Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers

Olivier Devillers, Andreas Fabri

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports de Recherche

N°1882

Programme 4

Robotique, Image et Vision

SCALABLE ALGORITHMS FOR BICHROMATIC LINE SEGMENT INTERSECTION PROBLEMS ON COARSE GRAINED MULTICOMPUTERS

Olivier DEVILLERS
Andreas FABRI

Mars 1993

Institut National
de Recherche
en Informatique
et en Automatique

2004 route des Lucioles
B.P. 93
06902 Sophia-Antipolis
France

Scalable Algorithms for Bichromatic Line Segment Intersection Problems on Coarse Grained Multicomputers*

Problèmes d'intersection de segments de deux couleurs. Algorithmes parallèles à échelle adaptative pour machine à gros grain

Olivier Devillers† **Andreas Fabri**†

Programme 4 : Robotique, Image et Vision.

Keywords: Computational Geometry, Data Structures, Decomposability, Parallel Algorithms, Coarse Grained Multicomputer

* *This work was partially supported by the ESPRIT Basic Research Action Nr. 7141 (ALCOM II).*

† INRIA, B.P.93, 06902 Sophia-Antipolis cedex (France), Phone: +33 93 65 78 55, Fax: +33 93 65 76 43, E-mail: `Firstname.Name@sophia.inria.fr`.

Abstract

We present output-sensitive scalable parallel algorithms for bichromatic line segment intersection problems for the Coarse Grained Multicomputer model. Under the assumption that $n \geq p^2$, where n is the number of line segments and p the number of processors, we obtain an intersection counting algorithm with a time complexity of $O(\frac{n \log n \log p}{p} + T_s(n \log p, p))$, where $T_s(m, p)$ is the time used to sort m items on a p processor machine. An additional $O(\frac{k}{p})$ time is spent on the reporting of the k intersections. As the sequential complexity is $O(n \log n)$ and $O(k)$ for counting and reporting, respectively, we obtain a speedup of $\frac{p}{\log p}$. As a byproduct we present algorithms for the trapezoidal decomposition and 1-dimensional range query reporting with the same speedup.

Résumé

Cet article présente des algorithmes parallèles à échelle adaptative qui calculent les intersections de segments bichromatiques dans un modèle d'ordinateur parallèle à gros grain. En supposant que $n \geq p^2$, où n est le nombre de segments de droite et p le nombre de processeurs, on obtient un algorithme qui compte les intersections en un temps $O(\frac{n \log n \log p}{p} + T_s(n \log p, p))$, où $T_s(m, p)$ désigne le temps nécessaire au tri de m données sur une machine à p processeurs. Il faut un temps supplémentaire de $O(\frac{k}{p})$ pour énumérer les k intersections. Comme le temps de calcul séquentiel, pour compter et énumérer les intersections est de $O(n \log n)$ et $O(k)$ respectivement, le gain de temps est de $\frac{p}{\log p}$. Nous présentons en corollaire des algorithmes de décomposition trapezoïdale et de requête en dimension 1 pour lesquels le gain de temps est analogue.

1 Introduction

The *bichromatic line segment intersection problems* can be stated as follows. Let R (resp. B) be two sets containing n red (resp. blue) non-intersecting line segments. The problem is to *count* or to *report* the intersections between red and blue segments. This problem is well understood in a sequential setting [4, 11] and its sequential time complexity is $\Theta(n \log n)$ for counting and $\Theta(n \log n + k)$ for reporting, where k is the number of intersections. In the parallel setting there is an optimal algorithm for the CREW-PRAM model solving the problem in time $O(\log n)$ with $O(n + \frac{k}{\log n})$ processors [7].

In this paper we consider the parallel complexity of this problem in the *Coarse Grained Multicomputer* model, or $CGM(m, p)$ for short. In this model a set of p processors, each with $O(\frac{m}{p})$ local memory, are connected by some arbitrary interconnection network. The model is *coarse grained*, as the size $O(\frac{m}{p})$ of each local memory is defined to be considerably larger than $O(1)$. Throughout this paper we assume that $\frac{m}{p} \geq p$, unless otherwise stated. Interprocessor communication is done in *communication rounds*. An operation performed in a communication round is typically a sorting or a broadcasting step. Such operations are usually available as system calls on commercially available parallel machines.

The coarse grained multicomputer model has recently been used to solve problems in computational geometry [6] and image processing [1]. The first paper uses a decomposition of the Euclidean space and the second paper a data compression technique. In this paper we present a further decomposition technique which can be applied to tree-like data structures, in our case segment trees. We further present a load balancing scheme to solve the 1-dimensional range query reporting problem. The combination of these techniques yields efficient scalable parallel algorithms, which are independent of the communication network. The main objective in the design of algorithms for this model is to use a constant or small number of communication rounds. Roughly, this is achieved by reorganizing the n data in such a way that each processor can sequentially treat an independent subproblem of size $O(\frac{n}{p})$.

