PAPER Special Section on Discrete Mathematics and Its Applications

Designing Efficient Parallel Algorithms with Multi-Level Divide-and-Conquer*

Wei CHEN[†a)] ***and*** **Koichi WADA**[†], ***Regular Members***

SUMMARY *Multi-level divide-and-conquer (MDC)* is a generalized divide-and-conquer technique, which consists of more than one division step organized hierarchically. In this paper, we investigate the paradigm of the MDC and show that it is an efficient technique for designing parallel algorithms. The following parallel algorithms are used for studying the MDC: finding the convex hull of discs, finding the upper envelope of line segments, finding the farthest neighbors of a convex polygon and finding all the row maxima of a totally monotone matrix. The third and the fourth algorithms are newly presented. Our discussion is based on the EREW PRAM, but the methods discussed here can be applied to any parallel computation models.

key words: *design of parallel algorithm, multi-level divide-and-conquer, convex hull problem of discs, upper envelope problem of segments, farthest neighbors problem of polygons*

1. Introduction

Divide-and-conquer (DC) is one of the most important techniques used for designing parallel algorithms. It consists of two steps: the division step which divides a problem into subproblems and finds the solution of each, and the merging step which merges the solutions of the subproblems into that of the original problem. To solve a problem efficiently with the DC, one should find a division which divides the problem into a set of proper subproblems such that (i) the total size of the subproblems does not exceed the size of the original problem much and (ii) the solutions of the subproblems can be easily merged into that of the original problem. Sinc some problems do not have such divisions, the DC is not always efficient. Fortunately, for some problems, although proper subproblems can not be found by one division step, they can be found by carrying out several division steps which are organized hierarchically. We are interested in such problems.

We introduce a concept *multi-level divide-and-conquer (MDC)*. It is the same as that of the ordinary DC except that one division step is replaced by more than one hierarchically organized division step. The input of the first division step is that of the original problem, and the input of the ith ($i \geq 2$) division step is the output of the $(i-1)$th division step. In this paper, we discuss the structures and the mechanisms of the MDC and show that it is an excellent method for designing parallel algorithms. There are some important parameters in the MDC: the number of division levels (steps), the sizes of subproblems and the number of subproblems in each division level. We give a standard way for designing these parameters. We also show that by adjusting the parameters and adding some other auxiliary approaches such as pruning technique, the MDC will become much more efficient.

In this paper, we use several algorithms to explain the MDC technique. In these algorithms, some are known and some are newly presented. We would focus on the MDC technique itself but not these algorithms. Therefore, when the details of the algorithms prevent us from understanding the technique we leave them to the references. We use the EREW PRAM as a computational model, but conclusions can be applied to any parallel computation models. A parallel algorithm in the PRAM is cost optimal if the product of its running time and the number of the processors is of the same order as the running time of the fastest known sequential algorithm. It is time optimal if it is the fastest possible algorithm using a polynomial number of processors. In this paper, the following MDC-based parallel algorithms are used for studying the MDC method: (1) finding the convex hull of n discs in $O(\log^{1+\epsilon} n)$ time using $O(n/\log^{\epsilon} n)$ processors, (2) finding the upper envelope of n line segments in $O(\log n)$ time using $O(n)$ processors, (3) finding the farthest neighbors of a convex polygon with n vertices and (4) finding all the row maxima in an totally monotone matrix with n rows and n columns in $O(\log^{1+\epsilon} n/\log\log n)$ time using $O(n/\log^{\epsilon} n)$ processors, where $\epsilon > 0$ is an arbitrary constant. Some other problems such as finding the convex hull of curves in the plane and finding the upper envelope of h-intersecting segments in the plane are also used. In these algorithms, the first and the second are known [3], [4], and the third and the fourth are newly presented. All the above problems have been studied for quite some time and each has many applications. As so far, without using the MDC technique the first and the second problems are solved in $O(\log^2 n)$ time using $O(n/\log n)$ processors [10], [14], and the third and the fourth problem are solved in $O(\log n)$ time using

Manuscript received August 31, 2000.
Manuscript revised November 17, 2000.

[†]The authors are with the Department of Electrical and Computer Engineering, Nagoya Institute of Technology, Nagoya-shi, 466-8555 Japan.
a) E-mail: chen@elcom.nitech.ac.jp

*This research was supported by the Grant-in-Aid for Scientific Research (B)(2) 10205209 from the Ministry of Education, Science, Sports and Culture of Japan.

$O(n)$ processors [2] and in $O(\log^2 n/\log\log n)$ time using $O(n/\log n)$ processors [5].

We introduce a standard MDC technique in Sect. 2. In Sect. 3 and Sect. 4, we use the first and the second algorithms to show how to design and adjust parameters and how to prune the recursive processes in the MDC, respectively. We describe the third and the fourth algorithms in Sect. 5 to show how to use the MDC more flexibly and skillfully.

2. Multi-Level Divide-and-Conquer

Given a problem, we consider the following three types of divisions: EDHM (Easily Dividing Hard Merging) divisions, in which the total size of the subproblems is close to that of the original problem but the solutions of the subproblems are difficult to be merged, HDEM (Hard Dividing Easily Merging) divisions, in which the total size of the subproblems is much larger than that of the original problem, but the solutions of the subproblems are easily merged, and finally EDEM (Easily Dividing Easily Merging) divisions. Structurally, in contrast with one merge step following one division step in the ordinary DC, one merge step follows k-division steps in a k-level MDC, where the first $k-1$ division steps are the EDHM ones and the last one is the EDEM one. One should not confuse the concept of a k-level MDC with a k-DC. The latter is the ordinary DC where the division step divides the given problem into k subproblems. In a k-level MDC, the k division steps are executed one after another in a sequential way, therefore, the value of k should not be too large.

In the following, we show a standard MDC technique. We use it to solve the convex hull problem of a set S of N discs in the plane, which is defined as the smallest convex region that contains all the discs of S. The convex hull of S, denoted as $CH(S)$, is represented by the sequence of arcs (portions of circles) which lie on the boundary of $CH(S)$ in clockwise order. There are at most $2N-1$ arcs in $CH(S)$ [9], [12]. A straight line which passes through the leftmost and the rightmost points separates $CH(S)$ into two parts: the upper hull $UH(S)$ and the lower hull $LH(S)$. In the following, we show an algorithm to compute $UH(S)$. The lower hull $LH(S)$ can be found in the same way. Two sets of arcs are said to be *separated* if the arcs of two sets lie on the different sides of a vertical line. Let n be the number of discs for a given subset S' of S. We find $UH(S')$ in the following. We can find $UH(S)$ by setting $S'=S$.

