Linköping Electronic Articles in
Computer and Information Science
Vol. 2(1997): nr 3

Parallel Algorithms for Batched Range Searching on Coarse-Grained Multicomputers

Per-Olof Fjällström

Department of Computer and Information Science
Linköping University
Linköping, Sweden

This work has been submitted for publication elsewhere.
Copyright may then be transferred,
and the present version of the article may be superseded by a revised one.
The WWW page at the URL stated below will contain up-to-date information
about the current version and copyright status of this article. Additional
copyright information is found on the next page of this document.

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1997/003/

Published on April 1, 1997 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

©*1997 Per-Olof Fjällström*
Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in
Computer and Information Science, Vol. 2(1997): nr 3.
http://www.ep.liu.se/ea/cis/1997/003/. *April 1, 1997.*

This URL will also contain a link to the author's home page.

The publishers will keep this article on-line on the Internet
(or its possible replacement network in the future)
for a period of 25 years from the date of publication,
barring exceptional circumstances as described separately.

The on-line availability of the article implies
a permanent permission for anyone to read the article on-line,
and to print out single copies of it for personal use.
This permission can not be revoked by subsequent
transfers of copyright. All other uses of the article,
including for making copies for classroom use,
are conditional on the consent of the copyright owner.

The publication of the article on the date stated above
included also the production of a limited number of copies
on paper, which were archived in Swedish university libraries
like all other written works published in Sweden.
The publisher has taken technical and administrative measures
to assure that the on-line version of the article will be
permanently accessible using the URL stated above,
unchanged, and permanently equal to the archived printed copies
at least until the expiration of the publication period.

For additional information about the Linköping University
Electronic Press and its procedures for publication and for
assurance of document integrity, please refer to
its WWW home page: http://www.ep.liu.se/
or by conventional mail to the address stated above.

Abstract

We define the batched range-searching problem as follows: given a set S of n points and a set Q of m hyperrectangles, report for each hyperrectangle which points it contains. This problem has applications in, for example, computer-aided design and engineering. We present several parallel algorithms for this problem on coarse-grained multicomputers. Our algorithms are based on well-known average- and worst-case efficient sequential algorithms. One of our algorithms solves the d-dimensional batched range-searching problem in $O(T_s(n \log^{d-1} p, p) + T_s(m \log^{d-1} p, p) + ((m+n)\log^{d-1}(n/p) + m \log^{d-1} p \log(n/p) + k)/p)$ time on a p-processor coarse-grained multicomputer. ($T_s(n,p)$ denotes the time globally to sort n numbers on a p-processor multicomputer, and k is the total number of reported points.)
Keywords Parallel algorithms, coarse-grained multicomputers, range searching.

The work presented here is funded by CENIIT (the Center for Industrial Information Technology) at Linköping University.

1 Introduction

In many applications, such as geographic information systems, computer-aided design and engineering, statistics, etc., we need to answer the following *range-searching* query: given a set S of n points, which points lie within a given hyperrectangle? (A hyperrectangle is the Cartesian product of intervals on distinct coordinate axes.) Usually, we need to answer many such queries for the same set of points. In some situations, we know the set of queries in advance. That is, we want to solve the following *batched* range-searching problem: given a set S of n points and a set Q of m hyperrectangles, report for each hyperrectangle which points it contains. For example, this is an important subproblem in computer simulation of deformation processes, such as vehicle collisions and mechanical forming processes. In such simulations finding all contacts between components of finite-element models of physical objects is necessary. This can be simplified by approximating surface segments with hyperrectangles, and then determining which vertices these hyperrectangles contain [1, 2].

In this paper, we present parallel algorithms for batched range searching on *coarse-grained multicomputers*. A coarse-grained multicomputer consists of several processors connected by an interconnection network. Each processor is fairly powerful, i.e., it delivers workstation-class performance. Since off-the-shelf hardware can be used, coarse-grained multicomputers are relatively inexpensive. Most commercially available parallel computers are of this type.

Most of the research on parallel algorithms for geometric problems has focused on fine-grain parallel models of computation [3, 4, 5]. It is only during the last couple of years that researchers have designed parallel geometric algorithms for coarse-grained multicomputers [6, 7, 8, 9, 10, 11, 12, 13, 14]. In this model of computation we can assume that the size of each local memory is large. For example, it is common to assume that the size of each local memory is larger than the number of processors. This property allows the algorithm designer to balance communication latency with local computation time.

Our parallel algorithms for batched range searching are based on well-known worst- and average-case efficient sequential algorithms. One of our algorithms is based the range-tree method, and solves the d-dimensional batched range-searching problem in $O(T_s(n \log^{d-1} p, p) + T_s(m \log^{d-1} p, p) + ((m+n)\log^{d-1}(n/p) + m \log^{d-1} p \log(n/p) + k)/p)$ time on a p-processor coarse-grained multicomputer. ($T_s(n,p)$ denotes the time globally to sort n numbers on a p-processor multicomputer, and k is the total number of reported points.) We also give algorithms based on the cell method. This method has poor worst-case performance, but since it can be very efficient in practice, we believe that developing parallel algorithms based on this approach is important.

