ELSEVIER

Theoretical Computer Science 203 (1998) 175–203

Theoretical Computer Science

Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model[1]

Armin Bäumker, Wolfgang Dittrich, Friedhelm Meyer auf der Heide *

Department of Mathematics and Computer Science and Heinz Nixdorf Institute, University of Paderborn, 33102 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 use Valiant's notion of 1-optimality. Intuitively, a 1-optimal parallel algorithm for p processors achieves speed-up close to p. 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 $m \leqslant n \geqslant p$ we present a deterministic BSP* algorithm that is 1-optimal, if $n/p \geqslant \log^2 n$. For $m > n \geqslant p$, we present a randomized BSP* algorithm that is 1-optimal with high probability, if $m \leqslant 2^p$ and $n/p \geqslant \log^3 n$. Both results hold for a wide range of BSP* parameters where the range becomes larger with growing input size n. We further report on implementation work. Previous parallel algorithms for Multisearch were far away from being 1-optimal in our model and did not consider blockwise communication.

Keywords: Parallel algorithms; Bulk synchronous computing; Data structures; Multisearch

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:

* Corresponding author. E-mail: fmadh@uni-paderborn.de.

[1] Work supported in part by DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen", by DFG Leibniz Grant Me872/6-1, and by the EC Esprit Long Term Research Project 20244 (ALCOM-IT).

0304-3975/98/$19.00
PII S0304-3975(98)00020-6

- 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 [5].
- 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).

In order to overcome the first objection mentioned above, several proposals for more realistic computation models have been made that explicitly charge for communication costs: The Bulk Synchronous Parallel (BSP) model due to Valiant [17], the LogP model due to Culler et al. [5], the BPRAM of Aggarwal et al. [1], and the CGM due to Dehne et al. [6] to name a few. In order to deal with the second objection, Kruskal et al. [10] have proposed a complexity theory which considers speed-up. Further, Valiant has proposed a very strong notion of work optimality of parallel algorithms, 1-*optimality*. It gives precise information about the speed-up on computation models that explicitly charge for communication cost. Let T_{seq} be the sequential complexity for some computational problem. Then, a parallel algorithm for p processors is 1-optimal if the computation time is in $T_{\text{seq}}/p \cdot (1 + o(1))$ and the communication time is in $o(T_{\text{seq}}/p)$. Thus, the speed-up of a 1-optimal algorithm on real machines should be close to p. Besides [4, 7] there are seemingly no systematic efforts undertaken to design parallel algorithms with respect to this strong optimality criterion.

In this paper we focus on 1-optimal Algorithms and on reducing the number and complexity of communication rounds. For this purpose we employ the BSP* model, an extension of Valiant's BSP model that rewards blockwise communication, i.e. the model captures the benefits of combining the data into few large messages before sending it. 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-optimal with high probability. 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 $q \in s_i$, $q \in \{s_1 \cup \cdots \cup s_{i-1}\}$, or $q \in \{s_{i+1} \cup \cdots \cup s_m\}$. 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

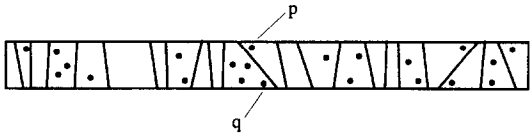

Fig. 1. Strip with segments and query points. Note, that p lies left to q but $s(p)$ is right to $s(q)$.

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 Fig. 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*.

1.2. BSP, BSP* and 1-optimality

The BSP model: In the BSP (Bulk-synchronous parallel) model [17] of parallel computation, a parallel computer 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 synchronizations. 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. The BSP model characterizes a parallel computer by parameters p, L and g.
- The parameter p is the number of processor/memory components.
- L is the minimum time between successive synchronization operations. Thus L is the minimum time for a superstep.
- g is the ratio of the total throughput of the whole system in terms of basic computational operations to the throughput of the router in terms of messages delivered.

The BSP model*: Many routers of real parallel machines support the exchange of large messages and achieve much higher throughput for large messages compared to small ones. In order to make things clear consider the following experiments: We let the router realize an h-relation, i.e. a communication pattern in which each processor sends and receives at most h messages of size one. We measure the throughput in terms of bytes delivered per second. Then, we perform the same experiment several times, each time increasing the size of the messages. A common observation is that the throughput increases with increasing message size until a certain message size, the "optimal" message size, is reached. The main reason for this is that in real parallel systems high start-up costs are involved in the transmission of each message (see

[5]). So the optimal message size may be considerably high. Experiments suggest that on the Parsytec GCel transputer system the messages should be of size at least 1 kBytes. On the Intel Paragon under the OSF/1 operating system and the NX message passing library the messages should have the size of 10 kBytes in order to get half of the possible communication throughput (see [9]). What we can learn from this is that parallel algorithms should be designed such that the communicated data can be combined into few large messages. This is what we call blockwise communication. To incorporate this aspect of real parallel systems in our model we extend the BSP model to the BSP* model by adding the parameter B, and changing the parameter g. Parameters p and L remain unchanged.

