Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 1297–1310
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Coarse grained gather and scatter operations with applications

Laurence Boxer[a,b,*], Russ Miller[c,1]

[a] Department of Computer and Information Sciences, Niagara University, NY 14109, USA
[b] Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY 14260, USA
[c] Department of Computer Science and Engineering & Center for Computational Research, State University of New York at Buffalo, Buffalo, NY 14260, USA

Received 10 February 2003; received in revised form 18 December 2003

Abstract

We introduce asymptotically optimal algorithms for gathering and scattering a small-to-moderate sized set of data on a coarse grained parallel computer. We use these operations to obtain efficient to optimal solutions to several fundamental problems in image processing and string matching (exact or approximate) for coarse grained parallel computers.

Keywords: Scaleable parallel algorithm; Coarse grained parallel computer; Semigroup operation; Parallel prefix; Digital picture; Connected component; Convex hull; Distance transform; Hausdorff metric; String matching

1. Introduction

Many recent papers use coarse grained models of parallel computation. The coarse grained multicomputer (CGM) model was introduced in [DFR93]. Computational geometry algorithms for the CGM have been presented in [BxHr01,BMR98,BMR99,DFR93,FRU95,DDDFK95]; also see [Dehn99]. Other computational models for coarse grained parallel computers include [Vali90,CKPSSSSE, HaK93].

Some papers using the CGM model of [DFR93] assume that data is communicated among processors only via sorting operations. This has the advantage that sorting can be treated as a general "black box" operation; running times may be expressed in terms of sorting time, so that analysis is not dependent on the architecture of the parallel computer. For many problems, this is a reasonable approach, as sorting has been implemented efficiently on all practical models of parallel computing. However, real machines are generally not constrained to sort-based data communications, which are often non-optimal. Recent papers, including [FKRU99,MoSo01,SaSo99], have used the CGM model, allowing communication among processors not based on sorting. In the current paper, we will not require interprocessor communications to be based on sorting.

We show how gather and scatter operations can be implemented to compress a small set of data to one processor, and, respectively, reverse such a compression, both in worst-case optimal time. These operations are often useful when key steps of an algorithm require processing a reduced set of data. Our applications of gather and scatter include the fundamental operations of semigroup computation and parallel prefix, and solutions to several problems in image processing and string matching (exact and approximate).

Our solutions, as far as we know, are the most efficient to appear in the literature. Many are adapted from well-known solution strategies to the CGM model. In a few cases, our running times yield domains of optimality that are proper subsets of the range of processors considered. This illustrates the principle, so often met in the study of fine-grained parallel algorithms, that architecture matters (notice also that many of the algorithms presented in papers cited above have

* Corresponding author. Department of Computer and Information Sciences, Niagara University, NY 14109, USA. Fax: +1-716-286-8445.
E-mail address: boxer@niagara.edu (L. Boxer).
[1] Research partially supported by NSF grant ACI-0204918.

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.07.002

running times expressed as $\Theta(T_{\text{sort}}(n, p))$; asymptotic analysis of $\Theta(T_{\text{sort}}(n, p))$ depends on architecture). We show that the mesh architecture is particularly useful for these problems.

2. Preliminaries

2.1. Model of computation

The *coarse grained multicomputer*, or $CGM(n, p)$, is described in [DFR93]. A $CGM(n, p)$ has a set of p processors that operate on $\Theta(n)$ data items. Every processor is assumed to have $\Omega(\frac{n}{p})$ local memory cells, each of $\Theta(\log n)$ bits. The term "coarse grained" means the size $\Omega(\frac{n}{p})$ of each local memory is "considerably larger" than $\Theta(1)$. It has become customary to interpret "considerably larger" to mean that $\frac{n}{p} \geqslant p$, a convention we use in the current paper. Thus, each processor has at least enough local memory to store a unique ID for every other processor. The processors may share memory or may be arranged in some interconnection network.

In [DFR93], it is assumed that all communications among processors are handled by sorting operations. However, this restriction appears to be a barrier to optimal performance, so some recent papers do not assume this restriction. We assume that processors may communicate data without sorting. Any directly connected pair of processors may exchange a unit of data in $\Theta(1)$ time. Similarly, in a shared memory system, any pair of processors may exchange a unit of data in $\Theta(1)$ time.

We regard a $CGM(n, p)$ as a connected graph G in which the vertex set is the set of processors and the edge set is the set of communications links that join pairs of processors (if the $CGM(n, p)$ is a shared memory machine, we regard it as a complete graph, i.e., every pair of processors is regarded as joined by an edge).

Algorithms for coarse-grained parallel computers are often designed to be *scaleable*, i.e., they are described so that they may be implemented over the full range of processors, $1 \leqslant p \leqslant n^{1/2}$, in terms of which the coarse grained models are described, except as noted otherwise (among our algorithms is one that calls for $p \leqslant n^{1/3}$).

2.2. Fundamental operations

We say a list $x_1, x_2, \ldots, x_n$ is *evenly distributed* among the processors of a $CGM(n, p)$ if its members are partitioned among the processors such that each processor stores $\Theta(\frac{n}{p})$ of the members.

For a given problem, suppose T_{seq} and T_{par} are, respectively, the running times of the problem's best sequential and best parallel solutions. If

$$T_{\text{par}} = \Theta\left(\frac{T_{\text{seq}}}{p}\right) \tag{1}$$

then the parallel algorithm is optimal, to within a constant factor. In practice, we often must add to $\Theta(\frac{T_{\text{seq}}}{p})$ the time necessary for interprocessor communications and/or data exchanges (e.g., in global sorting operations) in order to evaluate T_{par}.

We denote by $T_{\text{sort}}(n, p)$ the time required by the most efficient algorithm to sort $\Theta(n)$ data on a $CGM(n, p)$. Sorting is a fundamental operation that has been implemented efficiently on all models of parallel machines (theoretical and existing). Sorting is important not only in its own right, but also as a basis for a variety of parallel communications operations, such as the following (see [BMR98,DFR93] for efficient CGM implementations of these and others).

- *Multinode broadcast*: Every processor sends a copy of the same unit of data to every other processor. That is, every processor P_i sends a copy of a locally stored data value d_i to every processor P_j, $j \neq i$.
- *Total exchange*: Every processor sends a (not necessarily the same) unit of data to every other processor. That is, there is a set of values $\{d_{i,j}\}_{i=1}^{p}$ stored in processor P_j, $j \in \{1, \ldots, p\}$, such that P_j sends $d_{i,j}$ to P_i for all $i, j \in \{1, \ldots, p\}$.
- *Semigroup operation*: Let $X = \{x_1, \ldots, x_n\}$ be a set of data distributed evenly among the processors and let $\circ$ be a binary operation on X that is associative and that may be computed in $\Theta(1)$ serial time. Compute $x_1 \circ x_2 \circ \cdots \circ x_n$ and make the result known to all processors. Examples of such operations include *total*, *product*, *minimum*, *maximum*, *and*, and *or*.
- *Parallel prefix*: Let $X = \{x_1, \ldots, x_n\}$ be a set of data distributed evenly among the processors and let $\circ$ be a binary operation on X that is associative and that may be computed in $\Theta(1)$ serial time. Compute all n members of $\{x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \cdots \circ x_n\}$.

Another fundamental sort-based operation we use in this paper is *concurrent read* [MiBo00,MiSt96], in which a set of processors requires, for each member x of a set X, a set of $\Theta(1)$ keys associated with x, without a priori knowledge of which processor stores the data associated with any key. Below, we present a somewhat simplified version of concurrent read.

Proposition 2.1. *Let X and Y be sets of n data apiece, distributed evenly throughout the processors of a $CGM(n, p)$ G. Suppose for each $x \in X$, there is a unique $y \in Y$ such that x must be associated with a value k_x currently associated with y. Then a concurrent read operation that forms the pairs (x, k_x) for every $x \in X$ can be performed in $\Theta(T_{\text{sort}}(n, p))$ time.*

Proof. We give the following algorithm, which follows the outlines of concurrent read operations given in [MiBo00,MiSt96].

1. In parallel, every processor creates, for each of its members y of Y, a *master record* of the form

[$key, returnaddress, data$, "*Master*"], where *key* is the value of the key field of *y* to be used to find its corresponding member of *X*, $returnaddress$ is the processor in which *y* is stored, and *data* is the value associated with *y* that serves as k_x for some $x \in X$. Since *Y* is evenly distributed, this takes $\Theta(n/p)$ time.