Other researchers have developed parallel algorithms for range

searching on coarse-grained multicomputers. Devillers and Fabri [7] give an algorithm for the one-dimensional case. Recently, Ferreira et al [14] present algorithms for the d-dimensional case. They construct a distributed range tree in time $O(s/p + T_s(s,p))$, where $s = n\log^{d-1} n$. They can then answer a set of $m = O(n)$ range queries in time $((s\log n + k)/p + T_s(s,p))$.

We organize the rest of the paper as follows. In Section 2, we give additional information about coarse-grained multicomputers, and describe some basic operations used by our algorithms. In Sections 3 and 4, we present parallel range-searching algorithms based on the range-tree and cell methods, respectively.

2 Model of Computation

Coarse-grained multicomputers consist of a set of processors connected through an interconnection network. The number of processors usually varies between 16 and 256. The memory is physically distributed over the processors, and interaction between processors is through message passing. Each processor can execute a different program independent of the other processors. However, it is common to let each processor execute the same program asynchronously. That is, except a few global communication steps, processors execute the same program independently of each other. Common interconnection networks are 2D meshes (Paragon XP/S), 3D meshes (Cray T3E), hypercubes (nCUBE 2), and fat trees (CM-5).

Our algorithms use a few basic and extensively studied communication operations. We next describe these operations, and give their time complexities for a square 2D mesh with p processors, which are assumed to be indexed from 1 through p. For a detailed description and analysis of the operations, see Kumar et al [15].

Monotone routing: Each processor $P(i)$ sends at most one m-word message. The destination address, $d(i)$, of the message sent by $P(i)$ is such that if both $P(i)$ and $P(i')$, $i < i'$, send messages, then $d(i) \leq d(i')$. The time complexity, $T_{mr}(m, p, r_{max})$, is $O((r_{max} + m)\sqrt{p})$, where r_{max} is the maximum number of words received by any processor.

Segmented broadcast: Processors with indexes $i_1 < i_2 \ldots < i_q$, are selected; each processor $P(i_j)$ sends the same m-word message to all processors $P(i_j + 1)$ through $P(i_{j+1} - 1)$. The time complexity, $T_{sb}(m,p)$, is $O(m\sqrt{p})$.

Multinode broadcast: Every processor sends the same m-word message to every other processor. The time complexity, $T_{mb}(m,p)$, is $O(mp)$.

Total exchange: Every processor sends a distinct m-word message to every other processor. The time complexity, $T_x(m,p)$, is $O(mp\sqrt{p})$.

Prefix sums and *reduction*: Let $a_1, a_2, \ldots, a_n$ be a list of numbers

evenly distributed over the processors and let $\otimes$ be an associative operator. The prefix sums operation computes $s_i = a_1 \otimes \cdots \otimes a_i$, and stores s_i in the same processor as a_i. The time complexity, $T_p(n,p)$, is $O(n/p + \sqrt{p})$. The reduction operation computes $s = a_1 \otimes \cdots \otimes a_n$, and stores s in each processor. The time complexity, $T_r(n,p)$, is $O(n/p + \sqrt{p})$. In the *segmented* versions of these operations, we apply them to sublists of $a_1, a_2, \ldots, a_n$. The time complexity is the same as for the ordinary operations.

Global sort: Given a list $a_1, a_2, \ldots, a_n$ of numbers evenly distributed over the processors, the global sort operations sorts the list, and returns it evenly distributed over the processors. The time complexity, $T_s(n,p)$, is $O(n(\log(n/p) + \sqrt{p})/p)$.

We end this section by showing how some of the above operations can be used to solve a simple *data-copying* problem. This is an important subproblem in the algorithms to be presented in this paper. The data-copying problem is as follows. A set R of n equal-sized data records is evenly distributed over the processors of a p-processor multicomputer. With each record r is associated a nonnegative integer $n(r)$. The task is to create $n(r)$ additional copies of each record r such that the work of creating the records is uniformly distributed over the processors. We do this as follows.

1. Let $R' = \{r \in R : n(r) > 0\}$, and let $w = \sum_{r \in R'} n(r)$. Decompose R' into subsets $R'(i)$, $i = 1, 2, \ldots, p$, such that $\sum_{r \in R'(i)} n(r) = \lfloor w/p \rfloor$ for $i \leq p\lceil w/p \rceil - w$, and $\sum_{r \in R'(i)} n(r) = \lceil w/p \rceil$ otherwise.

2. For $i = 1, 2, \ldots, p$, copy $R'(i)$ to the processor $P(i)$. Create the copies of the records in $R'(i)$ in the processor $P(i)$.

