HAL
open science

On the multisearching problem for hypercubes

Mikhail J. Atallah, Andreas Fabri

To cite this version:

Mikhail J. Atallah, Andreas Fabri. On the multisearching problem for hypercubes. [Research Report] RR-1990, INRIA. 1993. inria-00074682

HAL Id: inria-00074682
https://inria.hal.science/inria-00074682v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports de Recherche

N°1990

Programme 4

Robotique, Image et Vision

ON THE MULTISEARCHING PROBLEM FOR HYPERCUBES

Mikhail J. Atallah
Andreas Fabri

23 Juin 1993

Institut National
de Recherche
en Informatique
et en Automatique

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

On the Multisearching Problem for Hypercubes*

A Propos du Problème des Recherches Multiples sur les Hypercubes

Mikhail J. Atallah† Andreas Fabri‡

Programme 4 : Robotique, Image et Vision.

Keywords: Computational Geometry, Parallel Algorithms, Hypercube, Multisearching, Pointlocation

* *This work was supported in part by the National Science Foundation under Grant CCR-9202807, and by the ESPRIT Basic Research Action Nr. 7141 (ALCOM II).*

† *Department of Computer Science, Purdue University, West Lafayette, IN 47906, USA. Email:* `mja@cs.purdue.edu`

‡ *INRIA, BP 93, 06902 Sophia-Antipolis Cedex, France. Email:* `fabri@sophia.inria.fr`

Abstract

We build on the work of Dehne and Rau-Chaplin and give improved bounds for the *multisearch problem* on a hypercube. This is a parallel search problem where the elements in the structure S to be searched are totally ordered, but where it is not possible to compare in constant time any two given queries q and q'. This problem is fundamental in computational geometry, for example it models planar point location in a slab. More precisely, we are given on a n-processor hypercube a sorted n-element sequence S, and a set Q of n queries, and we need to find for each query $q \in Q$ its location in the sorted S. Note that one cannot solve this problem by sorting $S \cup Q$, because every comparison-based parallel sorting algorithm needs to compare a pair $q, q' \in Q$ in constant time. We present an improved algorithm for the multisearch problem, one that takes $O(\log n(\log\log n)^3)$ time on a n-processor hypercube. This essentially replaces a logarithmic factor in the time complexities of previous schemes by a $(\log\log n)^3$ factor. The hypercube model for which we claim our bounds is the standard one, with $O(1)$ memory registers per processor, and with one-port communication. Each register can store $O(\log n)$ bits, so that a processor knows its ID.

Résumé

En partant des travaux de Dehne et Rau-Chaplin, nous améliorons les temps de calcul pour le problème des recherches multiples sur un hypercube. Il s'agit d'une recherche parallèle, où les éléments dans la structure S sont totalement ordonnés, mais où il n'est pas possible de comparer deux requètes en temps constant. C'est un problème fondamental en géometrie algorithmique, par exemple pour la location des points dans la subdivision d'une tranche. Plus précisement, étant donné un hypercube à p processeurs, une séquence triée S à n éléments et un ensemble Q de n requètes, on veut localiser chaque $q \in Q$ dans S. Notons, qu'on ne peut pas trier $S \cup Q$, car chaque algorithme de tri basé sur les comparaisons doit être capable de comparer une paire $q, q' \in Q$ en temps constant. La complexité en temps de l'algorithme presenté dans cet article est de $O(\log n(\log\log n)^3)$ sur un hypercube à n processeurs. Nous avons donc remplacé un facteur logarithmique dans le temps de calcul des precedentes algorithmes par un facteur $(\log\log n)^3$. Le modèle d'hypercube est le modèle standard avec $O(1)$ registres de mémoire locale par processeur et avec communication *one-port*. Chaque registre peut stocker $O(\log n)$ bits, et chaque processeur peut donc connaître son identité.