2. In parallel, every processor creates, for each of its members *x* of *X*, a *request record* of the form [$key, returnaddress, data$, "*Request*"], where *key* is the value of the key associated with *x* to be used to match a member of *Y*, $returnaddress$ is the processor in which *x* is stored, and *data* is uninitialized (eventually to be used for the desired value k_x). Since *X* is evenly distributed, this takes $\Theta(n/p)$ time.
3. Sort the union of the Master and Request records by the *key* field. Where there are ties, place a Master record before its (unique) Request record. This takes $\Theta(T_{\text{sort}}(n, p))$ time.
4. In the sorted list, every Request record now gets for its *data* field the desired k_x value from the *data* field of the preceding record, which is the corresponding Master record. This takes $\Theta(n/p)$ time.
5. Return all the records to their original processors by sorting on the $returnaddress$ field in $\Theta(T_{\text{sort}}(n, p))$ time.

The algorithm uses $\Theta(\frac{n}{p} + T_{\text{sort}}(n, p))$ time. This simplifies as $\Theta(T_{\text{sort}}(n, p))$ time, since we have, for sorting, $\frac{n}{p} = O(\frac{T_{\text{seq}}}{p}) = O(T_{\text{sort}}(n, p))$. □

In previous papers on CGM algorithms (c.f., [BMR98, BMR99,DFR93,DDDFK95]), it was often found that algorithms were dominated by operations requiring global communication of data, so that if communications are sort-based, then a problem with $\Theta(n)$ input or output is solved in $\Theta(T_{\text{sort}}(n, p))$ time. In the current paper, we show how not requiring communications to be sort-based enables us to obtain coarse grained algorithms that are efficient to optimal for the problems we discuss.

3. Gather and scatter operations

In this section, we define *gather* and *scatter* operations and develop efficient implementations for coarse grained parallel computers. The definitions of gather and scatter operations given below are related to, but somewhat different from, those of [MiSt96].

Definition 3.1. Let *S* be a nonempty set of data distributed among the processors of a parallel computer *C*. Let P_0 be one of the processors of *C*.

- We say *S* is gathered into P_0 by a data movement operation that sends copies of all the members of *S* to P_0.
- We say *S* is scattered from P_0 by a data movement operation that sends all members of *S* from P_0 back to their original processors.

We show in this section that a CGM can gather and scatter a small set *S* of data in time linear in the size of *S*. To direct the flow of data in our gather/scatter algorithms, the following is useful.

Lemma 3.2. *Let G be a $CGM(n, p)$. Let R be any processor of G. In $O(p)$ time, a spanning tree T for G with R as the root node of T can be determined such that all processors of G know their parent and child nodes in T. In the worst case, the running time $\Theta(p)$ is optimal.*

Proof. The algorithm given below essentially consists of a scatter operation to identify parents, followed by a gather operation to identify children. Subsequent scatter and gather operations may be more efficient, as a processor in the current algorithm will communicate with all of its neighbors (not only parent and children) in order to determine these relationships. However, in the worst case, this loss of efficiency causes an increase of running time of only a constant factor.

The worst case assumes that a processor must communicate sequentially with its neighboring processors. We note that some parallel architectures permit a processor to send a unit of information to all neighboring processors in $\Theta(1)$ time. Were we to use such an assumption, steps of the algorithm given below run faster than claimed. However, we will show that the running time of the algorithm would remain $\Theta(p)$ in the worst case.

We give the following algorithm:

1. In parallel, each processor creates an ID record containing its processor ID and its parent processor ID; the latter is initialized to *null*. Also, each processor initializes a list of its children in *T* as empty. This takes $\Theta(1)$ time.
2. *R* sends its ID record to all its neighbors. This takes $O(p)$ time.
3. In parallel, each processor *P* does the following.
   (a) If $P \neq R$, do the following.
      i. Receive a neighbor's ID record. This takes $\Theta(1)$ time, after the time during which *P* waited for the message to arrive. The waiting time is analyzed below.
      ii. Set $P.parent$ (second component of the record) equal to the received processor ID (first component of the received record). This takes $\Theta(1)$ time.
      iii. Send the processor's own ID record to all neighboring processors. The time for this step is $O(p)$.
   End if.
   (b) From each neighboring processor *Q* such that $Q \neq P.parent$, receive the ID record of *Q*. If $Q.parent = P$, add *Q* to the list of children of *P* in the spanning tree. This step takes $O(p)$ time, plus waiting time. We discuss the waiting time below.

End in parallel

To analyze the time a processor waits for messages from its neighbors in the algorithm above, we separately analyze the time spent waiting for its parent's ID record, and the time spent waiting for the ID records of its non-parental neighbors.

To analyze the time a processor waits until receiving its parent's ID record, we observe that if P and Q are neighboring processors in G, then the respective distances d_P and d_Q of these processors from R in G satisfy

$$|d_P - d_Q| \leqslant 1. \tag{2}$$

The time spent waiting by a processor for its parent's message may be analyzed as follows. Let $d_{\max}$ be the maximum distance of a processor from R in G. For $i \in \{-2, -1, 0, \ldots, d_{\max} + 1\}$, let

$$A_i = \{P \mid P \in V(G),\ d(P, R) = i\},$$

where the distance $d(P, R)$ is the smallest number of edges in a connecting path in G from P to R. Notice that $A_{-2} = A_{-1} = A_{d_{\max}+1} = \phi$ and that

$$\Sigma_{i=-2}^{d_{\max}+1} |A_i| = p. \tag{3}$$

Suppose we say a *unit time step* is the time required for a processor to send its ID record to one of its neighbors. Let n_i be the maximum number of unit time steps until a member of A_i receives its parent's ID record, with $n_0 = 0$. We will show that for $i \in \{0, 1, \ldots, d_{\max}\}$,

$$n_i \leqslant 3(\Sigma_{j=-2}^{i-2} |A_j|) + 2|A_{i-1}| + |A_i|. \tag{4}$$

To show that inequality (4) is valid for $i \in \{0, 1, \ldots, d_{\max}\}$, we argue by induction on i. Inequality (4) is trivial for $i = 0$. Now suppose k is an integer, $0 \leqslant k < d_{\max}$, such that inequality (4) is true for $i \leqslant k$. Let $P \in A_{k+1}$. There exists $Q \in A_k$ such that P and Q are neighbors in G. Then the number of unit time steps until P receives its message from its parent node is less than or equal to the number unit time steps until P receives its message from Q. From inequality (2), all neighbors of Q belong to $A_{k-1} \cup A_k \cup A_{k+1}$. In the worst case, P is the last neighbor of Q to receive a record from Q. It follows that

$$\begin{aligned} n_{k+1} &\leqslant n_k + |A_{k-1}| + |A_k| + |A_{k+1}| \\ &\leqslant \text{(by the inductive hypothesis) } 3(\Sigma_{j=-2}^{k-1} |A_j|) \\ &\quad + 2|A_k| + |A_{k+1}| \end{aligned}$$

as desired. This completes the induction.

From Eq. (3) and inequality (4), $n_i \leqslant 3p$, so the waiting time for every processor to receive its parent's ID record is $O(p)$.

The time spent awaiting messages from non-parental neighbors is analyzed as follows. Let P and Q be neighboring processors. Once P receives its parent's ID record, P sends its own ID record to all its neighbors. In the worst case, Q is the last of the neighbors of P to receive the ID record from P, waiting $O(p)$ unit time steps. In the worst case, P is then the last neighbor of Q to receive the ID record of Q, waiting $O(p)$ unit time steps to receive the ID record of Q. Thus, in $O(p)$ unit time steps, P and Q exchange ID records. Taking the maximum over all neighbors Q of P, it follows that P waits $O(p)$ unit time steps before receiving the last of its neighbors' ID records.

Therefore, the entire algorithm takes $O(p)$ time.

To show $\Theta(p)$ is optimal in the worst case, we observe that the root processor R could be an end processor of a linear array, in which case the communication with the processor at the opposite end of the linear array requires $\Omega(p)$ time. □

We use Lemma 3.2 to obtain the following.

Theorem 3.3. *Let S be a set of N data items such that $N = \Omega(p)$ and $N = O(n/p)$, and suppose S is distributed among the processors of a $CGM(n, p)$, G.*

- *A gathering of S to any processor of G can be performed in $\Theta(N)$ time, which is optimal in the worst case.*
- *A scattering of S can be performed in $\Theta(N)$ time, which is optimal in the worst case.*

Proof. Since the processor R to which S is gathered reads $\Theta(N)$ data sequentially in the worst case, a gathering must take $\Omega(N)$ time in the worst case. In scattering S, R transmits $\Theta(N)$ data sequentially in the worst case, so a scatter must take $\Omega(N)$ time in the worst case.

