polygons. This problem was solved recently by Welzl [10] and Asano et al. [8] independently in $O(n^2)$ time. The shortest path between two points in the
plane with polygonal obstacles can be computed by applying Dijkstra's algorithm to the visibility graph of the obstacles. This problem is of current interest because it is an instance of a general class of important problems in robotics known as collision avoidance problems (see, for example, Lozano-Perez and Wesley [11]).

2.2. COMPUTING THE PARALLEL VIEW

A parallel view of a set of objects consists of a partition of a line. Each part of the line corresponds to an object in the set (from the direction of view). The lowest part of the objects in the interval \((k, k+1)\) is visible, and we want to find the lowest part in all intervals, i.e., the lower envelope of the set of objects. (See Figure 4.)

To each part of the line we assign the index of the corresponding object. If the part corresponds to a place where one can look through the set we assign NULL to it. It is possible that different parts of the line correspond to the same object and hence are assigned the same index (see, for example, object 2 in Figure 4). A partition point corresponds to the leftmost point of an object
Direction of view

Fig. 3. A parallel view of a set of objects.

(Figure 5, A) or the rightmost point of an object (Figure 5, B), with respect to the direction of view. (A part of the line might consist of one point if it corresponds to a line segment in the direction of view. In this case we treat it as a double partition point). It can be proved that the parallel view of a set of \( n \) objects from a fixed direction consists of at most \( 2n + 1 \) parts and at most \( 2n \) partition points [9].

For computing the view of a set of objects from a fixed direction we will use the divide-and-conquer technique. Divide the set \( S \) of \( n \) objects into subsets \( A \) and \( B \) containing approximately equal numbers of objects. Let the partition points of a view of \( A \) be \( \{a_0, a_1, \ldots, a_k\}, \ k \leq 2n - 1 \), and the parts of the view of \( A \) be \( \{a_0a_1a_2a_3a_k\} \). Similarly, let the partition points of a view of \( B \) be \( \{b_0, b_1, \ldots, b_k\}, \ k \geq 2n - 1 \), and the parts of the view of \( B \) be \( \{b_0b_1b_2b_3b_k\} \). A part of the view of \( A, a_ia_{i+1} \), is a part of the view of \( S = A \cup B \) iff projecting \( a_ia_{i+1} \) to the view does not cross any other objects, that is, no part of the view of \( B \) falls (even partially) in the interval \( (a_i, a_{i+1}) \),
or part $b_j b_{j+1}$ falls in the interval but $a_i a_{i+1}$ is "closer" to the observer than $b_j b_{j+1}$. We can compute the view of $S = A \cup B$ by checking at each partition point of both views whether or not this point is also a partition point of the total view. As a result of the definition of a simple object, this checking can be done in constant time.

Recursively partition the problem into two equal-sized subproblems, compute the views of the two subsets simultaneously, and merge the results. The two phases involved in the merge are as follows:

(i) Each partition point $a_i$ in the view of $A$ finds the part $b_j b_{j+1}$ such that $a_i$ falls in the interval $b_j b_{j+1}$, and vice versa. (Figure 6 is a reference.)

(ii) Decide whether $b_j b_{j+1}$ prevents the part $a_i a_{i+1}$ from being visible, and vice versa.

---

Fig. 4.

Fig. 5. Two partition points: (A) at the leftmost point of an object and (B) at the rightmost point of an object.
2.3. **MCC IMPLEMENTATION**

In an MCC consisting of $N$ PEs, each PE has a distinct index $j$, $j = 0, 1, \ldots, N - 1$. We assume $N = 2^k$, with $k$ being an integer, and represent $j$ in binary code $(j_{k-1}j_{k-2} \cdots j_1j_0)$. The PEs can be ordered in row major (see
Figure 7a) or shuffled row major (see Figure 7b). The shuffled row major is obtained by shuffling the binary representation of the indices of the PEs ordered in row major.

The common data-routing operations of sorting and random access read/write (RAW/RAR) can be performed on a mesh-connected computer [1, 2].

**Sorting**

Distribute $N$ data items on a mesh, with one item per PE. The sorting problem is to move the $j$th smallest data item to the processor indexed by $j$, for all $j = 0, 1, \ldots, N - 1$. The sorting of $N$ data items can be completed in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh [1].

**Random Access Write / Read (RAW / RAR)**

PE($i$) specifies an ID, $f(i)$, of a PE from which it desires to obtain a record or to which it has a record to transmit, where $f: \{0, \ldots, N - 1\} \rightarrow \{0, \ldots, N - 1\}$ may be a many-to-one function. It has been shown that both RAW and RAR can be accomplished in $O(\sqrt{n})$ steps [2].

Distribute the partition points of a view on an MCC, one point per PE. Since there are at most $2n$ partition points in a view of $n$ objects, $2\sqrt{n} \times \sqrt{n}$ or $\sqrt{n} \times 2\sqrt{n}$ PEs are sufficient. The PE containing partition point $a_i$ maintains the part $a_i, a_{i+1}$.

