Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP Model*

(Extended Abstract)

Armin Bäumker, Wolfgang Dittrich
and Friedhelm Meyer auf der Heide

Department of Mathematics and Computer Science
and Heinz Nixdorf Institute, University of Paderborn
33095 Paderborn, Germany

Abstract

In this paper we design and analyse parallel algorithms with the goal to get exact bounds on their speed-ups on real machines. For this purpose we define an extension of Valiant's BSP model, BSP*, that rewards blockwise communication, and uses Valiant's notion of c-optimality. Intuitively a c-optimal parallel algorithm for p processors achieves speed-up close to p/c. We consider the Multisearch problem: Assume a strip in 2D to be partitioned into m segments. Given n query points in the strip, the task is to locate, for each query, its segment. For $m \leq n$ we present a deterministic BSP* algorithm that is 1-optimal, if $n = \Omega(p \log^2 p)$. For $m > n$, we present a randomized BSP* algorithm that is $(1+\delta)$-optimal for arbitrary $\delta > 0$, $m \leq 2^p$ and $n = \Omega(p \log^2 p)$. Both results hold for a wide range of BSP* parameters where the range becomes larger with growing input sizes m and n. We further report on implementation work in progress. Previous parallel algorithms for Multisearch were far away from being c-optimal in our model and do not consider blockwise communication.

1 Introduction

The theory of efficient parallel algorithms is very successful in developing new original algorithmic ideas and analytic techniques to design and analyse efficient parallel algorithms. For this purpose the PRAM has proven to be a very convenient computation model, because it abstracts from communication problems. On the other hand, the asymptotic results achieved only give limited information about the behaviour of the algorithms on real parallel machines. This is (mainly) due to the following reasons.

- The PRAM cost model (communication is as expensive as computation) is far away from reality, because communication is by far more expensive than internal computation on real parallel machines [6].
- The number of processors p is treated as an unlimited resource, (like time and space in sequential computation) whereas in real machines p is small (a parallel machine (MIMD) with 1000 processors is already a large machine).

There are several approaches to define more realistic computation models and cost measures to overcome the first objection mentioned above: The BSP model due to

* email: {abk,dittrich,fmadh}@uni-paderborn.de, Fax: +49-5251-603514. Supported in part by DFG-Sonderforschungsbereich 1511 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen", by DFG Leibniz Grant Me872/6-1, and by the Esprit Basic Research Action Nr 7141 (ALCOM II)

Valiant [14], the LogP model due to Culler et al. [6], the BPRAM of Aggarval et al. [1], and the CGM due to Dehne et al. [7] to name a few. Note that most of the models except the BPRAM neglect the negative effects of communicating small packets.

To deal with the second objection, Kruskal et al. [10] have proposed a complexity theory which considers speed-up. Valiant has proposed a very strong notion of work optimality of algorithms, *c-optimality*. It gives precise information about the possible speed-up on real machines, the speed-up of a c-optimal algorithm should be close to p/c. Besides [8] and [5] there are seemingly no systematic efforts undertaken to design parallel algorithms with respect to this strong optimality criterion.

The computation model used in this paper is the BSP enhanced by a feature that rewards blockwise communication. We design and analyse two algorithms for a basic problem in computational geometry, the Multisearch problem. Our first algorithm is deterministic and 1-optimal. It works for the case of many search queries compared to the number of segments. The second algorithm is designed for the case if only few search queries are asked. It is randomized and proven to be $(1+\delta)$-optimal with high probability, for $\delta > 0$ arbitary. Both results hold for wide ranges of BSP* parameters.

1.1 The Multisearch Problem

Multisearch is an important basic problem in computational geometry. It is the core of e.g. planar point location algorithms, segment trees and many other data structures.

Given an ordered *universe* U and a partition of U in *segments* $S = \{s_1, \ldots, s_m\}$. The segments are ordered in the sense that, for each $q \in U$ and segment s_i, it can be determined with unit cost whether $q \in s_i, q \in \{s_1 \cup \ldots \cup s_{i-1}\}$, or $q \in \{s_{i+1} \cup \ldots \cup s_m\}$. We assume that, initially, the segments and queries are evenly distributed among the processors. Each processor has a block of at most $\lceil m/p \rceil$ consecutive segments and arbitrary $\lceil n/p \rceil$ queries, as part of the input. The *Multisearch problem* is: Given a set of queries $Q = \{q_1, \ldots, q_n\} \subseteq U$ and a set of segments $S = \{s_1, \ldots, s_m\}$, find, for each q_i, the segment it belongs to (denoted $s(q_i)$). Sequentially this needs time $n \log m$ in the worst case.

An important example is: A strip in 2D is partitioned into segments, and queries are points in the strip, see Figure 1. The task is to determine for each query point which segment it belongs to. Note that sorting the points and merging them with the segments would not solve the problem, as our example shows. In case of $n < m$ we refer to *Multisearch with few queries*, otherwise to *Multisearch with many queries*.

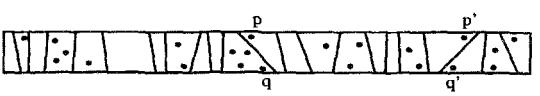

Fig. 1. Strip with segments and query points. Note, that p lies left to q (q' left to p') but $s(p)$ is right to $s(q')$ ($s(q')$ right to $s(p')$).

1.2 BSP, BSP* and c-Optimality

The BSP (Bulk-synchronous parallel) model [14] consists of:

- a number of processor/memory components,
- a router that can deliver messages point to point among the processors, and
- a facility to synchronize all processors in barrier style.

A computation on this model proceeds in a succession of *supersteps* separated by synchronisations. For clarity we distinguish between *communication* and *computation supersteps*. In computation supersteps processors perform local computations on data

that is available locally at the beginning of the superstep. In communication supersteps all the necessary exchange of data between the processors is done by the router. An instance of the BSP model is characterized by the parameters p, L and g.