Our algorithms call for each processor to maintain an array $from[1 \ldots p]$ used to keep track of which data reached the processor from which neighboring processor. This is necessary so that during a scatter, data can be routed efficiently among the processors. In every processor P_i, we will define entries of the array *from* as follows:

$$\begin{aligned} &from[j] \\ &\quad = k \text{ if data originating in } P_j \text{ was sent to } P_i \text{ by } P_k. \end{aligned} \tag{5}$$

We give the following algorithm for a gather operation:

1. In $\Theta(1)$ time, each processor P_i sets its $from[i] = i$.
2. In $O(N)$ time, each processor P_i tags each of its members $s \in S$ via $s.processorOrigin = i$.
3. If a spanning tree for G with R as the root is not already known, use the algorithm of Lemma 3.2 to configure a spanning tree T of G so that R is the root processor and every processor P knows its parent processor $parent(P)$ and its child processors in T. This takes $O(p)$ time.
4. In parallel, every processor P sends members of S to $parent(P)$ and receives members of S from its children until there are no more members of S for P to send. As P receives members of S from its children, P marks the appropriate entry of its *from* array in accord

with Eq. (5):

$$from[s.processorOrigin] = k \text{ if } s \text{ reached } P \text{ from } P_k.$$

Since each processor handles $O(N)$ data with a total waiting time of $O(N+p)$, this takes $O(N+p) = O(N)$ time.

Thus, this algorithm requires $O(N)$ time.

To show that a scattering requires $O(N)$ time, we give the following algorithm.

1. The root processor R does the following. For each $s \in S$, if $s.processorOrigin$ is not R then send s to the neighboring processor $P_{from[s.processorOrigin]}$. This takes $O(N)$ time.
2. All other processors P_i do the following in parallel. For at most N members s of S, receive s from a neighboring processor. If $s.processorOrigin \neq i$ then send s to $P_{from[s.processorOrigin]}$. Since waiting for data to arrive from neighboring processors requires $O(N+p) = O(N)$ time, this takes $O(N)$ time.

This algorithm takes $O(N)$ time. At the end of the latter step, every $s \in S$ has returned to its original processor. □

4. Applications to fundamental algorithms

In this section, we give efficient CGM solutions to several fundamental problems. All of our algorithms make use of the gather and/or scatter operations, and all are asymptotically faster or are efficient to optimal over wider ranges of the parameter p than previous counterparts that assumed sort-based communications.

Broadcasting a unit of data x stored in processor P to all other processors on a $CGM(n,p)$ can be done by a multinode broadcast operation, in which all processors $Q \neq P$ send a unit of dummy data. This can be done in $\Theta(T_{\text{sort}}(p^2,p))$ time [BMR98]. Even if we base our parallel sort on a linear-time sequential sort such as BinSort, for sorting $\Theta(p^2)$ data we have

$$T_{\text{par}} = \Omega\left(\frac{T_{\text{seq}}}{p}\right) = \Omega(p^2/p) = \Omega(p).$$

Therefore, the running time asserted below improves upon the result of [BMR98].

Theorem 4.1. *Let x be a unit of data in one processor P_a of a $CGM(n,p)$. In $O(p)$ time, x can be broadcast to all processors.*

Proof. We observe that depending on the computer's architecture, an algorithm asymptotically faster than $\Theta(p)$ may exist (c.f., [MiBo00]). The following algorithm runs in $\Theta(p)$ time.

1. Let S be a set of dummy values distributed 1 per processor. By Theorem 3.3, we can gather S to P_a in $\Theta(p)$ time.
2. In $\Theta(p)$ time, P_a tags each member of S with x.
3. By Theorem 3.3, we scatter S in $\Theta(p)$ time. At the end of this step, every processor has the value of x.

The algorithm given above takes $\Theta(p)$ time. □

Next, we give an algorithm for efficient broadcasting of a small string.

Theorem 4.2. *Let P be a string of m characters in a $CGM(n,p)$, where $m = O(n/p)$. Then every processor can obtain a copy of P in $O(m+p)$ time.*

Proof. We give the following algorithm.

- Gather P into one processor, say, PE_1, in $O(m)$ time.
- Gather one dummy unit of data x_i from PE_i (for all i) into PE_1 in $\Theta(p)$ time.
- For each $p_j \in P$, tag each x_i with p_j and scatter $\{x_i\}_{i=1}^{p}$. Since scattering $\Theta(1)$ data takes $O(p)$ time and we can pipeline the data being scattered, this takes $O(m+p)$ time. □

Both multinode broadcast and total exchange operations can be implemented on a $CGM(n,p)$ in $\Theta(T_{\text{sort}}(p^2,p))$ time when communications operations are sort-based [BMR98]. Depending on a parallel computer's architecture, this could be faster than the $\Theta(p^2)$-time algorithm given below for a $CG(n,p)$; however, note the best upper bound we can give in general is the sequential bound, $T_{\text{sort}}(p^2,p) = O(p^2 \log p)$. Below, we give CGM algorithms for these operations.

Theorem 4.3.

- *Let S be a set distributed one member per processor in a $CGM(n,p)$. Then a multinode broadcast operation, at the end of which every processor has the entire set S, can be performed in $O(p^2)$ time.*
- *Let $\{s_{i,j}\}_{i=1}^{p}$ be a set of p units of data initially stored in processor P_j of a $CGM(n,p)$, for $j \in \{1,\ldots,p\}$. Then a total exchange operation, in which each $s_{i,j}$ is sent to P_i, can be performed in $O(p^2)$ time.*

Proof. We remark that depending on the architecture of the $CGM(n,p)$, algorithms asymptotically faster than $O(p^2)$ may exist. We give algorithms that run in $\Theta(p^2)$ time below.

Suppose S is a set distributed one member per processor in a $CGM(n,p)$; say, the member s_j of S is stored in P_j. We can implement a multinode broadcast via a total exchange operation, *using* $s_{i,j} = s_j$ *for* $i \in \{1,\ldots,p\}$, as described above. Therefore, both assertions will be established when we show a total exchange operation can be implemented in $\Theta(p^2)$ time.

Consider the following algorithm for a total exchange operation.

For $i = 1$ to p, gather $\{s_{i,j}\}_{j=1}^{p}$ to P_i.

This takes $\Theta(p^2)$ time. When the loop finishes, every processor P_i has $\{s_{i,j}\}_{j=1}^{p}$. □

A semigroup operation can be implemented on a $CGM(n, p)$ in $\Theta(\frac{n}{p} + T_{\text{sort}}(p^2, p))$ time when sort-based communications are used [BMR98]. Below, we obtain a faster running time.

Theorem 4.4. *Let $X = \{x_i\}_{i=1}^{n}$ be a set distributed $\Theta(n/p)$ per processor in a $CGM(n, p)$. Then the semigroup computation $r = x_1 \circ x_2 \circ \cdots \circ x_n$ can be performed in optimal $\Theta(n/p)$ time. At the end of this algorithm, every processor holds the value of r.*

Proof. To simplify our presentation, we assume each processor P_j holds $\{x_i\}_{i=\frac{(j-1)n}{p}+1}^{\frac{jn}{p}}$. We give the following algorithm.

1. In parallel, each P_j computes the partial result
$$r_j = x_{\frac{(j-1)n}{p}+1} \circ x_{\frac{(j-1)n}{p}+2} \circ \cdots \circ x_{\frac{jn}{p}}.$$
This takes $\Theta(n/p)$ time.
2. Gather $\{r_j\}_{j=1}^{p}$ into P_1. This takes $\Theta(p)$ time.
3. P_1 computes the desired value,
$$r = r_1 \circ r_2 \circ \cdots \circ r_p,$$
in $\Theta(p)$ time.
4. In $\Theta(p)$ time, P_1 tags each member of $\{r_j\}_{j=1}^{p}$ with the value of r.
5. In $\Theta(p)$ time, scatter the data gathered above. At the end of this step, every processor has r.

Since $p \leqslant n/p$, our algorithm runs in $\Theta(n/p)$ time. This is optimal, since a sequential semigroup operation can be performed in optimal $\Theta(n)$ time. □

A parallel prefix operation can be executed on a $CGM(n, p)$ in $\Theta(\frac{n}{p} + T_{\text{sort}}(p^2, p))$ time when communications are restricted to sort-based operations [BMR98]. Below, we give an optimal parallel prefix algorithm for the $CGM(n, p)$.

Theorem 4.5. *Let $X = \{x_i\}_{i=1}^{n}$ be a set distributed $\Theta(n/p)$ per processor in a $CGM(n, p)$. Then the parallel prefix computation of*

$$x_1,\ x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \cdots \circ x_n$$

can be performed in optimal $\Theta(n/p)$ time. At the end of this algorithm, the processor that holds x_j also holds $x_1 \circ \cdots \circ x_j$.

Proof. To simplify our presentation, we assume each processor P_j holds $\{x_i\}_{i=\frac{(j-1)n}{p}+1}^{\frac{jn}{p}}$. We give the following algorithm.