Lemma 1 *We can solve the data-copying problem in $O(T_{mr}(n/p, p, (n+w)/p) + (n+w)/p)$, time where w is the total number of copies and $n \geq p^2$.*

Proof. Regard R' as an ordered set $\{r_1, r_2, \ldots, r_m\}$. We begin Step 1 by computing the prefix sums $s_1, s_2, \ldots, s_m$, where $s_k = \sum_{j=1}^{k} n(r_j)$. To simplify the description of how to decompose R' into subsets, we assume that w is an integer multiple of p. Extending our description to the general case is easy. Let $l_k = \lfloor s_k/(w/p) \rfloor$. For each record r_k, if $l_{k-1} = l_k$, then we assign r_k to the subset $R'(l_{k-1}+1)$. Otherwise, let $d_k = l_k - l_{k-1}$. Next, we create new records $r_{k,j}$, $j = 0, 1, \ldots, d_k$, such that we (1) assign $r_{k,0}$ to the subset $R'(l_{k-1}+1)$ and set $n(r_{k,0}) = (l_{k-1}+1)w/p - s_{k-1}$, (2) assign r_{k,d_k} to the subset $R'(l_k+1)$ and set $n(r_{k,d_k}) = s_k - l_k w/p$, and (3) assign $r_{k,j}$, $0 < j < d_k$, to subset $R'(l_{k-1}+1+j)$ and set $n(r_{k,j}) = w/p$. Observe that no subset contains more than $\lceil w/p \rceil$ elements. In Step 2, we first identify nonlocal subsets, i.e., subsets whose elements lie in several processors. To do this each processor sends the indexes

of the lowest- and highest-indexed subset that it contains to every other processor. Using monotone routing, we then copy the nonlocal subsets directly to their final destinations. Some processors may completely contain one or more subsets. We handle this by copying all such subsets to the final destination of the lowest-indexed subset in the processor. If a processor completely contains more than one subset, we then use segmented broadcast to transfer the subsets to their correct destinations. □

3 A Worst-Case Efficient Algorithm for Range Searching

Let us again state the problem in which we are interested. The input consists of a set S of n points and a set Q of m hyperrectangles. The task is to report, for each hyperrectangle, which points it contains. In our development of parallel algorithms for this problem, we assume that initially each processor stores n/p points and m/p hyperrectangles, and that m and n are both greater than or equal to p^2. The output consists of hyperrectangle-point pairs, that is, for each hyperrectangle q and point p such that p is contained in q, the pair (q, p) is created.

In this section we present a parallel algorithm inspired by the sequential range-tree method [16]. This is a worst-case efficient method. We can use it to solve the d-dimensional, $d \geq 2$, batched range-searching problem in time $O(n \log^{d-1} n + m \log^d n + k)$, where k is the total number of reported points. For a set S of points in the plane, the corresponding range tree consists of a binary search tree on the x-coordinates of the points. That is, every node v represents an interval $I(v)$ such that a leaf node represents the interval between two consecutive x-coordinates, and an interior node represents the union of the intervals of its children. (We call these intervals *standard intervals*.) With every node v is associated a y-sorted list $S_y(v)$ of the points with x-coordinate within $I(v)$. To determine which points are contained in a hyperrectangle q, partition the x-range of q into standard intervals. More specifically, interval $I(v)$ is part of the partition if the x-range of q contains $I(v)$ but not $I(p(v))$, where $p(v)$ is the parent of node v. Then, for every interval $I(v)$ in the partition, decide by a binary search which points in $S_y(v)$ lie within the y-range of q. We can thus decompose a two-dimensional range-searching problem into a collection of one-dimensional range-searching problems.

We give first a parallel algorithm for the one-dimensional case. We then show how we can extend this algorithm to higher dimensions. The algorithm consists of three parts and the details are as follows.

Part I:

1. Globally sort S into nondecreasing order by x-coordinate. Divide the sorted list into equal-sized sublists, $S(i)$, $i = 1, 2, \ldots, p/2$.

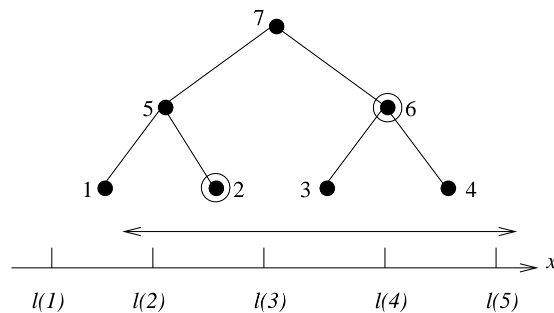

Figure 1: The tree T_p for $p = 8$. T_p has $p/2$ leaf nodes and $p-1$ nodes in total. The given x-range is partitioned into the standard intervals corresponding to the circled nodes. It contains the intervals of leaves 2, 3 and 4, and intersects the interval of leaf 1. We index nodes from left to right, beginning with the leaves.