- The parameter p is the number of processor/memory components.
- L is the minimum time between successive synchronisation operations. Thus L is the minimum time for a superstep.
- The parameter g is the time the router needs to deliver a packet when in continuous use.

Many routers of real parallel machines support the exchange of large messages and achieve much higher throughput for large messages compared to small packets. Thus good parallel algorithms should try to communicate only large packets, i.e. use *blockwise communication*. Communicating small messages is not significantly faster than communicating messages up to a certain size. To incorporate this crucial property in our model we extend the BSP model to the BSP* model by adding the parameter B, the minimum size the packets must have in order to fully exploit the bandwidth of the router.

For a computation superstep with at most t local operations on each processor we charge $\max\{L, t\}$ time units. For an h-relation, i.e. a routing request where each processor sends and receives at most h messages, we charge $\max\{g \cdot h \cdot \lceil \frac{s}{B} \rceil, L\}$ time units in a communication superstep, if the messages have maximum size s. Thus in the BSP* model it is worthwhile to send messages of size at least B.

Let A^* be the best sequential algorithm on the RAM for the problem under consideration, and let $T(A^*)$ be its worst case runtime. Let c be a constant with $c \geq 1$. In order to be *c-optimal* (with respect to p, L, g and B) a BSP* algorithm A has to fulfill the following requirements [8]:

- The ratio between the time spent for computation supersteps of A and $T(A^*)/p$ has to be in $c + o(1)$.
- The ratio between the time spent for the communication supersteps of A and $T(A^*)/p$ has to be in $o(1)$.

All asymptotic bounds refer to the problem size as $n \to \infty$.

1.3 Known Results

Sequentially Multisearch can be done in time $n \log m$ in the worst case. There are some parallel algorithms for this problem on a variety of parallel models, mainly for the case $n = m$. Multisearch can be solved optimally by a trivial algorithm for the CREW-PRAM. For the EREW-PRAM it is already a very complicated problem. Reif and Sen [13] developed an asymtotically optimal randomized EREW-PRAM algorithm, which works also on the butterfly network. It runs in time $O(\log n)$, with high probability, using n processors. It is easily seen that large constant factors are involved and that it performs badly on the BSP* model. Further it is not obvious how to generalize the algorithm work optimally for the case $n < m$. Ranade [12] has developed a multisearch algorithm for the p processor butterfly network for $n = p \log p$ queries and $m = O(p^c)$ segments. For the case $m \geq n$ this algorithm is asymptotically optimal but not for the case $m < n$. As in the case of the algorithm mentioned above this algorithm has large constant factors and does not consider blockwise communication. Atallah and Fabri [3] achieved (non-optimal, deterministic) time $O(\log n(\log \log n)^3)$ on a n-processor hypercube. A $O(\sqrt{n})$ time algorithm on a $\sqrt{n} \times \sqrt{n}$ mesh network is from Atallah et al. [2]. Dehne et al. [7] have developed some algorithms on the CGM for geometric problems including Mulitsearch. In contrast to the other results quoted

above Dehne et al. assume that the machine is much smaller than the problem size, i.e. $p \leq \sqrt{n}$. On their model Multisearch can be solved with p processors in time $O(\frac{n}{p} \log n)$ using a constant number of communication rounds, where constant factors and blockwise communication are not considered. Some 1-optimal algorithms for Sorting and Gauss-Jordan Elimination have been developed by Valiant et al. [8]. McColl [11] has developed some algorithms for matrix problems also on the BSP model.

1.4 New Results

We present and analyse two parallel algorithms for Multisearch. The first algorithm (*ManyQueries*) works for Multisearch with many queries, i.e. $m \leq n$. It is a deterministic BSP* algorithm that is 1-optimal, if $n = \Omega(p \log^2 p)$. The second algorithm (*FewQueries*) works for Multisearch with few queries, i.e. for $2^p \leq m > n$. It is a randomized BSP* algorithm that is $(1+\delta)$-optimal with probability $1 - p^{-(\log^\beta p)}$ for arbitrary $\delta > 0$, β small enough and $n = \Omega(p \log^2 p)$. These results hold for a wide range of BSP* parameters. E.g. $L \leq n^\eta, B \leq n^\xi$ and $g = o(B \log n)$ suffice for ξ and η small enough. Note that p, g and B may grow with the problem size. Therefore we can expect that our algorithms are fast even on machines with relatively slow routers that need large packets. Our algorithms use routines for broadcast, parallel prefix and variants of load balancing as basic routines. BSP* algorithms for these problems are part of our work. Due to page limitations, this extended abstract only sketches most of the algorithms and proofs. A full version of the paper appears as technical report [4].

1.5 Experiments

We have performed preliminary measurements of the Algorithm ManyQueries for the case $n \geq m$ on the GCel from Parsytec. The GCel is a network of T800 transputers as processors, a 2-dimensional mesh as router and Parix as its operation system. In order to implement our algorithm in BSP* style we have realized a library on top of Parix containing the basic routines mentioned above.

First expermiments show, for $n = 1572864$ and $m = 524288$, a speed-up of 49 with 128 processors, where 12288 points and 4096 segments are stored in each processor. Further experiments are in progress, especially with more processors and for the randomized algorithm.

1.6 Organisation of the Paper

In Section 2 we give an outline of the algorithm and introduce some notations, in Section 3 we describe some basic routines which are used by the two Multisearch algorithms presented in Sections 4 and 5.

2 Outline of the Algorithm and Notations

In a preprocessing phase a suitable balanced search tree will be constructed from the input segments. In order to guarantee few communication supersteps and to achieve blockwise communication we choose the search tree to be of high degree (large nodes) and therefore low height.

In the course of the algorithm the queries travel through the search tree along their search paths from the root to their leaves level by level. The algorithm proceeds in rounds, each consisting of a small number of supersteps. The number of rounds corresponds to the height of the search tree. In round i it will be determined which query visits which node on level $i+1$ of the search tree. In order to obtain an efficient BSP* implementation we have to cope with the following problems.