1. In parallel, each processor P_j computes the partial results
$$s_{j,1} = x_{\frac{(j-1)n}{p}+1}, \quad s_{j,k+1} = s_{j,k} \circ x_{\frac{(j-1)n}{p}+k+1}, \quad k \in \left\{1, 2, \ldots, \frac{n}{p} - 1\right\}.$$
This takes $\Theta(n/p)$ time.
2. In $\Theta(p)$ time, gather $\{s_{j,n/p}\}_{j=1}^{p}$ into one processor, say, P_1.
3. In $\Theta(p)$ time, P_1 uses a sequential prefix algorithm to compute
$$S_1 = s_{1,n/p},\ S_{m+1} = S_m \circ s_{m+1,n/p}, \quad m \in \{1, \ldots, p-1\}.$$
4. P_1 marks each member of $\{s_{j,n/p}\}_{j=2}^{p}$ with the corresponding S_{j-1}. This takes $\Theta(p)$ time.
5. Scatter the data gathered above, so that P_i receives S_{i-1}. This takes $\Theta(p)$ time.
6. In parallel, each processor P_i, $i > 1$, distributes S_{i-1} over its partial results, replacing $s_{i,k}$ with $S_{i-1} \circ s_{i,k}$, $k \in \{1, \ldots, n/p\}$. This takes $\Theta(n/p)$ time. Notice the desired results are the updated values of $s_{i,k}$, $i \in \{1, \ldots, p\}$, $k \in \{1, \ldots, n/p\}$.

Since $p \leqslant n/p$, the algorithm runs in $\Theta(n/p)$ time. This is optimal, since there is an optimal sequential solution with $\Theta(n)$ running time [MiBo00]. □

Our algorithms for (exact or approximate) parallel string matching problems are all based on efficient sequential solutions. This poses the following problem: The pattern P will be tested for (exactly or approximately) matching substrings P' of a text T such that the first character of P' and the last character of P' are not initially stored in the same processor of our $CGM(n, p)$. Thus, we want processors to share segments of T efficiently. We have the following.

Proposition 4.6. *Let T be a text of n letters evenly distributed among the processor of a $CGM(n, p)$ G so that the portion of T in each processor is a substring of T. Let m be a positive integer such that $m = O(n/p)$. Then, in parallel, segments of m entries apiece of T stored in PE_{i+1} can be sent from PE_{i+1} to PE_i, $i \in \{1, \ldots, p-1\}$, in the following time.*

- *In $\Theta(m)$ time, if for all i we have PE_i and PE_{i+1} adjacent.*
- *In $\Theta(T_{\text{sort}}(mp, p))$ time, in general.*

Proof.

- If for all i we have PE_i and PE_{i+1} adjacent, this can be done in $\Theta(m)$ time by having pairs of processors perform parallel gather operations. That is, in parallel, for all even $i < p$, PE_i gathers the desired characters from PE_{i+1};

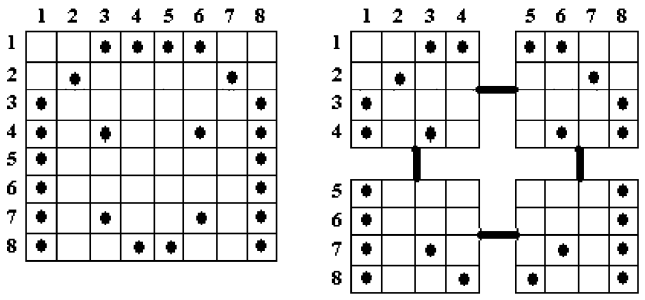

Fig. 1. An 8 × 8 digital picture and its storage on a 2 × 2 mesh. The "border" rows/columns are those numbered 1, 4, 5, and 8.

then, for all odd $i < p$, PE_i gathers the desired characters from PE_{i+1}.

- In general, this can be done via a random access read operation in $\Theta(T_{\text{sort}}(mp, p))$ time. □

5. Image analysis

In this section, we show how the CGM can make use of gather and scatter operations to obtain optimal solutions to a variety of problems in image analysis.

We will assume that an $n^{1/2} \times n^{1/2}$ digital picture is stored so that each processor has a rectangle of dimensions either $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$ (a square) or $\frac{n^{1/2}}{p} \times n^{1/2}$ (each processor storing $\frac{n^{1/2}}{p}$ entire rows of the digital picture). We speak of the first and last rows and columns of these subpictures as *border rows* and *border columns*, respectively, of the subpictures (see Fig. 1).

5.1. Image component labeling

In this section, we give efficient to optimal algorithms to label the foreground components of a binary $n^{1/2} \times n^{1/2}$ digital picture. We assume the digital picture is stored in a $CGM(n, p)$ so that each processor contains a $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$ square section of the digital picture. We assume also that connectedness is determined by 4-adjacency, i.e., two foreground pixels are adjacent if and only if one is North, South, East, or West of the other. Our presentation is easily modified for 8-adjacency to obtain algorithms with the same asymptotic running times.

A solution to the component labeling problem consists of assigning a label to every foreground (i.e., black) pixel such that all members of the same connected component have the same label, and members of distinct components have distinct labels. This may be done as follows.

1. Give each black pixel an initial label corresponding to its unique location in the digital image (e.g., its row-major index or its coordinates—if the latter, coordinates may be ordered lexicographically).
2. Take steps that result in each foreground pixel being assigned the minimum of the initial labels of foreground pixels in the same component.

Below, we give implementations of this strategy for coarse grained parallel computers. Our first algorithm performs optimally for a mesh architecture, but is less efficient on other architectures. Our second algorithm performs optimally on all architectures, but is limited to $p^3 \leqslant n$ rather than our usual restriction of $p^2 \leqslant n$. Both algorithms make use of the following.

Lemma 5.1. *Let X be a binary $n^{1/2} \times n^{1/2}$ digital picture. Let p be an integer, $1 < p \leqslant n^{1/2}$. Suppose X is partitioned into p subpictures, each of dimensions $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$. Suppose the component labeling problem has been solved for each of these subpictures such that local components in different subpictures have distinct component labels, i.e., if C_0 and C_1 are components of distinct subpictures, then C_0 and C_1 have distinct component labels. Let B be the union of the border pixels of the subpictures. Suppose B is stored in a single processor. Let $N = |B|$. Then every member of B can be given its component label relative to X in $\Theta(N)$ time.*

Proof. Since there are p subpictures, each with $4(\frac{n}{p})^{1/2} - 4$ border pixels,

$$N = |B| = \Theta(n^{1/2} p^{1/2}).$$

We treat our problem as a graph component labeling problem. The graph $G = (V, E)$ will be represented by adjacency lists. The vertex set V will be the set of foreground pixels in B, and the edge set E is initially empty. We take care to build E so that its size is at most linear in $|V|$. We give the following algorithm.

1. Create V by scanning B, adding $b \in B$ to V if and only if b is a foreground pixel. This takes $\Theta(N)$ time. Note $|V| = O(N)$.
2. Sort V by the local component labels of the pixel records. Since there are $\Theta(N)$ possible labels, this may be done via the Binsort algorithm in $\Theta(N)$ time.
3. Scan the sorted list so that if successive members v_i and v_{i+1} have the same label, then each of these vertices is added to the other's adjacency list. Thus, $O(N)$ edges are created in this step. Notice that every pair of vertices representing foreground border pixels with the same local component label is connected by a path in G. This step takes $O(N)$ time.
4. Sort V by the x-coordinates of the pixels as the primary key, using the y-coordinates of the pixels as the secondary key. Since there are $\Theta(N)$ pairs (x, y) under consideration, this may be done via the Binsort algorithm in $\Theta(N)$ time.
5. Adjacent border pixels with equal x-coordinates from distinct subpictures are now successive elements of V. Scan the sorted list V so that if successive members v_i and v_{i+1} are neighboring pixels from distinct subpic-

tures, then each of these vertices is added to the other's adjacency list. Thus, $O(N)$ edges are created in this step. This takes $O(N)$ time.

6. Repeat the previous two steps with the roles of the x- and y-coordinates interchanged, so that adjacent border pixels with equal y-coordinates from distinct subpictures appear in each other's adjacency lists. Thus, $O(N)$ edges are created in this step. This takes $\Theta(N)$ time.
7. Solve the component labeling problem for the graph G. Since $|E| = O(N)$, the time required is [MiBo00]

$$\Theta(|V| + |E|) = O(N).$$

This algorithm uses $\Theta(N)$ time. □

The special case of the next result is stated for a mesh architecture. Notice that it could be implemented on other architectures that have mesh-like connections, e.g., PRAM, hypercube, pyramid. The significance of the mesh architecture is that it permits adjacent subpictures of a digital image to be stored in adjacent processors; this allows us to perform efficient gather and scatter operations within rows and columns of processors in the mesh. We say a digital picture X is *naturally mapped into a mesh* if

- for each row r, if x_{ru} and x_{rv} are stored, respectively, in processors P_{ab} and P_{cd}, then $a = c$, and $u < v$ implies $b \leqslant d$; and
- for each column c, if x_{pc} and x_{qc} are stored, respectively, in processors P_{ab} and P_{cd}, then $b = d$, and $p < q$ implies $a \leqslant c$.