(We assume that p is an integer power of two.)

2. For each sublist $S(i)$, find $l(i)$, the smallest x-coordinate in the sublist (for sublist $S(p/2)$ find also $l(p/2+1)$, the largest x-coordinate in $S(p/2)$). Broadcast the l-values to all processors.

3. In every processor build a binary search tree T_p on the l-values. Identify each node in T_p by a unique index in the range 1 through $p-1$. See Figure 1.

Part II:

1. For each hyperrectangle q and leaf node i, if q's x-range intersects but does not contain $I(i)$, create the node-hyperrectangle pair (i, q).

2. For each leaf node i, determine $e(i)$, the number of node-hyperrectangle pairs with node index i. Compute $e = \sum_{i=1}^{p/2} e(i)$. If $e = 0$, continue to Part III.

3. Globally sort the node-hyperrectangle pairs by node index.

4. For each leaf node i, compute $\bar{p}(i) = \lceil e(i)/\lceil 2e/p \rceil \rceil$ and $\bar{f}(i) = \sum_{j=1}^{i-1} \bar{p}(j)$. If $\bar{p}(i) > 0$, continue as follows.

   (a) Copy $S(i)$ to the processors $P(\bar{f}(i)+1)$ through $P(\bar{f}(i)+\bar{p}(i))$.

   (b) Divide the node-hyperrectangle pairs with node index i into equal-sized subsets $\bar{Q}(i,j)$, $j = 1, 2, \ldots, \bar{p}(i)$. Move $\bar{Q}(i,j)$ to the processor $P(\bar{f}(i)+j)$.

   (c) Find $k(i,j)$, the output size of the range-searching problem with input $S(i)$ and $\bar{Q}(i,j)$. Compute $\bar{k} = \sum_{i=1}^{p/2} \sum_{j=1}^{\bar{p}(i)} k(i,j)$. If $\bar{k} = 0$, continue to Part III.

5. For each subset $\bar{Q}(i,j)$, compute $p(i,j) = \lfloor k(i,j)/(k'/p) \rfloor$, where $k' = \max(\bar{k}, n)$, and $f(i,j) = \sum_{k=1}^{i-1} \sum_{l=1}^{\bar{p}(k)} p(k,l) + \sum_{l=1}^{j-1} p(i,l)$. If $p(i,j) = 0$, solve the range-searching problem with input $S(i)$ and $\bar{Q}(i,j)$ in the processor $P(\bar{f}(i)+j)$. Otherwise, continue as follows.

   (a) Copy $S(i)$ and $\bar{Q}(i,j)$ to the processors $P(f(i,j)+1)$ through $P(f(i,j)+p(i,j))$.

   (b) Divide $\bar{Q}(i,j)$ into the subsets $Q'(i,j,l), l = 1, 2, \ldots, p(i,j)$, such that $\sum_{(i,q) \in Q'(i,j,l)} k(i,q)$ is $O(k'/p)$.

   (c) Solve the range-searching problem with input $S(i)$ and $Q'(i,j,l)$ in the processor $P(f(i,j)+l)$.

Part III:

1. For each leaf node i, determine $d(i)$, the number of hyper-rectangles whose x-range contains the interval $I(i)$. Compute $d = \sum_{i=1}^{p/2} d(i)$. If $d = 0$, end the execution.

2. For each leaf node i, compute $p(i) = \lceil d(i)/\lceil 2d/p \rceil \rceil$ and $f(i) = \sum_{j=1}^{i-1} p(j)$. Copy $S(i)$ to the processors $P(f(i)+1)$ through $P(f(i)+p(i))$.

3. If $d > m$, then do as follows.

   (a) Copy the hyperrectangles in each processor to every other processor.

   (b) For each hyperrectangle q and leaf node i, if q's x-range contains $I(i)$, create the node-hyperrectangle pair (i,q).

   (c) For each leaf node i, divide the node-hyperrectangle pairs with node index i into equal-sized subsets $Q(i,j)$, $j = 1, 2, \ldots, p(i)$.

4. If $d \leq m$, then do as follows.

   (a) For each hyperrectangle q and leaf node i, if q's x-range contains $I(i)$, create the node-hyperrectangle pair (i,q).

   (b) Globally sort the pairs by node index.

   (c) For each leaf node i, divide the node-hyperrectangle pairs with node index i into equal-sized subsets $Q(i,j)$, $j = 1, 2, \ldots, p(i)$. Move $Q(i,j)$ to the processor $P(f(i)+j)$.