An ordinary DC technique finds $UH(S')$ in the following two ways. (1) the first method divides the n discs into δ equally-sized subsets of discs, recursively finds the upper hull of each subset in parallel, and then merges these δ upper hulls into $UH(S')$. (2) The second method partitions the n discs into δ equally-sized separated sets of arcs by $\delta-1$ vertical lines, recursively computes the upper hull of each subset in parallel, and then merges the δ upper hulls into $UH(S')$. The first method uses an EDHM division: to merge the δ upper hulls, one should find the intersections of the hulls which may reach totally $\Theta(\delta n)$. The second method uses an HDEM division: after partitioning, each subsets may contain $\Theta(n)$ arcs which means that the total size of the subproblems may be $\Theta(\delta n)$ (Fig. 1). Therefore, both methods are inefficient.

A 2-level MDC combined by the above two division methods solves the problem as follows:

First division step: Divide n discs into δ equally-sized subsets of discs, and recursively construct the upper hull of each subset in parallel (Fig. 2(a)). Note that these upper hulls may intersect with each other.

Second division step: Using $n/\delta-1$ vertical lines, partition the δ upper hulls produced in the first recursive step which contain at most $2n$ arcs totally into n/δ equally-sized separated parts (Fig. 2(b)), and then recursively find the upper hull of each part in parallel.

Merge step: Merge the $2n/\delta$ separated upper hulls produced in the second recursive step into the upper hull of S'.

In the second division step, when using vertical lines to partition the upper hulls, one line intersects with each upper hull at most once. The second division step is an EDEM one, since (1) partitioning δ upper hulls by $n/\delta-1$ vertical lines increases at most $n-\delta$ arcs, i.e., the size of each subset does not exceed 3δ and the total size of the subsets does not exceed $3n$,

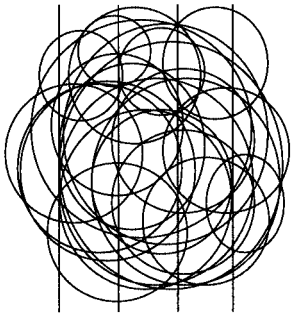


Fig. 1 A set of discs difficult to be separated.

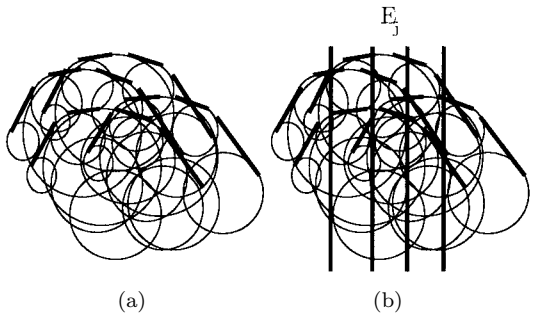


Fig. 2 An MDC technique used for finding the convex hull of discs.

and (2) the separated n/δ upper hulls produced in the second division step can be merged easily: for each upper hull computing the portion which belongs to the solution, and connecting each portion from the left to right. In the algorithm if the size of the problem is small enough, we solve it by any sequential algorithm in constant time. Let $T(n)$ and $P(n)$ denote the running time and the number of the processors, respectively. Partitioning δ upper hulls by $n/\delta - 1$ vertical lines in the second division step can be done in $O(\log n)$ time using $O(n/\log n)$ processor, and the merging step can be done in $O(\log n)$ time using $O(n/\log n)$ processor [4]. In the first and the second division steps, δ subproblems of size n/δ each and n/δ subproblems of size 3δ each are recursively found in parallel, respectively. Therefore, we have the following recurrences:

$$T(n) \leq \begin{cases} T(n/\delta) + T(3\delta) + O(\log n), \\ \text{if } n > 4 \\ c, \quad \text{if } n \leq 4 \end{cases}$$

$$P(n) \leq \begin{cases} \max\{O(n/\log n), \delta P(n/\delta), n/\delta P(3\delta)\}, \\ \text{if } n > 4 \\ 1, \quad \text{if } n \leq 4 \end{cases}$$

Replacing δ by $n^{1/2}$, we get $T(n) = O(\log n \log\log n)$ and $P(n) = O(n \log^{\log 3} n)$. Therefore, $UH(S)$ can be found in $O(\log N \log\log N)$ time using $O(N \log^{\log 3} N)$ processors. This means that the MDC solves the problem faster than the ordinary DC. In Sect. 3, we will adjust the parameters of the above MDC to make it cost optimal.

The above example gives a common behavior of the MDC: in each division step total size of the subproblems should not exceed the size of the original problem too much, and in the merging step the solutions of the subproblems should be easily merged into the solution of the original problem. Generally, let a k-level MDC solve a problem of size n in $T(n)$ time using $P(n)$ processors. We assume that the ith division step divides the output of the $(i-1)$th division step (the first division step divides the input of the problem) into h_i $(1 \leq i \leq k)$ subproblems of size n_i each in $T_D(n,i)$ time using $P_D(n,i)$ processors, and assume the merge step merges the outputs of the final division step into the solution of the original problem in $T_M(n)$ time using $P_M(n)$ processors. Since the subproblems are solved recursively, in the ith division step we can solve the subproblem of size n_i in $T(n_i)$ time using $P(n_i)$ processors. Let $T_D(n) = \sum_{i=1}^{k} T_D(n,i)$ and $P_D(n) = \max_{i=1}^{k} P_D(n,i)$. The problem can be solved in $T(n) = \sum_{i=1}^{k} T(n_i) + T_D(n) + T_M(n)$ time and using $P(n) = \max_{i=1}^{k}\{h_i P(n_i),\ P_D(n),\ P_M(n)\}$ processors. In an h-DC (it divides the problem into h-subproblems), h is often designed as a positive integer or as a function of the size n of the problem. In the latter case, for solving more subproblems in parallel, h is often designed as n^{ϵ}, where $0 < \epsilon < 1$. When designing a k-level MDC, we usually set $h_i = n^{1/k}$ $(1 \leq i \leq k)$ first. The size n_i of each subproblem in the ith level should not exceed $cn^{1/k}$, where $c > 0$ is a constant. In this situation, if $T_D(n) + T_M(n) = O(1)$, then $T(n) = kT(cn^{1/k}) + O(1) = O(\log n)$, and if $T_D(n) + T_M(n) = O(\log n)$, then $T(n) = kT(cn^{1/k}) + O(\log n) = O(\log n \log\log n)$. Also if $P_D(n) = P_M(n) = O(n)$, then $P(n) = \max_{i=1}^{k}\{n^{1/k} P(cn^{1/k}),\ O(n)\} = O(n \log^{\log c} n)$.