Theorem 5.2. *Let X be a binary $n^{1/2} \times n^{1/2}$ digital picture, viewed as a black image on a white background, in a $CGM(n, p)$ G. Suppose each processor holds a $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$ subpicture of X. Then the connected components of the black pixels can be uniquely labeled as follows*:

- *In $\Theta(\frac{n}{p} + p^{1/2} T_{\text{sort}}(n^{1/2} p^{1/2}, p))$ time, in general.*
- *In $\Theta(n/p)$ time, if G is a mesh and X is naturally mapped into G.*

Proof. The significance of the mesh is that it permits us to replace sort-based concurrent read operations (used in the general case) by more efficient parallel gather operations. We cannot use gather operations in general, as a row or a column of the digital picture from which we will want to accumulate data may not be stored in a connected set of processors in G.

We give the following algorithm.

1. In parallel, every processor solves the component labeling problem for the square section of the binary image stored by the processor. This takes $\Theta(n/p)$ time [MiBo00].
2. *For $\Theta(p^{1/2})$ iterations, do the following.*
   - Pixels on the border of their processor's rectangle (i.e., pixels with an adjacent pixel in the binary image that is stored in a different processor) exchange labels (if any) with the (at most 2, for pixels in a corner of their processor's rectangle) adjacent border pixel(s) stored in different processors. Notice there are $4(\frac{n}{p})^{1/2} - 4$ border pixels per processor, so the number of pixels involved in this operation is less than $4n^{1/2} p^{1/2}$. An exchange of labels between pairs of adjacent foreground pixels stored in distinct processors can be implemented as follows.
     - In the general case, this may be done via a concurrent read operation, using the algorithm of Proposition 2.1, in $\Theta(T_{\text{sort}}(n^{1/2} p^{1/2}, p))$ time.
     - If G is a mesh and X is naturally mapped into G, we proceed as follows. In parallel, adjacent pairs of processors perform gather operations, so that each gathers the other's adjacent border pixels. Since a processor may have pixels adjacent to pixels stored in 4 other processors, we perform 8 rounds of parallel gather operations. Since the length of the adjacent border edges is $(\frac{n}{p})^{1/2}$, this takes $O((\frac{n}{p})^{1/2})$ time.
   - In parallel, each processor uses the algorithm of Lemma 5.1 to compute, for each border pixel stored in the processor, the pixel's correct label with respect to the union of the pixel's processor's subpicture and the adjacent subpictures. Since each processor has $\Theta((\frac{n}{p})^{1/2})$ border pixels, the time for this step is $\Theta((\frac{n}{p})^{1/2})$.

   End for

   The time used by the loop is as follows:
   - In general,

$$\begin{aligned}\Theta\left(p^{1/2}\left(\frac{n}{p}\right)^{1/2} + p^{1/2} T_{\text{sort}}(n^{1/2} p^{1/2}, p)\right) &= \Theta(n^{1/2} + p^{1/2} T_{\text{sort}}(n^{1/2} p^{1/2}, p)) \\ &= O\left(\frac{n}{p} + p^{1/2} T_{\text{sort}}(n^{1/2} p^{1/2}, p)\right).\end{aligned}$$

   - $\Theta\left(\left(\frac{n}{p}\right)^{1/2} p^{1/2}\right) = \Theta(n^{1/2}) = O(n/p)$, when X is naturally mapped into a mesh G.

   The loop uses $\Theta(p^{1/2})$ iterations, as each iteration of the loop increases by one or two the number of processor-blocks of rows and columns of pixels (in the mesh, the number of rows and columns of processors) with respect to the union of whose subpictures each border pixel is correctly labeled. Therefore, when we exit the loop, the members of the set B of subpicture border pixels have their correct component labels with respect to the entire digital picture.
3. In parallel, each processor P_j solves the component labeling problem for the $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$ subpicture stored by P_j. This takes $\Theta(n/p)$ time. Now, the entire digital picture has its components labeled.

The running time of the algorithm is as follows:

- $\Theta(\frac{n}{p} + p^{1/2} T_{\text{sort}}(n^{1/2} p^{1/2}, p))$, in general.
- $\Theta(n/p)$, if X is naturally mapped into a mesh G. □

The time of the algorithm above for the digital picture mapped naturally to the mesh, $\Theta(n/p)$, is optimal, since the sequential version of the problem has an optimal $\Theta(n)$-time solution [MiBo00]. The time for the general case is optimal when $p^{1/2}\,T_{\text{sort}}(n^{1/2}p^{1/2}, p) = O(n/p)$. Since our general upper bound for sorting time is the sequential bound, it follows that our general case running time is optimal if $p^4 = O(\frac{n}{\log^2 n})$.

A different approach yields an asymptotically optimal algorithm for the $CGM(n, p)$ for the larger range of processors $1 < p^3 \leqslant n$, without the requirement of naturally mapping to a mesh architecture. The assumption $p^3 \leqslant n$ implies $n^{1/2}p^{1/2} \leqslant n/p$, so a set of size $O(n^{1/2}p^{1/2})$ can be gathered into one processor. The number N of foreground border pixels satisfies

$$N \leqslant \left[4\left(\frac{n}{p}\right)^{1/2} - 4\right]p = \Theta(n^{1/2}p^{1/2}) = O(n/p). \quad (6)$$

Theorem 5.3. *Suppose we have $p^3 \leqslant n$. Given an $n^{1/2} \times n^{1/2}$ digitized binary picture, viewed as a black image on a white background, the connected components of the black pixels can be uniquely labeled on a $CGM(n, p)$ in optimal $\Theta(n/p)$ time.*

Proof. We give the following algorithm.

1. In parallel, every processor solves the component labeling problem for the $(\frac{n}{p})^{1/2} \times (\frac{n}{p})^{1/2}$ section of the binary image stored by the processor, so that local components in distinct processors get distinct labels. This should be done so that every foreground pixel that is on the border of its processor's rectangle gets a local component label that is a nonnegative integer less than $4n/p$. This is possible, in light of inequality (6). This takes $\Theta(n/p)$ time [MiBo00].
2. By inequality (6), we can gather the foreground pixels that are on the border of their processor's subpicture, with their local component labels, into one processor, say, P_1. This takes $\Theta(N)$ time.
3. Use the algorithm of Lemma 5.1 to obtain, in P_1, the correct component labels of every processor's border pixels in $\Theta(N)$ time.
4. In $\Theta(N)$ time, scatter the border pixels from P_1 with their correct component labels.
5. In parallel, each processor P_i now solves its local component labeling problem in $O(n/p)$ time.

It follows from inequality (6) that our algorithm uses $\Theta(n/p)$ time, which is optimal, since the problem can be solved sequentially in optimal $\Theta(n)$ time [MiBo00]. □

5.2. Convex hull

In this section, we give a solution to the problem of computing the convex hull of a binary image. That is, we wish to mark those black pixels that represent the extreme points of the convex hull of the set of black pixels in a binary picture. Note the set of black pixels need not be connected.

Theorem 5.4. *The convex hull of an $n^{1/2} \times n^{1/2}$ binary digital picture, stored in row-major fashion, $\Theta(\frac{n^{1/2}}{p})$ rows per processor, can be computed on a $CGM(n, p)$ in $O(n/p)$ time, which is optimal.*

Proof. We give the following algorithm.

1. In parallel, every processor uses a simple scan operation to find the leftmost and rightmost black pixels (if they exist) for each of the $\Theta(\frac{n^{1/2}}{p})$ rows stored in the processor. These are the only pixels in their respective rows that can be extreme points of the convex hull. This step takes $O(n/p)$ time.
2. Note the set S of row extrema has at most $2n^{1/2} = O(n/p)$ points in the entire image. Therefore, we can gather the set S into one processor, say, P_1, in $O(n^{1/2})$ time.
3. In $O(n^{1/2})$ time, P_1 can compute the convex hull from S as follows.

- Use the BinSort algorithm [MiBo00] to sort the row extrema by row in $O(n^{1/2})$ time.
- Use the post-sorting steps of the Graham sweep algorithm [Grah72,MiBo00], starting with the right lowest row extreme point and proceeding through the right extrema, then back through the left extrema, to solve the Convex Hull problem for these row extrema in an additional $O(n^{1/2})$ time. A row extreme point that is the only extreme point in its row is not eliminated as a candidate for a hull extreme point unless it is eliminated as both a right candidate and a left candidate; as described in [Grah72,MiBo00], a right or left candidate is eliminated from consideration as a hull extreme point if the path from the predecessor candidate to the current candidate to the successor candidate fails to turn leftward at the current candidate.