5. For each pair $(i,q) \in Q(i,j)$ and each point $p \in S(i)$, create the pair (q,p).

Theorem 1 *We can solve the one-dimensional range-searching problem in time $O(T_s(n,p) + T_s(m,p) + (m\log(n/p) + k)/p)$.*

Proof. In Step 1 of Part II, we use binary search to decide which pairs to create. Since each hyperrectangle intersects at most two intervals, we create the corresponding pairs locally. In Step 2, each processor first determines how many of its node-hyperrectangle pairs have node index i, $i = 1, 2, \ldots, p/2$. After a total exchange operation, the processor $P(i)$ computes $e(i)$. $P(i)$ then broadcasts $e(i)$ to every processor. Step 4(a) is done by monotone routing followed by segmented broadcasting. In Step 4(b), a segmented prefix sums operation determines the rank of each pair (i, q) among the pairs with node index i. The rank decides to which subset $\bar{Q}(i, j)$ that (i, q) belongs. We then move $\bar{Q}(i, j)$ to the processor $P(\bar{f}(i) + j)$ using techniques similar to those used in Step 2 of the algorithm for the data-copying problem (Section 2). In Steps 4(c) and 5(c), we use binary search. In total, Parts I and II take $O(T_s(n, p) + T_s(m, p) + (m \log(n/p) + \bar{k})/p)$ time.

In Step 1 of Part III, each processor first determines, for each leaf node i, how many of its hyperrectangles contain the interval $I(i)$. This is done as follows. For each hyperrectangle q and node i, if q's x-range contains $I(i)$ but not $I(p(i))$ (where $p(i)$ is the parent of node i), increment a counter associated with node i. To compute, for each leaf node i, how many local hyperrectangles contain $I(i)$, sum the counters associated with nodes along the path from i to the root of T_p. Then continue as in Step 2 of Part II. In Step 3(b), each processor that has received a copy of $S(i)$ decides which hyperrectangles contain the interval $I(i)$, and creates the corresponding pairs. In Step 4(a), we create the pairs using a modified version of the algorithm for the data-copying problem (Section 2). Since we assume in Step 4 that $d \leq m$, the total time complexity of Step 4 is $O(T_s(m, p))$. Step 5, finally, takes $O(d\, n/p^2)$ time. Since we assume that $n \geq p^2$, if $d > m$, the time complexity of Step 5 asymptotically exceeds the time complexities of Steps 1 and 3. The total time for Part III is thus $O(T_{sb}(n/p, p) + T_s(m, p) + dn/p^2)$. □

Giving an algorithm for the two-dimensional case is now easy. It too consists of three parts, where Parts I and II are essentially the same as above. In Part II, we use the batched range-searching algorithm of Edelsbrunner and Overmars [17]. They give a divide-and-conquer algorithm for batched range searching that runs in $O((m + n)\log^{d-1} n + m \log m + k)$ time and uses $O(m + n)$ space. It is only Part III that deviates significantly from the one-dimensional case. The details of Part III are now as follows.

Part III:

1. For each hyperrectangle q and node i, if q's x-range contains $I(i)$ but not $I(p(i))$, create the node-hyperrectangle pair (i, q).

2. For each node i, determine $c(i)$, the number of node-hyperrectangle pairs with node index i. Compute $c = \sum_{i=1}^{p-1} c(i)$. If $c = 0$, end the execution.

3. For each point p and each node i such that p's x-coordinate is contained in $I(i)$ and $c(i) > 0$, create the node-point pair (i, p).

4. Solve the one-dimensional range-searching problem with input consisting of the node-point pairs and the node-hyperrectangle pairs. That is, for each node-hyperrectangle pair (i, q), find the node-point pairs (i, p) such that p is contained in q's y-range.

Theorem 2 *We can solve the two-dimensional range-searching problem in time* $O(T_s(n \log p, p) + T_s(m \log p, p) + (m \log p \log(n/p) + k)/p)$.

Proof. Part I is exactly as in the one-dimensional case. Part II is the same as in the one-dimensional case, except Steps 4(c) and 5(c) which now use the batched range-searching algorithm of Edelsbrunner and Overmars. (In Step 4(c), we modify this algorithm to compute just how many points are contained in each hyperrectangle.) Parts I and II take together $O(T_s(n, p) + T_s(m, p) + ((m + n) \log(n/p) + \bar{k})/p)$ time.

In Step 1 of Part III, it takes $O(\log p)$ time for each hyperrectangle q to find all nodes i in T_p such that q's x-range contains $I(i)$ but not $I(p(i))$. Step 2 is similar to Step 2 of Part II. Step 3 takes $O(n \log p/p)$ time. In Step 4, we apply our one-dimensional range-searching algorithm to the node-hyperrectangle and node-point pairs created in Steps 1 and 3. In our one-dimensional algorithm we assume that the input is evenly distributed over the processors, and that the number of points and the number of hyperrectangles are both greater than or equal to p^2. These assumptions are not necessarily satisfied by the node-hyperrectangle and node-point pairs. We can easily remedy this by adding dummy input as follows. Each processor counts how many node-point pairs it stores. By a reduction operation, we then find n_{max}, the maximum number of such pairs contained in any processor. Finally, each processor adds dummy pairs until it has exactly $\max(n_{max}, p)$ pairs. The same approach is used for the node-hyperrectangle pairs. Since no processor stores more than $O(n \log p/p)$ node-point pairs and $O(m \log p/p)$ node-hyperrectangle pairs, it takes $O(T_r(p, p) + (m + n) \log p/p)$ time to add the dummy input. □

Generalizing the above approach to higher dimensions is straightforward. We can easily derive the following result.