Let a *submesh* of size $2^r$, for some $0 \leq r \leq k$, consist of the set of PEs whose indices agree on the $k - r$ most significant bits (MSBs) of their binary representation. Submeshes $\alpha$ and $\beta$ of size $2^r$ are said to be *adjacent* if the
indices of a PE in submesh \( \alpha \) and a PE in submesh \( \beta \) agree on the first \((k - r - 1)\) MSBs but differ in the \((r + 1)\)th LSB.

The partition of the problem into subproblems in computing the parallel view is now mapped to the partition of the mesh into submeshes. Let submesh \( \alpha \) contain the view of subset \( A \in S \) and submesh \( \beta \) contain the view of subset \( B \in S \). The size of the two subsets can be recursively reduced by half, and so can the size of the corresponding submesh. The merging of the views of subsets \( A \) and \( B \) is performed on submeshes \( \alpha \) and \( \beta \). If the sizes of the submeshes \( \alpha \) and \( \beta \) are \( 2^r \) each, then the merging involves \( 2^{r+1} \) PEs. In a binary tree fashion, the merging is performed recursively. That is, all the merges at a level of the computation tree are performed in parallel. In an iteration, each two adjacent submeshes are “sewn” together to obtain the solution of the problem whose size is twice that of the problem solved in each submesh. The “sewing” of the submeshes alternates between horizontal and vertical steps, beginning with a horizontal step. In a horizontal sewing step, each submesh is merged with the submesh immediately to the right or left of it, and in a vertical sewing step, with the submesh immediately below or above it (see Figure 8).

To perform phase (i) of the merging operation described in Section 2.2, each partition point \( a_i \) in the view of \( A \) must find the part \( \overline{b_i b_{i+1}} \) such that \( a_i \)
falls in the interval $b_jb_{j+1}$ and vice versa. To allow concurrency in the data movement, we adopt the following technique.

Sort all the partition points in the views of $A$ and $B$, respectively, and concurrently by $x$-coordinates in nondecreasing order. Denote the rank of a point, say $p$, in the sorted list as $\text{local\_rank}(p)$. Then sort all the partition points in $A$ and $B$ together and denote the rank of point $p$ in the sorted list as $\text{global\_rank}(p)$. We have

$$\text{global\_rank}(a_i) - \text{local\_rank}(a_i) - 1 = \text{local\_rank}(b_j),$$

i.e., within constant steps of computing, each partition point in the view of $A$ can find points $b_j$ in the view of $B$ such that $a_i$ falls in the interval $b_jb_{j+1}$. For example, $\text{global\_rank}(a_4) = 10$, $\text{local\_rank}(a_4) = 4$ in Figure 6, and

$$\text{local\_rank}(b_5) = \text{global\_rank}(a_4) - \text{local\_rank}(a_4) - 1 - 10 - 1 - 5,$$

i.e., $a_4$ falls in the interval $b_5b_6$.

To complete phase (ii) in the merging operation described previously, we need a transformation of the coordinate system. Let the angle of the observing direction be $\alpha$. Rotate the coordinate axes by $\alpha$ to obtain the new coordinate system. The point with the coordinates $(x', y')$ under the old axis system will have coordinates $(x, y)$ under the new axis system such that

$$x' = x \cos \alpha - y \sin \alpha,$$

$$y' = x \sin \alpha - y \cos \alpha.$$

Then the view of a single object is just the horizontal line segment with the leftmost and rightmost points of the object as its endpoints. The projection of
the object in subset $A$, say $i$, will cross the object in subset $B$, say $j$, iff $y_i \geq y_j$, where $y_i$ is the $y$ coordinate of the leftmost or rightmost point of object $i$ and $y_j$ is the $y$ coordinate of the leftmost or rightmost point of object $j$. The MCC algorithm will be given in Algorithm \texttt{PARALLEL VIEW}.

The record maintained in each PE includes the following field:

\textit{VIEW 1: Record}

\begin{itemize}
  \item $i$ /* index of the PE; can also be used as index of the object */
  \item $x, y$ /* coordinates of the leftmost or rightmost point of the object */
  \item $see$ /* the index of the part in the view, NULL if no object is visible */
  \item $local\_rank$ /* indicating in $r$th iteration the rank of the partition point in the view of $2^{r-1}$ objects */
  \item $global\_rank$ /* indicating in $r$th iteration the rank of the partition point in the view of $2^r$ objects */
  \item $base$ /* recording in $r$th iteration of the $[(\log n) + 1 - r] \text{ MSBs}$ of the PE index */
  \item $bias$ /* temporary variable to record the $local\_rank$ of a partition point in the view to be merged */
\end{itemize}

\texttt{Algorithm \texttt{PARALLEL VIEW}}

1. Distribute the $n$ objects on the MCC, so that each consecutive two PEs contain the same object, say $i$.
2. $\text{PE}(2k)$ finds the leftmost point of the object it contains, and $\text{PE}(2k+1)$ finds the rightmost point of the object it contains, for $k = 0, 1, \ldots, n - 1$. /* initialize the partition points*/
   Record the coordinates of the points found as $(x, y)$.
   $\text{PE}(2k)$ sets $see = i$. $\text{PE}(2k + 1)$ sets $see = \text{NULL}$.
   \texttt{for $r := 1$ to $\log n$ do the following:}
3. $base = (\log n + 1 - r) \text{ MSBs of } i$. /* represented by $b_k \cdots b_1 b_0$ */
4. Sort the partition points in nondecreasing order by $x$, on the $2^r$ submesh.
   Find the $global\_rank$ of each point.
5. Sort the partition points in nondecreasing order by $x$, on the $2^{r-1}$ submesh. Find the local_rank of each point.