The first problems concern the initial distribution of the nodes of the search tree among the processors. If $m < n$ an one-to-one-mapping of tree nodes to processors works fine, but if there are more nodes than processors, as in the case of $m > n$ (large search trees), a deterministic distribution causes contention problems: One can always find an input such that only nodes mapped to the same processor are accessed. In order to make contention unlikely we distribute the nodes of the search tree randomly.

This leads to another problem. Random distribution destroys locality and therefore makes it more difficult to communicate in a blockwise fashion. In order to cope with this we use a partially random distribution that preserves some locality properties.

During the travel of the queries through the tree the following problems arise. For each node v and each query q visiting v, the next node to be visited has to be determined. It may occur that some nodes are visited by many queries and that other nodes may only be visited by few queries. Thus a careful load balancing has to be done.

As it might happen that a processor holds many queries visiting different nodes, it can be too expensive to send all the appropriate nodes to that processor. In that case we send the queries to the processors that hold the appropriate nodes.

In order to present the algorithm we need some notation for describing the distribution of the queries among the processors:

For a node v of the search tree we define the *job* at node v to be the set of queries visiting v. *Executing this job* means determining, for each query of the job, which node of the search tree it has to visit next. Let w be a child node of v and let J be the job at node v, then the job at node w is called a *successor job* of J. If J is a job at a node v on level i of the search tree then J is called a *job on level i*.

Let $J_1, \ldots, J_k$ be some jobs on the same level of the search tree and $P_i, \ldots, P_{i+l}$ some consecutive processors. Let the queries from $J_1 \cup \ldots \cup J_k$ be numbered $1, \ldots, n'$, $n' \leq n$, with the first $|J_1|$ jobs numbered $1, \ldots, |J_1|$, and so on. A distribution of the queries of $J_1, \ldots, J_k$ among the processors $P_i, \ldots, P_{i+l}$ is *balanced*, if no P_j gets more than $\frac{n}{p}$ of the queries. It is *ordered*, if the queries in P_j are lower numbered than those in P_{j+1}. The processors holding queries from the same job form a *group*. The first processor of a group is the *group leader*. If the group of a job consists of one processor the job is called *exclusive*, if not it is *shared*. For an example see Figure 2.

Let α, α' be constants with $0 < \alpha' < \alpha < 1$. Further conditions on α, α' can be derived from the analysis of the algorithms. We define *small jobs* to be of size smaller than $(n/p)^{\alpha}$ and *large jobs* to be of size at least $(n/p)^{\alpha}$. The nodes of the search tree will have degree $(n/p)^{\alpha'}$, thus a job can have up to $(n/p)^{\alpha'}$ successor jobs.

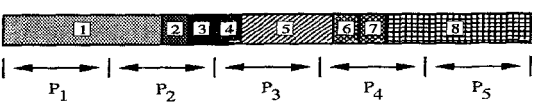

Fig. 2. Jobs $1, \ldots, 8$ are distributed in an ordered and balanced way among processors $P_1, \ldots, P_5$. 1, 5 and 8 are shared jobs, the others are exclusive ones. Job 1, $(5, 8)$ is held by the processor group $\{P_1, P_2\}$ $(\{P_3, P_4\}, \{P_4, P_5\})$.

3 Communication Routines

In this section we describe and analyse BSP* algorithms for several communication routines: Broadcast, Parallel-Prefix, Distribute, DistributeFQ and Load-Balance. All these are auxiliary routines for the two Multisearch algorithms described in the next

two sections. There, these routines will be executed by groups of consecutive processors. These groups can be of any size between two and p processors. We present the communication routines as they are needed for the execution on the group of all p processors $P_1, \ldots, P_p$. Thus the analysis will give an upper bound for the complexity of the routines. The algorithms and their analysis are only briefly sketched in this extended abstract.

3.1 Broadcast

Consider a vector of size d stored in processor P_1. The task is to send this vector to all the other processors.

Algorithm Broadcast: The processors are organized as a binary tree with root P_1. The other processors are called internal processors. P_1 splits the vector in $\lceil d/b \rceil$ packets, each of size at most b. The algorithm proceeds in rounds, each consisting of a communication and a computation superstep. In the i-th round P_1 makes two copies of the i-th packet and sends them to its children processors. Each internal processor that got a packet in the previous round makes two copies of it and sends them two its children. Thus the packets travel through the tree in a pipelined fashion. The algorithm performs $\lfloor \log p \rfloor + \lceil d/b \rceil$ rounds. Each round takes time $\max\{g\lceil b/B \rceil, L\}$ for the communication superstep and time $\max\{\lceil b/B \rceil, L\}$ for the computation supersteps.

Result 1 *Broadcast requires communication time* $O((\log p + \lceil d/b \rceil) \cdot \max\{g\lceil b/B \rceil, L\})$ *and computation time* $O((\log p + \lceil d/b \rceil) \cdot \max\{\lceil b/B \rceil, L\})$. *In particular, if* $d \geq B \cdot L \log p$ *set* $b = B \cdot L$ *thus communication time* $O(g\frac{d}{B})$ *and computation time* $O(\frac{d}{B})$ *is needed.*

3.2 Parallel-Prefix

Let p vectors of size d are given where the i-th vector is stored in the i-th processor. The task of the algorithm *Parallel-Prefix* is to compute, for each $k \in \{1, \ldots, d\}$, the prefix sums of the k-th components of the vectors. The resulting prefix sums are stored in the corresponding processors. As in the broadcast algorithm the processors are organized as a balanced binary tree. We employ the standard parallel prefix algorithm that proceeds in two phases. In the first phase the sums move from the leaves of the tree, in the second from the root to the leaves making some calculations at each level. The vectors are split into $\lceil d/b \rceil$ packets of size at most b in order to proceed in a pipelined fashion as in the algorithm Broadcast.

