Contour Ranking On Coarse Grained Machines: A Case Study for Low-Level Vision Computations*

Farooq Hameed
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
hameed@cs.purdue.edu

Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
seh@cs.purdue.edu

Ashfaq A. Khokhar
Department of Electrical Engineering and
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA
ashfaq@eecis.udel.edu

Jamshed N. Patel
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, USA
jamshed@ecn.purdue.edu

November 29, 1995

Abstract

In this paper we present parallel solutions for performing image contour ranking on coarse-grained machines. In contour ranking, a linear representation of the edge contours is generated from the edge contours of a raw image. We describe solutions that employ different divide-and-conquer approaches and that use different communication patterns. The combining step of the divide-and-conquer solutions uses efficient sequential techniques for merging information about subimages. The proposed solutions are implemented on Intel Delta and Intel Paragon machines. We discuss performance results and present scalability analysis using different image and machine sizes.

Keywords: Parallel processing, coarse-grained machines, contour ranking, list ranking, computer vision, scalability.

*Research supported in part by ARPA under contract DABT63-92-C-0022ONR. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing official policies, expressed or implied, of the U.S. government.

1 Introduction

Edge operators in low-level vision tasks generate edge contours represented as edge maps in a 2-dimensional image plane. For efficient processing of these edge contours in subsequent mid- and high-level vision tasks, a more compact and linearized representation is desirable [2, 8, 14]. We refer to the process of generating such a linear representation as *contour ranking*. Contour ranking can be done by performing list ranking on each edge contour. Straightforward sequential linear-time solutions exist for list ranking. On parallel machines, numerous fine-grained solutions for list ranking exist, many of them designed for the PRAM [6, 7, 11, 10, 12]. These algorithms are communication intensive and the resulting communication patterns are data-dependent and irregular. Thus, they do not perform well on coarse-grained message-passing machines.

In this paper, we present efficient coarse-grained algorithms for contour ranking. Our algorithms exploit the property that an element of an edge contour is connected to one of its eight nearest neighbors in the image plane. This allows our parallel solutions to employ regular communication patterns which result in a reduction in the communication overhead. Our contour ranking algorithms use a divide-and-conquer approach. Different ways of dividing the image result in algorithms with different communication and computation requirements. We present performance and scalability results for these algorithms on the Intel Touchstone Delta and the Intel Paragon. We discuss how the communication and computation behavior of the different algorithms impacts the algorithm design of other problems in computer vision.

Let I be an image of size $m \times n$. We refer to a pixel on an edge contour in image I as an *edge point*. For each edge point e, $succ(e)$ points to either one of e's eight immediate successors on the edge contour or it is nil. The *successor set* of an edge point e is the set containing e and all other edge points in the reflexive and transitive closure of the $succ$-relation. An edge point e with $succ(e) = nil$ is called a *head*. We assume that an edge point is the successor of at most one other edge point and that the successor set of every edge point contains exactly one head. In contour ranking we determine, for every edge point e, the head in the successor set of e and the size of the successor set of e. We refer to the size of this set as the *rank* of e. Once the rank of every edge is known, a final data movement step generates the linear representation.

We present algorithms for performing contour ranking on a p-processor machine. Our algorithms make no assumption about the communication network underlying the parallel machine. For simplicity, we assume that p is a perfect square and that m and n, the dimensions of image I, are both multiples of $\sqrt{p}$. We assume that image I is partitioned into p rectangular subimages, each of size $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. We number these subimages from I_0 to I_{p-1} using a row-major numbering scheme and assign subimage I_k to processor P_k.

In the next section we describe our parallel algorithms in an architecture-independent way. In Section 3, we present experimental results of implementations based on these algorithms on the Intel Touchstone Delta and Intel Paragon. In the final section, we address how our algorithms and experimental results impact the design of coarse-grained algorithms for other computer vision and image processing problems.

2 Coarse-grained Contour Ranking Algorithms

Efficient coarse-grained algorithms are generally a combination of fine-grained parallel and sequential problem-solving approaches. On coarse-grained machines, divide and conquer strategies often produce efficient solutions. Such strategies typically have a merging step in which the results computed by different processors are combined to obtain the final solution. Different merging patterns have varying communication and computation requirements which can, depending on machine parameters, significantly impact overall performance.

In this section we describe different divide-and-conquer algorithms for contour ranking. We start by defining in Section 2.1 the notations used in the paper, along with a description of the basic concepts. In Section 2.2, we describe the sequential computations performed by the processors. In Section 2.3, we describe the divide-and-conquer patterns employed in our algorithms.

2.1 Notation and Concepts

Let I' be a subimage of I. An edge point h of I' is a head of subimage I' if $succ(h)$ corresponds to either an edge point not in subimage I' or if $succ(h) = nil$. An edge point t of I' is a tail of subimage I' if no edge point of I' has t as its successor. Of particular interest are: (i) heads of I' that lie on the boundary of I' and have a non-nil successors outside I', and (ii) the tails that lie on the boundary of I'. Assume that these heads and tails are stored in the *head list* and the *tail list* of subimage I', respectively. For every edge pixel in the head or tail list we maintain rank and head information. At some point in the algorithm, this rank and head information is correct with respect to (w.r.t.) subimage I'. At later iterations of the algorithm, this information is correct w.r.t. some other subimage of I which contains I'. Eventually, head and rank information will be correct w.r.t. image I.

Let I_k be the subimage assigned to processor P_k, $0 \leq k \leq p-1$. Our algorithms consist of the following three main steps.

- Step 1: Creating Head and Tail Lists.

  Every processor P_k, $0 \leq k \leq p-1$, forms the head list and tail list of subimage I_k. This is done by executing a sequential list ranking on the edge pixels in subimage I_k within processor P_k. For each edge point on a contour, the rank and head information with respect to subimage I_k is determined. This step requires no communication among the p processors.

- Step 2: Merge and Update.

  For each edge point h in the head list of subimage I_k, determine the rank and head information of h with respect to image I. In order to compute this information, head and tail lists of subimages are merged over a number of iterations and the resulting lists form head and tail lists of larger subimages. Once the final information for the larger subimages has been computed, it is used to update the information for the smaller subimages. The choice of subimages to be merged and the choice of processors involved in further computations, determines the communication pattern arising in the algorithm.

- Step 3: Final Updating.

  Every processor P_k, $0 \leq k \leq p-1$, determines the rank and head information w.r.t. image I for every edge pixel in subimage I_k. This final step is similar to the first one and requires no communication between processors.

Step 1 and 3 are identical for each contour ranking algorithm and can be viewed as as preprocessing and postprocessing, respectively. They are "embarassingly" parallel since each processor operates on its own subimage and no communication is needed. The sequential algorithms in Step 1 and Step 3 are straightforward and are not discussed further. The following two sections describe and analyze various parallel solutions for Step 2. We start with a description of the sequential computations arising in Step 2 and then discuss different merging and communication patterns.