The main result of this paper can be stated as follows. Given two sets of n non-intersecting red and n non-intersecting blue line segments, we show how to solve the intersection counting problem on a $CGM(n \log p, p)$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n \log p}{p} + T_s(n \log p, p))$. For reporting the k intersections we need a machine with $\max(k, n \log p)$ memory and it takes additional time $O(\frac{k}{p})$, that is our algorithm is output-sensitive.

We further show how to obtain a trade off between the memory and the number of communication rounds. We can solve the counting problem on a $CGM(n\frac{\log p}{r}, p)$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n \log p}{p} + r \cdot T_s(n\frac{\log p}{r}, p))$, for $1 \leq r \leq \log p$. That is, we can vary between $O(n \log p)$ memory and $O(1)$ communication rounds and $O(n)$ memory and $\log p$ communication rounds. As we need $O(k)$ memory for the reporting problem in order to store the output, we can obtain a similar trade off between memory and number of communication rounds, if $k \leq O(n \log p)$.

Our algorithms are *scalable* in the following sense. They have two parameters, the problem size n and the number of processors p, and the speedup of the algorithm is maintained when one of these parameters changes. Hence, the speedup is a function of p, not of n. This condition is crucial as the following example shows. Suppose the sequential time complexity for a problem is $O(n \log n)$ and the time complexity of a parallel algorithm for the $CGM(n,p)$ is $O(\frac{n \log^2 n}{p})$. Then, a speedup of $\frac{p}{\log n}$ would be obtained, and hence the parallel algorithm is only faster than the sequential algorithm for $n \leq 2^p$. The design of scalable algorithms is one of the main goals of the recent High Performance Computing and Communication Initiative [8].

This paper is organized as follows. In Section 2 we shortly discuss the underlying communication model. In Sections 3, 4 and 5, we present a binary search, a trapezoidal decomposition and a 1-dimensional range query reporting algorithm, which are interesting in their own right and which are used as building blocks for the bichromatic line segment intersection algorithms presented in Section 6.

2 Communication Model

The processors of a $CGM(m,p)$ communicate via an interconnection network in which each processor can exchange messages, of size $O(\log m)$ bits each, with any one of its immediate neighbors in constant time. Commonly used interconnection networks for CGM include 2D-mesh (e.g. Intel Paragon), hypercube (e.g. Intel iPSC/860) and the fat-tree (e.g. Thinking Machines CM-5). We refer the reader to [2, 3, 9, 10] for a more detailed discussion of the different architectures and algorithms.

We will now outline the four operations involving interprocessor communication which we use in this paper and give the time complexity of the operations. The first two operations concern all n data. Assume that the p processors of the $CGM(m,p)$ are numbered from 0 to $p-1$.

1) global sort: $T_s(m,p)$ refers to the time to sort $O(m)$ data items stored on a $CGM(m,p)$. Initially, $O(\frac{m}{p})$ data items are on each processors. When the sort is complete, the data are distributed such that data on processor p_i are greater than data on p_j if $i > j$, and data are sorted on each processor.

The time complexity[1] is $T_s(m,p) = \Theta(\frac{m}{p}(\log m + \sqrt{p}))$ for a 2D-mesh, $T_s(m,p) = O(\frac{m}{p}(\log m + \log^2 p))$ for a hypercube and $T_s(m,p) = \Theta(\frac{m}{p} \log m)$ for a fat-tree, which is optimal provided that $m \geq 2^{\sqrt{p}}$, $p^{\log p}$, p respectively.

2) segmented broadcast: In a segmented broadcast operation, $q \leq p$ processors with numbers $j_1 < j_2 < \ldots < j_q$ are selected. Each such processor p_{j_i} broadcasts $O(\frac{m}{p})$ data from its local memory to the processors p_{j_i+1} to $p_{j_{i+1}-1}$. The time complexity is $T_{sb}(m,p) = \Theta(\frac{m}{p}\sqrt{p})$ for a 2d-mesh and $T_{sb}(m,p) = \Theta(\frac{m}{p} \log p)$ for a hypercube and a fat-tree. If the data to broadcast are on different processors we either first have to sort

[1]The time complexity for the hypercube and the 2D-mesh is based on Batcher's bitonic sort[2]. Note that for the hypercube better deterministic [5] and randomized [13] sorting algorithms exist, which, however, are not of practical use.

them or, when they are in the correct order, we have to move them using monotonic routing operations. The latter can be done in the same time as the segmented broadcast.

3) multinode broadcast: In a multinode broadcast operation, every processor (in parallel) sends one message to all other processors. The time complexity $T_b(p)$ for any interconnection network is $T_b(p) = \Theta(p)$.

4) total exchange: In a total exchange operation, every processor (in parallel) sends a different message to each other processors. The time complexity $T_x(p)$ of the total exchange is $T_x(p) = \Theta(p^{\frac{3}{2}})$ for a 2D-mesh, and $T_x(p) = \Theta(p \log p)$ for a hypercube and a fat-tree.