6. Each PE computes

$$bias = global\_rank - local\_rank - 1$$

and

$$addr = base(\overline{b_0}) \circ bias,$$

where $base(\overline{b_0})$ is obtained from $base$ with the LSB complemented, and $\circ$ is a concatenation operator.

7. Each PE performs a RAR from PE($addr$) to get $y_{addr}$ and $see_{addr}$.

8. if $((y > y_{addr}) \land (see_{addr} = \neg \text{NULL})) \lor ((y \leq y_{addr}) \land (see = \text{NULL}))$

$$see = see_{addr}.$$

end ( /* of Algorithm \textsc{parallel\_view} */ )

2.4. TIME COMPLEXITY

Assuming that the merge of the two subproblems whose sizes sum to $n$ needs time $M(n)$, the total time needed, $T(n)$, is given by the recurrence

$$T(n) = T(n/2) + M(n).$$

We show below the analysis of the bound of the merge time $M(n)$ and the derivation of the time complexity of our algorithm \textsc{parallel\_view}.

Step 1 takes $O(\sqrt{n})$ time. Step 2 needs only constant time, because the objects are simple. Constant time is sufficient for the computation involved in step 3. In step 4, $O(\sqrt{2^r})$ time is required for iteration $r$. Step 5 requires $O(\sqrt{2^{r-1}})$ time only. Step 6 involves only constant time of computing. Step 7 involves RAR, which needs $O(\sqrt{2^r})$ time in iteration $r$. Step 8 takes constant time. Thus, the time required in iteration $r$ is bounded by $O(\sqrt{2^r})$, and the total time needed in all the iterations is

$$T(n) = \sqrt{2} + \sqrt{2^2} + \cdots + \sqrt{2^{\log n}} = O(\sqrt{n}).$$

We have considered the view from a given direction, i.e., a parallel view. However, there is another interesting type of view, called a perspective view,
which consists of the portions of the objects that are visible from a given point. The problem of finding the perspective view from an arbitrary point is discussed in the following section.

2.5. **COMPUTING THE PERSPECTIVE VIEW**

Let \( S \) be a set of \( n \) simple objects, and let \( q \) be an arbitrary query point. We want to find the parts of the objects in \( S \) that are visible from \( q \), that is, find the *perspective view*.

A perspective view of a set of objects consists of a partition of a circle. Each circle segment corresponds to the part of an object that one can see from the fixed point or to the place where one can look through the set. It can be verified that a perspective view contains at most \( 2n \) partition points and hence at most \( 2n + 1 \) parts \[9\].

Consider a polar coordinate system with the point \( q \) as the origin and the positive \( y \) axis as the reference. Denote the polar angle of a point \( p_i \) by \( \theta(p_i) \), where the polar angle increases counterclockwise around \( q \) (see Figure 7). The polar coordinates of a point can be represented as \((r, \theta)\). The leftmost point \( p_l \) or the rightmost point \( p_r \) of an object is the tangency point such that the line emanating from \( q \) is tangent to the object at it and \( \theta(p_l) > \theta(p_r) \).

Computing the perspective view is similar to computing the parallel view. A similar divide-and-conquer technique is adopted. If \( p_0, p_1, \ldots, p_k, k \leq 2n - 1 \), denote the partition points, then \( p_i p_{i+1} \) indicates part of the view. We will distribute the partition points on the mesh with each PE containing one partition point and maintaining the record of the part \( p_i p_{i+1} \).

The problem of finding the perspective view can be decomposed into the following two subproblems.

(i) Each partition point \( a_i \) in the view of \( A \) finds the part \( \overline{b_j b_{j+1}} \) such that \( \theta(b_j) \leq \theta(a_i) < \theta(b_{j+1}) \), and vice versa.

(ii) Decide the visible part of the objects in the interval \((p_i, p_{i+1})\).

The visibility can be examined by determining which is the nearest to \( q \) among those objects with the ray emanating from \( q \), extended from \( \theta(p_i) \) to \( \theta(p_{i+1}) \), passing through its interior. If we cut the plane along the ray emanating from \( q \) and spread it out according to the angular and radial orders, the spread-out view is similar to the one we discussed in the previous sections. Thus a similar approach can be adopted to complete phase (i) in the merging operation. For a partition point \( a_i \) in the perspective view of subset \( A, A \in S \), find the difference of its global rank and local rank. The result gives the local rank of partition point \( b_j \) in the prospective view of \( B, B \in S \), such that \( \theta(b_j) \leq \theta(a_i) \leq \theta(b_j + 1) \).
The transformation of the coordinate system needs not only a rotation but also a translation of the axes. Let \((x',y')\) be the coordinates before the rotation, \((x'',y'')\) the ones before the translation, and \((h,v)\) the position of the observer in the old coordinate system. We have

\[
x' = x \cos \alpha - y \sin \alpha,
\]
\[
y' = x \sin \alpha - y \cos \alpha
\]

and

\[
x'' = x' + h,
\]
\[
y'' = y' + v.
\]

In addition,

\[
\rho^2 = x^2 + y^2,
\]
\[
\tan \theta = x / y,
\]

will complete transforming the Cartesian system to a polar system. We consider below the algorithm \textsc{perspective view}, an MCC algorithm for computing a perspective view from a given point.