Result 2 *Parallel-Prefix needs communication time* $O((\log p + \lceil d/b \rceil) \cdot \max\{g\lceil b/B \rceil, L\})$ *and computation time* $O((\log p + \lceil d/b \rceil) \cdot \max\{b, L\})$. *In particular, if* $d \geq B \cdot L \log p$ *set* $b = B \cdot L$ *thus communication time* $O(g\frac{d}{B})$ *and computation time* $O(d)$ *is needed.*

3.3 Distribute and DistributeFQ

Input: A set of at most p large jobs distributed among the p processors in a balanced and ordered way (refer to Section 2 for the notations).

Output: Input jobs of size at most $\frac{n}{p}$ (we have at most p of them) are mapped to different processors. Let r_i be the size of the job which is mapped to processor P_i. Then there is enough space for $\frac{n}{p} - r_i$ additional queries on P_i. We call this value the *gap* of P_i. Input jobs of size larger than $\frac{n}{p}$ are distributed among the processors such that they fill up the gaps. See Figure 3 for an example. Note that afterwards each processor stores queries of at most three different input jobs.

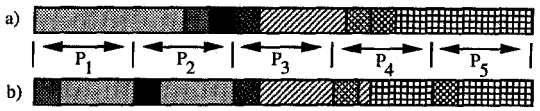

Fig. 3. a) shows the situation before and b) after Distribute. Before P_4 holds queries of four jobs. Afterwards P_4 holds queries of 3 jobs.

For multisearch with few queries we need a slightly different version of the algorithm Distribute. We call it *DistributeFQ*.

Input: Small jobs $J_1, \ldots, J_k$ and large jobs $J'_1, \ldots, J'_{k'}$. Each of the small jobs is stored in a processor such that no processor holds more than $(1+\delta)\frac{n}{p}$ queries of these jobs (δ will be specified later). The large jobs are stored in an ordered way among the processors.

Output: Let r_i be the amount of queries of the small jobs stored in processor P_i. We call the value $(1+\delta)\frac{n}{p} - r_i$ the gap of P_i. The queries of $J'_1, \ldots, J'_{k'}$ are redistributed such that they fill up the gaps of the processors as in the algorithm Distribute. The small jobs are not moved.

Result 3 *Algorithms Distribute and DistributeFQ need time $O(\frac{n}{p})$ for computation, time $O(g(\frac{n}{pB} + (\frac{n}{p})^{1-\alpha}) + \frac{n}{p})$ for communication, if $\frac{n}{p} \geq \log p \max\{L, g\}$. Further, they need space $O(\frac{n}{p})$ and $O((1+\delta)\frac{n}{p})$, respectively.*

3.4 Load-Balance

Let J be a large job on a certain level of the search tree and let $J_1, \ldots, J_t$ be the successor jobs of J.

Input: The queries of J are distributed among the processors such that each processor holds at most $\frac{n}{p}$ queries of J. P_1 holds a vector A with the sizes of all successor jobs of J. Further each query is labelled with the number of the successor job it belongs to. A query that belongs to the k-th successor job of J gets the label k, with $1 \leq k \leq (\frac{n}{p})^{\alpha'}$.

Output: The jobs $J_1, \ldots, J_t$ distributed among the processors in an ordered and balanced way. See Figure 4 for an example.

The redistribution specified above can easily be done. If P_1 broadcasts the vector A to the other processors, they have the necessary information in order to calculate the appropriate target processor for each query of J. But if the queries are directly sent to their target processor a problem arises: Many processors could hold only few queries for a certain target procesor, especially less than the block size B. So $\Theta(\frac{n}{p})$ small packets may be sent to a processor. This would require communication time $O(g\frac{n}{p})$ which is too large (if $g = o(B \log n)$). In order to cope with this situation we have to combine these queries to larger packets before we can finally send them to their target processors.

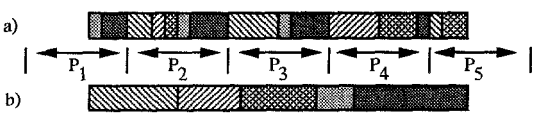

Fig. 4. a) shows the situation before and b) after Load-Balance for a group.

Result 4 *Let $0 < \alpha' < 1$, $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$, and $B \leq (\frac{n}{p})^{(1-\alpha')/2}$. Then algorithm Load-Balance needs computation time $O(\frac{n}{p})$, communication time $O(g(B(\frac{n}{p})^{\alpha'} + \frac{n}{Bp}))$, and space $O(\frac{n}{p})$.*

4 Multisearch with Many Queries

In this section we show how to do Multisearch for the case $n \geq m$ such that the internal work is almost exactly the same as for the sequential algorithm and the ratio of communication time to computation time is in $o(1)$.

In order to simplify the presentation we consider only the case $m = n$. The case $n > m$ can easily be concluded. The main idea of the search procedure is the following. Construct a balanced search tree over S and let the queries "flow" through the search tree level by level from the root to the leaves. The main problem arises from the fact that many queries can visit a node of the search tree and many nodes are visited per level.

Preprocessing: At most $\lceil n/p \rceil$ consecutive segments lie on each processor as part of the input, they form an *interval*. For each processor P_i we denote the largest segment held by P_i as a *separating segment*. Now we build a balanced binary search tree T

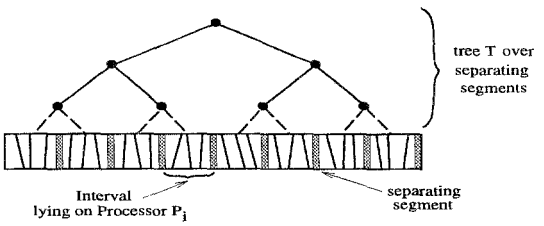

Fig. 5. Tree T above the set of separating segments.

where the leaves are formed by the intervals and the internal nodes are formed by the separating segments, compare Figure 5. We choose the nodes of T to be of degree 2^μ where μ is the largest integer such that $2^\mu - 1 < \lceil (\frac{n}{p})^{\alpha'} \rceil$.