3 Binary Search

In this section we present a parallel binary search algorithm for the coarse grained multicomputer $CGM(n, p)$. The algorithm is not particularly difficult, but should be considered as a "warmup", as it shows a basic loadbalancing technique which will be used and refined in the next sections. The algorithm is given in sufficient detail, which permits to skip over similar details later on.

The problem can be stated as follows: given a sequence S of n elements perform a binary search on S for each element of a set Q of n queries. This problem can easily be solved by sorting $S \cup Q$, if S and Q come from a totally ordered universe, but that is not the case in our application later on (point location in a planar subdivision). Essentially, it is the lack of a total order for the queries, which makes it necessary to perform n independent search processes.

The idea of the algorithm is to determine for the p subsequences S_i of size $\frac{n}{p}$ how many queries end up in each of them. Then, we make the appropriate number of copies of each subsequence, in order to balance the ratio of queries per subsequence. The details of the algorithm are as follows:

0. S_i, the i^{th} subsequence of size $\frac{n}{p}$ of the sorted sequence S, and a set Q_i of $\frac{n}{p}$ queries are on processor p_i.

1. Perform a multinode broadcast, where each processor p_i sends the biggest element of S_i to all other processors. Let $\hat{S}$ denote this ordered sequence of size p.

2. Each processors p_i performs the binary search on $\hat{S}$ for its $\frac{n}{p}$ queries Q_i. Let q_{ij} be the number of queries from Q_i which end up at entry $\hat{S}[j]$.

3. Perform a total exchange, where each processor p_i sends q_{ij} as message to processor p_j, $1 \leq j \leq p$. Each processor p_i then computes $q_i := \sum_{j=1}^{p} q_{ji}$, which is the overall number of queries which want to do binary search in S_i.

4. In order to loadbalance the number of queries per copy of a set S_i we need $\lceil \frac{q_i}{n} p \rceil$ copies of S_i. Perform a segmented broadcast to create them. Note that we end up with at most $2p - 1$ copies, if $\frac{q_i}{n} = \frac{n}{p} + 1$, for $1 \leq i \leq p$. Hence, each processor has to treat at most two copies of sets S_i.

5. Perform a multinode broadcast, where each processor p_i sends q_i as message to all other processor. Given this information each processor generates the $O(\frac{n}{p})$ dummy queries, which are needed, for that the number of queries falling in S_i is an exact multiple of $\frac{n}{p}$. Globally sort the queries which yields that queries which ended up at $\hat{S}[j]$ in step 2 are on a processor holding a copy of S_j.

6. Locally continue the binary search.

The correctness of the algorithm is easy to see. The time complexity of the binary search can be analyzed as follows. The local binary search in Steps 2 and 6 takes time $O(\frac{n}{p}\log p)$ and $O(\frac{n}{p}\log n)$, respectively. The communication complexity is $2T_b(p) + T_x(p) + T_{sb}(n,p) + T_s(n,p)$, where the sorting step is the dominating term. We thus can state the following lemma.

Lemma 1 *Given a sorted sequence S of n elements and a set Q of n queries we can perform the binary search on a CGM(n,p), $\frac{n}{p} \geq p$, in time $O(\frac{n\log n}{p} + T_s(n,p))$.*

4 Trapezoidal Decomposition

The *trapezoidal decomposition* of a set of n non-intersecting line segments in the plane is a planar subdivision and obtained as follows. Starting at each endpoint of each segment in S we draw two vertical rays, one upwards one downwards, each extending until it hits another segment from S. Various sequential algorithms compute this subdivision in optimal $\Theta(n\log n)$ time and $\Theta(n)$ space. In this section we will first present a less efficient sequential algorithm, its parallelization and its modification in order to obtain a scalable parallel algorithm with speedup $\frac{p}{\log p}$.

The idea of the non-optimal sequential algorithm is as follows. We build a segment tree [12] on the x-coordinates of the segment endpoints. Each node v represents an interval and stores the catalogue $S(v)$ of segments covering the interval of the node but not the interval of its parent. As the segments do not intersect we can order the segments in $S(v)$ by the relation *below*. As each segment is stored in at most $2\log n$ catalogues, the size of the segment tree is $O(n\log n)$. For each segment endpoint we traverse the tree and perform a binary search on the catalogue $S(v)$ for each node v on the path from the root to the appropriate leaf. The binary search gives us the closest segments above and below the query point in $S(v)$, that is we have to determine a kind of "minimum" of $\log n$ segments, to determine the closest segments in S. The time complexity is thus $O(n\log^2 n)$ and the space requirement is $O(n\log n)$.

We obtain an obvious parallelization of the query phase, by unfolding a loop of the sequential algorithm. Instead of letting a query s traverse the nodes of a path q of length $\log n$ in the tree, we create $\log n$ queries for each segment s. Each query consists of pairs (v,s) for each node v on the path q. Suppose that the nodes of T are uniquely numbered. For these $n\log n$ queries we perform a binary search on the concatenation of the catalogues $S(v)$ of all nodes $v \in T$ with increasing node numbers, which is a totally ordered sequence of length $O(n\log n)$. We finally determine the closest segment in S by