In Algorithm \textsc{perspective view}, the record \textsc{view} 2 maintained in each PE is similar to the record \textsc{view} 1 given in Section 2.2, but the field "\(x, y\)" is changed to "\(\rho, \theta\)." The data record maintained in each PE and the detailed algorithm \textsc{perspective view} are presented as follows.

\textsc{view} 2: Record

\[
i / * \text{index of the PE; can also be used as index of the object} */
\]
\[
\rho, \theta / * \text{polar coordinates of the leftmost or rightmost point of the object} */
\]
\[
\text{see} / * \text{the index of the part in the view; NULL if no object is visible} */
\]
\[
\text{local_rank} / * \text{indicating in } r \text{th iteration the rank of the partition point in the view of } 2^{r-1} \text{ objects} */
\]
\[
\text{global_rank} / * \text{indicating in } r \text{th iteration the rank of the partition point in the view of } 2^r \text{ objects} */
\]
SOLVING VISIBILITY PROBLEMS 177

base / * recording in r th iteration the \((\log n) + 1 - r\) MSBs of the PE index */

bias / * temporary variable to record the local_rank of a partition point in the view to be merged */

end ( / * of record */ )

Algorithm perspective view

1. Distribute the \(n\) objects on the MCC so that each consecutive two PEs contain the same object, say \(i\).
2. PE(2\(k\)) finds the leftmost point of the object it contains, and PE(2\(k + 1\)) finds the rightmost point of the object it contains, for \(k = 0, 1, \ldots, n - 1\).
   /* initialize the partition points */
   Record the coordinates of the points found as \((\rho, \theta)\).
   PE(2\(k\)) sets \(\text{see} = i\). PE(2\(k + 1\)) sets \(\text{see} = \text{NULL}\).

for \(r := 1\) to \(\log n\) do the following:
3. base = \((\log n + 1 - r)\) MSBs of \(i\). /* represented by \(b_k \cdots b_1 b_0\) */
4. Sort the partition points in nondecreasing order by \(\theta\) on the \(2^r\) submesh.
   Find the global_rank of each point.
5. Sort the partition points in nondecreasing order by \(\theta\) on the \(2^{r-1}\) submesh.
   Find the local_rank of each point.
6. Each PE computes

   \[
   \text{bias} = \text{global_rank} - \text{local_rank} - 1
   \]

   and

   \[
   \text{addr} = \text{base}(\overline{b_0}) \circ \text{bias},
   \]

   where \(\text{base}(\overline{b_0})\) is obtained from \(\text{base}\) with the LSB complemented, and \(\circ\) is a concatenation operation.

7. Each PE performs a RAR from PE(addr) to get \(\rho_{addr}\) and \(\text{see}_{addr}\).
8. if \(((\rho > \rho_{addr}) \land (\text{see}_{addr} = \neg \text{NULL})) \lor ((\rho \leq \rho_{addr}) \land (\text{see} = \text{NULL}))\)

   \[
   \text{see} = \text{see}_{addr}.
   \]

end ( / * of Algorithm perspective view */ )

Similarly, step 1 of Algorithm perspective view takes \(O(\sqrt{n})\) time. Steps 4, 5, and 7 each require \(O(\sqrt{2^r})\) time in iteration \(r\). Steps 2, 3, 6, and 8 each need constant time only. The total time needed to find a perspective view of a set of \(n\) simple objects is thus bounded by \(O(\sqrt{n})\).
The visibility problem from a point for a set of \( h \) (not necessarily convex) disjoint polygons with \( n \) edges in total can be solved with the same time complexity by applying the above algorithms to the edges of those polygons. We first compute the visible portion of the boundary of each polygon from the point. The result is a sequence of edges from each polygon. Then the sequence can be decomposed where the visible parts in the view are found.

A visibility graph of \( n \) arbitrary oriented segments is a graph whose vertices are endpoints of those segments and whose edges are the straight-line segments joining vertices that are visible from each other. This graph can be constructed by solving the visibility problem from each vertex for the given segments. As an application of this result, the shortest path between two points in the plane with polygonal obstacles having \( n \) edges can be solved by Dijkstra's algorithm, provided that the visibility graph is available.

3. SOLVING A PROBLEM ON AN MCC OF SMALLER SIZE THAN THE PROBLEM

The results described in the previous sections were obtained using the unbounded model of parallel computation, i.e., we imposed no limit on the number of processors used by our algorithms. My coworkers and I have discussed in several papers [12-15] the methods for solving some geometrical problems using MCCs of the same size as the problem, that is, problems for which the number of elements to be processed is equal to the number of processors in an MCC. Obviously, in any practical situation we will be required to handle varying problem sizes with a fixed number of processors.