3. Adjusting Parameters of the MDC

A k-level MDC is called as an $(h_1, h_2, \ldots, h_k)$-MDC if its ith division step divides the input of this step into h_i subproblems and solves them recursively. The algorithm of finding $UH(S)$ in Sect. 2 is a $(\delta, 2n/\delta)$-MDC, where $\delta = n^{1/2}$. As we stated before, instead of setting $h_i = c$ (c is a positive integer) we prefer to set $h_i = n^{\epsilon}$ $(0 < \epsilon < 1)$ so that n^{ϵ} subproblems can be solved in parallel. But it does not always give the best algorithm since the price paid for dividing and merging may be high when the number of subproblems is big. In this section, we show how to adjust the value of h_i such that it does not depend on the sizes of subproblems too much.

Let us consider the $(\delta, 2n/\delta)$-MDC used for finding $UH(S)$ in Sect. 2 once again. Instead of setting $\delta = n^{1/2}$ in the algorithm of Sect. 2, we define that $\delta = \delta_i$ $(1 \leq i \leq c)$, if $d_i < n \leq d_{i+1}$, where $\delta_i = \min(d_i, n/d_i)$, $d = \log^{1/c} N$ and $d_i = 2^{d^i}$ $(= 2^{\log^{i/c} N})$ and c is a positive integer which will be decided later. We should note that N is the size of S and d is fixed when the input S and c are given. From the definition, we know that δ (therefore, the number of the subproblems in both division steps) keeps the same value when n changes from d_i to $\leq d_{i+1}$. We solve the problem with the same algorithm in Sect. 2 as follows: if the size n of the problem ($n = N$ at the beginning of the algorithm) satisfies the condition $d_t < n \leq d_{t+1}$ $(1 \leq i \leq c-1)$, we solve the problem by the $(\delta_t, 2n/\delta_t)$-MDC $(1 \leq t \leq c)$. If $n \leq d_1$, we stop the recursion and solve the problem directly by the known algorithm which runs in $O(\log^2 n)$ time using $O(n/\log n)$ processors [14]. Replacing δ with δ_t and changing the boundary condition in the recurrences of Sect. 2, we get the following recurrences for the revised algorithm:

$$T(n) \leq \begin{cases} T(n/\delta_t) + T(3\delta_t) + O(\log n), \\ \text{if } d_t < n \leq d_{t+1}\ (1 \leq t \leq c-1) \\ O(\log^2 n), \quad \text{if } n \leq d_1 \end{cases}$$

$$P(n) \leq \begin{cases} \max\{O(n/\log n), \delta_t P(n/\delta_t), n/\delta_t P(3\delta_t)\}, \\ \text{if } d_t < n \leq d_{t+1}\ (1 \leq t \leq c-1) \\ n/\log n, \quad \text{if } n \leq d_1 \end{cases}$$

We can prove that $T(N) = O(\log^{1+1/c} N)$ and $P(N) = O(N/\log^{1/c} N)$ [4]. For any constant $\epsilon > 0$, by setting $c = \lceil 1/\epsilon \rceil$ and $c = \lceil \log\log N \rceil$, the algorithm runs in $O(\log^{1+\epsilon} N)$ time using $O(N/\log^{\epsilon} N)$ processors, and in $O(\log N \log\log N)$ time using $O(N \log^{1+\epsilon} N)$ processors, respectively. The first result is cost optimal and the second one is faster. If there are only $2^{O(\log^{\alpha} N)}$ different kinds of radii in N discs for any α $(0 < \alpha < 1)$, from the first result $UH(S)$ is constructed not only cost optimally but also time optimally in $O(\log N)$ time using $O(N)$ processors.

The same MDC can be used to solve the following generalized convex hull problems. Let E be a set of N convex curves in the plane and let the boundary of the convex hull of E consist of at most hN portions of curves. In the special case that E is a set of circle, $h = 2$. Let O_I, O_{II} and O_{III} denote the operations of finding the common tangents of two curves, the intersections of two curves and the intersections of one curve and one straight line, respectively, in the plane, and A, B and C denote the running time of the operations O_I, O_{II} and O_{III}, respectively. By replacing the operations on arcs of circles by the ones on arcs of curves, the convex hull of E can be computed by the same algorithm in $O(\frac{A+B+C}{\log^{1+\epsilon} N})$ time using $O(N/\log^{\epsilon} N)$ processors, or in $O(\frac{A+B+C}{\log N \log\log N})$ time using $O(N \log^{(1+\epsilon)\log h} N)$ processors for any positive constant ϵ. For quadratic curves in the plane, A, B and C are constants, and $h = 4$.

The same MDC can be also used to solve the upper envelope problem. Given a collection S of N segments which are h-intersecting in the plane, that is, they intersect pairwise at most h times, regarding the segments as opaque barriers, their upper envelope, denoted as $UE(S)$, consists of the portions of the segments visible from point $(0, +\infty)$. $UE(S)$ contains $\Theta(\lambda_{h+2}(N))$ segments, where $\lambda_h(N) = O(N)$, $\Theta(N\alpha(N))$, and $O(N\alpha(N)^{\alpha(N)^{h-3}})$ for $h \leq 2$, $h = 3$ and $h > 3$, respectively, and $\alpha(N)$ is the extremely slowly growing functional inverse of Ackermann's function [9], [12]. Let d_t and δ_t be defined as before, and let S' be a subset of S with n h-intersecting segments. The following algorithm finds $UE(S')$. $UE(S)$ can be found by setting $S' = S$ and $n = N$.