4. Scatter the data gathered above so that every member of S is returned to its original processor with knowledge as to whether or not it is a vertex of the convex hull, in $O(n^{1/2})$ time.

Since $n^{1/2} = O(n/p)$, the running time of this algorithm is $O(n/p)$. This is optimal, since the image convex hull problem has an optimal $O(n)$-time sequential solution [MiBo00]. □

5.3. Distance problems

We assume in this section that the distance function d for our grid of pixels is the L_1 metric d_1, defined for a pair of points in the Euclidean plane by

$$d_1((x_0, y_0), (x_1, y_1)) = |x_0 - x_1| + |y_0 - y_1|.$$

We have the following properties of the L_1 metric.

Lemma 5.5 (Shonkwiler [Shon89]). *Let y be a black pixel in a binary digital picture. Let x_0, x_1 be pixels (not necessarily black) in the digital picture. Suppose x_0 and y are in the same row r_0 and x_0 and x_1 are in the same column. Using the metric d_1, if y is a closest black pixel to x_0 among the pixels in row r_0, then y is a closest black pixel to x_1 among the pixels in row r_0.*

Corollary 5.6 (Corrigendum to [BxMi00]). *Consider a range of row indices*

$$R = \{r \mid r_{\min} \leqslant r \leqslant r_{\max}\}$$

for pixels of a binary digital picture X. Let y be a black pixel of X with row index in R. Let x_0, x_1 be pixels of X (not necessarily black). Suppose x_0 has row index in R, x_0 and x_1 are in the same column, and x_1 has row index $r_1 \notin R$. Using the metric d_1, if x_0 is a closest pixel to x_1 among the pixels with row index in R (that is, if x_1 has row index less than $r_{\min}$ then x_0 has row index $r_{\min}$; otherwise, x_0 has row index $r_{\max}$) and y is a closest black pixel to x_0 among the pixels with row index in R, then y is a closest black pixel to x_1 among the pixels with row index in R.

5.3.1. L_1 distance transform

A distance transform (DT) [RoPf68] computes, for each pixel (black or white) of a binary digital picture, the distance to a nearest black pixel. Applications of this operation are listed in [BxMi00].

In this section, we give an algorithm for computing the L_1 DT on a mesh $CGM(n, p)$.

Theorem 5.7. *Consider an $n^{1/2} \times n^{1/2}$ binary digital picture X mapped naturally to a mesh $CGM(n, p)$ G such that every processor stores a square subpicture of $(n/p)^{1/2} \times (n/p)^{1/2}$ pixels of X. Then the L_1 distance transform for X can be computed in $\Theta(n/p)$ time, which is optimal.*

Proof. We give the following algorithm.

1. Each pixel x of X learns a nearest black pixel (if one exists) of X in the segment of its row stored in the same processor. This is done as follows.
   In parallel, each processor P does the following.
   (a) P performs a prefix operation so that every pixel x of X learns a nearest black pixel (if one exists) not to its right, in the same row and processor. This takes $\Theta(n/p)$ time.
   (b) P performs a postfix operation so that every pixel x of X learns a nearest black pixel (if one exists) not to its left, in the same row and processor. This takes $\Theta(n/p)$ time.
   (c) In P, each pixel compares its nearest not-right and its nearest not-left black pixel to find a nearest black pixel in its row and processor. This takes $\Theta(n/p)$ time.
   End parallel
2. Every pixel x_0 learns a nearest black pixel of X stored in the same row as x_0 (if any), over all processors. A nearest black pixel in the same row to x_0 is either a nearest black pixel in the same row and processor, or else a nearest black pixel to a processor-border pixel in the same row but in a different processor. We achieve this as follows.
   (a) Since $n^{1/2} \leqslant n/p$, $O(n^{1/2})$ data can be gathered into one processor. In parallel, each row of processors performs a gather operation to collect all the column border pixels of the subpictures stored by this row of processors. Each processor has at most $2(\frac{n}{p})^{1/2}$ foreground border column pixels, so each row of $p^{1/2}$ processors has $O(n^{1/2})$ foreground border column pixels that are gathered into one processor of the block in $O(n^{1/2})$ time.
   (b) By parallel sweep operations, the column border pixels gathered into one processor learn nearest not-left and nearest not-right foreground pixels in their respective rows, in $O(n^{1/2})$ time.
   (c) In parallel scatter operations, each row of processors sends the column border pixels to their original processors. This takes $O(n^{1/2})$ time.
   (d) By simple sweep operations, the column border pixels distribute their closest not-left and not-right black pixels to all pixels in the same row that are stored in the same processor, so each pixel can determine a closest black pixel in the same row (by comparing any previously marked closest foreground point in the segment of the row stored in the same processor with the at most 4 closest points received from the column border pixels). This takes $\Theta(n/p)$ time.
3. Every pixel x_0 learns a nearest black pixel over all rows of X in the same row of processors as x_0. By Lemma 5.5, this may be achieved by comparing the nearest black pixel in the row of x_0 with all nearest black pixels to x_1 in the row of x_1, for all x_1 in the same column and processor as x_0. This is done by performing prefix and postfix computations over every segment of a column stored in the same processor, for all columns. At the end of this step, every pixel x_0 knows a nearest black pixel of X (if any) stored in the same row of processors as x_0. This takes $\Theta(n/p)$ time.
4. Mimic the steps above, interchanging the roles of rows and columns, to distribute data within columns so that, by Corollary 5.6, every pixel knows a nearest black pixel in X. This takes $\Theta(n/p)$ time.

Since $n^{1/2} \leqslant n/p$, the algorithm takes $\Theta(n/p)$ time. This is optimal, since the DT can be computed in optimal sequential $\Theta(n)$ time (*Corrigendum* to [BxMi00]). □

5.3.2. Hausdorff Metric

The *Hausdorff metric* measures how well two compact nonempty subsets of a metric space S approximate each other's location in S. Although two sets may have similar location and yet have very different geometric properties, the fact that the Hausdorff metric is often efficiently computed has led many researchers to consider it as a tool for computational geometry and image analysis [ABB91,Atal83,Boxe97,BCMR91,BMR98,ChR96, CK92,Doug90,HuK90,HuKK92,PuRa81,Rote91,Shon89].

Let $d(a, b)$ be a metric for S. We abuse notation and write

$$d(z, A) = \min\{d(z, a) \mid a \in A\}.$$

The "non-symmetric" or "one-way" Hausdorff measure is

$$H^*(A, B) = \max_{a \in A} d(a, B).$$

The Hausdorff metric $H(A, B)$ is defined [Nadl78] by

$$H(A, B) = \max\{H^*(A, B), H^*(B, A)\}.$$

It is easily seen that computing the distance transform is a useful tool for computing the Hausdorff metric. We have the following.

Theorem 5.8 (Shonkwiler [Shon89])**.** *Let X and Y be $n^{1/2} \times n^{1/2}$ binary digital pictures. Then $H(X, Y)$ can be computed with respect to the L_1 metric in serial $\Theta(n)$ time.*

We give a CGM algorithm to compute the Hausdorff metric for two $n^{1/2} \times n^{1/2}$ binary digital pictures X and Y.

Theorem 5.9. *Let X and Y be $n^{1/2} \times n^{1/2}$ binary digital pictures, each mapped naturally to a mesh $CGM(n, p)$ such that every processor has corresponding square subpictures of $(n/p)^{1/2} \times (n/p)^{1/2}$ pixels of both X and Y. The Hausdorff distance $H(X, Y)$ between $n^{1/2} \times n^{1/2}$ binary digital pictures X and Y can be computed with respect to the L_1 metric in $\Theta(n/p)$ time, which is optimal.*

Proof. We give the following algorithm.

1. Compute the L_1 distance transform so that every black pixel x of X learns a nearest black pixel of Y. By Theorem 5.7, this takes $\Theta(n/p)$ time.
2. Perform a semigroup (maximum) operation to find $H^*(X, Y)$, the maximum over all black pixels x of X of the distance from x to a nearest black pixel of Y. By Theorem 4.4, this takes $\Theta(n/p)$ time. At the end of this operation, every processor has the value of $H^*(X, Y)$.
3. Repeat the operations above with the roles of X and Y interchanged to find $H^*(Y, X)$ in $\Theta(n/p)$ time. At the end of these operations, every processor has the values of $H^*(X, Y)$ and $H^*(Y, X)$.
4. In $\Theta(1)$ time, every processor computes

$$H(X, Y) = \max\{H^*(X, Y), H^*(Y, X)\}.$$

Thus, the algorithm uses $\Theta(n/p)$ time. This is optimal, in light of the optimal sequential result of Theorem 5.8. □

6. Pattern matching problems

Problems discussed in this section take the following form. Given two strings P and T (respectively, known as the *pattern* and the *text*) such that

$$|P| = m \leqslant n = |T| \tag{7}$$

find every instance of a copy (exact or approximate) of P in T. Note that [Gusf97] uses

$$|P| = n \leqslant m = |T|.$$

We prefer the notation of statement (7) in order to preserve the convention that the volume of input to the problem is $\Theta(n)$.