Fact 1 *Let $0 < \alpha' < 1$ be arbitrary, and $B \leq (\frac{n}{p})^{(1-\alpha')/2}$. Then the preprocessing needs computation time $O(\max\{\frac{n}{p}, L\} \log p)$ and communication time $O(\max\{g(\frac{n}{p})^{\alpha'}, L\} \log p)$. The constructed tree T has at most p nodes each of degree $2^\mu \leq \lceil (\frac{n}{p})^{\alpha'} \rceil$. T has $O(\frac{\log p}{\alpha' \log(n/p)})$ levels and each processor stores the segments of at most one interval and one node of T. The data structure needs $O(\frac{n}{p} + (\frac{n}{p})^{\alpha'})$ storage per processor. The root of T is stored on processor P_1.*

Executing Queries: The algorithm has three phases. The first phase is subdivided into rounds. In the i-th round level i of T is considered. During each round only large jobs are executed. Queries of small jobs are directly sent to the processors which hold the appropriate nodes. They are not considered further until Phase 3. The second phase handles jobs at leaf nodes of T after Phase 1. For each such job the correct interval is already computed. Finally, in the last phase the small jobs which have been put aside in the first phase are processed. These can only be small jobs. No two of these need to inspect the same interval, therefore they can be broadcasted to the processors, which store the intervals they have to inspect (in order to find the correct segment) and the processors can compute the correct segments independently.

Algorithm ManyQueries:
Description of Phase 1: Phase 1 proceeds in rounds. In round i the jobs on level i of T will be executed.

Input for round i: Large jobs on level i of the search tree T, distributed in a balanced and ordered way among the processors (For notations compare Section 2).
Output of round i: Large jobs on level $i+1$ of T, distributed in a balanced and ordered way among the processors. Small jobs are directly sent to the processors which hold the appropriate nodes.

Thus we have as input for round i exclusive jobs and we have shared jobs with a group of consecutive processors allocated to each of them. Note that, since we are restricted to large jobs, there are at most $(\frac{n}{p})^{1-\alpha}$ jobs mapped exclusively to one processor. This is crucial for the analysis. We now describe how the algorithm executes the input jobs of round i:

Each processor P executes the exclusive jobs which are mapped to it as follows: For each of its exclusive jobs it fetches the nodes from the processors that store them. These can only be $(\frac{n}{p})^{1-\alpha}$ many nodes, since there cannot be more exclusive jobs mapped to one processor. Thus, this step is not too expensive. After that each processor determines the successor jobs by means of binary search and stores the large successor jobs ordered in its memory. Successor jobs of smaller size are directly sent to the processors that hold the appropriate node for that job.

A group of processors $P_i, \ldots, P_j$ executes a shared job J as follows: The group leader fetches the appropriate node for that job and broadcasts it to the other group members. Each processor of that group now locally determines to which successor job its queries belong by means of binary search and labels them accordingly. After that the size of each successor job of the shared job J is determined by a parallel prefix computation. With this information the algorithm *Load-Balance* is called which redistributes the queries of J such that afterwards the successor jobs of J are distributed in a balanced and ordered way among the processors $P_i, \ldots P_j$. After that small successor jobs are directly sent to the processors that hold the appropriate node for that job.

We describe the algorithm as if a processor was only involved in either the execution of one shared job or the execution of some exclusive jobs. In fact it can be involved in the execution of up to two shared jobs and up to $(\frac{n}{p})^{1-\alpha}$ exclusive ones (see Section 2). It is not difficult to schedule the instructions such that the performance is not affected. The algorithm Load-Balance ensures that each processor has to perform local binary search on at most $\frac{n}{p}$ queries in each round. Here comes the algorithm in detail. The input for the first round is the job at the root node of T consisting of all queries.

1. For each large job (exclusive or shared) the corresponding node of T has to be fetched. Therefore each group leader fetches the node of T and broadcasts it to the processors of its group. Additionally each processor holding exclusive jobs fetches for each exclusive job the corresponding node of T.
2. Each processor determines by means of binary search (on the fetched nodes) for each of its queries which node to visit next. The queries visiting the same node in the next level belong to the same successor job and are marked with the same label.
3. The processors of each group compute the size of the new successor jobs by means of parallel prefix on vectors, where the i-th component of the vector corresponds to the number of queries which visit the i-th child node of the current fetched node of T. The group leader knows the size of each of there successor jobs afterwards.
4. The successor jobs of every shared job are redistributed in a balanced and ordered way among the processors of the group by executing procedure Load-Balance. The processors holding exclusive jobs compute the reorganisation sequentially (by a bucket sort approach).

5. Each processor sends the queries of each small successor job it holds to the processor which holds the node of T that successor job wants to visit next. If a group has reached a leaf of T it leaves this phase. Goto Step 1.

Description of Phase 2: In this phase the large jobs which have reached the leaves of T after Phase 1 are processed. Remember T has p leaves, therefore the queries are partitioned into at most p jobs. Unfortunately a processor can hold queries of up to $(\frac{n}{p})^{1-\alpha}$ large jobs, since a large job can be as small as $(\frac{n}{p})^{\alpha}$.

1. The processors redistribute the queries of the jobs by means of Distribute such that each processor holds queries of at most three jobs.
2. Each processor which has received an exclusive job and each group leader of a shared job fetches the corresponding interval. Additionally each group leader initiates a broadcast of the interval to the processors of its group.
3. Eventually each processor performs binary search for each query on the appropriate interval. Note that a processor has to store at most three intervals.

Description of Phase 3: In this phase the small jobs are considered. Remember that small jobs are sent (during Phase 1) to the processors that hold the nodes of T they have to visit next.

1. Each processor which has received a small job broadcasts it to the processors storing the intervals reachable from the corresponding node of T.
2. Each processor determines for the received queries the correct segment by means of binary search on the segments of its interval.