Base: If $n \leq d_1$, solve the problem directly by the known algorithm which runs in $O(\log^2 n)$ time using $O(\frac{\lambda_{h+1}(n)}{\log n})$ processors [10]. If $n > d_1$, do the following steps.

First division step: Find $t > 1$ such that $d_t < n \leq d_{t+1}$. Divide n segments of S' into δ_t equally-sized subsets of segments, and recursively construct the upper envelope of each subset in parallel. Note that these upper envelopes may intersect with each other.

Second division step: Using $n/\delta_t - 1$ vertical lines, partition the δ_t upper envelopes produced in the first recursive step which contain at most $\lambda_{h+2}(n)$ segments totally into $\lambda_{h+2}(n)/\delta_t$ equally-sized separated parts, and then recursively find the upper envelopes of each part in parallel.

Merge step: Connect the $\lambda_{h+2}(n)/\delta_t$ separated upper envelopes produced in the second recursive step from the left to the right to get the upper envelope of S'.

We can prove that $UE(S)$ can be found by the above algorithm in $O(\log^{1+\epsilon} N)$ time using $O(\frac{\lambda_{h+1}(N)}{\log^{\epsilon} N})$ processors for any $\epsilon > 0$ [3].

4. Using Prune Technique in the MDC

In a k-level MDC, the k division steps are organized hierarchically and executed one after another. Since the subproblems in each division step are usually solved recursively, there are k recursive processes which must be executed sequentially. If we can remove some recursive processes from the division steps, we can solve the problems more efficiently. In the following, we show how to use prune technique to remove the recursive processes of some division steps.

At the end of Sect. 3, we gave an algorithm of finding the upper envelope of a set S of N h-intersecting segments in the plane. Here, we restrict the segments to be line segments and show that by adding a prune technique we can find $UE(S)$ both time and cost optimally.

S has *a left (or right) base* if all the left (right) endpoints of the segments of S lie on a same vertical line. It is known that the problem of finding the upper envelope of N segments can be reduced, in $O(\log N)$ time using $O(N)$ processors, into that of finding the upper envelope of N segments which has the left (or the right) base [3]. Therefore, in the following we assume that S has a left base and find $UE(S)$. When S has a left base, the segments of S is 1-intersecting and the size of $UE(S)$ is at most $2N - 1$. We define set $S[l_1, l_2]$ $= \{$the subsegment of e lying between vertical lines l_1 and l_2 , $e \in S\}$.

Algorithm **FindUEWithBase**(S)

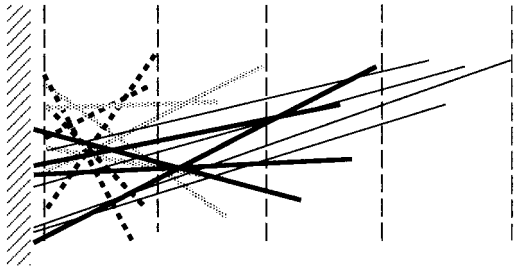
(Input) S and $P(S)$, where S is the set of N line segments in the plane and S is sorted in increasing slope, and $P(S)$ is the set consisting of the right endpoints of the segments of S and $P(S)$ is sorted in increasing x-coordinate.

(Output) $UE(S)$.

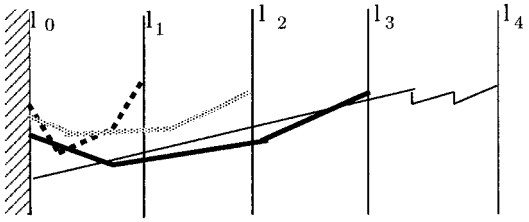
[method] Let S' be a subset of S with n segments. The segments of S' are sorted in increasing slope. $P(S')$ consists of the right endpoints of the segments of S' which are sorted in increasing x-coordinate. The following procedure finds $UE(S')$ with an $(n^{1/2}, n^{1/2})$-MDC. $UE(S)$ can be found by setting $S' = S$ and $n = N$.

Procedure **UEWithBase**(S')

(Base step) If $n = 1$, $UE(S') = S'$. This completes the algorithm.



(a) The first division step



(b) The second division step

Fig. 3 Algorithm for finding the upper envelope of the lines segments with a left base.

(First division step) Divide S' into $n^{1/2}$ subsets S_1, S_2, ..., $S_{n^{1/2}}$ such that S_i $(1 \le i \le n^{1/2})$ consists of the segments whose right endpoints belong to P_i, where P_i contains $((i-1)n^{1/2}+1)$th to $(in^{1/2})$th points of $P(S')$ (Fig. 3(a)). For each i, recursively construct $UE(S_i)$ in parallel.

(Second division step) Let l_i $(1 \le i \le n^{1/2})$ be the vertical line passing through the rightmost endpoint of S_i (Fig. 3(b)). Let $\overline{S}$ be the set of all the segments of $UE(S_1)$, $UE(S_2)$, ..., $UE(S_{n^{1/2}})$. Separate $\overline{S}$ which contains at most $2n-1$ segments by lines l_0, l_1, ..., $l_{n^{1/2}}$ into $n^{1/2}$ separated parts $\overline{S}[l_0, l_1]$, $\overline{S}[l_1, l_2]$, ..., $\overline{S}[l_{n^{1/2}-1}, l_{n^{1/2}}]$ such that each part contains $O(n^{1/2})$ segments (or subsegments), where l_0 is the left base of S'. For each i, recursively construct $UE(\overline{S}[l_{i-1}, l_i])$ in parallel.