2.2 Algorithms Merge and Update

Step 2 of our contour ranking algorithms consists of a number of iterations. Initially, each iteration takes head and tail lists of subimages and determines the head and tail lists of the image resulting from the union of the subimages. An iteration consists of a communication and a computation stage. Communication is discussed in Section 2.3. In this section we discuss Algorithms Merge and Update which are the sequential computations performed after communication. **Algorithm Merge** performs the actual merge of subimages. Assume $I'_1, \ldots, I'_r$ are r rectangular subimages that are contiguous and whose union forms a rectangular subimage I'. Algorithm Merge determines the head and tail lists of I' from the head and tail lists of the r subimages. At some later time during Step 2, rank and head information of the edge points in the head list of I' is correct w.r.t. image I. **Algorithm Update** generates, from the final head list for I', the head and rank information of every edge point in the head lists of $I'_1, \ldots, I'_r$ w.r.t. image I. Algorithm Merge is executed by processors in a forward phase and Algorithm Update is performed in either an explicit or the implicit backward phase. Forward and backward phases are discussed in Section 2.3 along with the communication.

Before giving a description of Algorithm Merge, we describe relevant details of the data structure used for representing the head and tail lists. All the heads (resp. tails) on the boundary of a subimage are represented by a linked list. Entries in the linked list are arranged as heads (resp. tails) are encountered in a clockwise traversal of the boundary of the subimage. Every entry in a head list contains the position of the head in the subimages and the position of the successor of this head in the subimage. Every entry in a tail list contains the position of the tail in the subimage and the rank of the tail edge point in the subimage. Let t be a tail edge point in the tail list. The entry corresponding to t in the tail list also records the position of t's head h in the subimage. In addition, the entry contains a pointer to the entry corresponding to h in the head list of the subimage. These links are needed to achieve linear time in the merging of subimages. Besides these two linked list structures, no data structures are used during Step 2 of the contour ranking algorithms.

Let $I'_1, \ldots, I'_r$ be r rectangular subimages whose union forms a rectangular subimage I'. Algorithm Merge creates the head and tail lists of I' from the head and tail lists of the r subimages. It uses time linear in the total number of edge points in the head and tail lists of the r subimages. We describe the algorithm for the case when $r = 2$. Its generalization is straightforward.

W.l.o.g. assume that I'_1 and I'_2 are horizontally adjacent, as shown in Figure 1. Notice that every edge point in the head (resp. tail) list of I' is also an edge point in the head (resp. tail) list of either I'_1 or I'_2, but the converse is not true. The successor of a head on the common boundary of I'_1 and I'_2 (excluding the four corner points) lies inside I'. Such heads are not included in the head list of I'. Heads on the four corner points of the common boundary are included in the head list of I' only if their successor lies outside I'. Edge points included in the tail list of I' are selected in a similar way. Figure 1 shows the edge points of head and tail lists of two sample subimages and those of their union.

Next, we explain the procedure for updating the head and rank information of an edge point t in the tail list of I'. W.l.o.g., assume that t is an edge point in the tail list of I' that is also an edge point in the tail list of I'_1. Let h be the head of t in subimage I'_1. When h is an edge point not on the boundary of I'_1, the rank and head information of t do not change. This situation applies to t_2 and h_2 of I'_2 in Figure 1. Similarly, when h is an edge point of the head list of I'_1 and $succ(h)$ lies outside I'_1 as well as outside I'_2, the rank and head information of t do not change. This situation applies to t_1 and h_1 of I'_1 in Figure 1. The head (and thus the rank) of t changes only when $succ(h)$ is an edge point on the boundary of I'_2, as shown in Figure 2. Recall that there exists a link from edge point t in the tail list to h in the head list. Let $\hat{t} = succ(h)$. Then, $\hat{t}$ is in the tail list of I'_2. In order to determine the new head and rank of t, Algorithm Merge locates $\hat{t}$ in the tail list of I'_2 and then finds $\hat{t}$'s rank and head in I' recursively. Let h^* be the new head and let a be the rank of $\hat{t}$ in I', as shown in Figure 2. Then, h^* is the new

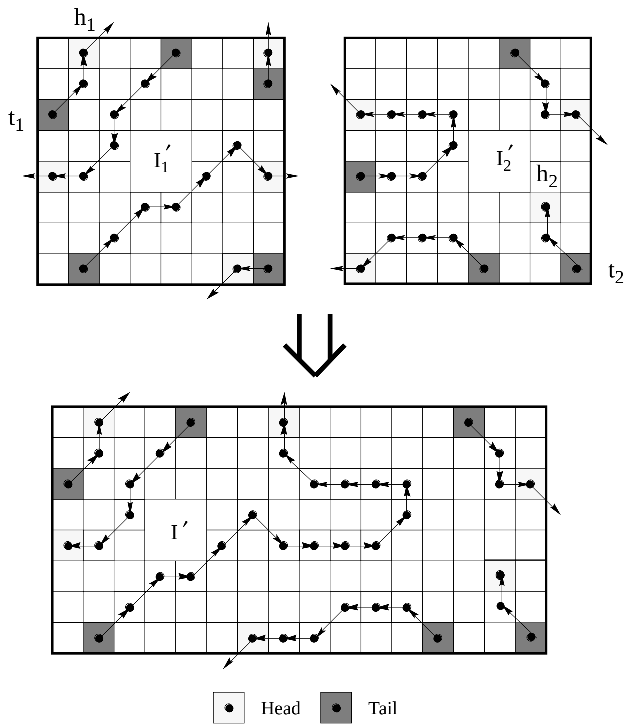

Figure 1: Merging two subimages I_1' and I_2'

head of t in I' and the rank of t in I' is equal to the rank of t in I_1' plus $a+1$.

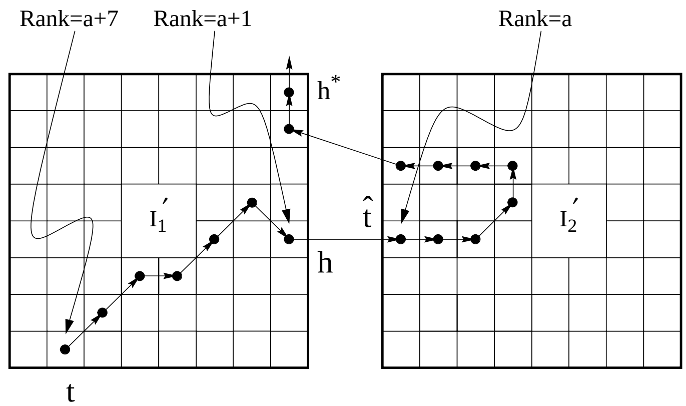

Figure 2: Following a tail in Algorithm Merge

The position of $\hat{t}$ in the tail list of I_2' could be located through binary search. However, doing so would cost $O(\log m)$ time per search, where m is the size of the tail list of I_2' (assuming the head and tail lists were represented accordingly). Algorithm Merge achieves linear time by exploiting the connectivity of the edge contours. The algorithm first creates links from the heads on the common boundary of I_1' and I_2' to the corresponding tails. To set up these links, the head and tail lists containing the edge points on the common boundary are traversed once. Let h_k and t_k be the $k-th$ element of the head list of I_1' and the tail list of I_2', respectively. Assume a link from h_i to the entry corresponding to t_j was created; i.e., $succ(h_i) = t_j$. The link from h_{i+1} to its successor is established next. The edge point corresponding to $succ(h_{i+1})$, cannot occur before t_{j-1} in the tail list of I_2'. Figure 3 shows the only case when $succ(h_{i+1}) = t_{j-1}$. In order to make the link for h_{i+1}, Algorithm Merge considers consecutive edge points starting with t_{j-1} and terminating with $succ(h_{i+1})$. Observe that an arbitrary number of edge points may be traversed. The linking process requires time linear in the number of heads and tails on the common boundary. The updating of the head and tail information for edge points in the tail lists also requires linear time. Hence, the merging of two subimages can be done in linear time.