1 Introduction

Consider the situation depicted in Figure 1: We have a horizontal slab partitioned by a set S of n nonintersecting segments. For a set Q of n points, we need to determine for each point which region of the slab it belongs to. Both the segments and the points are initially stored in a n processor hypercube.

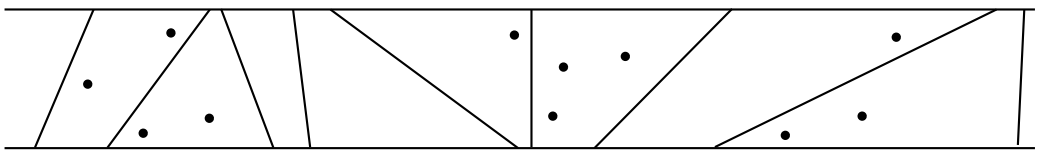

Figure 1: Point location in a subdivided slab.

This problem would be trivial, if the partitioning segments were vertical, but the fact that they are slanted makes it impossible to solve the problem by (e.g.) simply mergesorting $S \cup Q$ according to x-coordinates. The method we give for solving this multisearch problem works for more general versions of this problem: The basic assumption is that any pair x, y in a processor can be compared in constant time if $x \in S \cup Q$ and $y \in S$, but not so if both x and y are in Q. In [DR89] Dehne and Rau-Chaplin gave an $O(\log^2 n)$ time algorithm for this problem. Their algorithm is easy to implement and thus of practical interest, and they later generalized it for doing fractional cascading on a hypercube [DFR92]. A randomized $O(\log n)$ time scheme for multisearching was given by Reif and Sen [RS91]. Since searching is related to sorting and there is a deterministic $O(\log n \log\log n)$ time sorting algorithm [CP90], the question was open, if there exists an algorithm for the multisearch problem that runs faster than $O(\log^2 n)$. This paper gives a step in the right direction, by presenting an algorithm with time complexity $O(\log n(\log\log n)^3)$ for a n processor hypercube. Our result is more of theoretical than of practical interest, because it uses the sorting algorithm of [CP90] as a subroutine. However, any practical improvement to sorting would immediately make our algorithm more practical.

The paper is organized as follows. In Section 2 we review the definition of a hypercube interconnection network and some basic algorithms for this parallel machine. Then in Section 3 we sketch a very preliminary solution that is worse than the one we claim, but that serves as a "warmup" for the later improved algorithms. Section 4 gives an algorithm that is almost as good as what we claim, except that it requires

each processor to have $\Theta(\log\log n)$ memory registers (rather than $O(1)$ registers). Section 5 gives the algorithm that achieves the bounds we claim. Section 6 concludes by discussing some implementation issues and details.

2 The Model of Computation

This section is a brief review of the model, and in particular of some operations on that model that we will make use of.

The hypercube model for which we claim our bounds is the standard one, with $O(1)$ memory registers per processor, and with one-port communication. Each register can store $O(\log n)$ bits, so that a processor knows its ID. Recall that a hypercube of dimension d consists of $n = 2^d$ processors which are uniquely labeled with bit-strings of length d. Two processors are connected *along dimension* i, iff their labels differ in exactly the ith bit. In this paper we are interested in SIMD (Single Instruction Multiple Data) machines, that is, all processors execute the same instruction simultaneously. An instruction is either an operation on data in the local memory, or a communication step with a processor adjacent along a particular dimension. An instruction takes time $O(1)$.

We shall use as subroutines certain operations on sequences of size n, with time complexity $O(\log n)$. These operations include *segmented parallel prefix* and *monotonic routing* which together allow a *monotonic read.* Thus the read is monotonic, iff for any pair of processors p_i and p_j, with $i < j$, which want to read data on processors p_h and p_k, we have $h \leq k$. We refer to [Lei92, NS81] for a detailled discussion of these operations. Another operation we use is sorting n numbers, which can be done in time $O(\log n \log\log n)$ [CP90].