- The parameter B is the "optimal" message size, i.e. the size the messages must have in order to fully exploit the bandwidth of the router.
- g is the ratio of the total throughput of the whole system in terms of basic computational operations to the throughput of the router in terms of messages of size B delivered.

For a computation superstep with at most t local operations on each processor we charge $\max\{L, t\}$ time units. The costs for a communication superstep is determined as follows: Assume that a processor P sends h messages of sizes $s_1, \ldots, s_h$ and receives h' messages of sizes $s'_1, \ldots, s'_{h'}$ in one superstep. Then P's communication cost is

$$\max\left\{L, g \cdot \max\left\{\sum_{i=1}^{h}\left(\left\lceil\frac{s_i}{B}\right\rceil\right), \sum_{i=1}^{h'}\left(\left\lceil\frac{s'_i}{B}\right\rceil\right)\right\}\right\}.$$

The time for the communication superstep is defined as the maximum over the communication costs of all processors. Note that messages of size smaller than B are treated as if they were of size B. Thus in the BSP* model it is worthwhile to send messages of size at least B. Messages of size smaller than B are strictly punished. So blockwise communication means to have messages of size B in the average.

In our algorithms, each processor will always combine messages that are sent to the same processor in the same superstep to one big message in order to take advantage from blockwise communication. Therefore we get the following simplified bound for the cost of a communication superstep. Assume that each processor sends messages of total size at most s to at most r processors, and each processor receives messages of total size at most s' from at most r' processors. Then the cost of the communication superstep is bounded by runtime

$$\max\left\{g \cdot \max\left\{\frac{s}{B} + r, \frac{s'}{B} + r'\right\}, L\right\}.$$

In [13], Miller suggests an extension to the BSP model that is very similar to ours. There, a parameter $n_{1/2}$ is introduced that corresponds to the BSP* parameter B. The messages must have the size of $n_{1/2}$ in order to exploit half of the possible bandwidth of the router. So the results in this paper can immediately translated to the BSP extension in [13].

In many situations it is easy to achieve blockwise communication, i.e. large messages. Assume that in some superstep each processor has to send and receive $p^{1+\varepsilon}$ words of data. Since the data that each processor sends and receives can be combined into at most p messages, the average message size is p^{ε}. So it is more interesting to consider situations in which n/p, the input size per processor, is small. Our algorithms achieve blockwise communication even if n/p is polylogarithmic in n.

1-*Optimality*: Let A^* be the best sequential algorithm on the RAM for the problem under consideration, and let $T_{A^*}(n)$ be its worst case runtime. Let A be a BSP* algorithm for the problem and let $M_A(n)$ and $C_A(n)$ be the time that A needs for communication and computation supersteps, respectively. In order to be 1-*optimal* (with respect to p, L, g and B) A has to fulfill the following requirements [7]:

- The ratio between $C_A(n)$ and $T_{A^*}(n)/p$ has to be in $1 + o(1)$.
- The ratio between $M_A(n)$ and $T_{A^*}(n)/p$ has to be in $o(1)$.

All asymptotic bounds refer to the problem size as $n \to \infty$. Thus, the parallel runtime $T_A(n) = C_A(n) + M_A(n)$ is bounded by $(T_{A^*}(n)/p) \cdot (1 + o(1))$, i.e., the speed-up tends to p with growing input size.

1.3. Known results

Sequentially, Multisearch can be done in time $n \log m$ in the worst case by simply executing binary search on the segments for each query. This sequential algorithm can easily be parallelized optimally in the case of $p \leqslant n$ on the CREW-PRAM. For the EREW-PRAM it is already a very complicated problem. Reif and Sen [16] developed an asymptotically optimal randomized EREW-PRAM algorithm, which works also on the butterfly network. It runs in time $O(\log n)$, with high probability, for $n = m = p$. However, large constants are involved and it performs badly on the BSP* model. Further it is not obvious how to generalize the algorithm work optimally for the case $n < m$. The EREW-PRAM algorithm of Paul et al. [14] is capable of executing n search queries on a search tree with m nodes in time $O(\log m + \log n)$. Unfortunately, their approach would require too much fine grained communication on the BSP* model. Ranade [15] has developed a multisearch algorithm for the p processor butterfly network for $n = p \log p$ queries and m polynomial in p. For the case $m \geqslant n$ 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. [6] have developed some algorithms on the CGM for geometric problems including Multisearch. In contrast to the other results quoted above Dehne et al. assume that the machine is much smaller than the problem size, i.e. $p \leqslant \sqrt{n}$. 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. [7].