The merging of more than two subimages (i.e., $r > 2$) is done in a similar fashion. We first create the links between heads and tails lying on the common boundaries of the subimages. We then identify which edge points belong to the head and tail lists of I'. Finally, rank and head

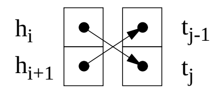

Figure 3: Case when the successor of h_{i+1} occurs before the successor of h_i

information of the edge points on the tail list of I' is updated, using the above technique. The time required for constructing the head and tail lists of I' is linear in the total number of edge points in the head and tail lists of the r subimages.

We conclude this section with a brief description of Algorithm Update. Assume the rank and head information of the edge points in the head list of I' is correct w.r.t. I. Algorithm Update uses this information to determine the head and rank information of every edge point in the head lists of $I'_1, \ldots, I'_r$ w.r.t. image I. When Update is performed, the links from the head lists to the successor edge points in the tail lists are available. Every edge point h in the head list of subimage I'_j, $1 \leq j \leq r$, determines its head in image I' (by following links between head and tail lists). The rank and head information of the so determined heads is available w.r.t. image I. The rank and head information of every h w.r.t. I can now be determined. The time required for Algorithm Update is linear in the total number of edge points in the head and tail lists of the r subimages.

2.3 Communication Patterns for Merging

The choice of which subimages to merge and the number of processors actually performing a merge, determine the number of iterations and the communication characteristics arising in Step 2. At the beginning of Step 2, each processor P_k has determined the head and tail lists of its assigned subimage I_k and has computed rank and head information w.r.t. I_k.

Our contour ranking algorithms differ on which communication pattern is employed in Step 2. Each one of our algorithms can be classified as either a 2-phase or 1-phase algorithm. The fundamental difference underlying the 2-phase and 1-phase approach is as follows. Assume processor P_i contains the head and tail lists of subimage I'_i and processor P_j contains the head and tail lists of subimage I'_j. Assume the next step is to determine the head and tail lists of subimage $I' = I'_i \cup I'_j$. This can be done by P_i sending its head and tail lists to P_j and having P_j determine the head and tail lists of I'. At some later point in time, processor P_j will contain the head and tail lists of I' w.r.t. image I. P_j can then determine the head and tail lists of I'_i and I'_j w.r.t. image I. The head and tail lists of I'_i w.r.t. I are sent to processor P_i. This approach can be generalized from 2 to an arbitrary number of processors. We refer to it as the *2-phase* approach. In the forward phase of a 2-phase approach, subimages are merged in order to compute the head and tail lists of image I. Initially, the number of processors performing computations decreases and by the end of the forward phase a single processor performs computations. After that, the backward phase starts. In the backward phase, head and rank information that is correct w.r.t. image I "flows back" to the smaller subimages. The number of processors performing computations increases and the backward phase terminates after every processor P_k has updated the head and tail lists of subimage I_k w.r.t image I.

An alternative to the 2-phase is the *1-phase* approach in which both P_i and P_j send their head and tail lists to each other. This avoids an explicit backward phase at the cost of more communication in a forward phase. Processors P_i and P_j both, after receiving the other processor's head and tail lists, compute the head and tail lists of subimage I'. Hence, each processor continues to merge subimages, and the number of processors merging identical subimages, and thus performing identical computations, increases with every iteration. Processors continue to merge subimages until each one knows the head and tail lists of image I. At this point the head and tail lists of subimage I_k w.r.t. image I are determined within each processor P_k. The computation done in the backward phase of a 2-phase algorithm is now done within each processor and requires no communication. We refer to this computation as the implicit backward phase.

The communication arising in a 2-phase algorithm are all-to-one communication in the forward phase and one-to-all communication in the backward phase. A 1-phase algorithm executes all-to-all broadcasts in the forward phase; i.e., every processor broadcasts a message to every other processor. Each communication operation is performed on subgroups of processors, with the number of processors in each subgroup depending on the algorithm. We next describe three 2-phase algorithms and then their 1-phase counterparts.

2-Phase Algorithms

The first one of our 2-phase algorithms for computing head and rank information w.r.t. image I employs parallelism in an almost trivial way. Every processor P_k sends its head and tail lists to one common processor, say processor P_0. Processor P_0 determines, for each edge point h in a head list, the rank and head information of h w.r.t. image I. The updated head lists are then sent back to the corresponding processors. In this algorithm, called Direct 2-Phase, the communication between processors occurs in the form of one all-to-one and one one-to-all operation over the entire machine. Figure 4 contains a brief overview of this algorithm, along with the other 2-phase algorithms.

If the communication network of the p-processor machine is (or contains) a mesh, merging the subimages in the row-column (or column-row) pattern is natural. Figure 4 contains an outline of a row-column algorithm, which we call Algorithm Row-Col 2-Phase. For clarity, we change the indexing so that processor $P_{i,j}$ is now assigned subimage $I_{i,j}$, $0 \leq i, j \leq \sqrt{p} - 1$. In the forward phase of Row-Col 2-Phase, the head and tail lists of subimage $I_{i,j}$ are first sent to processor $P_{i,0}$; i.e., the algorithm operates on its rows first. $P_{i,0}$ creates the head and tail lists of subimage $I_{i,*} = \bigcup_{0 \leq j \leq \sqrt{p}-1} I_{i,j}$ and sends them to processor $P_{0,0}$. In the first step of the backward phase, $P_{0,0}$ sends the head list of subimage $I_{i,*}$ w.r.t. image I to processor $P_{i,0}$. Processor $P_{i,0}$ now updates the head list of subimage $I_{i,j}$ w.r.t. image I and sends it to $P_{i,j}$. When we first operate on the columns and then on the rows, we refer to the resulting algorithm as Algorithm Col-Row 2-Phase.

The third algorithm merges the subimages in a quad-tree like fashion. In the description we assume that processors are arranged (and can thus be indexed) in a mesh pattern. However, the quad-tree like merging can be employed efficiently on many other interconnection networks. In Algorithm Quad-Tree 2-Phase, the head and tail lists of four adjacent subimages are merged until the head list of image I is known. In the backward phase, head and rank information w.r.t. image I flows back to the smaller subimages until it reaches subimages $I_{i,j}$. An outline of the algorithm is given in Figure 4.

The above described solutions can also be viewed as partitioning a p-processor machine into one, two, and $\log_4 p$ conceptual levels. In each level, processors are partitioned into groups, with

Algorithm Direct 2-Phase

1. Every processor P_k sends its head and tail list to processor P_0.
2. Processor P_0 merges the p lists and determines, for each edge point in a head list, its rank and head information w.r.t. image I.
3. Processor P_0 sends the updated head lists back to the corresponding processors.