We shall occasionally need to solve problems on *subcubes* of a hypercube. We can obtain subcubes of dimension $\hat{d} \leq d$ by selecting all $2^{\hat{d}}$ nodes matching a constant bitpattern on $d-\hat{d}$ bits. Two patterns which occur frequently are the following. Fixing the first $d/2$ bits yields $\sqrt{n}$ *consecutive* subcubes, fixing the last $d/2$ bits yields $\sqrt{n}$ *interlaced* subcubes. Using the interlaced subcubes we can easily copy the contents of one of the consecutive subcubes to the other consecutive subcubes in $O(\log n)$ time.

3 A Preliminary $o(\log^2 n)$ Time Solution

The $O(\log^2 n)$ time complexity of the algorithm in [DR89] results from the fact that the queries in Q perform a *binary* search; that is, each query performs $\log n$ comparisons with elements of S and, in order to read the element of S for their next comparison, the queries perform a monotonic read. One thought that comes to mind in trying to improve on this algorithm is to try to perform a *rootish* search, e.g., a $\sqrt{n}$-ary search (recursively), to bring the height of the search tree down to $\log\log n$. In such a scheme the outdegree of a node v of the search tree would be $n^{(\frac{1}{2})^k}$, where k is the level of v in the search tree, $1 \leq k \leq \log\log n$. However, in such a scheme, a typical search tree node (say) v would have too many children: To decide which child of v to go to, the queries "currently at v" could recursively solve a similar problem restricted to the children of v. Using this idea, the following (flawed) algorithm might come to mind:

1. Partition Q into $\sqrt{n}$ chunks of size $\sqrt{n}$ each, and solve each chunk recursively with respect to that chunk's own private copy of $\hat{S}$ where $\hat{S}$ is a $\sqrt{n}$-sample of S. That is, $\hat{S}$ consists of $\sqrt{n}$ evenly spaced elements of S: The $\sqrt{n}$th, $2\sqrt{n}$th, ..., nth elements of S.

2. Let $S_1, S_2, \ldots, S_n$ be the partition of S induced by the elements of $\hat{S}$. Let Q_i denote the subset of Q which belongs in S_i. Partitioning Q into $Q_1, \ldots, Q_{\sqrt{n}}$ is easily done by sorting the queries of Q based on which S_i they belong to.

3. Since Q_i could be much larger than $\sqrt{n}$, we do not want to recurse on Q_i itself, so we partition it into $m_i = \lceil Q_i/\sqrt{n} \rceil$ pieces, call them $Q_{i,j}$, $1 \leq j \leq m_i$. Recursively solve in parallel each $Q_{i,j}$ with respect to that $Q_{i,j}$'s own private copy of S_i (making m_i copies of S_i, etc). There are no more than $2\sqrt{n}$ such recursive calls: At most $\sqrt{n}$ *full* recursive calls for which $|Q_{i,j}| = \sqrt{n}$, and another $\sqrt{n}$ *non-full* recursive calls for which $|Q_{i,m_i}| < \sqrt{n}$.

The alert reader has undoubtedly observed many flaws in the above:

Difficulty 1: Carrying out Step 1 requires $O(\log\log n)$ registers in each processor. This is because the total space $S(n)$ satisfies the recurrence $S(n) = \sqrt{n}S(\sqrt{n}) +$

$c_1 n$, $S(1) = c_2$, where c_1, c_2 are constants. This implies $S(n) = \Theta(n \log\log n)$, which contradicts our assumption that each processor has $O(1)$ registers.