(Merge step) Concatenate $UE(\overline{S}[l_{i-1}, l_i])$ from $i = 1$ to $i = n^{1/2}$ in the increasing order of i. □

In Second division step, one vertical line passes through at most $n^{1/2}$ upper envelopes. If in the above algorithm we move l_i $(1 \le i \le n^{1/2})$ such that it passes through the $2i|\overline{S}|/n^{1/2}$th end points of $\overline{S}$, then each part contains at most $3n^{1/2}$ segments. Except the recursive parts, the rest of the procedure of finding $UE(S')$ can be done in $O(\log n)$ time using $O(n)$ processors. Therefore the whole procedure can be executed in $T(n) = T(n^{1/2}) + T(3n^{1/2}) + O(\log n)$ time using $P(n) = \max\{n^{1/2}P(n^{1/2}),\ n^{1/2}P(O(n^{1/2})),\ n\}$, that is, it can be executed in $O(\log n \log\log n)$ time using $n\log^{\log 3} n$ processors. Therefore, $UE(S)$ can be found in $O(\log^N \log N \log N)$ time using $N\log^{\log 3} N$ processors. Actually, by adding a prune technique into procedure UEWithBase to get rid of the recursive structure in the second division step, we can find $UE(S)$ both cost and time optimal.

A line is said to be *induced from* a segment, if the segment lies on the line. A set of lines is said induced from a set S of segments, denoted as $L(S)$, if it consists of all the lines which are induced from the segments of S. Let set $\hat{S}$ consist of some subsegments of S. For any pair $s \in S$ and $s' \in \hat{S}$, s is *the origin* of s' if s' is a subsegment of s. We use $Ori(\hat{S}, S)$ to denote the subset of S which consists of all the origins of the segments in $\hat{S}$. Usually, $UE(S)$ is a simple polygonal chain, but when no confusion occurs, we also consider it as a set of the line segments which are the edges of the chain. The following lemma reveals the relation between the upper envelope of lines and that of segments.

Property 1: Let T be a set of line segments and $L(T)$ be the set of the lines induced from T. If T has both the left base l_1 and the right base l_2 (i.e., all the segments begin from l_1 and end at l_2), then $UE(T)$ is the portion of $UE(L(T))$ lying between l_1 and l_2, that is, $UE(T) = UE(L(T))[l_1, l_2]$. □

From the above property, if a set of line segments has both the left base and the right base, we can treat it as a set of lines. The problem of finding the upper envelope of n lines is dual to that of finding the convex hull of n points which can be solved both time and cost optimally in $O(\log n)$ time using $O(n)$ processors. The key point of our prune technique here is to remove the line segments which can be treated as lines from the recursive process of the second division step, i.e., remove them from $\overline{S}[l_{i-1}, l_i]$.

For a set T of segments and two vertical lines l_1 and l_2, where l_1 lies on the left of l_2, we can divide $T' = T\ [l_1, l_2]$ into two subsets: *passing-through-segment set* $Pas(T')$ and *falling-in-segment set* $Fal(T')$, where $t \in Pas(T')$ if t' begins from l_1 and ends at l_2 and $t \in Fal(T')$ if at least one endpoint of t' lies in the slab between l_1 and l_2 excluding l_1 and l_2. It is easily seen that $UE(T') = UE(UE(Pas(T')) \cup UE(Fal(T')))$. Thus, in the second division step of the algorithm, $UE(\overline{S}[l_i, l_{i+1}]) = UE(UE(Pas(\overline{S}[l_i, l_{i+1}])) \cup UE(Fal(\overline{S}[l_i, l_{i+1}])))$. From Property 1, $UE(Pas(\overline{S}[l_i, l_{i+1}]))$ can be found by the upper envelope algorithm for lines. That is, we can prune $Pas(\overline{S}[l_i, l_{i+1}])$ from $\overline{S}[l_i, l_{i+1}]$. Furthermore, from the definitions of S_i and $Fal(\overline{S}[l_i, l_{i+1}])$, $Ori(Fal(\overline{S}[l_i, l_{i+1}]), S) = S_i$, thus, $UE(Fal(\overline{S}[l_i, l_{i+1}])) = UE(S_i)[l_i, l_{i+1}]$. $UE(S_i)[l_i, l_{i+1}]$ can be easily gotten from $UE(S_i)$ which has been found in the first division step. Therefore, we remove the recursive process from the second division step successfully and complete it in $O(\log n)$ time using $O(n)$ processors. We get the formulas $T(n) = T(n^{1/2}) + O(\log n)$ and $P(n) = \max\{n^{1/2}P(n^{1/2}),\ n\}$. Thus, $T(n) = O(\log n)$ and $P(n) = n$. That is, the upper envelope of N line segments in the plane can be found both time and cost

optimally in $O(\log N)$ time using $O(N)$ processors.

Algorithm FindUEWithBase(S) gives some other results: (1) the upper envelope of N line segments in the plane can be found in $O(N \log\log N)$ time sequentially if the segments are sorted in slope and the segment endpoints are sorted in x-coordinate, and (2) the upper hull of N nonintersecting line segments can be found in $O(\log N)$ time using $O(N/\log N)$ processors, if the segment endpoints are sorted in x-coordinate. For more details, see [3].

5. Finding All the Farthest Neighbors of Convex Polygons

In this section, we give some other instances, in which the MDC is used more flexible and skillfully. Let $P = (p_1, \ldots, p_n)$ be a convex polygon represented by the sequence of the vertices listed in counter-clockwise order. We use $P[i \ldots j]$ to denote the contiguous vertices of P listed from p_i to p_j inclusive. We define distance $d(p, q)$ to be the length of the line segment connecting p and q. Vertex q is said to be *the farthest neighbor* of p in P, if $d(p, q) \geq d(p, q')$ for any other vertex q' of P. The farthest neighbor of p may be not unique. The farthest neighbors problem of P is to find a farthest neighbor in P for each vertex of P. In this setion, we first present a parallel algorithm which solves the problem in $O(\log^2 n/\log\log n)$ time using $O(n/\log n)$ processors, and then using the MDC improve it to run in $O(\log^{1+\epsilon} n/\log\log n)$ time using $O(n/\log^{\epsilon} n)$ processors.

5.1 A Basic Algorithm

Let $Q_1 = (u_0, u_1, \ldots, u_{s-1})$ and $Q_2 = (v_0, v_1, \ldots, v_{t-1})$ be two subsequences of $P = (p_1, p_2, \ldots, p_n)$, and let u and v be the vertices of Q_1 and Q_2, respectively. Vertex v is said to be the farthest neighbor of vertex u in Q_2, denoted as $FN(u : Q_2)$, if for any vertex w of Q_2, $d(u, v) \geq d(u, w)$. We use $AFN(Q_1 : Q_2)$ to denote the sequence $(FN(u_0 : Q_2), FN(u_1 : Q_2), \ldots, FN(u_{s-1} : Q_2))$. In the following we find $AFN(Q_1 : Q_2)$. The problem of finding all the farthest neighbors of a convex polygon P can be solved by finding $AFN(P : P)$.