Algorithm Row-Col 2-Phase

1. Processor $P_{i,j}$ sends its head and tail lists to processor $P_{i,0}$, $0 \leq i,j \leq \sqrt{p}-1$.
2. Processor $P_{i,0}$ merges the received lists, creating head and tails list of subimage $I_{i,*}$, $0 \leq i \leq \sqrt{p}-1$.
3. Processor $P_{i,0}$ sends the newly formed head and tail lists to processor $P_{0,0}$, $0 \leq i \leq \sqrt{p}-1$.
4. Processor $P_{0,0}$ forms the head list of image I. It then updates the head list of subimage $I_{i,*}$ so that head and rank information is correct w.r.t. image I, $0 \leq i \leq \sqrt{p}-1$.
5. Processor $P_{0,0}$ sends the updated head list of $I_{i,*}$ to processor $P_{i,0}$, $0 \leq i \leq \sqrt{p}-1$.
6. Using the head and rank information of subimage $I_{i,*}$ w.r.t. image I, processor $P_{i,0}$ determines the head and rank information of $I_{i,j}$ w.r.t. image I, $0 \leq i,j \leq \sqrt{p}-1$.
7. Processor $P_{i,0}$ sends the updated head list of $I_{i,j}$ to processor $P_{i,j}$.

Algorithm Quad-Tree 2-Phase

1. Form $p/4$ groups, each containing 4 processors, so that processors $P_{2i,2j}$, $P_{2i+1,2j}, P_{2i,2j+1}$, and $P_{2i+1,2j+1}$, $0 \leq i,j \leq \sqrt{p}/2-1$ belong to the same group. Processor $P_{2i,2j}$ is made the leader of the group. Every processor sends its head and tail lists to the leader in its group.
2. Let $I'_{2i,2j} = I_{2i,2j} \cup I_{2i+1,2j} \cup I_{2i,2j+1} \cup I_{2i+1,2j+1}$. Leader processor $P_{2i,2j}$ determines the head and tail lists of subimage $I'_{2i,2j}$.
3. The leader processors recursively merge their subimages. After the recursion, processor $P_{2i,2j}$ contains the head list of subimage $I'_{2i,2j}$ w.r.t. image I.
4. Each leader processor determines the head lists for subimages $I_{2i,2j}$ $I_{2i+1,2j}$, $I_{2i,2j+1}$, and $I_{2i+1,2j+1}$ w.r.t image I.
5. Processor $P_{2i,2j}$ sends the updated head lists back to the corresponding processors in its group.

Figure 4: Three 2-Phase Algorithms

communication occurring only between processors in the same group. In [5], we have used this notion of conceptual levels to define a k-level algorithm. One of the conclusions of that work was that for communication operations, 1-, 2-, and $\log p$-level algorithms give good performance on existing coarse-grained machines. The main reason for this is the number of processors in existing coarse-grained machines, which is in the hundreds and not in the thousands. Based on the results of this work, we did not consider general k-level solutions for contour ranking, even though the concepts underlying the described solutions can be generalized.

1-Phase Algorithms

Each one of the 2-phase algorithms has a corresponding 1-phase algorithm. Three 1-phase algorithms are outlined in Figure 5. Direct 1-Phase employs an all-to-all broadcast on p processors as the single communication operation. Row-Col 1-Phase (resp. Col-Row 1-Phase) employs two partial all-to-all broadcasts, the first one within the rows (resp. columns) and the second one within the columns (resp. rows). Quad-Tree 1-Phase performs $p/4$ partial all-to-all broadcasts, each involving four processors, in each one of the $\log_4 p$ iterations. Figure 6 shows the all-to-all patterns arising in the two iterations of Algorithm Quad-Tree 1-Phase on a 4×4 mesh. The processors communicating in the all-to-all broadcast in the second iteration are linked with arrows of the same type.

Variations on the Basic Algorithms

In this section we describe two variations on the above described approaches. The first one tries to take advantage of the fact that in many images a significant number of edges exhibit locality. By this we mean that edges tend to be short in length. Short edge contours can often be ranked by each processor using the subimages assigned to it and its neighboring processors. Hence, each one of the algorithms described can be augmented with a neighbor preprocessing phase. In the neighbor preprocessing phase a processor sends its head and tail lists to its four adjacent processors (horizontally or vertically adjacent). Edge contours that span across two subimages are ranked and their entries are removed from the corresponding lists. Contours that span over more than two subimages are ranked as before. We added the neighbor preprocessing

Algorithm Direct 1-Phase

1. Every processor sends its head and tail lists to every other processor.
2. Every processor $P_{i,j}$ merges the p lists it received and determines, for each edge point on the head list for subimage $I_{i,j}$, the rank and head information w.r.t. image I.

Algorithm Row-Col 1-Phase

1. Processor $P_{i,j}$ sends its head and tail lists to every other processor in row i, $0 \leq i, j \leq \sqrt{p} - 1$.
2. Processor $P_{i,j}$ merges the received lists, creating head and tails list of subimage $I_{i,*}$, $0 \leq i, j \leq \sqrt{p} - 1$.
3. Processor $P_{i,j}$ sends the head and tail lists of subimage $I_{i,*}$ to every processor in column i, $0 \leq i, j \leq \sqrt{p} - 1$.
4. Processor $P_{i,j}$ forms the head lists of image I. It then determines the head list of subimage $I_{i,j}$ with respect to image I. This is done by first updating the head list of $I_{i,*}$ w.r.t image I, $0 \leq i, j \leq \sqrt{p} - 1$.

Algorithm Quad-Tree 1-Phase

1. Form $p/4$ groups, each containing 4 processors, so that processors $P_{2i,2j}$ $P_{2i+1,2j}, P_{2i,2j+1}$, and $P_{2i+1,2j+1}$, $0 \leq i, j \leq \sqrt{p}/2 - 1$ belong to the same group. Number the processors in a group from 1 to 4. Every processor sends its head and tail lists to every other processor in the same group.
2. Let $I'_{2i,2j} = I_{2i,2j} \cup I_{2i+1,2j} \cup I_{2i,2j+1} \cup I_{2i+1,2j+1}$. A processor in the same group with $P_{2i,2j}$ determines the head and tail lists of subimage $I'_{2i,2j}$.
3. All the processors with number l, $1 \leq l \leq 4$, recursively merge their subimages. After the recursion, every processor in the group with $P_{2i,2j}$ contains the head list of subimage $I'_{2i,2j}$ w.r.t. image I.
4. Processor $P_{i,j}$ determines the head list for subimage $I_{i,j}$ w.r.t image I.

Figure 5: Three 1-Phase Algorithms

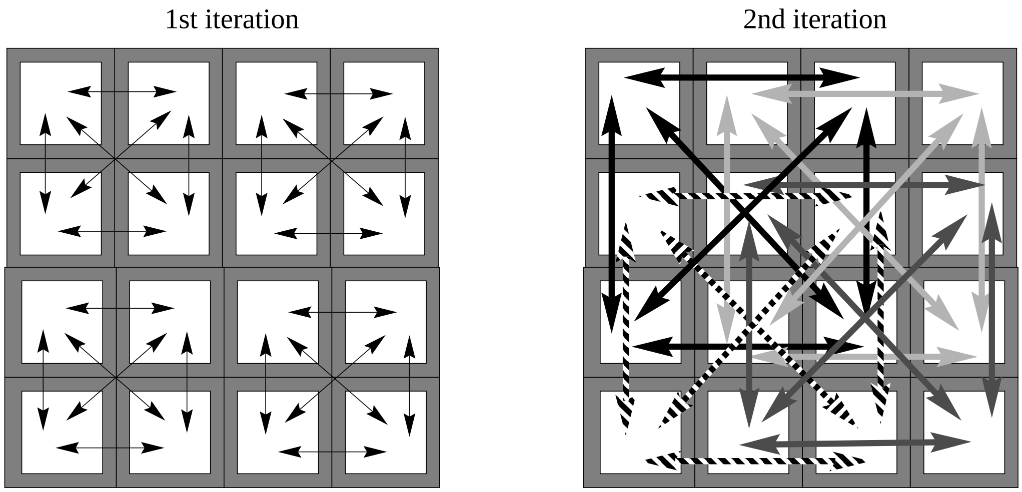

Figure 6: All-to-all Communication patterns for Algorithms Quad-Tree 1-Phase

phase to Algorithms Direct 1-phase, Direct 2-phase, and Quad-Tree 1-phase and will discuss the observed advantage in the next section.