We have not obtained solutions for the full range $1 \leqslant m \leqslant n$. Although it is theoretically desirable to do so, note that typical applications are in search operations of word processors and other popular software applications, and in molecular biologists' efforts to recognize strands of DNA in a genome. In such applications, typically $m \ll n$. In the following, we therefore assume that $m = O(n/p)$.

6.1. Exact matching

In this section, we give a solution to the exact matching problem for coarse grained parallel computers. This is the pattern matching problem as described above for which we require exact copies of P in T. For sequential computers, a naive algorithm solves this problem in $O(m(n - m))$ time [Gusf97]. However, this is far from optimal. There exist solutions (including those known as the *Knuth–Morris–Pratt*, *Boyer–Moore*, and *Apostolico–Giancarlo* algorithms), that solve the problem in optimal $\Theta(n)$ time [ApGi86,BoyM77,Gusf97,KMP77]. We have the following.

Theorem 6.1. *Let T be a text of n letters evenly distributed among the processors of a $CGM(n, p)$ G so that the portion of T in each processor is a substring of T. Let P be a pattern of m letters, where $m = O(n/p)$. Then every exact copy of P in T can be identified in a running time satisfying the following.*

- *In optimal $\Theta(\frac{n}{p})$ time, if for all i we have PE_i and PE_{i+1} adjacent.*
- *In general, in $O(\frac{n}{p} + T_{sort}(mp, p))$ time.*

Proof. We give the following algorithm. To simplify the exposition, we assume processor PE_i has the substring $T[\frac{(i-1)n}{p}+1,\ldots,\frac{in}{p}]$ of $T=T[1,\ldots,n]$, $i\in\{1,\ldots,p\}$.

1. Broadcast the pattern P so that every processor has a copy of P. By Theorem 4.2, this takes $O(m+p)=O(n/p)$ time.
2. Perhaps a copy of P starts at an entry of T stored in PE_i and ends in PE_{i+1}. Use the algorithm of Proposition 4.6 to send the $m-1$ entries $T[\frac{(i+1)n}{p}+1,\ldots,\frac{(i+1)n}{p}+m-1]$ of T from PE_{i+1} to PE_i, $i\in\{1,\ldots,p-1\}$.
   - If for all i we have PE_i and PE_{i+1} adjacent, this takes $\Theta(m)$ time.
   - More generally, this takes $\Theta(T_{\text{sort}}(mp,p))$ time.

   At the end of this step, processor PE_i has the substring
   $$S_i = T\left[\frac{(i-1)n}{p}+1,\ldots,\frac{in}{p}+m-1\right],\quad i\in\{1,\ldots,p-1\};$$
   PE_p has $S_p = T[\frac{(p-1)n}{p}+1,\ldots,n]$.
3. In parallel, each processor PE_i applies a linear-time sequential algorithm to the text S_i and the pattern P. This takes $\Theta(\frac{n}{p})$ time.

We summarize the running time of our algorithm as follows.

- If for all i we have PE_i and PE_{i+1} adjacent, the algorithm runs in optimal $\Theta(\frac{n}{p})$ time.
- In general, the algorithm uses $\Theta(\frac{n}{p}+T_{\text{sort}}(mp,p))$ time. This reduces to optimal $\Theta(\frac{n}{p})$ time if $T_{\text{sort}}(mp,p)=O(\frac{n}{p})$. Since the best general upper bound for sorting time is the sequential bound, it follows that the general case has optimal running time when $mp^2\log mp = O(n)$. □

6.2. Approximate string matching

Given integers k,m,n such that $0\leqslant k\leqslant m\leqslant n$, a pattern P of size $m>0$, and a text T of size $n\geqslant m$, it is often useful to allow k mismatches in the string matching problem. That is, we seek all substrings P' of T such that $|P'|=m$ and P' is a copy of P except for at most k mismatched characters. This is the *k-approximate matching problem*. Note the k-approximate matching problem generalizes the exact matching problem; for $k=0$, the two problems are identical.

The problem of finding approximate matches is closely related to the *match-count problem*, in which, for every substring T' of T such that $|T'|=m$, we find the number of characters of T' that match the corresponding character of P. There is a sequential algorithm for the match-count problem based on the fast Fourier transform (FFT) [Gusf97]. The algorithm runs in $T_{\text{seq}} = O(n\log n)$ time. We give an efficient parallel solution below.

Theorem 6.2. *Suppose T is a text of n characters distributed evenly in a $CGM(n,p)$ G, and P is a pattern of size $m=O(n/p)$ stored in G. The match-count problem can be solved in G as follows*:

- *In $O(\frac{n\log n}{p})$ time, if for all i we have PE_i and PE_{i+1} adjacent.*
- *In $O(\frac{n\log n}{p}+T_{\text{sort}}(mp,p))$ time, in general.*

Proof. We give the following algorithm.

1. Broadcast P so every processor has a copy. By Theorem 4.2, this takes $O(m+p)=O(n/p)$ time.
2. Send $T[\frac{(i+1)n}{p}+1,\ldots,\frac{(i+1)n}{p}+m-1]$ from PE_{i+1} to PE_i so PE_i has $T[\frac{in}{p}+1,\ldots,\frac{(i+1)n}{p}+m-1]$. By Proposition 4.6, this takes $\Theta(m)=O(n/p)$ time, if for all i we have PE_i and PE_{i+1} adjacent. In the general case, the time required is $\Theta(T_{\text{sort}}(mp,p))$.
3. In parallel, each PE_i solves the match-count problem for P and $T[\frac{in}{p}+1,\ldots,\frac{(i+1)n}{p}+m-1]$. The time for this step is $O(\frac{n\log(n/p)}{p})$, which simplifies as $O(\frac{n\log n}{p})$, since $p=O(n^{1/2})$.

It follows that the running time of our algorithm is as claimed above. □

Notice that our algorithm for the match count problem achieves the goal of Eq. (1) if for all i we have PE_i and PE_{i+1} adjacent; also, in the general case, if $T_{\text{sort}}(mp,p)=O(\frac{n\log n}{p})$.

Corollary 6.3. *For integers m,n, pattern P, and text T as described above, a $CGM(n,p)$ can solve the k-approximate matching problem for any k satisfying $0\leqslant k\leqslant m$ as follows.*

- *In $O(\frac{n\log n}{p})$ time, if for all i we have PE_i and PE_{i+1} adjacent.*
- *In $O(\frac{n\log n}{p}+T_{\text{sort}}(mp,p))$ time, in general.*

Proof. We give the following algorithm:

1. Solve the match-count problem, using the algorithm of Theorem 6.2.
2. For each $matches[i]$, compute
   $$mismatches[i] = m - matches[i].$$
   This takes $O(n/p)$ time. Note for each index i, the substring of T of length m starting at $T[i]$ is a k-approximate match for P if and only if $mismatches[i]\leqslant k$.

Since the running time of this algorithm is dominated by the first step, the assertion follows. □

6.3. String matching with differences

More general that the approximate string matching problem is the problem of *string matching with k differences*, described in [LaVi88]. In this problem, a pattern P of length m, a text T of length n, and an integer $k\geqslant 0$ are input. Output consists of identification of all substrings P' of T such that

P' matches P with at most k differences of the following kinds:

- A character of P corresponds to a different character of $P' \subset T$. A difference of this kind is a *mismatch* between the corresponding characters.
- A character of P corresponds to no character of P'. A difference of this kind is an *insertion*.
- A character of P' corresponds to no character of P. A difference of this kind is a *deletion*.

Since the *edit distance* between strings P and P' is the minimum number of character substitutions, insertions, or deletions necessary to transform P to P' [Gusf97], our problem is to find all substrings P' of T such that the edit distance between P and P' is at most k. A sequential algorithm for this problem is given in [LaVi88] with running time $O(m+nk^2)$, assuming an alphabet of fixed size.

We have the following.