Theorem 1. *Let $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$. For $0 < \alpha' < 1$ there is a constant $\eta > 0$ such that algorithm ManyQueries needs time $(1+o(1))\frac{n}{p}\log n + O(\frac{\log p}{\alpha' \log(n/p)}\frac{n}{p})$ for computation, $O(g\frac{\log p}{\alpha' \log(n/p)}\frac{n}{pB} + g\cdot\frac{n}{pB})$ for communication and $O(\frac{n}{p})$ space per processor, if $B = (\frac{n}{p})^{\eta}$. Thus algorithm ManyQueries is 1-optimal*

- *for $n \geq p^{1+\epsilon}$, $\epsilon > 0$ arbitrary, if $g = o(B \log n)$, and*
- *for $n \geq p \log^2 p$, if $g = o(B \log\log n)$.*

Proof of Theorem 1:
Analysis of Phase 1: Let h be the number of levels of T, i.e. $h = O(\log p/(\alpha' \log(n/p)))$. The Steps 1 to 6 are repeated at most h times.

Step 1: Each processor holds queries of at most $O((\frac{n}{p})^{1-\alpha})$ large jobs. Therefore each processor has to fetch $O((\frac{n}{p})^{1-\alpha})$ nodes. This can be realized in two communication supersteps. In the first one the requests for nodes will be sent to the processors that hold them. Thus an $O((\frac{n}{p})^{1-\alpha})$-relation with packets of size 1 is realized. In the second superstep the nodes will be sent to the requesting processors. Thus an $O((\frac{n}{p})^{1-\alpha})$-relation with packets of size $O((\frac{n}{p})^{\alpha'})$ is routed. Since $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$ these two supersteps need time $O(g \cdot (\frac{n}{p})^{1-\alpha} \cdot (\frac{n}{p})^{\alpha'}/B)$. The broadcast needs time $O(g(\frac{n}{p})^{\alpha'}/B)$ by Result 1.

Step 2: Binary search is performed during h rounds. For each query at most $\log p + 1$ comparisons are made, $\lceil\frac{n}{p}\rceil$ queries are handled by each processor therefore at most $\lceil\frac{n}{p}\rceil(\log p + 1)$ comparisons are made during Phase 1.

Step 3: By Result 2, if $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$, it needs communication time $O(g(\frac{n}{p})^{\alpha'}/B)$ and computation time $O((\frac{n}{p})^{\alpha'})$ for the parallel prefix on vectors of size $O((\frac{n}{p})^{\alpha'})$.

Step 4: By Result 4, if $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$ and $B \leq (\frac{n}{p})^{(1-\alpha')/2}$, Load-Balance needs communication time $O(g(B(\frac{n}{p})^{\alpha'} + \frac{n}{Bp}))$ and computation time $O(\frac{n}{p})$. For exclusive jobs the reorganisation is done sequentially in time $O(s + (\frac{n}{p})^{\alpha'})$, where $s \leq (n/p)$ is the total size of the exclusive jobs on a processor.

Step 5: The queries of small successor jobs are directly sent to the nodes they have to visit next. A job can have at most $\lceil(\frac{n}{p})^{\alpha'}\rceil$ different small successor jobs. A processor has at most $(\frac{n}{p})^{1-\alpha}$ exclusive jobs therefore a processor has to send $O((\frac{n}{p})^{1-\alpha+\alpha'} + \frac{n}{pB})$ packets. A processor receives small jobs of size $O((\frac{n}{p})^{\alpha})$, therefore it has to receive $O((\frac{n}{p})^{\alpha}/B)$ packets. Thus Step 5 can be done in one communication superstep that needs time $(g((\frac{n}{p})^{1-\alpha+\alpha'} + \frac{n}{pB}))$.

Thus Phase 1 needs computation time $\leq \frac{n}{p}\log(p) + O(\frac{\log p}{\alpha' \log(n/p)} \cdot \frac{n}{p} + \frac{n}{p})$ and communication time $O(g \cdot \frac{\log p}{\alpha' \log(n/p)} \cdot (B(\frac{n}{p})^{\alpha'} + \frac{n}{Bp} + (\frac{n}{p})^{1-\alpha+\alpha'}))$.

Analysis of Phase 2: Step 1: By Result 3 it requires time $O(g(\frac{n}{pB} + (\frac{n}{p})^{1-\alpha}) + \frac{n}{p})$ for communication and $O(\frac{n}{p})$ for computation, if $\frac{n}{p} \geq \log p \max\{L, g\}$. *Step 2:* By Result 1 it requires communication time $O(g\frac{n}{Bp})$ and $O(\frac{n}{p})$ computation time, if $\frac{n}{p} \geq B \cdot L \log p$. *Step 3:* It requires computation time $\lceil\frac{n}{p}\rceil \log\lceil\frac{n}{p}\rceil \leq \frac{n}{p}\log n - \frac{n}{p}\log p + O(\frac{n}{p})$.

Analysis of Phase 3: By Result 1 Step 1 needs communication time $O(g(\frac{n}{p})^{\alpha}/B)$ and computation time $O(\frac{n}{p})$, if $(\frac{n}{p})^{\alpha} \geq B \cdot L \log p$. Step 2 needs computation time $\lceil(\frac{n}{p})^{\alpha}\rceil \log\lceil\frac{n}{p}\rceil = O(\frac{n}{p})$

Adding up the communication and computation times for Phases 1, 2, 3 yields the time bounds claimed in Theorem 1. Note that from the analysis we get conditions on the constant α. The value of η in Theorem 1 depends on α and α'. □

5 Multisearch with Few Queries

In the sequel only few queries are asked: we allow now $n < m \leq 2^p$, rather than $n \geq m$. Recall that the sequential time needed is $n \log m$ in the worst case. It is easily seen that algorithm ManyQueries is far from optimal in this case, e.g. if all queries happen to belong to segments stored in the same processor. Therefore we need a slightly different search tree and a new way to distribute its nodes among the processors.