On architectures in which the assignment of subimages to processors preserves locality, Algorithm Quad-tree 1-phase experiences the following communication imbalance. The size of the boundary of the subimages, and thus the size of the lists sent between processors, increases in subsequent iterations. In initial phases, processors communicate over short distances. As the algorithm proceeds, the communication distances and associated congestion increases. This is shown in Figure 6 for teh mesh architecture. For the mesh, this imbalance can be reduced by performing a permutation that sends head and tail lists from processor $P_{i,j}$ to processors $P_{rev(i),rev(j)}$, where $rev(i)$ is the index obtained by applying the bit-reversal to the binary expansion of i. The result of applying this permutation is that processors initially communicate over long distances and, as the size of the lists increases, the distance between communicating processors and thus edge contention, decreases. We will discuss the performance of Algorithm Quad-Tree 1-phase with this balancing variant in the next section.

This completes our discussion of the different patterns for merging boundary lists. We point out that the merging of boundaries is the basis of parallel algorithms for other problems on images; e.g., for determining the connected components. Coarse-grained connected component algorithms based on the merging of boundaries are described in [1, 3]. Our study of 1-phase versus 2-phase algorithms, of varying the number of boundaries to be merged, and of the effect of locality on future iterations is broader than done in previous studies. In the next section we discuss the performance of our different solutions on the Intel Delta and Paragon.

3 Experimental Results

The algorithms described in the previous section were implemented on the Intel Paragon and the Intel Delta. The programs were written in C using the NX message passing library. Compilation on the Paragon and Delta was done with optimization level 4. In this section we discuss performance and analyze the scalability of the algorithms on these machines.

Image	Size	Description
Picnic	256 x 256 512 x 512 1204 x 1024 2048 x 2048	A group photo of a picnic
House	512 x 512	A house with a car in front
Earth	1024 x 768	Earth taken from a satelite
Van Gogh	640 x 480	A painting by Van Gogh
Anatomy	336 x 896	Human Anatomy
Cockpit	944 x 704	An empty Cockpit
Text	512 x 512	Typewritten text
Diagonal Lines	512 x 512	Image filled with diagonal lines
Vertical Lines	512 x 512	Image filled with vertical lines
Semi Dense Vertical Lines	512 x 512	Image half filled with vertical lines

Figure 7: Description of Images

In our experiments, we used both real and synthetic images. A description of the images is given in Figure 7 and three of these images are shown in Figure 8. The edge contours of the images were obtained by applying a sequential edge linking algorithm [4]. As can be seen in Figure 8, the Van Gogh image has a high edge point density compared to the Earth or Picnic image. The edge point density measures the fraction of pixels that are edge points. Also, the shape and size of edge contours vary significantly in the images. Some images, e.g., image Text, contain only short edge contours, while others contain a mixture or only long edge contours. The last three images listed in Figure 7 are synthetic images. They have an edge point density of either 50 or 100% and contain only long edge contours. Note that in an image of 100% edge

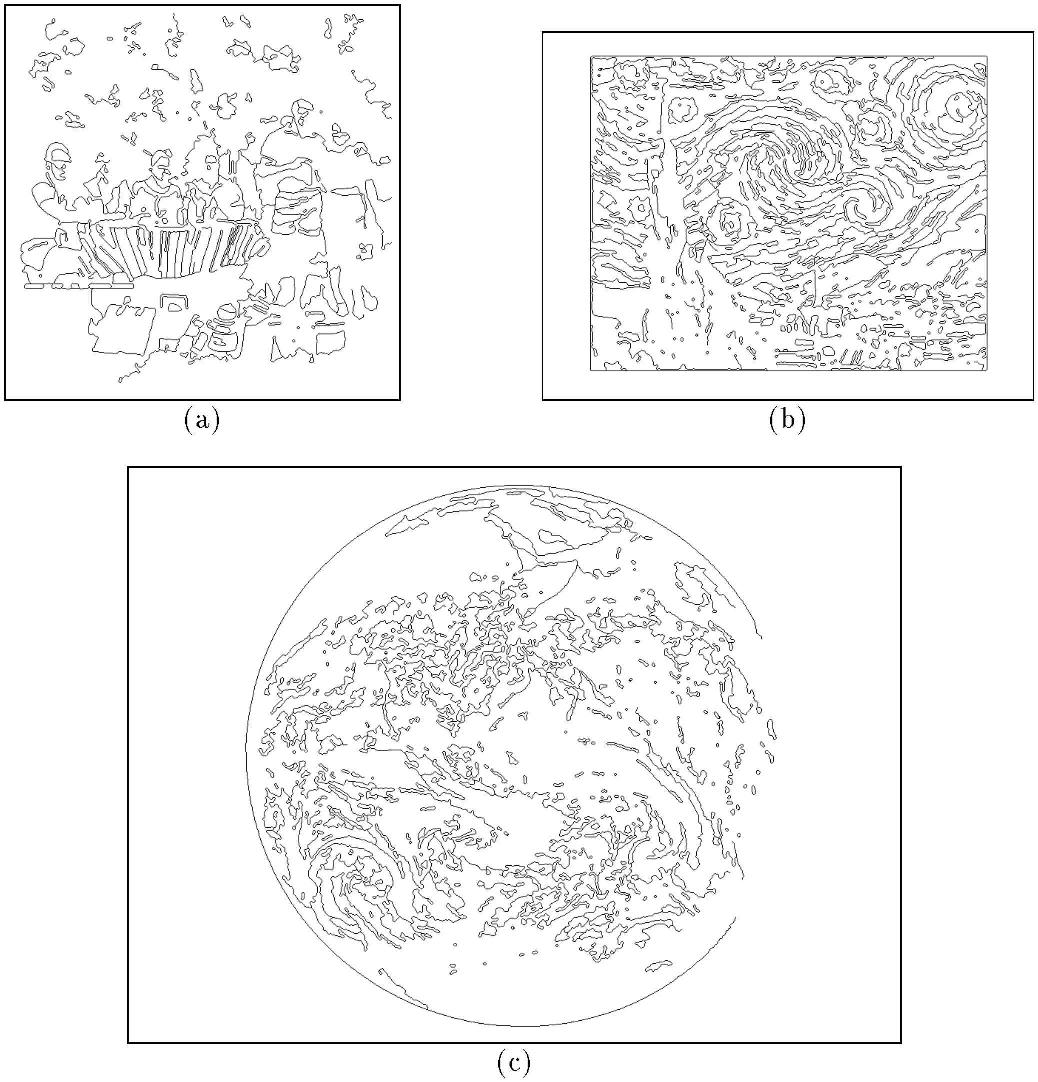

Figure 8: Picnic, Van Gogh, and Earth image

point density, every pixel is an edge pixel (the *succ*-relation still identifies valid edge contours). We use the synthetic images to gain insight into the behavior of the algorithms on images of high edge point densities and to study the effect of large head and tail lists on computation and communication.