McColl [12] 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. $m \leqslant n \geqslant p$. It is a deterministic BSP* algorithm that is 1-optimal, if $n/p \geqslant \log^2 n$. The second algorithm (*FewQueries*) works for Multisearch with few queries, i.e. for $2^n \geqslant m > n \geqslant p$. It is a randomized BSP* algorithm that is 1-optimal with probability $1 - n^{-c}$ for arbitrary $c > 0$ and $n/p \geqslant \log^3 n$. These results hold for a wide range of BSP* parameters. E.g. $L, B \leqslant (n/p)^\eta$ and $g = o(B \log\log n)$ suffice for $\eta < 1$ small enough. The algorithms use blockwise communication. Without that, the algorithms would need $g = o(\log\log n)$ in order to be 1-optimal. Note that p, L, 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 messages. Our algorithms use routines for broadcast, parallel prefix and load balancing as basic routines. BSP* algorithms for these problems are part of our work.

1.5. Organisation of the paper

In Section 2 we give an outline of the algorithms 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. Section 6 discusses implementation issues on parallel machines.

2. Outline of the algorithm and notations

Outline: Both algorithms start with a preprocessing phase where a suitable balanced search tree is 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 small depth. The nodes of the search tree are then mapped to the processors. We have different search trees and different mapping strategies for the case $p \leqslant n \leqslant m$ and $p \leqslant n > m$.

After the preprocessing phase the queries are processed. In both algorithms 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 depth of the search tree. In round i the algorithm determines for each query which node it visits 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: In the case of $m \leqslant n$ the preprocessing provides us with a search

tree with at most p nodes. Therefore a one-to-one-mapping of tree nodes to processors works fine. But in the case of $m > n$ our search tree has much more nodes than processors and a deterministic distribution causes contention problems: One can always find an input such that in one round only nodes mapped to the same processor are accessed. In order to make contention unlikely we distribute the nodes of the search tree randomly. The randomization 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 design and analyse a distribution which preserves locality, i.e. allows blockwise communication, but randomizes sufficiently to guarantee low contention with high probability.

During the travel of the queries through the tree the following problems arise. 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.

Notations: 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 a 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. Let the queries of these jobs be distributed among the processors. A distribution of the queries of $J_1,\ldots,J_k$ among the processors is *balanced*, if no processor gets more than n/p of the queries. The distribution is *ordered* if the following holds: The queries of each job are distributed among consecutive processors, only the first and the last of these processors of may hold queries from other jobs. Processors holding queries from the same job form a *group*. The first processor of a group is the *group leader*. If the group consists of one processor the job is called *exclusive*, if not it is *shared*. The distribution of the queries of $J_1,\ldots,J_k$ among the processors is *compact* if for each shared job the following holds: All but the first and the last processor of the respective group store n/p queries of the job. For an example see Fig. 2.

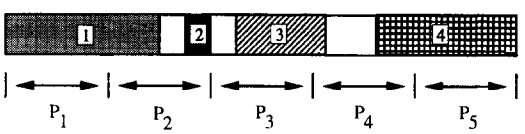

Fig. 2. The distribution of the jobs $1,\ldots,4$ among the processors $P_1,\ldots,P_5$ is ordered, balanced and compact. Jobs 1, 3 and 4 are shared jobs, job 2 is an exclusive job. Job 1, (3,4) is held by the processor group $\{P_1,P_2\}$ $(\{P_3,P_4\},\{P_4,P_5\})$.

3. Communication routines

In this section we describe and analyse BSP* algorithms that perform several communication tasks: Broadcast, Parallel-Prefix, Distribute and Load Balance. All these are auxiliary routines for the two Multisearch algorithms described in the next two sections. In this section we present the algorithms as they work on all p processors $P_1,\ldots,P_p$. Later when we apply the algorithms for the multisearch algorithms, we need to execute several instances of the algorithms concurrently on disjoint groups of processors. The algorithms presented here can easily be adapted such that several instances run on disjoint groups of processors at the same time.