Difficulty 2: Step 3 requires $n \log n$ processors, because of the excessive duplication of the S_i's. More specifically, the number of processors $P(n)$ satisfies the recurrence $P(n) \geq 2\sqrt{n}P(\sqrt{n})$, which implies that $P(n) = \Omega(n \log n)$. The factor of 2 in the $P(n)$ recurrence comes about because we are solving the non-full subproblems in parallel with the full subproblems. If we try to avoid this factor of 2 by doing one additional parallel recursive call for the non-full subproblems (i.e., *after* the call for the full ones return), then we damage the time complexity: There would then be *three* consecutive recursive calls, and an unwelcome factor of $(\log n)^{1.59}$ shows up in the time complexity (because it would satisfy the recurrence $T(n) = 3T(\sqrt{n}) + c \log n \log\log n$).

Treating Difficulty 1 is postponed until Section 5. The way we get around Difficulty 2 is by treating the full subproblems in a different way from the non-full ones. This will be the subject of the next section.

4 Improving the Time Complexity

In this section we temporarily assume that each of the n processors available has $O(\log\log n)$ memory registers. This is needed not only because of Difficulty 1, but also because the way we get around Difficulty 2 will itself require a factor of $\log\log n$ extra space. In the next section we show how to get rid of this assumption. Subject to this assumption, we now show how to achieve $O(\log n(\log\log n)^3)$ time.

We have already argued that Steps 1 and 2 pose no problem so long as we have $O(\log\log n)$ memory registers in each of the n processors. The main issue is how to avoid one of the three recursive calls mentioned in the previous section, when discussing Difficulty 2. We create $\sqrt{n}$ subproblems of size $\sqrt{n}$ each, where each subproblem can be of two types: Either a full subproblem in the same sense as in Section 3, or a subproblem *derived* from the non-full subproblems of Section 3 in the following way.

Recall that the non-full subproblems of Step 3 are described by the queries Q_{i,m_i} and the elements S_i. For a non-full problem, let $l_i = |Q_{i,m_i}|$; note that $l_i < \sqrt{n}$

since the subproblem is assumed to be non-full. Let Q' be the concatenation of $Q_{1,m_1}, \ldots, Q_{\sqrt{n},m_{\sqrt{n}}}$. Partition Q' into ℓ contiguous chunks of size $\sqrt{n}$ each, call them $Q'_1, \ldots, Q'_\ell$, and observe that the number of full subproblems is $\sqrt{n} - \ell$.

We create for each Q'_j a corresponding set of elements $S'_j \subset S$, in the following way. Each Q_{i,m_i} that has a nonempty intersection with Q'_j contributes to S'_j a subset $S'_{i,j} \subset S_i$ defined as follows. Let $l_{i,j} = |Q_{i,m_i} \cap Q'_j| > 0$. Note that for a particular j, at most two indices i have $l_{i,j} < l_i$ (for all the other i's such that $l_{i,j} > 0$, we have $l_{i,j} = l_i$). Then $S'_{i,j}$ consists of $l_{i,j}$ evenly spaced elements of S_i. It is not hard to see that computing all the Q'_j's and S'_j's can be done in $O(\log n)$ time by using monotonic routing operations.

The ℓ derived subproblems (Q'_j, S'_j), $1 \leq j \leq \ell$, are solved recursively in parallel with the full subproblems of Section 3. Hence the second parallel recursive call consists of a total of $\sqrt{n}$ subproblems of size $\sqrt{n}$ each: The $\sqrt{n} - \ell$ full ones, and the ℓ derived ones.

Our main problem now is in using the outcome of this second parallel recursive call in order to obtain the overall solution. Clearly this is not an issue for the full subproblems. But for the derived subproblems (Q'_j, S'_j), $1 \leq j \leq \ell$, it is not clear. We explain how this is done for a typical Q'_j, S'_j. It suffices to show how this is done for the elements in $Q_{i,m_i} \cap Q'_j$, with $l_{i,j} > 0$. The recursive call for (Q'_j, S'_j) tells us the positions of the elements of $Q_{i,m_i} \cap Q'_j$ with respect to $S'_{i,j}$. Letting μ_k be the number of queries in $Q_{i,m_i} \cap Q'_j$ that end up in the k-th position within $S'_{i,j}$, $1 \leq k \leq l_{i,j}$, we further locate these μ_k queries in their correct positions in S_i in logarithmic time and $O(\mu_k \sqrt{n}/l_{i,j})$ processors. This is done by creating all the query-element pairs needed (there are μ_k queries and $|S_i|/|S'_{i,j}| = \sqrt{n}/l_{i,j}$ elements). That there are enough processors to do this is seen by the following analysis. For each $Q_{i,m_i} \cap Q'_j$ with $0 < l_{i,j} < \sqrt{n}$, the number of processors needed is