Theorem 3 *We can solve the d-dimensional range-searching problem in time* $O(T_s(n \log^{d-1} p, p) + T_s(m \log^{d-1} p, p) + ((m+n) \log^{d-1}(n/p) + m \log^{d-1} p \log(n/p) + k)/p)$.

4 Average-Case Efficient Algorithms for Range Searching

In this section, we present parallel algorithms for range searching that are based on the cell method [18]. In its simplest version, this

method is as follows. First, find the smallest hyperrectangle B that contains the set S. Divide B into equal-sized hyperrectangular cells, and record for each cell which points it contains. We call the resulting data structure a *cell directory*. To decide which points a hyperrectangle q contains, do as follows. For each cell intersected by q, access the corresponding entry in the cell directory and test, for each point contained in the cell, if it is included within q.

It is common to divide B into $O(n)$ cells, in which case we can build the cell directory (e.g., a multidimensional array of pointers) in $O(n)$ time. The total cost of solving the batched range-searching problem is then $O(m + n + s + t)$ time, where s and t denote the total number of cell accesses and point inclusion tests, respectively. (The time complexity increases linearly with the dimension d of the problem. In this paper, we assume that d is a small constant.)

As already mentioned, the worst-case performance of this method is poor. We can easily create an input such that $s + t$ is $\Omega(m\,n)$, although the output size k is zero. However, in many applications the cell method may outperform more sophisticated methods. For example, in an experimental evaluation of methods for range searching, we [2] found it to be much faster than the range-tree method. This is due to its relative simplicity (small constants of proportionality). Moreover, one can show that if the points are evenly distributed in space and the shape of the query hyperrectangles is similar to the shape of the cells, then $s+t$ is $O(k)$. Finding an efficient parallelization of the cell method as described above is therefore important.

Algorithm I: Our first algorithm is based on the assumption that storing a copy of S and Q in each processor is possible. The first step of the algorithm achieves this by multinode broadcasting. Then, each processor executes the sequential cell method. To load-balance the computations, we divide Q into subsets for which the total number of cell accesses and point inclusion tests is about the same.

1. Copy the points and hyperrectangles in each processor to every other processor.

2. Locally build a cell directory for S. That is, compute B, the smallest hyperrectangle containing S. Divide B into $O(n)$ equal-sized hyperrectangular cells, and record, for each cell, which points it contains.

3. For each hyperrectangle q, find $s(q)$, the number of cells it intersects. Compute $s = \sum_{q \in Q} s(q)$.

4. For each hyperrectangle q, find $t(q)$, the total number of points contained in the cells intersected by q. Compute $t = \sum_{q \in Q} t(q)$.

5. For each hyperrectangle q, let $r(q) = s(q) + t(q)$. Divide Q into the subsets $Q(j)$, $j = 1, 2, \ldots, p$, such that $\sum_{q \in Q(j)} r(q) = O(\max(n, (s+t)/p))$.

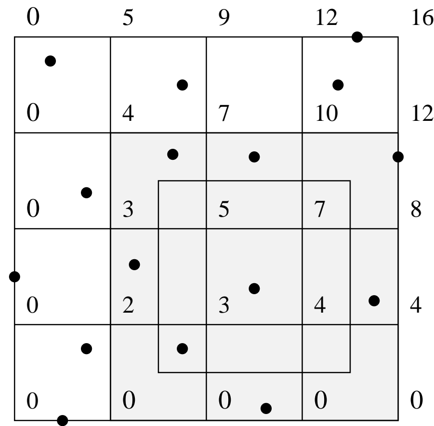

Figure 2: Example with sixteen points and one hyperrectangle. The number besides each grid vertex is the number of points dominated by the vertex. The shaded region is the block of cells intersected by the hyperrectangle. The number of point inclusion tests is $12+0-4-0=8$.