Preprocessing: In contrast to ManyQueries we now organize the m segments in a search tree T^* of size $m/(\frac{n}{p})^{\alpha'}$ with degree $(\frac{n}{p})^{\alpha'}$ and depth $\log m/\alpha' \log(\frac{n}{p})$, with $\frac{1}{2} < \alpha' < 1$ (see Section 2). The segments within each node are organized as a binary search tree. They will always be processed sequentially.

As m can be very large the number of nodes of T^* can become much larger than p. Thus, for every distribution of its nodes among the processors, there is a bad set of queries which causes high contention. The reason is that many of the queries have to travel through different nodes mapped to the same processor. A standard technique to reduce contention is randomization: If the nodes are randomly distributed among the processors, the above event becomes very unlikely for arbitrary sets of queries. For our purposes, however, a naive random distribution is not suitable, because the neighbours of the nodes stored in a processor are likely to be distributed almost injectively. Thus the communication to be executed when the queries travel through the tree is fine grained, blockwise communication is not possible. Therefore our approach uses a randomized distribution which ensures that children of nodes that are mapped to the same processor are distributed among few processors only. This is done by the following algorithm. It works for any tree T with degree d and depth h.

Algorithm Build-Up:

$i = 0$: The root will be placed on an arbitrary processor.

$i > 0$: If level i of T has at most p nodes, they are mapped to the processors such that each processor gets at most one node. If level i has more than p nodes, they are distributed as follows: Let R be the subset of nodes on level $i-1$ of T that have been placed on processor P. P chooses a random set of $d^{1+\epsilon}$ processors (neighbour processors) and distributes the children of nodes of R randomly among these neighbour processors. For our algorithm ϵ has to be chosen such that $\alpha' \cdot (1+\epsilon) < 1$.

The following is easy to check:

Fact 2 *The algorithm Build-Up needs time $O(Lh + g(\frac{m}{p}))$ and $O(\frac{m}{p})$ space per processor.*

The next lemma captures properties of the distribution produced by algorithm Build-Up that are crucial for the use of blockwise communication.

Lemma 2. *a) Consider a tree T with degree d and depth h distributed among p processors by algorithm Build-Up. Then for every $\delta > 0$ and ζ with $0 < \zeta < 1$, there exists an $\eta > 0$ such that the following holds: Fix l, $l \leq M$, nodes $v_1, \ldots, v_l$ of a level of T and give them non-negative weights $g_1, \ldots g_l$ each at most $(\frac{M}{p})^{\zeta}$, such that the total weight of $v_1, \ldots, v_l$ is at most M. The probability that the total weight of nodes placed on the same processor exceeds $(1+\delta)\frac{M}{p}$ is less than $4ph/2^{d^{\eta}}$.*

b) For each processor P_i, the parents of the nodes stored in P_i are distributed among $O(d^{1+\epsilon})$ processors, with probability at least $1 - 2^{-d}$.

The proof can be found in the full version of this paper [4].

Executing Queries:

The algorithm FewQueries proceeds similar to the first phase of algorithm ManyQueries with the following exceptions:

- It works on the data structure generated by algorithm Build-Up that represents T^* rather than on the data structure used by algorithm ManyQueries.
- Many small jobs may be generated in this setting. Fetching nodes for each of them is too expensive. Therefore we send these jobs to the processors that hold the appropriate nodes. The treatment of small jobs is not postponed to a later phase as in algorithm ManyQueries. Small jobs are handled together with large ones.

In the i-th round of algorithm FewQueries we have the following input and output:

Input for round i: The jobs on level i are distributed among the processors as follows: Each small job on level i is placed on the processor that holds the appropriate node for that job. Let r_i be the number of queries of these small jobs that are stored on processor P_i. Lemma 2 guarantees that r_i is not larger than $(1+\delta)\frac{n}{p}$. The value $\frac{n}{p} - r_i$ is called the gap of P_i. The large jobs on level i are distributed in an ordered way among the processors such that they fill up the gaps.

Output for round i: Each small job on level $i+1$ is placed on the processor that holds the appropriate node for that job. Large jobs on level $i+1$ are distributed in an ordered way among the processors such that they fill up the gaps.

In each round i the small jobs of level i are sent to processors holding the appropriate nodes for these jobs. The new data structure guarantees that a processor has to send the small jobs only to $(\frac{n}{p})^{\alpha'(1+\epsilon)}$ neighbour processors. Remember that ϵ is chosen such that $\alpha'(1+\epsilon) < 1$. Thus jobs can be combined in order to form large messages. This is crucial in order to achieve blockwise communication.

As with the algorithm ManyQueries in each round exclusive and shared input jobs have to be executed. Here we may have much more exclusive jobs mapped to one processor, but the nodes for small exclusive jobs are already stored on the same processor. They need not to be fetched from other processors.

Algorithm FewQueries:
If h is the depth of T^* then the algorithm makes h iterations. In each iteration it executes the following 6 steps:

1. For each shared job the group leader fetches the appropriate node of T^*. Each processor fetches for each of its large exclusive jobs the appropriate node of T^* (Small jobs are already placed together with their appropriate nodes on a processor). Each group leader broadcasts the fetched node to the other group members.
2. Each processor P_i determines by means of binary search for each of its queries which node to visit next. The queries visiting the same node in the next level belong to the same successor job and are marked with the same label. It is guaranteed by Step 6 that P_i has to perform binary search for at most $(1+\delta)\frac{n}{p}$ queries.

3-5. These steps are the same as Step 3-5 in Phase 1 of algorithm ManyQueries.

6. The processors execute algorithm *DistributeFQ*. This guarantees that the queries of large jobs of level $i+1$ are distributed ordered among the processors such that they fill up the gaps that have been left by the small jobs of level $i+1$.