In Section 3.1, we discuss the performance of all algorithms on the Intel Paragon for the Picnic image of size 512×512, varying machine size. In Section 3.2, we discuss the performance of the algorithms on a 16×16-processor Intel Paragon for three images shown in Figure 7 and three synthetic images. We discuss how image density influences algorithm performance. In

Section 3.3, we compare the performance of the algorithms on the Intel Delta and the Intel Paragon.

3.1 Performance and Scalability

We use the 512×512 Picnic image to illustrate the relative performance of the different algorithms on the Intel Paragon when the machine size varies from 4 to 512 processors. The conclusions we draw are valid for the other images considered.

Algorithms	Paragon Size						
	2x2	4x4	4x8	8x8	8x16	16x16	16x32
Direct 2-Phase	111.52	37.35	22.17	15.73	15.08	21.37	33.39
Direct 1-Phase	111.37	36.42	20.88	13.88	11.93	14.81	22.49
Quad-Tree 2-Phase	111.68	36.96	20.91	13.48	9.99	8.06	7.35
Quad-Tree 1-Phase	111.62	36.83	20.66	13.18	9.89	8.27	7.54
Quad-Tree 1-Phase Balanced	112.02	37.33	21.33	13.52	9.89	8.89	7.74
Row-Col 1-Phase	111.52	36.86	20.45	12.77	9.58	8.46	9.15
Col-Row 1-Phase	111.46	37.14	20.54	13.10	9.93	9.31	10.40
Row-Col 2-Phase	113.76	37.08	22.30	14.59	11.86	12.26	14.09
Col-Row 2-Phase	113.23	36.87	21.33	14.70	13.93	16.02	13.47
Neighbor Direct 1-Phase	118.14	39.11	22.36	14.90	12.50	16.00	22.62
Neighbor Direct 2-Phas	118.08	39.47	22.78	15.71	13.88	19.78	29.80
Neighbor Quad-Tree 1-Phase	118.25	39.40	22.54	14.22	10.97	8.52	7.96

Figure 9: Picnic Scene (512 x 512 image, time in msec)

Figure 9 gives the running times of all the algorithms on different sizes of Intel Paragon[1]. The algorithms include the 2-phase algorithms described in Figure 4 and the 1-phase algorithms described in Figure 5. In addition, they include four algorithms based on extensions and modifications described in Section 2.3. Algorithm Quad-Tree 1-Phase Balanced is the variant of the 1-phase quad-tree algorithm including the load balancing step. Neighbor Direct 2-Phase, Neighbor Direct 1-Phase, and Neighbor Quad-Tree 1-Phase perform the neighbor communication before proceeding with the actual algorithm.

[1]When a program is run multiple times (e.g. inside a loop) on the Paragon, it has been observed that that first execution of the program is significantly slower than subsequent executions of the same code. This is due to the use of demand paging by the operating system. This behavior is regarded as an anomaly of the current operating system [13], and we expect the problem to be fixed in the next generation of parallel systems software. To eliminate the effects of demand paging on the performance results, we execute the entire program multiple times in a loop. Throughout, we report the execution time of the third run.

For machines consisting of fewer than 64 processors, there is no significant difference in the performance between the different algorithms. This was observed not only for the Picnic image, but for all images we considered. Recall that each algorithm consists of the three steps described in Section 2.1. For small machine sizes, Steps 1 and 3 of an algorithm constitute a significant portion of overall time. To illustrate this point, Figure 10 provides a breakdown of the total time into communication and computation times for three of the algorithms. (The breakdown was obtained by observing one of the processors.) Figure 10(a) shows the time spent on computation in Steps 1 and 3 and in Algorithms Merge and Update of Step 2. Figure 10(b) shows the time spent on communication (which occurs only in Step 2). Figure 10(c) shows the computation time of Step 2; i.e., the time spent by Algorithms Merge and Update. It can be seen that, for small machine sizes, Steps 1 and 3 represent at least 50% of the overall time and that the time of Merge and Update is a relatively small part of the total execution time.

Figure 10(b) shows that for small machines (4 to 64 processors) the communication time decreases as the machine size increases. This is due to a decrease in synchronization costs. On small machines, processors have large subimages assigned and these subimages have different edge point densities. Hence, the time spent by processors in Step 1 on setting up the original head and tail lists varies. Step 2 involves interprocessor communication. A processor P_i finished with Step 1 and waiting for the head and tail lists of some other processor, say P_j, must wait for P_j to complete Step 1. For processor P_i the waiting period counts as communication time. This synchronization overhead increases the communication time on small machines. As machine size increases, the size of the subimage assigned to a processor decreases, thereby reducing the synchronization time. The communication time decreases with machine size until the communication overheads dominate the synchronization cost. After that point, communication time starts increasing again.

For machines of size less than 64, Algorithms Direct 1-Phase, Quad-Tree 1-Phase, and Row-Col 1-Phase experience very similar communication times. One reason is that for small machine sizes there is not much difference between the communication patterns arising in different algorithms. For small machine sizes, the size of the subimage assigned to a processor is large and thus head and tail lists contain more entries. However, because of the high bandwidth of