The situation where the size of the MCC we have is smaller than the problem size occurs very often. However, nothing that I am aware of has been written on this subject, although a few researchers understand that the solution to this problem can be addressed. In this section, the results of solving problems of size \( n \) on MCCs with \( p \) processors, where \( p < n \), are discussed, and, more important, the time complexity of the solution is analyzed, and the limitations imposed by the computational and communication requirements are explored.

3.1. BASIC APPROACH

If a problem has \( n \) pieces of data initially distributed one per PE in a mesh of size \( n \), we now consider what happens when we try to solve the problem on a mesh of size \( p \), \( 1 \leq p \leq n \), where each PE is initially given \( n/p \) pieces of data. This requires that the processors have sufficient memory to handle the largest problem size that will be encountered. A processor with its local storage is referred to as a node.
The basic approach to solve problems using MCCs of smaller size is to combine parallel and sequential processing on the MCCs. Previously developed MCC algorithms are used for internode processing, whereas sequential algorithms are used for intranode processing.

The algorithms include two phases:

(i) Each PE oper:

As described above, \( n \) pieces of input data are distributed on a \( \sqrt{p} \times \sqrt{p} \) MCC, with \( n/p \) pieces of data per PE. Two sorting orders are considered, consecutive order and cyclic order. In consecutive order, \( n/p \) successive elements of the sorted sequence are stored in each node, with successive sets of \( n/p \) elements being stored in nodes in order of increasing node address (see Figure 10a). In cyclic order, node \( i \) stores the elements in the set \( \{ j | i = \text{rank}(j) \mod p \} \) such that \( i = j \mod p \). We describe below the details of sorting into consecutive order (see Figure 10b). Rearrangement of consecutive to cyclic storage order (or vice versa) can be carried out in time \( O(n/p + p) \) by pipelining the data transfers.

The sorting is carried out first by intranode processing and then by internode processing. A local sort is performed initially. Batcher's [16] odd-even merge is then mapped onto the MCC. When performing the local sort, each PE sorts the data it contains independently. This can be done by using sequential algorithms within \( O((n/p) \log(n/p)) \) time. After that, the \( n/p \) elements in a node are part of a sorted sequence.

The remaining task is to merge the given sorted sequences on the mesh. The questions we need to answer are whether Batcher's odd-even merge can be mapped on to the mesh of smaller size, how to operate on the data items with more than one item sorted in a PE, and what kind of time complexity can be expected.

Recall that Batcher's odd-even merge of two sorted sequences \( \{u_i\} \) and \( \{v_i\} \) is performed in the following stages recursively.

(i) Merge the "odd sequences" \( \{u_1, u_3, u_5, \ldots, u_{2j+1}, \ldots\} \) and \( \{v_1, v_3, v_5, \ldots, v_{2j+1}, \ldots\} \) concurrently with the merging of the "even sequences" \( \{u_2, u_4, \ldots, u_{2j}, \ldots\} \) and \( \{v_2, v_4, \ldots, v_{2j}, \ldots\} \). Let \( \{s_1, s_2, \ldots\} \) and \( \{t_1, t_2, \ldots\} \) be the merged sequences of odds and evens, respectively.
(ii) Interleave the two merged sequences, \(\{s_1, s_2, \ldots\}\) and \(\{t_1, t_2, \ldots\}\).

(iii) Perform a single parallel comparison–interchange step.

In Thompson and Kung's MCC sorting algorithm [1], stage (i) is implemented by columns containing either all evens or all odds, unshuffling each row so that all evens are on the left half of the mesh and all odds are on the right half. It takes \(O(k)\) time to accomplish the unshuffle for a row of length \(k\). Odd–even merge is recursively invoked on each half independently. Interleaving in stage (ii) is implemented by shuffling each row on the mesh. \(O(k)\) time is needed to complete the shuffle for a row of length \(k\). Comparison–interchange in stage (iii) is implemented on pairs of adjacent processors, and constant time is sufficient. Thus the time complexity of merging is dependent on the length of the rows.
Consider the interleaving on the mesh of smaller size. It can be carried out by the same shuffling approach on each row using the triangular interchange pattern. However, we need to perform the interleaving first on nodes and then on elements. After interleaving on nodes, the contents of successive nodes alternate between \( s_i \)'s and \( t_i \)'s. Figure 11 illustrates the interleaving on nodes of a row using the triangular interchange pattern. It can be observed that \( k \) steps are needed to interleave a row with \( k \) nodes. In each step, \( O(n/p) \) time is required for operating on the \( n/p \) elements in a node. The time complexity for interleaving of nodes on a row of length \( k \) is thus \( O((n/p)k) \). Following the interleaving on nodes, an interleaving on elements is performed between
two adjacent nodes. After interleaving on elements, the content of the successive memory in each node alternates between \( s_i \) and \( t_i \) (Figure 11b). Obviously, the time needed to interleave the elements is \( O(n/p) \).