3.1. Broadcast

Consider a vector of size ℓ 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 balanced binary tree with root P_1. The other processors are called internal processors. P_1 splits the vector in $\min\{\ell, \log p\}$ packets, each of size at most $\lceil \ell/\log p\rceil$. The algorithm proceeds in rounds, each consisting of a communication and a computation superstep. In the ith round P_1 makes two copies of the ith 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 $O(\log p)$ rounds. Each round takes time $\max\{2g\lceil\lceil \ell/\log p\rceil/B\rceil, L\}$ for the communication superstep and time $\max\{\lceil \ell/\log p\rceil, L\}$ for the computation supersteps.

Result 1. *Let $M_{br}(\ell)$ and $C_{br}(\ell)$ be the time for communication and computation supersteps, respectively, that the algorithm Broadcast needs for broad-casting a vector of size ℓ. Then*

- $M_{br}(\ell) = O(\ell + \log p + L \cdot \log p)$,
- $C_{br}(\ell) = O(g \cdot (\ell/b + \log p) + L \cdot \log p)$, *and*
- *space* $O(\ell)$ *per processor is needed.*

3.2. Parallel-prefix

Let p vectors of size ℓ be given where the ith vector is stored in the ith processor. The task of the algorithm *Parallel-Prefix* is to compute, for each $k \in \{1,\ldots,\ell\}$, the prefix sums of the kth 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. But now the tree has p leaves, i.e. it is bigger than in the broadcast algorithm. So each processor represents one leave node and one internal node. We employ a pipelined version of the standard parallel prefix algorithm of Ladner and Fischer [11] that proceeds in two phases. In the first phase the sums move from the leaves of the tree to the root, in the second from the root to the

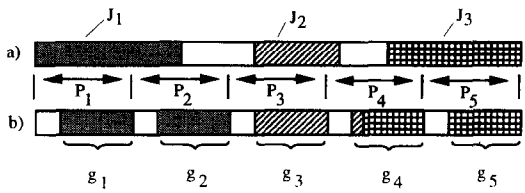

Fig. 3. (a) The situation before and (b) after Distribute.

leaves making some calculations at each level. The vectors are split into $\min\{\ell, \log p\}$ packets of size at most $\lceil \ell/\log p \rceil$ in order to proceed in a pipelined fashion as in the algorithm Broadcast. Note that each processor represents two nodes of the tree. So in each superstep the processors first perform the operations for one node and then for the other node. The following result can easily be concluded.

Result 2. *Let $M_{ppr}(\ell)$ and $C_{ppr}(\ell)$ be the time for communication and computation supersteps, respectively, that the algorithm Parallel-Prefix needs for calculating the prefix sums for vectors of size ℓ. Then*
- $M_{ppr}(\ell) = O(\ell + \log p + L \cdot \log p)$,
- $C_{ppr}(\ell) = O(g \cdot (\ell/b + \log p) + L \cdot \log p)$, *and*
- *space $O(\ell)$ per processor is needed.*

3.3. Distribute

The following algorithm performs a redistribution task. As input we have a couple of jobs (refer to Section 2 for the notations) distributed in a certain way among the processors, as output we get a certain redistribution of the queries of these jobs among the processors. First we define our special redistribution problem and then we give the algorithm. Remind that we describe the problem and the algorithm for all p processors. The algorithm can easily be modified such that it runs on a smaller group of processors. This is needed because in subsequent sections we let several instances of the algorithm run on disjoint groups of processors.

Definition 1. A redistribution problem for jobs of size at least t *is defined as follows. See Fig.* 3 *for an example.*
- *Input*: A set of input jobs $J_1, \ldots, J_k$, each of size at least t, and *gaps* $g_i \leqslant \frac{n}{p}$ for each processor P_i, where $\sum_i g_i$ is at least the total size of the input jobs. The jobs are distributed among the p processors in an ordered, balanced and compact way. Further, each processor knows the size of each job.
- *Output*: Input jobs $J_1, \ldots, J_k$ are redistributed in an ordered way among the processors such that each processor P_i holds at most g_i of the queries from $J_1, \ldots, J_k$.

Next we present the algorithm *Distribute* which solves the redistribution problem for jobs of size at least t.

Algorithm Distribute. (i) The queries of $J_1,\ldots,J_k$ are labeled with numbers from 1 to $\sum_{i=1}^{k}|J_i|$ such that different queries get different numbers and queries on processor P_i get a lower number than queries on processor P_j and queries of J_i get a lower number than queries of J_j, if $i<j$. The labeling can easily be done by a parallel prefix computation.