We first show that if $s \geq \log t$, $AFN(Q_1 : Q_2)$ can be found in $O(s)$ time using $O(t/s)$ processors. The algorithm is as follows.

(1) If $s \geq t$, compute $AFN(Q_1 : Q_2)$ by sequential algorithm in $O(s)$ time [13], else do the following steps.

(2) Divide Q_2 into t/s subsequences such that the ith one is $Q_2^i = Q_2[is \ldots ((i+1)s - 1)]$ $(0 \leq i \leq t/s - 1)$. Compute $AFN(Q_1 : Q_2^i)$ for each i, in parallel.

(3) Decide $FN(u : Q_2)$ for each vertex u of Q_1 by computing the maximum of $FN(u : Q_2^0)$, $FN(u : Q_2^1)$, $\ldots$, $FN(q : Q_2^{t/s-1})$.

To avoid the concurrent reading of Q_1 in Step (2), we should first make t/s copies of Q_1, which can be done in $O(\log t)$ time using $O(t/\log t)$ processors [11]. Both Q_1 and Q_2^i $(0 \leq i \leq t/s - 1)$ have only s vertices, therefore, $AFN(Q_1, Q_2^i)$ can be found in $O(s)$ time sequentially [13]. Thus, step (2) can be executed in $O(s)$ time using $O(t/s)$ processors. Step (3) can be executed in $O(\log t)$ time using $O(t/\log t)$ processors by prefix maxima computing [11].

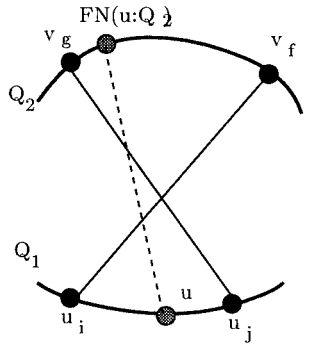


Fig. 4 The property of the farthest neighbors.

Lemma 1: Let Q_1 and Q_2 be two subsequences of a convex polygon with s and t vertices, respectively. $AFN(Q_1, Q_2)$ can be found in $O(s)$ time using $O(t/s)$ processors if $s \geq \log t$. □

The following property can be easily proven and it leads us to find $AFN(Q_1 : Q_2)$ with an ordinary DC.

Property 2: [5] Let u_i and u_j $(i \leq j)$ be the ith and the jth vertices of Q_1. If $FN(u_i : Q_2) = v_f$ and $FN(u_j : Q_2) = v_g$, then for any vertex u of $Q_1[i \ldots j]$, $FN(u : Q_2)$ belongs to $Q_2[f \ldots g]$ (Fig. 4). □

Given a subsequence $Q_1' = (u_{h_1}, u_{h_2}, \ldots, u_{h_r})$ of Q_1, where u_h is the hth $(h_1 \leq h \leq h_r)$ vertex of Q_1, if $AFN(Q_1' : Q_2) = (v_{k_1}, v_{k_2}, \ldots, v_{k_r})$, then Property 2 implies that $AFN(Q_1[h_i \ldots h_{i+1}] : Q_2) = AFN(Q_1[h_i \ldots h_{i+1}] : Q_2[k_i \ldots k_{i+1}])$ for all i $(1 \leq i \leq r)$. Let $d = \log t$, the following algorithm computes $AFN(Q_1 : Q_2)$.

Algorithm AllFarthestNeighbor(Q_1, Q_2)

[Input] Two subsequences of convex polygon P: $Q_1 = (u_0, u_1, \ldots, u_{s-1})$ and $Q_2 = (v_0, v_1, \ldots, v_{t-1})$

[Output] $AFN(Q_1 : Q_2)$

(Step 1) If $s \leq d$, find $AFN(Q_1 : Q_2)$ by Lemma 1. This completes the algorithm.

(Step 2) Let $Q_1' = (u_0, u_{(s-1)/d}, u_{2(s-1)/d}, \ldots, u_{s-1})$, i.e., the ith $(0 \leq i \leq d)$ vertex of Q_1' is the $(i(s-1)/d)$th vertex of Q_1. Find $AFN(Q_1' : Q_2)$ by Lemma 1.

(Step 3) Let $AFN(Q_1' : Q_2) = (v_{k_0}, v_{k_1}, \ldots, v_{k_d})$. Recursively find $AFN(Q_1[i(s-1)/d \ldots (i+1)(s-1)/d)] : Q_2[k_i \ldots k_{i+1}])$ for each i $(0 \leq i \leq d-1)$, in parallel. □

Since $d = \log t$, Step 1 and Step 2 can be executed in $O(d)$ time using $O(t/d)$ processors by Lemma 1. Step 3 is a recursive step. Let $T(s, t)$ and $P(s, t)$ denote the running time and the number of the processors of the algorithm, and let $t_i = k_{i+1} - k_i$, we have the

following recurrences:

$$T(s,t) \le \begin{cases} \max_{i=0}^{d-1} T(s/d, t_i) + O(d), \text{ if } s > d \\ O(d), \text{ if } s \le d \end{cases}$$

$$P(s,t) \le \begin{cases} \max\{\sum_{i=0}^{d-1} P(s/d, t_i), t/d\}, \text{ if } s > d \\ O(t/d), \text{ if } s \le d \end{cases}$$

Thus, $T(s,t) = O(d \log s / \log d)$ and $P(s,t) = O(t/d)$.

Lemma 2: Let Q_1 and Q_2 be two subsequences of a convex polygon with s and t vertices each. $AFN(Q_1 : Q_2)$ can be computed in $O(d \log s / \log d)$ time using $O(t/d)$ processors, where $d = \log t$. □

We compute $AFN(P : P)$ by setting $Q_1 = Q_2 = P$ and $s = t = n$. From Lemma 2, $AFN(P : P)$ can be computed in $O(\log^2 n / \log\log n)$ time using $O(n/\log n)$ processors.