Batcher's odd–even merge can be applied on the mesh with \( n/p \) memory per PE in the following way. To merge two single columns, the lengths of the columns to be merged are halved recursively. Separating odds and evens can be completed by performing an unshuffle on a column. Unshuffling is the reverse operation of shuffling, and the operation on a column is symmetric to that on a row. The time needed for an unshuffle on a column of length \( j \) is \( O((n/p)j) \). The odd and even subsequences can be rearranged into two horizontal adjacent subcolumns, and the interleaving can be performed between each pair of horizontal adjacent processors in parallel. \( O(n/p) \) time is sufficient to complete the interleaving for any node containing \( n/p \) elements. Also, single comparison–interchange needs time no greater than \( O(n/p) \). Thus, the time needed for merging two single columns, each of length \( j \), is \( O((n/p)j) \). Letting \( T_M(i,j) \) denote the time needed for merging two submeshes each with \( j \) rows and \( k \) columns, the time needed to merge two single columns can be expressed as

\[
T_M(j, 1) = T_M\left(\frac{j}{2}, 1\right) + O\left(\frac{n}{p} j\right).
\]

(See Figure 12 for an example.)

We have not yet considered the merging of each half of the mesh. In fact, this is also implemented recursively using Batcher's odd–even merge. The number of columns is halved recursively here. The following is the algorithm:

1. Perform single interchanges on even rows.
2. Unshuffle on elements. This moves odds to odd columns and evens to even columns.

![Fig. 12.](image_url)
3. Unshuffle on nodes of each row. This moves odds to the left half of the mesh and evens to the right half of the mesh.
4. Recursively merge on each half.
5. Interleave on nodes.
6. Interleave on elements.

Steps 1 and 2 each require $O(n/p)$ time. Step 3 takes $O((n/p)k)$ time for a row with $k$ nodes. If we are merging two submeshes each with $j$ rows and $k$ columns and need time $T_M(j, k)$, then the time needed by step 4 is $T_M(j, k/2)$. Step 5 can be completed in $O((n/p)k)$ time, and step 6 needs $O(n/p)$ time. Thus we have

$$T_M(j, k) = T_M\left(j, \frac{k}{2}\right) + O\left(\frac{n}{p} k\right).$$

(2)

As an example, the last merge step on a mesh with $n = 32$, $p = 16$ is shown in Figure 13.

When sorting on a $\sqrt{p} \times \sqrt{p}$ mesh, we divide the mesh in two dimensions, to reduce the size of both $j$ and $k$ simultaneously. Let $T_S(j, k)$ denote the time needed for sorting the elements in a mesh of $j$ rows and $k$ columns. We have

$$T_S(j, k) = T_S\left(\frac{j}{2}, \frac{k}{2}\right) + T_M\left(\frac{j}{2}, \frac{k}{2}\right) + T_M\left(\frac{j}{2}, k\right)$$

(3)

and

$$T_S(1, 1) = O\left(\frac{n}{p} \log \frac{n}{p}\right).$$

(4)

Substituting (1), (2), and (4) into (3), we have

$$T_S(j, k) = O\left(\frac{n}{p} \log \frac{n}{p}\right) + O\left(\frac{n}{p} j\right) + O\left(\frac{n}{p} k\right),$$

hence

$$T_S(\sqrt{p}, \sqrt{p}) = O\left(\frac{n}{p} \log \frac{n}{p} + \frac{n}{p} \sqrt{p}\right).$$
Single interchangers on even rows. Time: $O(n/p)$

Unshuffle on elements. Move odds to odd column and evens to even column. Time: $O(n/p)$

Unshuffle on nodes of each row. Move odds to left half mesh and evens to right half mesh. Time: $O((n/p)\sqrt{p})$
Recursive merge on each half Time: $T_m(J, k/2)$

Interleave on nodes Time: $O((n/p)\sqrt{p})$

Interleave on elements Time: $O(n/p)$

Fig. 13. (Continued)
Single interchange on even rows. Time: $O(n/p)$

Comparison-interchange of adjacent elements (every "even" with the next "odd"). Time: $O(n/p)$

Fig. 13. (Continued)
3.3. RAR AND RAW ON MCCS OF SMALLER SIZE

Similar to algorithms for performing RAR and RAW on MCCs with constant memory [2], our RAR and RAW algorithms on MCCs with \( n/p \) memory are performed using the following well-defined operations.

(1) **SORT.** as described before.

(2) **RANK.** The rank of a selected record is the number of selected records in PEs with a smaller index. For example, if eight records contained in four PEs are \(((6,4),(2,2*),(6,6*),(3*,4*))\), where the asterisk denotes the selection of a record, their ranks are \(((−,−),(−,0),(−,1),(2,3))\).

(3) **CONCENTRATE.** Assume that a record \( i \) initially has the rank \( H(i) = r \).
A CONCENTRATE results in the record being moved to PE\((r/(n/p))\), specifically at memory location \( r \mod(n/p) \). For example, \(((A,−),(−,B),(−,C),(−,D))\) becomes \(((A,B),(C,D),(−,−),(−,−))\) following a CONCENTRATE.

(4) **DISTRIBUTE.** Initially record \( i \) has a destination \( H(i) \), and \( H(i) < H(i+1) \). The purpose of a DISTRIBUTE is to route the record to PE\((H(i)/(n/p))\), specifically at memory location \( r \mod(n/p) \). For example, \(((A,B),(C,−),(−,−),(−,−))\) with \( H(0)=1, H(1)=5, \) and \( H(2)=6 \) becomes \(((−,A),(−,−),(−,B),(C,−))\) following a DISTRIBUTE.