(ii) Each processor which holds a query x_i (i corresponds to the labeling computed in the previous step) sends it to the processor with number $\lceil i/\lceil n/p\rceil\rceil$. So in this step a kind of compaction is performed. This step can be done in one communication superstep where all queries are sent in blocks of size B if possible.

(iii) The queries of the input jobs are to be distributed among the processors such that they fill up the gaps. Thus a target processor has to be computed for each query. This is done by a parallel prefix computation on the gaps g_i. Afterwards P_i knows i_1 and i_2, where x_{i_1} is the first and x_{i_2} is the last query of the input jobs P_i has to store in order to fill up the gap.

(iv) If a processor is the target processor for the $(k\lceil n/p\rceil)$th or the $((k+1)\lceil n/p\rceil-1)$th query of the input jobs it sends a message to P_k. From these messages P_k knows the target processors for the queries it holds.

(v) Every P_i broadcasts the queries of the input jobs it holds to the target processors calculated in the previous step. Note that for each broadcast the target processors are consecutive and that each processor can be a target processor for at most two broadcast operations.

(vi) Each processor checks which of the received queries it has to store in its gap, stores them and discards the others in one computation superstep.

Result 3. *Let $M_{\mathrm{dist}}(t)$ and $C_{\mathrm{dist}}(t)$ be the time for the communication and computation supersteps, respectively, that the algorithm* Distribute *needs in order to solve a redistribution problem for jobs of size at least t. Then*

- $M_{\mathrm{dist}}(t)=\mathrm{O}(g(\frac{n}{pB}+\frac{n}{pt}+\log p)+L\cdot\log p)$,
- $C_{\mathrm{dist}}(t)=\mathrm{O}(\frac{n}{p}+\log p+L\cdot\log p)$, *and*
- $\mathrm{O}(n/p)$ *space per processor is needed.*

Proof. The correctness of the algorithm is obvious. We prove the time bounds.

Step (i): The parallel prefix computation needs time $M_{ppr}(1)$ and time $C_{ppr}(1)$ for communication and computation, respectively. Additionally Step 1 requires computation time $\mathrm{O}(n/p)$ for local counting and numbering of queries and jobs.

Step (ii): Each processor sends at most n/p queries to at most 2 different processors. Each processor receives at most n/p queries from at most $\frac{n}{p\cdot t}$ different processor. This communication superstep needs time $\mathrm{O}(g(n/pB)+(n/pt)+L)$.

Step (iii): Step 3 has the same time bounds as step 1.

Step (iv): It requires communication time $\mathrm{O}(g+L)$.

Step (v): This step needs time $M_{br}(n/p)$ and time $C_{br}(n/p)$ for communication and computation, respectively. Note that each processor needs to broadcast *all* its queries to the different target processors. If we would send the queries directly to the target

processors, the following could happen. A processor might hold n/p queries that have different target processors. In this case the processors would need to send n/p messages of size one, which is not according to the philosophy of blockwise communication.

Step (vi): This computation superstep needs time $O((n/p)+L)$.

Result 3 can be concluded by adding up the above time bounds. □

3.4. Load-Balance

The algorithm *Load-Balance* solves a redistribution problem of the following type. Remind that we describe the problem and the algorithm for all p processors. The algorithm can easily be modified such that it runs on a smaller group of processors. This is needed because in subsequent sections we let several instances of the algorithm run on disjoint groups of processors.

Definition 2. Let J be a job on a certain level of the search tree and let $J_1,\ldots,J_d$ be the d successor jobs of J. A *d-ary load-balancing problem* is defined as follows:

- *Input*: The queries of J are distributed among the processors such that each processor holds at most n/p queries of J. P_1 holds a vector V with the sizes of all successor jobs of J. Further each query is labeled with the number of the successor job it belongs to. A query that belongs to the kth successor job of J gets the label k, with $1 \leqslant k \leqslant d$.
- *Output*: The jobs $J_1,\ldots,J_d$ distributed among the processors in an ordered, balanced and compact way. See Fig. 4 for an example.

A d-ary load-balancing problem can easily be solved. If P_1 broadcasts the vector V 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 processors, a problem arises: Many processors could hold only few queries for a certain target processor, especially less than the block size B. Thus $\Theta(n/p)$ small packets may be sent to a processor. This would require communication time $O(gn/p)$ 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.