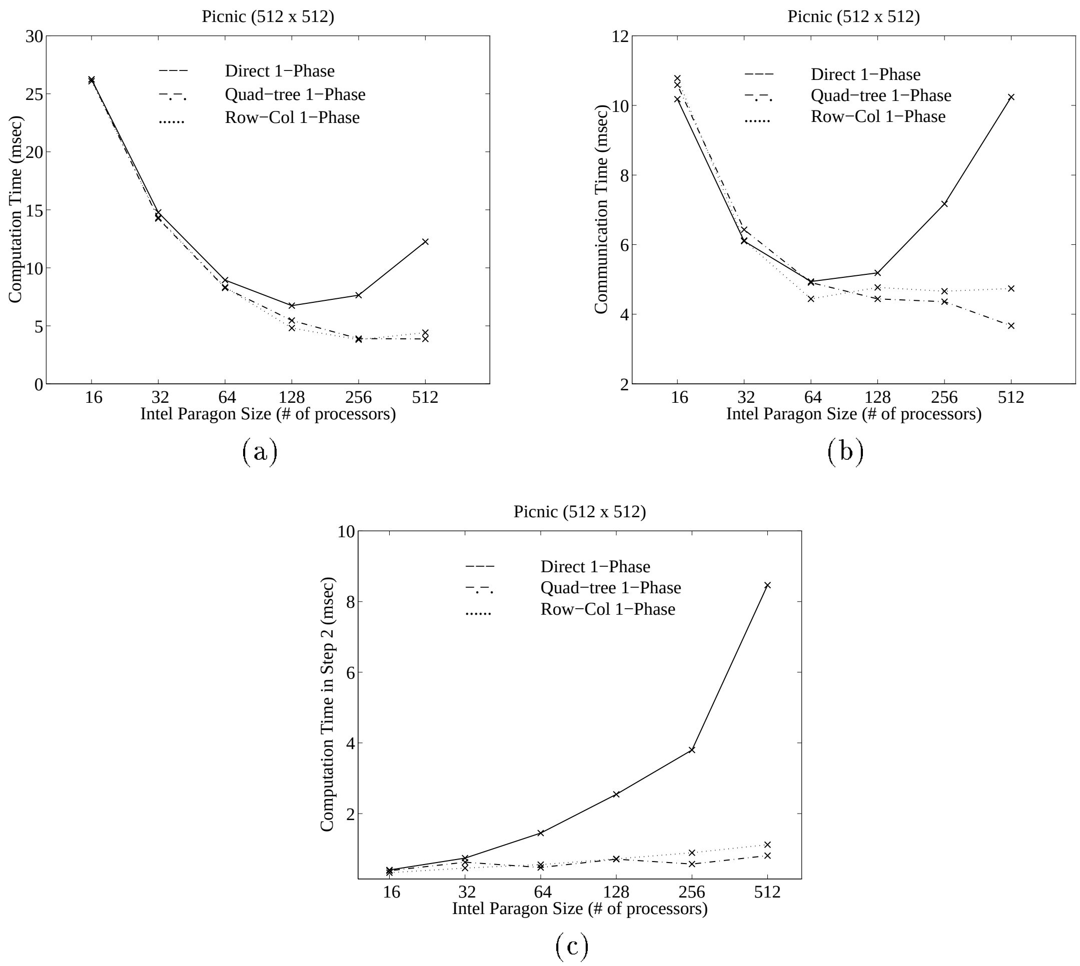

Figure 10: Breakdown into (a) Computation, (b) Communication, and (c) Computation in Step 2 only in Algorithms Direct 1-Phase, Quad-Tree 1-Phase, and Row-Col 1-Phase

the Paragon, the resulting increase in the message sizes does not translate into an increase in actual communication time.

For machines with more than 64 processors, quad-tree based algorithms and row/column based 1-phase algorithms give the best performance. The direct algorithms are the slowest, and the row-col 2-phase algorithms are in between. The computation and communication times shown in Figure 10 provide some explanation. For the sake of simplicity, assume we are dealing with square images. For the direct algorithms, we observed that the computation performed in Step 3 increases linearly with machine size. For p processors, p subimages get merged and their boundary lists are of size at most $4\frac{n}{\sqrt{p}}$. The increase in the total number of edge points involved in the merging of subimages is proportional to at most $\sqrt{p}$. Hence, it is not the increase in the number of edge pixels, but the increase in the number of head and tail lists and the increase in the associated overhead that dictates the observed performance. As the machine size increases, the handling of p head and tail lists dominates the computation time of Step 2 in the direct algorithms.

The row/column algorithms invoke Algorithm Merge twice, each time merging $\sqrt{p}$ subimages. The increase in the number of edge points in the head and tail lists is at most proportional to $\sqrt{p}$. We now observe an increase in the computation and communication time that is proportional to $\sqrt{p}$. With the exception of the largest machine considered (i.e., 16×32), Algorithms Row-Col 1-phase and Col-Row 1-phase match the performance of the quad-tree algorithms. The quad-tree algorithms invoke Algorithm Merge $\log_4 p$ times and overheads are thus proportional to $\log_4 p$.

Figure 9 indicates that 1-phase algorithms outperform their 2-phase counterparts. With the exception of the quad-tree algorithms, 2-phase algorithms were slower than their 1-phase counterparts. Recall that in 2-phase algorithms, leader processors perform algorithm Final_Update on all subimages assigned to them, while in the 1-phase algorithms a processor performs Final_Update only on one of its assigned subimages. Hence, the computation time of Step 2 in a 1-phase algorithm is significantly smaller than that of the corresponding 2-phase algorithm. For the direct algorithms, the computation done in Step 2 of 2-phase algorithms is more than double that of the computation in the 1-phase approach. This decrease in the computation time for 1-phase algorithms is made possible by an increase in the communication time. However, the high network bandwidth of machines like the Intel Paragon and Intel Delta is underutilized in the 2-phase algorithms. Hence, the additional communication arising in 1-phase algorithms does not result in a proportional increase in the communication time. For the quad-tree algorithms we observe a much smaller difference between 1- and 2-phase algorithms. In the 2-phase algorithm, a processor merges and updates always 4 subimages and thus relative difference in the computation is not significant. This factor, along with the presence of a high bandwidth network in Paragon, explains small differences between execution times of the quad-tree 1- and 2-phase algorithms.

Figure 9 also indicates that neither the neighbor preprocessing step nor the load balancing step in the quad-tree algorithms achieves a better performance. With respect to neighbor preprocessing, one might attribute this to a lack of short edges in the Picnic image. However, this is not the case. Even in images for which the neighbor preprocessing step eliminated almost all edge contours (e.g., image Text), we observed no improvement. This behavior is due to the large message passing overhead of the Paragon, compared to its enormous network bandwidth and processing power. This makes the preprocessing steps costlier than the savings that accrue from a reduction of message size in subsequent steps. The implementation of the quad-tree approach using the load balancing step fails for similar reasons.

3.2 Data Dependence

We next discuss the relative performance of the algorithms when the machine size is constant and the edge point density of the images varies. We present data for an Intel Paragon of size 16×16 on 6 images, each of size 512×512. Three real images: Picnic, House, and Text have edge point densities between 5 and 6%. The remaining three images are synthetic images with an edge density of either 50 or 100%. In the Semi-Dense Vertical Lines image, the pixels in every second columns are edge points and the *succ*-relation forms edges in the form of 8 vertical lines. In the Vertical Lines image, the *succ*-relation forms 16 vertical lines. In the Diagonal Lines image, every pixel is again an edge point and the edges form 31 diagonal lines of varying length. Figure 11 gives the edge point densities for these 6 images and the achieved running times.

	Image					
	Picnic	House	Text	S.D. Vert. Lines	Vert. Lines	Diag. Lines
Edge Density	5.25%	5.96%	6.22%	50%	100%	100%
Two Step	21.37	21.44	22.70	60.75	106.06	181.31
One Step	14.81	15.35	15.20	35.82	63.70	99.35
Quad-Tree 2-Phase	8.06	7.92	8.67	18.84	28.72	42.29
Quad-Tree 1-Phase	8.27	7.43	7.94	17.39	27.36	38.76
Quad-Tree 1-Phase Balanced	8.89	8.11	8.43	18.41	29.38	38.94
Row-Col 1-Phase	8.46	8.88	9.30	42.80	78.87	73.26
Col-Row 1-Phase	9.31	8.91	8.64	15.13	22.07	74.09
Row-Col 2-Phase	12.26	16.10	13.02	57.52	105.06	110.51
Col-Row 2-Phase	16.02	16.05	16.02	18.12	26.34	110.59
Neighbor Direct 1-Phase	16.00	14.98	15.36	36.43	63.15	100.28
Neighbor Direct 2-Phase	19.78	21.18	19.74	60.50	102.43	183.74
Neighbor Quad-Tree 1-Phase	8.52	8.31	8.24	18.56	30.00	38.76

Figure 11: Times (in msec) on a 16×16 Paragon for six images with different edge densities

Clearly, the running times increase with edge point density. When the edge density increases from about 5% to 50%, none of the algorithms experiences a proportional slowdown. At the same time, when the edge point density increases from 50 to 100%, the times double or triple. The reason lies in the underutilization of the network and processor bandwidth for images with low densities, as already discussed in the previous section. Figure 11 also shows that in addition to edge pixel density, the size of the head and tail lists also impacts performance. This can be observed by comparing the running times of the algorithms for image Vertical Lines and image Diagonal Lines. Although the edge pixel density is the same in both images, the size of the head and tail lists of image Diagonal Lines is twice the size of head and tail lists of image Vertical Lines.