(5) **GENERALIZE.** If a record \( i \) has a field \( H(i) \), and \( H(i) < H(i+1) \), GENERALIZE copies the record into PEs PE\((H(i−1)/(n/p))\) through PE\((H(i)/(n/p))\), starting from memory location \( H(i−1) \mod(n/p+1) \) at PE\((H(i−1)/(n/p))\) and ending at memory location \( H(i) \mod(n/p) \) at PE\((H(i)/(n/p))\). [Assume \( H(−1)=0. \) For example, record \(((A,B),(C,−),(−,−),(−,−))\) with \( H(0)=1, H(1)=5, \) and \( H(2)=6 \) becomes \(((A,A),(B,B),(B,B),(B,B),(C,−))\) following a GENERALIZE.

Denote the record \( H(i) \) referred to in operations 3, 4, and 5 as the *key record*. Following the example of RAR on a constant memory mesh [2], the RAR algorithm on an \( n/p \) memory mesh is composed of a set of previously described simple operations, such as SORT, RANK, CONCENTRATE, DISTRIBUTE, and GENERALIZE. We will describe below the data movement in CONCENTRATE; similar data movement will arise in other operations.

Let \( n = 2^k \) and \( p = 2^s \). The current address of a record and its destination can be expressed in binary representation \( b_{s−1},...,b_0 \). The \( (k−s) \) LSBs \( b_{k−s−1},...,b_0 \) indicate the intranode memory location. The \( s \) MSBs \( b_{s−1},...,b_{k−s} \) indicate the index of the node. We move a record to its destination in \( s \) steps. In step \( j \), the record is moved to an address that agrees with the destination of the record in the \( j \) LSBs. We can find that the first \( k−s \) steps involve only the intranode processing. Data are moved between memory locations of a PE. The number of data items in each PE is \( n/p \). The
total time needed to move the data to the location, which agrees with the destination of the data in $(k - s)$ LSBs, is

$$O\left(\frac{n}{p}(k - s)\right) = O\left(\frac{n}{p} \log \frac{n}{p}\right).$$

The last $s$ iterations involve internode processing. Each iteration follows the steps of \texttt{CONCENTRATE} explained earlier. The difference is that $n/p$ data items are transferred. In iteration $j$, $j = k - s, \ldots, k - 1$, the same routine will perform in a $2^j$-block on mesh independently and concurrently. Recall that a $2^j$-block consists of $2^j$ PEs whose indices differ only in the $j$ LSBs. The time needed for intranode processing in each kind of operation is described below.

\textbf{RANK.} The rank of a record in a $2^j$-block of PEs is the number of the selected records in that $2^j$-block with a lower address. Divide a $2^j$-block into two halves each containing $2^{j-1}$ consecutive PEs. Let $S(i)$ be the total number of selected records in the home $2^{j-1}$-block. Then, the rank of a selected record in a $2^{j-1}$-block is $H(i)$ if $i_{j-1} = 0$ (left $2^{j-1}$-block of a $2^j$-block) and $H(i) + S(i_{j-1})$ if $i_{j-1} = 1$ (right $2^{j-1}$-block of a $2^j$-block). The data broadcast in a $2^j$-block of $2^j$ nodes require $O(\sqrt{2^j})$ time, and each node needs $O(n/p)$ to perform the addition for all of $n/p$ data it contains. Thus, $n/p + \sqrt{2^j}$ time is sufficient in iteration $j$.

\textbf{CONCENTRATE.} For an integer $i$, let $i_b$ denote bit $b$ of the binary representation of $i$ and $i^{(b)}$ denote the number whose binary representation differs from that of $i$ in bit $b$ only. Concentration is carried out by first moving all records in each $2^j$-block of PEs, such that the PE index and $H(i)$ agree in bits $k - s$. The next routing ensures that PE indices and $H(i)$ agree in bits $k - s$ and $k - s + 1$; and so on until records have reached their destination PEs. To move to PE($[H(i)/(n/p)]$), interchange is initiated between PE($i$) and PE($i^{(b)}$) in iteration $b$. To finish all exchanges inside a node takes time no greater than $n/2p$. Internode exchanges can be performed in parallel in a $2^j$-block. $\sqrt{2^j}$ steps are required to reach the desired node, and in each step $n/p$ time is required for waiting in queue to access the desired memory. Thus, in iteration $j$, \texttt{CONCENTRATE} needs $O((n/p)\sqrt{2^j})$ time for $k - s \leq j < k$.

\textbf{DISTRIBUTE.} It is easy to see that a distribution is simply the inverse of a concentration. The time complexity of procedure \texttt{DISTRIBUTE} is clearly the same as that of procedure \texttt{CONCENTRATE}.