6. Solve the range-searching problem with input S and $Q(j)$ in the processor $P(j)$.

Theorem 4 *Algorithm I solves the batched range-searching problem in time $O(T_{mb}((m+n)/p, p) + m + n + (s+t)/p)$, where $m, n \geq p$.*

Proof. Steps 1 through 3 take $O(T_{mb}((m+n)/p, p) + m + n)$ time. In Step 4, we first compute for each grid vertex v how many points in S it *dominates*, that is, how many points lie in v's southwest quadrant. This can be done in $O(n)$ time. Let $d(v)$ denote the number of points dominated by the grid vertex v. Then, $t(q) = d(v_{NE}) + d(v_{SW}) - d(v_{NW}) - d(v_{SE})$, where v_{NE}, v_{SW}, v_{NW} and v_{SE} denote the northeast, southwest and northwest and southeast vertices, respectively, of the block of cells intersected by the hyperrectangle q. See Figure 2. Thus, Step 4 takes $O(n+m)$ time in total. In Step 5, the partitioning of Q into subsets can easily be done in $O(m)$ time. Finally, Step 6 takes $O(\max(n, (s+t)/p))$ time. □

If we can have a copy of S and Q on each processor, and if $s+t$ is large compared with $m+n$, then this algorithm can be quite efficient. However, in many applications storing a copy of the input on each processor would be impossible. This suggests that we should investigate alternative parallelizations of the cell method.

Algorithm II: Briefly, this algorithm is as follows. For each nonempty cell, we create a list of the points it contains. For each intersected cell, we create a list of (copies of) the hyperrectangles that intersect

it. For each cell that is both nonempty and intersected, we then combine the two lists, that is, we do the corresponding point inclusion tests.

1. Compute B, the smallest hyperrectangle containing S. Divide B into $O(n)$ equal-sized hyperrectangular cells. Identify each cell by a unique index.

2. Decide for each point p in which cell i it is contained. Create the cell-point pair (i, p).

3. For each hyperrectangle q and cell i such that q intersects cell i, create the cell-hyperrectangle pair (i, q). Compute s, the total number of cell-hyperrectangle pairs.

4. Globally sort the cell-point and cell-hyperrectangle pairs with respect to cell indexes. When comparing a cell-point pair and cell-hyperrectangle pair with the same index, let the latter pair win.

   If cell i is both nonempty and intersected, there is now a list of cell-point pairs with index i (denoted $pl(i)$), followed by a list of cell-hyperrectangle pairs with index i (denoted $hl(i)$). It remains to test each point in $pl(i)$ for inclusion within each hyperrectangle in $hl(i)$.

5. Let $n(i)$ and $m(i)$ denote the lengths of $pl(i)$ and $hl(i)$, respectively, and let $t(i) = n(i)m(i)$. Let A be the set of nonempty and intersected cells. For each cell $i \in A$, compute $n(i), m(i)$ and $t(i)$. Compute $t = \sum_{i \in A} t(i)$.

6. Let $I = \{i \in A : t(i) < \lceil t'/p \rceil\}$, where $t' = \max(t, n+s)$.

   (a) For each cell $i \in I$, gather $pl(i)$ and $hl(i)$ into the lowest-indexed processor that contains elements of $pl(i)$.

   (b) Let $I(j) = \{i \in I : pl(i) \text{ and } hl(i) \text{ are in the processor } P(j)\}$. For $j = 1, 2, \ldots, p$, compute $\hat{t}(j) = \sum_{i \in I(j)} t(i)$, $p(j) = \lfloor \hat{t}(j) / \lceil t'/p \rceil \rfloor$ and $f(j) = \sum_{k=1}^{j-1} p(k)$.

   (c) For $j = 1, 2, \ldots, p$, if $p(j) = 0$, then do the point inclusion tests corresponding to $I(j)$ in the processor $P(j)$. Otherwise, copy $pl(i)$ and $hl(i)$, $i \in I(j)$, to the processors $P(f(j)+1)$ through $P(f(j)+p(j))$. Decompose $I(j)$ into subsets $I(j,k)$, $k = 1, 2, \ldots, p(j)$, such that $\sum_{i \in I(j,k)} t(i)$ is $O(t'/p)$. Do the point inclusion tests corresponding to $I(j,k)$ in the processor $P(f(j)+k)$.

7. Let $E = A \setminus I$. For each cell $i \in E$, do as follows.

   (a) Compute $p(i) = \lfloor t(i) / \lceil t'/p \rceil \rfloor$ and $f(i) = \sum_{k \in E, k < i} p(k)$.

(b) Divide the longest list of $pl(i)$ and $hl(i)$ into equal-sized sublists, $ll(i,j)$, $j = 1, 2, \ldots, p(i)$. Move $ll(i,j)$ to the processor $P(f(i) + j)$.

(c) Create a copy of the shortest list of $pl(i)$ and $hl(i)$ in each processor $P(f(i) + 1)$ through $P(f(i) + p(i))$. Do the corresponding point inclusion tests.

Theorem 5 *Algorithm II solves the batched range-searching problem in* $O(T_s(n+s,p) + T_{sb}(m/p,p) + T_{sb}(t/p,p) + (m+n+s+t)/p)$*, time, where* $m, n \geq p^2$.