$$\sum_{1 \leq k \leq l_{i,j}} \mu_k \lceil \sqrt{n}/l_{i,j} \rceil = \sqrt{n} + l_{i,j} < 2\sqrt{n},$$

where we used the fact that $\sum_{1 \leq k \leq l_{i,j}} \mu_k = l_{i,j}$. Since there are at most $2\sqrt{n}$ such sets $Q_{i,m_i} \cap Q'_j$ that have $0 < l_{i,j} < \sqrt{n}$, the total number of processors is less than $(2\sqrt{n})(2\sqrt{n}) = 4n$. We do not have to worry about the factor 4 coming in, as this "conquer" step is not recursive in nature.

Since there are two recursive calls and the conquer step involves a constant number of monotonic routing steps and a single sorting step, the time complexity satisfies the recurrence $T(n) = 2T(\sqrt{n}) + c_1 \log n \log\log n$, $T(1) = c_2$, where c_1, c_2 are constants. This implies that $T(n) = O(\log n(\log\log n)^2)$.

The processor complexity is linear, since it satisfies the recurrence $P(n) = \max\{c_1 n, \sqrt{n}P(\sqrt{n})\}$, $P(1) = c_2$, where c_1, c_2 are constants.

The scheme uses a factor of $\log\log n$ too much space, because of the duplication of the subsets of S needed by the various recursive calls, and because it needs, in addition to the space taken by the recursive calls, to store S for completing the solution when the recursive calls return. Unlike Section 3, this requirement to set aside storage for (possibly all of) S, before recursing on many copies of only portions of S, occurs at two different places in the algorithm.

So far we have established the following.

Lemma 1 *Given a multisearch problem (Q, S) with $|Q| = |S| = n$, we can solve it in time $O(\log n(\log\log n)^2)$ on a n-processor hypercube, each processor of which has $O(\log\log n)$ registers.*

The next section deals with the space issue.

5 Improving the Space Complexity

We first observe that instead of having n processors with $O(\log\log n)$ registers each, we can transform the algorithm of the previous section, so that it runs on a $n \log\log n$ processor hypercube with $O(1)$ registers on each processor without any sacrifice in the time complexity. To see this, recall how the n-processor, $\log\log n$-space-per-processor algorithm of the previous section used the $\log\log n$ extra registers at each processor: If we think of these registers as belonging to *layers* numbered $1, \ldots, \log\log n$, then the information at layer j was needed only when the recursive call associated with layer $j+1$ returned (in the "conquer" step of the parallel divide and conquer). We can thus use an extra factor of $\log\log n$ in the processor complexity to simulate these $\log\log n$ layers: A cluster of $\log\log n$ of the $O(1)$-space processors can mimic a single $\log\log n$-space processor by (i) using a designated leader of the cluster to do all the calculations, and (ii) using all the other non-leader processors of the cluster only for

storage. Of course, reading from this storage by the leader now takes $O(\log\log n)$ time instead of constant time, but this is acceptable since there is only one such "read" for each layer j (in fact we could even afford to spend $O(\log n \log\log n)$ time for that "read" of layer j, since this is the time bottleneck we face anyway in other portions of the computation that follows that "read"). We summarize these observations in the following.