Theorem 3. *Let $m \leq 2^p$, $\frac{n}{p} \geq \log^2 p$ and $(\frac{n}{p})^{\alpha'} \geq B \cdot L \log p$. For $\frac{1}{2} < \alpha' < 1$ and $\delta > 0$ there are constants $\beta, \eta > 0$, such that algorithm FewQueries needs time $(1+\delta+o(1))\frac{n}{p}\log m + O(\frac{\log m}{\alpha' \log(n/p)}\frac{n}{p})$ for computation, $O(g\frac{\log m}{\alpha' \log(n/p)}(\frac{n}{Bp}))$ for communication and $O(\frac{n}{p}+\frac{m}{p})$ space per processor with probability at least $1-p^{-(\log^\beta p)}$, if $B = (\frac{n}{p})^\eta$. Thus algorithm FewQueries is $(1+\delta)$-optimal*

- *for $m \leq 2^p$, $n = p^{1+\zeta}$ with $\zeta > 0$, if $g = o(B \log n)$ and*
- *for $m \leq 2^p$ and $n = p\log^2 p$, if $g = o(B \log\log n)$.*

Proof of Theorem 3:
In the following we analyse the time bounds for each round i:

Step 1: Each processor fetches at most $(\frac{n}{p})^{1-\alpha}$ nodes. Mark each of these nodes with weight $(\frac{n}{p})^\alpha$. Lemma 2a) guarantees that Build-Up distributes these nodes such that the total weight of nodes placed on each processor is at most $(1+\delta)\frac{n}{p}$ with probability $1-p^{-(\log p)^\beta}$ for a certain $\beta > 0$. Thus each processor gets $O((\frac{n}{p})^{1-\alpha})$ requests for nodes of T^*. Therefore the nodes can be fetched in two communication supersteps. In the first one requests will be sent to the processors. In the second superstep the nodes are sent to the requesting processors. The first superstep realizes an $(\frac{n}{p})^{1-\alpha}$-relation of packets of size 1, the second realizes a $(\frac{n}{p})^{1-\alpha}$-relation of packets of size $(\frac{n}{p})^\alpha$. Thus this step has the same time bounds as Step 1 of Phase 1 of algorithm ManyQueries.

Step 2: At the start of each round each processor holds at most $(1+\delta)(\frac{n}{p})$ queries. This is guaranteed by Lemma 2a) and the execution of the routine DistributeFQ at the end of each round. Therefore in Step 2 of each round every processor performs at most $(1+\delta)\frac{n}{p}\alpha' \log(\frac{n}{p})$ comparisons.

Step 3-4: The analysis for these steps is the same as for Steps 3-4 of Phase 1 of algorithm ManyQueries.

Step 5: Each processor sends small jobs with total weight at most $(1+\delta)(\frac{n}{p})$ to at most $(\frac{n}{p})^{\alpha'(1+\epsilon)}$ neighbour processors which hold the appropriate nodes. Mark these nodes with the size of the respective jobs. By Lemma 2a) and b) we know that each

processor receives jobs of total weight at most $(1+\delta)(\frac{n}{p})$ from $O(d^{1+\epsilon})$ neighbour processors with probability $1-p^{-(\log p)^{\beta}}$ for an appropriate $\beta > 0$. Note that in order to achieve this probability we need $\alpha' > \frac{1}{2}$. Each processor needs computation time $O(\frac{n}{p})$ to combine the queries to large packets. Each processor sends and receives $O(\frac{n}{pB} + (\frac{n}{p})^{\alpha'(1+\epsilon)})$ packets of size B in one communication superstep. Thus the communication time is $O((\frac{n}{p \cdot B})g)$ for Step 5, if $B \leq (\frac{n}{p})^{1-\alpha'(1+\epsilon)}$.

Step 6: The complexity is the same as for the algorithm DistributeFQ (see Section 3).

Since the depth of T^* and therefore the number of iterations is $\frac{\log m}{\alpha \log(n/p)}$, we reach the resource bounds stated by Theorem 3. Note that from the analysis the conditions on α can be derived. The values of β and η in Theorem 3 depend on the values of α and α'. □

References

1. A. Aggarwal, A.K. Chandra and M. Snir, On communication latency in PRAM computations, Proc. ACM Symp. on Parallel Algorithms and Architectures, 1989, 11-21.
2. M.J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin and J.-J. Tsay, Multisearch Techniques for Implementing Data Structures on a Mesh-Connected Computer, Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991, 204–214.
3. M.J. Atallah and A. Fabri, On the Multisearching Problem for Hypercubes, Parallel Architectures and Languages Europe, 1994.
4. A. Bäumker, W. Dittrich, F. Meyer auf der Heide, Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model, Technical Report, Universität-Gesamthochschule Paderborn, Department of Mathematics and Computer Science, to appear.
5. R. H. Bisseling, W. F. McColl, Scientific computing on bulk synchronous parallel architectures, Proc. 13th IFIP World Computer Congress, Volume 1, 1994.
6. D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993.
7. F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable Parallel Computational Geometry for Coarse Grained Multicomputers, Proc. ACM Conf. on Comp. Geometry, 1993.
8. A.V. Gerbessiotis and L. Valiant, Direct Bulk-Synchronous Parallel Algorithms, Journal of Parallel and Distributed Computing, 1994.
9. W. Hoeffding, Probability inequalities for sums of bounded random variables, American Statistical Association Journal, 1963, 13–30.
10. C.P. Kruskal, L. Rudolph and M. Snir, A complexity theory of efficient parallel algorithms, Proc. 15th Int. Coll. on Automata, Languages, and Programming, 1988, 333–346.
11. W F McColl, The BSP Approach to Architecture Independent Parallel Programming, To appear in CACM on General Purpose Practical Models of Parallel Computation, 1995.
12. Abhiram Ranade, Maintaining dynamic ordered sets on processor networks, Proc. of the 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, 127–137.
13. J.H. Reif and S. Sen, Randomized Algorithms for Binary Search and Load Balancing on Fixed Connection Networks with Geometric Applications, SIAM J. Comput., Vol. 23, No. 3, June 1994, 633–651.
14. L. Valiant, A Bridging Model for parallel Computation, Communications of the ACM, August 1994, Vol. 33, No. 8.