5.2 Making the Algorithm Fast by the MDC

Let $d_h = 2^{\log^{h/(h+1)} s}$ $(h \ge 1)$, where s is the size of Q_1 and h is a positive integer. Let $Q'_1 = (x_1, x_2, \ldots, x_{s'})$ and $Q'_2 = (y_1, y_2, \ldots, y_{t'})$ be the subsequences of Q_1 and Q_2 with s' and t' vertices, respectively. We find $AFN(Q'_1 : Q'_2)$ by calling the following procedure ImprovAFN(Q'_1, Q'_2, $h+1$). $AFN(Q_1 : Q_2)$ can be found by setting $Q'_1 = Q_1$, $Q'_2 = Q_2$, $s' = s$ and $t' = t$.

Procedure ImprovAFN(Q'_1, Q'_2, $h+1$)

(Base step) If $h = 1$, compute $AFN(Q'_1 : Q'_2)$ by Lemma 2 in $O(d \log s / \log d)$ time using $O(t/d)$ processors. This completes the procedure. Now let $h > 1$. If $s' \le d_h$, recursively compute AFN $(Q'_1 : Q'_2)$ by ImprovAFN (Q'_1, Q'_2, h). Else compute AFN $(Q'_1 : Q'_2)$ as follows.

(First division step) Select a subsequence $\hat{Q}_1 = (x_0, x_{(s'-1)/d_h}, x_{2(s'-1)/d_h}, \ldots, x_{s'-1})$ from Q'_1, where $x_{i(s'-1)/d_h}$ is $i(s'-1)/d_h$th $(0 \le k \le d_h)$ vertex of Q'_1. Recursively compute $AFN(\hat{Q}_1 : Q'_2)$ by ImprovAFN($\hat{Q}_1$, Q'_2, h).

(Second division step) Suppose that AFN $(\hat{Q}_1 : Q'_2) = (y_{j_0}, y_{j_1}, \ldots, y_{j_{d_h}})$. Divide Q'_1 into $Q'_1(k) = Q'_1[k(s'-1)/d_h \ldots (k+1)(s'-1)/d_h]$ and divide Q'_2 into $Q'_2(k) = Q'_2[j_k \ldots j_{k+1}]$ $(0 \le k \le d_h - 1)$. For each i recursively compute AFN $(Q'_1(k) : Q'_2(k))$ by ImprovAFN($Q'_1(k)$,$Q'_2(k)$,$h+1$), in parallel. □

In the first division step, the division is not obvious: Q'_1 is divided into $\hat{Q}_1$ and $Q'_1 - \hat{Q}_1$ and only $AFN(\hat{Q}_1 : Q'_2)$ is computed recursively. The output of the second division step is $AFN(Q'_1 : Q'_2)$, therefore, the merge step is not necessary.

Lemma 3: If procedure ImprovAFN(Q'_1, Q'_2, h) $(h \ge 2)$ computes $AFN(Q'_1, Q'_2)$ in $O(\log^{1+1/h} s' / \log d)$ time using $O(t' / \log^{1/h} s')$ processors, ImprovAFN (Q'_1, Q'_2, $h+1$) computes $AFN(Q'_1, Q'_2)$ in $O(\log^{1+1/(h+1)} s' / \log d)$ time using $O(t' / \log^{1/(h+1)} s)$ processors.

Proof: Let $T(s', t')$ be the running time and $P(s', t')$ be the number of the processors for finding $AFN(Q'_1 : Q'_2)$ by ImprovAFN(Q'_1,Q'_2,$h+1$) $(h \ge 2)$, where the number of the vertices of Q'_1 and Q'_2 are s' and t', respectively. In ImprovAFN(Q'_1,Q'_2,$h+1$), if $s' \le d_h$ ImprovAFN(Q'_1,Q'_2,h) is called in Base Step, else ImprovAFN($\hat{Q}_1$,Q'_2,h) is called in First division step, where $|\hat{Q}_1| \le d_h$. According to the condition of this lemma, they can be solved in $O(\log^{1+1/h} d_h / \log d)$ time using $O(t' / \log^{1/h} d_h)$ processors. Let $t_k = j_{k+1} - j_k + 1$ and $M = \max_{k=0}^{d_h - 1} t_k$. In the second division step, d_h subproblems: finding AFN $(Q'_1(k) : Q'_2(k))$ for each of k $(0 \le k \le d_h - 1)$ are recursively solved by ImprovAFN($Q'_1(k)$,$Q'_2(k)$,$h+1$), where the sizes of $Q'_1(k)$ and $Q'_2(k)$ are s'/d_h and t_k. Therefore, we have the following recurrences.

$$T(s', t') \le \begin{cases} T(s'/d_h, M) + O(\log^{1+1/h} d_h / \log d), \\ \text{if } s' > d_h \\ O(\log^{1+1/h} d_h), \quad \text{if } s' \le d_h \end{cases}$$

$$P(s', t') \le \begin{cases} \max\{\sum_{i=0}^{d_h - 1} P(s'/d_h, t_i), \\ \quad O(t' / \log^{1/h} d_h)\}, \quad \text{if } s' > d_h \\ O(t' / \log^{1/h} d_h), \quad \text{if } s' \le d_h \end{cases}$$

By recalling that $\log d_h = \log^{h/(h+1)} s$, from the recurrences we can easily get $T(s,t) = O(\log^{1/(h+1)} s \log^{1+1/h} d_h / \log d) = O(\log^{1+1/(h+1)} s / \log d)$, and $P(s,t) = O(\sum_{i=0}^{d_h - 1} t_i / \log^{1/h} d_h) = O(t / \log^{1/(h+1)} s)$. □

Theorem 1: Let Q_1 and Q_2 be two sequences of a convex polygon with s and t vertices each. $AFN(Q_1, Q_2)$ can be computed in $O(\log^{1+1/h} s / \log d)$ time using $O(t / \log^{1/h} s)$ processors for any integer $h \ge 1$.