Proof. Steps 1 and 2 take together $O(T_r(n,p) + n/p)$ time. In Step 3, we can create the cell-hyperrectangle pairs by slightly modifying the algorithm for the data-copying problem (Section 2). By Lemma 1, this takes $O(T_{mr}(m/p, p, (m+s)/p) + (m+s)/p)$ time. The global sort in Step 4 takes $O(T_s(n+s,p))$ time. In Step 5, we compute $n(i)$ and $m(i)$ by segmented reduction in $O(T_r(n+s,p))$ time. We can identify all cells that are both nonempty and intersected in time $O((n+s)/p + T_{mr}(1,p,1))$. We then compute $t(i)$ and t in $O(T_r(n+s,p))$ time. Thus, in total Step 5 takes $O(T_r(n+s,p))$ time.

Step 6(a) takes $O(T_{mr}((n+s)/p, p, t'/p))$ time. This follows from the fact that, if cell $i \in I$, then $m(i) + n(i) \leq \lceil t'/p \rceil$. Thus, no processor receives more than $\lceil t'/p \rceil$ pairs. In Step 6(b), we compute and broadcast $\hat{t}(j)$ to every processor in $O((n+s)/p + T_{mb}(1,p))$ time. In Step 6(c), to decompose $I(j)$ into subsets of cost $O(t'/p)$ can easily be done in $O(t'/p)$ time. Step 6(c) takes $O(T_{mr}(t'/p, p, t'/p) + T_{sb}(t'/p, p) + t'/p)$ time.

Step 7(a) is similar to Step 6(b). To describe Step 7(b), we assume that list $pl(i)$ is longer than $hl(i)$. We divide $pl(i)$ into $p(i)$ sublists such that $p(i)\lceil n(i)/p(i) \rceil - n(i)$ sublists have length $\lfloor n(i)/p(i) \rfloor$, whereas the remaining sublists have length $\lceil n(i)/p(i) \rceil$. We use a segmented prefix sums computation to decide, for each list element, to which sublist it belongs. To move each sublist to its selected processor, we use the same techniques as in Step 2 of the algorithm for the data-copying problem (Section 2). Since no sublist has more than $O(t'/p)$ elements, the total time for Step 7(b) is $O(T_p(n+s,p) + T_{mb}(1,p) + T_{mr}((n+s)/p, p, t'/p) + T_{mr}((n+s)/p, p, (n+s)/p) + T_{sb}((n+s)/p, p))$. In Step 7(c), we copy the shortest list to the selected processors by monotone routing followed by segmented broadcasting. The length of a shortest list cannot exceed $\sqrt{t'}$. Thus, the total time for Step 7(c) is $O(T_{mr}((n+s)/p, p, \sqrt{t'}) + T_{sb}(\sqrt{t'}, p) + t'/p)$. Since we assume that $n \geq p^2$, it follows that $\sqrt{t'} \leq t'/p$. □

References

[1] Z-H. Zhong. *Finite Element Procedures for Contact-Impact Problems.* Oxford University Press, 1993.

[2] P-O. Fjällström, J. Petersson, L. Nilsson, and Z-H. Zhong. Evaluation of range searching methods for contact searching in mechanical engineering. To appear in *International Journal of Computational Geometry & Applications*.

[3] A. Aggarwal, B. Chazelle, L. Guibas, and C. O'Dunlaing. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[4] M.J. Atallah. Parallel techniques for computational geometry. *Proc. IEEE*, 80(9):1435–1448, 1992.

[5] S.G. Akl and K.A. Lyons. *Parallel Computational Geometry.* Prentice-Hall, 1993.

[6] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. 9th Annual ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[7] O. Devillers and A. Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In *Algorithms and Data Structures. Third Workshop, WADS'93*, pages 277–288, 1993.

[8] X. Deng. A convex hull algorithm on coarse grained multiprocessors. In *Proc. 5th Annual International Symposium on Algorithms and Computation (ISAAC 94)*, pages 634–642, 1994.

[9] F. Dehne, C. Kenyon, and A. Fabri. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 586–593, 1994.

[10] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A.A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, 1995.

[11] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing (ICS'96)*, 1996.

[12] P-O. Fjällström. Parallel algorithms for geometric problems on coarse grained multicomputers. Technical Report LiTH-IDA-R-96-38, Dep. of Computer and Information Science, Linköping University, 1996.

[13] P-O. Fjällström. Parallel interval-cover algorithms for coarse grained multicomputers. Technical Report LiTH-IDA-R-96-39, Dep. of Computer and Information Science, Linköping University, 1996.

[14] A. Ferreira, C. Kenyon, A. Rau-Chaplin, and S. Ubeda. *d*-Dimensional range search on multicomputers. Technical Report 96-23, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, 1996.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[16] J.L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[17] H. Edelsbrunner and M.H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[18] J.L. Bentley and J.H. Friedman. Data structures for range searching. *Computing Surveys*, 11:397–409, 1979.