Lemma 2 *Given a multisearch problem (Q, S) with $|Q| = |S| = n$, we can solve it in time $O(\log n(\log\log n)^2)$ on a $n \log\log n$-processor hypercube, each processor of which has $O(1)$ registers.*

We now use the above lemma to solve the multisearch problem using only n processors with $O(1)$ registers each, by solving smaller problems one after the other. More exactly, solving $\log\log n$ problems with only $n/\log\log n$ queries each, we can use the result from the previous section and Lemma 2, as we have enough processors. The algorithm that uses only n processors is as follows:

1. Partition Q into $t = \log\log n$ chunks of size n/t each, call these $Q_1, \ldots, Q_t$.

2. Partition S into $s = n/t$ chunks of size t each, call these $S_1, \ldots, S_s$. Call $\hat{S}$ the set of s elements that are at the boundaries of adjacent chunks.

3. For $i = 1, \ldots, t$ in turn, do the following:

   (a) Process Q_i against $\hat{S}$. By using Lemma 2, this takes $O(\log n(\log\log n)^2)$ time using the n available processors. Let $Q_{i,j}$ denote the subset of Q_i that goes into S_j, $1 \le j \le s$.

   (b) In parallel for all j, locate the queries of $Q_{i,j}$ in S_j. This can be done in logarithmic time by creating all query-element pairs (q, e), with $q \in Q_{i,j}$, $e \in S_j$, and $1 \le j \le s$. The number of processors needed is

$$\sum_{j=1}^{s} (|Q_{i,j}| \cdot |S_j|) = (\sum_{j=1}^{s} |Q_{i,j}|) \cdot t = |Q_i| \cdot t = n.$$

Each iteration of Step 3 takes $O(\log n(\log\log n)^2)$ time, and t of them are done one after the other, for a total of $O(\log n(\log\log n)^3)$ time. We thus obtain the following result.

Theorem 3 *Given a multisearch problem (Q, S) with $|Q| = |S| = n$, we can solve it in time $O(\log n(\log\log n)^3)$ on a n-processor hypercube, each processor of which has $O(1)$ registers.*

6 Implementation Notes

Note that we tacitly assumed that n was a perfect square, and thus the size of the problem on each level k of the recursion, namely $n^{\frac{1}{2^k}}$, was a power of two. The following observation is useful. If n is a power of two, then either $\sqrt{n}$ or $\sqrt{n/2}$ is a power of two. If we are in the latter case we solve two problems of size $n/2$ on the two interlaced hypercubes (with the last bit of the processor label fixed), with two interlaced subsequences of S. The final result can then easily be obtained by a comparison with the neighbour element in S.

Another detail that we did not dwell on is how to solve, in logarithmic time, a problem consisting of n_1 queries and n_2 elements by using $O(n_1 n_2)$ processors. This, however, is straightforward to do using standard hypercube operations (it is "brute force", since it uses so many processors).

References

[B68] K.E. Batcher. *Sorting networks and their applications.* Proc. AFIPS Spring Joint Computer Conference, pp. 307-314, 1968.

[CP90] R. Cypher, C.G. Plaxton. *Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers.* ACM Proceedings of the 22th Annual ACM Symposium on Theory of Computing, pp. 193–203, 1990.

[DR89] F. Dehne, A. Rau–Chaplin. *Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry*, Internal Report SCS–TR–152, Carlton University, 1989.

[DFR92] F. Dehne, A. Ferreira, A. Rau–Chaplin. *Parallel fractional cascading on hypercube multiprocessors.* Computational Geometry: Theory and Applications 2, pp. 141-167, 1992.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[NS81] D. Nassimi and S. Sahni. *Data broadcasting in SIMD computers.* IEEE Transactions on Computers, C-30(2):101–107, February 1981.

[RS91] J.H. Reif, S. Sen. *Randomized Algorithms for Binary Search and Load Balancing on Fixed Connection Networks with Geometric Applications.* Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures, 1990.