The relative performance of algorithms remains basically the same, regardless of edge point density. The algorithms quad-tree based algorithms give the best (or close to the best) performance. Even though Algorithm Col-Row 1-Phase is almost perfectly tailored towards image Vertical Lines, Algorithm Quad-Tree 1-Phase does not perform significantly worse. It is interesting to note that the load balancing step performed in Algorithm Quad-Tree 1-phase does not give the expected improvement for the fully dense image consisting of diagonal lines. As already stated in Section 3.1, the neighbor preprocessing phase does not improve performance even for image Text (which contains typed text).

3.3 Comparison Between Paragon and Delta

Figure 12 gives the execution times of the algorithms on the images given in Figure 8 on a 16×16 Intel Paragon and Intel Delta. The behavior of the algorithms on the Delta is similar to the one observed for the Paragon. Quad-tree based algorithms give the best performance, closely followed by the row-column based algorithms. The direct algorithms are the slowest. This similarity is not surprising because of the similar architectures of both machines.

Algorithms	Intel Pargon			Intel Delta		
	Earth 1024x768	V. Gogh 640x480	Picnic 512x512	Earth 1024x768	V. Gogh 640x480	Picnic 512x512
Direct 2-Phase	27.34	25.25	21.37	34.23	30.91	25.46
Direct 1-Phase	19.89	18.00	14.81	27.89	25.49	21.12
Quad-Tree 2-Phase	12.89	9.29	8.06	16.62	12.68	11.16
Quad-Tree 1-Phase	12.62	9.38	8.27	15.94	12.08	10.82
Quad-Tree 1-Phase Bal.	13.05	9.84	8.89	17.77	13.24	11.92
Row-Col 2-Phase	24.10	18.16	12.26	21.13	16.53	13.78
Row-Col 1-Phase	13.49	11.25	8.46	17.96	14.00	11.82
Col-Row 2-Phase	19.81	16.01	16.02	19.82	17.04	13.79
Col-Row 1-Phase	13.58	10.49	9.31	17.43	14.00	11.86
Neighbor Direct 1-Phase	19.58	16.61	16.00	26.86	24.06	20.95
Neighbor Quad-Tree 1-Phase	13.23	9.75	8.52	17.23	12.74	11.73

Figure 12: Performance of the algorithms on the Paragon and the Delta using 256 processors

The figure also indicates that the performance of the Paragon is about 10-30% faster compared to that of the Delta. From the technical specifications of the Paragon, a larger difference between the speeds of the two machines would be expected. In particular, the Paragon has a much higher network bandwidth compared to the Delta and the processors on the Paragon we used are 1.5 times faster than the processors on the Delta. The figure shows that the direct algorithms benefited the most and are 20-40% faster on the Paragon than on the Delta. The row/col based algorithms benefit the least, with speedups as low as 10%. We believe that this non-optimal speedup occurs because of an underutilization of the available resources inherent to the problem under consideration.

4 Concluding Remarks

We have presented parallel solutions for performing contour ranking on coarse-grained machines. These solutions employ different divide-and-conquer patterns and different communication patterns. They use efficient sequential techniques for merging information about subimages. The proposed solutions were implemented on Intel Delta and Intel Paragon machines. We discussed performance results and presented scalability analysis using different image and machine sizes.

Our results of the contour ranking algorithms provide insight into the behavior and interplay of various machine and problem parameters. The results also lead to a design philosophy for coarse-grained algorithms for a large class of low-level vision tasks. Our contour ranking algorithms use divide-and-conquer and merge information about subimages in order to compute the final values. The information needed about a subimage is proportional to the number of edge points on the boundary of this subimage. The time used for merging subimages is linear in the number of edge points in the boundary lists of these subimages. A number of other problems on images can be solved by algorithms following the same principle. The merging of boundaries is central to parallel connected component algorithms. We refer to [1, 3] and references therein for recent work on coarse-grained connected component algorithms. Other problems that can be solved by divide-and-conquer algorithms having similar characteristics include straight line approximations and region growing. The approaches used in our contour ranking algorithms can serve as templates for parallel algorithms for such problems. Unless sequential computation times and messages sizes change significantly, our observations about the performance of the different communication patterns will remain valid.

Various distance computations in an image [9] can also be performed by divide-and-conquer algorithms. However, the information needed about a subimage may now be quadratic in the size of the boundary. This results in more data to be communicated. In the presence of a high-bandwidth communication network, the performance of coarse-grained algorithms for distance problems is likely to follow the same trend.

5 Acknowledgements

We thank the referees for their helpful comments and suggestions.

References

[1] D.A. Bader, J. Ja'Ja', "Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study", Technical Report, CS-TR-3384, University of Maryland, December 1994.

[2] L. T. Chen, L. S. Davis, and C. P. Kruskal, "Efficient parallel processing of image contours," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, no. 1, pp. 69-81, 1993.

[3] A. Choudhary, R. Thakur, "Connected Component Labeling on Coarse Grain Parallel Computers", *JPDC*, Vol. 20, no. 1, pp. 78-83, 1994.

[4] P. H. Eichel and E. J. Delp, "Sequential edge linking," *Proceedings 22nd Allerton Conference on Communication, Control, and Computation*, pp. 782-791, 1984.

[5] S.E. Hambrusch, F. Hameed, and A. Khokhar, "Communication Operations on Coarse-Grained Architectures," *Parallel Computing*, Vol. 21, pp. 731-751, 1995.

[6] T. Heywood and S. Ranka, "Architecture independent analysis of sorting and list ranking on the hierarchical PRAM model," *Proceedings Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 531-4, 1992.

[7] J. JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[8] M.H. Kim, O.H. Ibarra, "Transformations Between Boundary Codes, Run Length Codes, and Linear Quadtrees," *Proceedings of the 8th International Parallel Processing Symposium*, pp. 120-125, 1994.

[9] R. Miller, Q. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh-connected Computer," *IEEE Trans. on PAMI*, pp. 216-228, 1985.

[10] J. N. Patel, A. A. Khokhar, and L. H. Jamieson, "Scalable Parallel List Ranking of Image Edges on Fine-grained Machines," *9th International Parallel Processing Symposium*, pp. 717-721, 1995.

[11] M. Reid-Miller, "List Ranking and List Scan on the Cray C-90," *Proceedings Symposium on Parallel Algorithms and Architectures*, Cape May, NJ, pp. 104-113, 1994.

[12] *Synthesis of Parallel Algorithms*, J.R. Reif, Editor, Morgan Kaufmann, 1993.

[13] S. Saini, H. Simon, "Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation," *Proceedings of the Scalable Parallel Libraries Conference*, pp. 232-239, 1993.

[14] H. Samet, *Applications of Spatial Data Structures, Computer Graphics, and Image Processing*, Addison Wesley, 1990.