\textbf{GENERALIZE.} Again, a $2^j$-block may be thought of as being composed of two $2^{j-1}$-blocks. One of these contains all PEs in the $2^{j-1}$-block with bit $j - 1$ of node index equal to 1 and the other all PEs with bit $j - 1$ of node index equal to 0. To carry out a \texttt{GENERALIZE} in a $2^j$-block, we first send a package of $n/p$ records from each PE in a $2^{j-1}$-block to the corresponding PE in the other
2^{i-1}-block. The newly arrived records are kept in registers labeled $G'[0,\ldots,n/p-1]$, other than $G[i]$. $G(i)$ is not to be replicated in the home $2^{i-1}$-block if $H(i)<i-i_{j-2;0}$, and $G'(i)$ is not to be replicated in the home $2^{j-1}$-block if $H'(i)<i-i_{j-2;0}$. The number of unit-routes is the same as that used by RANK, and the time needed in iteration $j$ is $(n/p)\sqrt{2^j}$ for $k-s\leq j<k$.

In other words, the time complexity for the intranode processing in the $j$th iteration is $O((n/p)2^{j/2})$ in the above operations. The total time needed for all the iterations is

$$O\left(\frac{n}{p} [1+\cdots+2^{j/2}]\right) = O\left(\frac{n}{\sqrt{p}}\right).$$

The total time needed for intranode processing and internode processing is

$$T = O\left(\frac{n}{\sqrt{p}}\right) + O\left(\frac{n}{p} \log \frac{n}{p}\right).$$

The RAW problem is similar to the RAR problem and has the same time complexity as RAW. Since the existing MCC algorithms are based on sorting, RAR, and RAW techniques, the results of sorting, RAR, and RAW on MCCs of smaller size provide the solutions to various problems on MCCs of smaller size.

### 3.4. LOWER BOUND TIME AND OPTIMAL SIZE

We have presented the parallel algorithms for solving problems on the mesh of unbounded model and on the mesh of smaller size. The new questions brought to our attention are: What is the trade-off between the time complexity and the number of processors? Is it true that the more processors we use, the less time is required?

Given $n$ elements distributed on $p$ processors with $n/p$ elements per PE, where $p<n$, the relationship of the computation time $T$ versus the number of processors $\sqrt{p}$ is shown in Figure 14. The line $T = \sqrt{p}$ indicates the influence of the diameter of the mesh. Two curves, $T = (n/p)\log n$ and $T = n/\sqrt{p}$, are also given that indicate the time needed for internode processing and for intranode processing, respectively.

Since moving a data item from, say, the upper left corner of a mesh of size $\sqrt{p} \times \sqrt{p}$ to the lower right corner needs time no less than $2\sqrt{p}$, $T$ should be
greater than $\sqrt{p}$. Thus our working area is above the line $T = \sqrt{p}$ in Figure 14. In the meantime, we can find that $n/\sqrt{p} > (n/p)\log n$ when $p > \log^2 n$. Our working area in Figure 14 is therefore above the line $T = \sqrt{p}$ and the curve $T = n/\sqrt{p}$. Observing the common shaded area, we note that the minimal $T$ is given at the point $(n, \sqrt{n})$. It means that if $n$ processors are provided, we can obtain the optimal time complexity, which is $O(\sqrt{n})$.

All the sequential algorithms to solve the previous problems use only one processor and have an optimal time complexity of $O(n \log n)$. With $p$ processors available, the desired time performance is $O((n/p)\log n)$. However, it
can be achieved only when a very small number of processors are used, as indicated by the area of $p \leq \log^2 n$ in Figure 14. We have $n / \sqrt{p} > (n / p) \log n$ and $T > O(\sqrt{n})$ in that area, and the utility of each processor is 100%.

When $p > \log^2 n$, our working area is bounded by $T = n / \sqrt{n}$. The $(n / p) \log n$ time performance can no longer be achieved, and the processors cannot be 100% utilized. This is because of the bandwidth limitation. The machine model we used, a mesh-connect computer, is a limited-connectivity processor network. We avoided the complicated interconnections in the machine building at the cost of some loss in time performance.

When more than $n$ processors are given, where $n$ is the size of the problem as indicated by $T > n$ area in Figure 14, we find, surprisingly, that it is not the case that the more processors we use the less computation time is required, but $T$ increases as $p$ increases. This demonstrates that to put more than $n$ processors in operation is wasteful. We cannot gain anything in time performance, because the time complexity is bounded by the diameter of the mesh.

In other words, $0 < p < \log^2 n$ corresponds to the computation bound region in Figure 14, $\log^2 n < p < n$ corresponds to the connection bound region, and $p > n$ corresponds to the diameter bound region.

4. CONCLUSIONS

Parallel MCC algorithms for solving visibility problems were presented. Given a set of $n$ simple disjoint objects such as line segments, circles, and simple polygons in the plane, our algorithms can find a parallel view or a perspective view of them on a $\sqrt{n} \times \sqrt{n}$ MCC and achieve $O(\sqrt{n})$ time complexity, which is optimal. Data movement has been carefully designed to allow concurrency and hence increase the efficiency.

Methods for solving the above tasks on MCCs with $p$ processors are also considered, where $p < n$. Parallel processing and sequential implementation are combined on the mesh for which we analyzed the time complexity obtained and the limitations imposed by the computational and communication requirements. The result is significant, since in practical MCC applications we will be required to handle varying problem sizes with a fixed number of processors. The research indicated the direction in which efforts should be made to better apply the small size of mesh-connected computers for solving large size problems.

This research is partially supported by the National Science Foundation under grant no. MIP8809328.
REFERENCES