Theorem 6.4. *Let T be a text of n characters on an alphabet of fixed size, distributed evenly among the processors of a $CGM(n, p)$ G. Let P be a pattern of m characters stored in G, where $m = O(n/p)$. Let k be an integer such that $0 \leqslant k \leqslant m$. Let T_{seq} be the running time of the sequential algorithm of [LaVi88], i.e., $T_{\text{seq}} = O(m + nk^2)$. Then the string matching with k differences problem can be solved in the following time.*

- $\Theta(\frac{T_{\text{seq}}}{p})$, *if for all i we have PE_i and PE_{i+1} adjacent.*
- $\Theta(\frac{T_{\text{seq}}}{p} + T_{\text{sort}}(mp, p))$, *generally.*

Proof. Note since $m = O(n/p)$, $T_{\text{seq}} = O(nk^2)$. We give the following algorithm.

1. Broadcast P so every processor has a copy. By Theorem 4.2, this takes $O(m + p) = O(n/p)$ time.
2. Send $T[\frac{(i+1)n}{p}+1, \ldots, \frac{(i+1)n}{p}+m-1+k]$ from PE_{i+1} to PE_i so PE_i has $T[\frac{in}{p}+1, \ldots, \frac{(i+1)n}{p}+m-1+k]$ for $i < p$. Since the edit distance between two strings is at least the difference of the strings' lengths, PE_i now has every substring of T starting in $T[\frac{in}{p}+1, \ldots, \frac{(i+1)n}{p}]$ that can match P with at most k differences. Since $k \leqslant m$, by Proposition 4.6, this takes $\Theta(m) = O(n/p)$ time, if for all i we have PE_i and PE_{i+1} adjacent. In the general case, the time required is $\Theta(T_{\text{sort}}(mp, p))$.
3. In parallel, each processor PE_i solves the problem for the text $T[\frac{(i-1)n}{p}+1, \ldots, \frac{in}{p}+m-1+k]$ and the pattern P. This takes $\Theta(\frac{T_{\text{seq}}}{p})$ time.

Clearly, the running time of the algorithm is as claimed above. □

7. Further remarks

We have shown that the $CGM(n, p)$ yields efficient to optimal algorithms for data movement and computational problems, fundamental problems for binary digital pictures, and important string matching problems.

Acknowledgments

We are grateful to the anonymous referees for helpful suggestions.

References

[ABB91] H. Alt, B. Behrends, J. Blömer, Approximate matching of polygonal shapes, in: Proceedings ACM Symposium on Computational Geometry,1991, pp. 186–193.

[ApGi86] A. Apostolico, R. Giancarlo, The Boyer–Moore–Galil string searching strategies revisited, SIAM J. Comput. 15 (1986) 98–105.

[Atal83] M.J. Atallah, A linear time algorithm for the Hausdorff distance between convex polygons, Inform. Process. Lett. 17 (1983) 207–209.

[Boxe97] L. Boxer, On Hausdorff-like metrics for fuzzy sets, Pattern Recognition Lett. 18 (1997) 115–118.

[BxHr01] L. Boxer, R. Haralick, Even faster point set pattern matching in 3-d, Technical Report 2001-01, SUNY, Buffalo Department of Computer Science and Engineering.

[BCMR91] L. Boxer, C-S. Chang, R. Miller, A. Rau-Chaplin, Polygonal approximation by boundary reduction, Pattern Recognition Lett. 14 (1991) 111–119.

[BxMi00] L. Boxer, R. Miller, Efficient computation of the Euclidean distance transform, Comput. Vision Image Understanding 80 (2000) 379–383 Corrigendum: Comput. Vision Image Understanding 86 (2002) 137–140.

[BMR98] L. Boxer, R. Miller, A. Rau-Chaplin, Scaleable parallel algorithms for lower envelopes with applications, J. Parallel Distributed Comput. 53 (1998) 91–118.

[BMR99] L. Boxer, R. Miller, A. Rau-Chaplin, Scalable parallel algorithms for geometric pattern recognition, J. Parallel Distributed Comput. 58 (1999) 466–486.

[BoyM77] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (1977) 762–772.

[ChR96] B.B. Chaudhuri, A. Rosenfeld, On a metric distance between fuzzy sets, Pattern Recognition Lett. 17 (1996) 1157–1160.

[CK92] L.P. Chew, K. Kedem, Improvements on geometric pattern matching problems, Proceedings Scandinavian Workshop on Algorithm Theory, 1992.

[CKPSSSSE] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation, in: Proceedings of the 4th ACM SIGPLAN Symposium on Principles of Parallel Programming,1993.

[Dehn99] F. Dehne (Ed.), Coarse Grained Parallel Algorithms, Algorithmica, special edition, vol. 24, no. 3/4, July/August, 1999.

[DFR93] F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable parallel geometric algorithms for multicomputers, in: Proceedings of the 9th ACM Symposium on Computational Geometry,1993, pp. 298–307.

[DDDFK95] F. Dehne, X. Deng, P. Dymond, A. Fabri, A. Khokhar, A randomized parallel 3D convex hull algorithm for coarse grained multicomputers, in: Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures,1995, pp. 27–33.

[Doug90] E.R. Dougherty, Hausdorff-metric interpretation of convergence in the Matheron topology for binary mathematical morphology, IEEE Proc. (1990) 870–875.

[FKRU99] A. Ferreira, C. Kenyon, A. Rau-Chaplin, S. Ubéda, *d*-dimensional range search on multicomputers, Algorithmica 24 (1999) 195–208.

[FRU95] A. Ferreira, A. Rau-Chaplin, S. Ubéda, Scalable 2d convex hull and triangulation algorithms for coarse grained multicomputers, in: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing,1995, pp. 561–569.

[Gusf97] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, Cambridge, 1997.

[Grah72] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, Inform. Process. Lett. 1 (1972) 132–133.

[HaK93] S. Hambrusch, A. Khokhar, C3: an architecture-independent model for coarse-grained parallel machines, Computer Sciences Technical Report CSD-TR-93-080, Purdue University, 1993.

[HuK90] D.P. Huttenlocher, K. Kedem, Computing the minimum Hausdorff distance for point sets under translation, in: Proceedings of the 6th Annual Symposium on Computational Geometry,1990, pp. 340–349.

[HuKK92] D.P. Huttenlocher, K. Kedem, J.M. Kleinberg, On dynamic Voronoi diagrams and the minimum Hausdorff distance for point sets under Euclidean motion in the plane, Technical Report TR 92-1271, Department of Computer Science, Cornell University, 1992.

[KMP77] D.E. Knuth, J.H. Morris, V.B. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.

[LaVi88] G.M. Landau, U. Vishkin, Fast string matching with *k* differences, J. Comput. System Sci. 37 (1988) 63–78.

[MiBo00] R. Miller, L. Boxer, Algorithms Sequential & Parallel: A Unified Approach, Prentice-Hall, Upper Saddle River, NJ, 2000.

[MiSt96] R. Miller, Q.F. Stout, Parallel Algorithms for Regular Architectures: Meshes and Pyramids, The MIT Press, Cambridge, MA, 1996.

[MoSo01] H. Mongelli, S.W. Song, Parallel pattern matching with scaling, Parallel Process. Lett. 11 (2001) 125–138.

[Nadl78] S.B. Nadler Jr., Hyperspaces of Sets, Marcel Dekker, New York, 1978.

[PuRa81] M.L. Puri, D.A. Ralescu, Différentielle d'une fonction floue, C. R. Acad. Sci. Paris Sér. I 293 (1981) 237–239.

[RoPf68] A. Rosenfeld, J.L. Pfalz, Distance function on digital pictures, Pattern Recognition Lett. 1 (1968) 33–61.

[Rote91] G. Röte, Computing the minimum Hausdorff distance between two point sets on a line under translation, Inform. Process. Lett. 32 (1991) 123–127.

[SaSo99] E.L.G. Saukas, S.W. Song, A note on parallel selection on coarse-grained multicomputers, Algorithmica 24 (1999) 371–380.

[Shon89] R. Shonkwiler, An image algorithm for computing the Hausdorff distance efficiently in linear time, Inform. Process. Lett. 30 (1989) 87–89.

[Vali90] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (1990) 103–111.

Laurence Boxer is Professor and Chair of Computer and Information Sciences at Niagara University, and Research Professor of Computer Science and Engineering at the State University of New York at Buffalo. He received his Ph.D. in Mathematics from the University of Illinois at Urbana-Champaign. His research interests are computational geometry, parallel algorithms, and digital topology.

Russ Miller is Distinguished Professor of Computer Science and Engineering, and Director of the Center for Computational Research, at the State University of New York at Buffalo; and senior research scientist at the Hauptman-Woodward Medical Research Institute. He received his Ph.D. in Mathematical Sciences from the State University of New York at Binghamton. His research interests include parallel algorithms, image processing, computational crystallography, and parallel processing education. Dr. Miller is on the editorial board of *Parallel Processing Letters*, has served on program committees of many conferences concerning parallelism and image processing, and is on the executive committee of the IEEE Technical Committee on Parallel Processing.