Proof: When $h = 1$, algorithm ImprovAFN(Q_1,Q_2,h) finds $AFN(Q_1 : Q_2)$ by Lemma 2 in $O(d \log s / \log d) = O(\log^2 s / \log d)$ time using $O(t/d) = O(t / \log s)$ processors. In general, assume that algorithm ImprovAFN(Q_1,Q_2,h) find $AFN(Q_1, Q_2)$ in $O(\log^{1+1/h} s / \log d)$ time using $O(t / \log^{1/h} s)$ processors for $h \ge 1$. By using Lemma 3, it finds $AFN(Q_1 : Q_2)$ in $O(\log^{1+1/(h+1)} s / \log d)$ time using $O(t / \log^{1/(h+1)} s)$ processors. Therefore, it finds $AFN(Q_1 : Q_2)$ in $O(\log^{1+\epsilon} s / \log d)$ time using $O(t / \log^{\epsilon} s)$ processors. □

For any given $0 < \epsilon \le 1$, let $h = \lceil 1/\epsilon \rceil$. By recalling that $d = \log t$ and P has n vertices, algorithm ImprovAFN(P, P, h) finds $AFN(P : P)$ in

$O(\log^{1+\epsilon} n/\log\log n)$ time using $O(n/\log^{\epsilon} n)$ processors for any $\epsilon \geq 0$.

Let A be a matrix having n rows and m columns with real entries, let A_i and A^j denote the ith row and the jth column of A, respectively, and let $j(i)$ be the smallest column index j such that $A(i,j)$ equals the maximum value in A_i. A is said to be monotone if $j(i_1) \leq j(i_2)$ for all $1 \leq i_1 \leq i_2 \leq n$. A is totally monotone if every submatrix of A is monotone. It is easy to prove that the algorithm of computing $AFN(P:P)$ can be used for solving the row maxima problem of totally monotone matrix [1]. Therefore, the row maxima problem of totally monotone matrix can be solved in $O(\log^{1+\epsilon} n/\log\log n)$ time using $O(n/\log^{\epsilon} n)$ processors for any $\epsilon \geq 0$.

6. Conclusions

We introduced a new technique called Multi-level divide-and-conquer (MDC). The MDC is the same as the ordinary DC except that one division step is replaced by more than one hierarchically organized division step. We investigated the paradigm of MDC and showed that it was an excellent method for designing parallel algorithms. We also showed that by adjusting the parameters and adding some other auxiliary approaches such as pruning technique, the MDC could become much more efficient. To discuss the structures and the mechanisms of the MDC, we used the algorithms of the following problems: the convex hull problem of discs, the upper envelope problem of segments, the farthest neighbors problem of convex polygons, and the row maxima problem of totally monotone matrices. The last two algorithms are first presented. We showed that by using the MDC the results of the above problem could be improved significantly. Our discuss is based on the EREW PRAM, but the conclusions can be applied to any parallel computation models.

References

[1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric applications of a matrix-searching algorithm," Algorithms, no.2, pp.195–208, 1987.
[2] M.J. Atallah and S.R. Kosaraju, "An efficient parallel algorithm for the row minima of a totally monotone matrix," J. Algorithms, vol.13, no.3, pp.394–413, 1992.
[3] W. Chen and K. Wada, "On computing the envelope of segments in parallel," Proc. 1998 International Conference on Parallel Processing, 1998.
[4] W. Chen, K. Wada, K. Kawaguchi, and D.Z. Chen, "Finding the convex hull of discs in parallel," International J. Computational Geometry and Applications, vol.8, no.3, pp.305–319, 1998.
[5] W. Chen, T. Masuzawa, and N. Tokura, "Find all farthest neighbors in convex polygons in parallel," IEICE Technical Report, COMP93-11, 1993.
[6] M.T. Goodrich, "Using approximation algorithms to design parallel algorithms that may ignore processors allocation," Proc. 34nd Annual Symposium on Foundations of Computer Science, pp.711–722, 1991.
[7] M.T. Goodrich, S.B. Shauck, and S. Guha, "Parallel methods for visibility and shortest-path problems in simple polygons," Algorithmica, vol.8, no.5/6, pp.461–486, 1992.
[8] S. Guha, "Parallel computation of internal and external farthest neighbors in simple polygons," J. Computational Geometry & Applications, vol.2, no.2, pp.175–190, 1992.
[9] D. Hart and M. Sharir, "Nonlinearity of davenport-schinzel sequences and of generalized path compression schemes," Combinatorica, no.6, pp.151–177, 1989.
[10] J. Hershberger, "Finding the upper envelope of n line segments in $O(n\log n)$ time," Inf. Process. Lett., vol.33, no.4, pp.169–174, 1989.
[11] J. JaJa, An Introduction to Parallel algorithms, Addison Wesley Publishing Company, 1992.
[12] M. Sharir and P.K. Agarwal, Davenport-Schinzel Sequences and Their Geometric Applications, Cambridge University Press, 1995.
[13] S. Suri, Computing geodesic furthest neighbors in simple polygons, J. Comput. Syst. Sci., no.39, pp.220–235, 1989.
[14] M. Yoshimori, W. Chen, and N. Tokura, "An efficient convex hull parallel algorithm for discs," IEICE Trans., vol.J78-D-I, no.5, pp.501–503, May 1995.

Wei Chen received the B.A. degree in mathematics from Shanghai Marine University in 1982, and received M.E., and Ph.D. degrees from the Department of Information Engineering, Faculty of Engineering Science, Osaka University in 1991 and 1994. Since 1994 she has been working at the Department of Electrical and Computer Engineering, Nagoya Institute of Technology. She is now an associate professor of that university. Her research interests include parallel and distributed computing, computational geometry and graph theory. She is a member of ACM, IEEE, LA Symposium and IPSJ.

Koichi Wada graduated in 1978 from the Department of Information Engineering, Faculty of Engineering Science, Osaka University, and received his M.S. and Ph.D. degrees both from the same university in 1980 and 1983, respectively. He was a research associate at Osaka University during 1983–1984. In 1984 he joined Nagoya Institute of Technology, where he is currently a professor in the Department of Electrical and Computer Engineering. He was a visiting associate professor at University of Minnesota, Duluth and University of Wisconsin, Milwaukee during 1987–1988. His research interests include graph theory, parallel/distributed algorithms and VLSI theory. Dr. Wada is a member of IEEE, ACM, LA Symposium, Japan SIAM and IPSJ.