Given the input distribution we call the queries of the kth successor job of J held by processor P_i the *ith fraction of* J_k. Let S_k denote the set of queries of fractions of J_k that are smaller than B. For each k the algorithm *Combine* redistributes S_k as

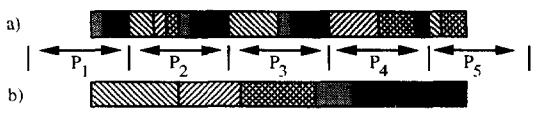

Fig. 4. (a) The situation before and (b) after Load-Balance for a group.

follows: Each of the first $\lfloor |S_k|/B \rfloor$ processors gets B queries of S_k, the $(\lfloor |S_k|/B \rfloor + 1)$th processor gets $|S_k| - \lfloor |S_k|/B \rfloor \cdot B$ queries of S_k.

After a call of *Combine* we like to have the following situation: For each k the following holds. Among the processors that hold queries of the successor job J_k there is at most one processor that holds fewer than B of these queries. Further we have $O(n/p)$ queries per processor.

We now describe the algorithm *Combine* in detail. It works on the input of a d-ary load-balancing problem and will later be called at the start of algorithm *Load-Balance*.

Algorithm Combine. (i) Each processor P_l computes a_k^l, the size of the lth fraction of the kth successor job. The processor P_l sets b_k^l to a_k^l if $a_k^l < B$ otherwise to 0 for each k. This step can be done in one computation superstep. Note that $k \leqslant d$.

(ii) A parallel prefix operation is performed on the d-vectors b^l, where $b^l = (b_k^l)$ and $l \in \{1, \ldots, p\}$. Afterwards each P_l holds the resulting vector r^l.

(iii) If $a_k^l < B$ processor P_l sends the queries belonging to the lth fraction of the kth successor job to the processor P_{c+1}, if $c \cdot B \leqslant r_k^l < (c+1) \cdot B$. This can be done in one communication superstep.

(iv) The processors which have received packets combine the queries belonging to the kth subjob for each k, that means they store queries which belong to the same subjob in the same list. This can be done in one computation superstep.

Result 4. *The algorithm Combine needs communication time* $O(g \cdot B \cdot d + L + M_{ppr}(d))$, *computation time* $O((n/p) + L + C_{ppr}(d))$, *and space* $O(n/p)$.

Proof. The correctness can easily be seen. For each successor job J_k at most one processor holds less than B queries of that job. Further $O(n/p)$ queries are stored by each processor. We now prove the time bounds.

Step (i): This computation superstep needs time $O((n/p) + L)$.

Step (ii): This step needs communication time $M_{ppr}(d)$ and computation time $C_{ppr}(d)$.

Step (iii): Each processor sends at most $B \cdot d$ queries to at most $2 \cdot d$ processors. Each processor receives at most $B \cdot d$ queries from at most $B \cdot d$ processors. Thus, this communication superstep needs time $O(g \cdot B \cdot d + L)$.

Step (iv): This computation superstep needs time $O((n/p) + L)$ for combining the different fractions locally.

Result 4 can be derived by adding up the time bounds for steps 1–4. □

Algorithm Load-Balance. (i) The processors call *Combine*.

(ii) Processor P_1 knows the size of all its successor jobs. It distributes this information to the processors of the group, i.e., it broadcasts a vector of size d, where the kth entry is the size of successor job J_k.

(iii) Now each processor P_l locally computes the prefix sums of the vector. The kth prefix sum is the offset for queries with label k. We call this offset s_k. This is done in one computation superstep.

(iv) Queries with the same label are numbered such that a query of a certain label being held by processor P_l gets a smaller number as a query of the same label held by processor P_{l+x}, if $x \geqslant 1$. This can be done by a parallel prefix computation. Together with the above computed offset s_k for each query with label k the target processor will be computed.

(v) Each processor sends the queries to the target processors computed in step 4. This is done in one communication superstep. The queries are send in blocks of size B, if possible.

Result 5. *Let $M_{lb}(d)$ and $C_{lb}(d)$ be the communication and computation time that algorithm Load-Balance needs in order to solve a d-ary load-balancing problem. Then*
- $M_{lb}(d) = O(g(B \cdot d + (n/Bp) + (d/B) + \log p) + L \cdot \log p)$,
- $C_{lb}(d) = O(d + (n/p) + \log p + L \cdot \log p)$, *and*
- *space* $O(n/p)$ *per processor is needed.*

Proof. The correctness of the algorithm is obvious. In the following we prove the time bounds.

Step (i): By Result 4 this step needs communication time $O(g \cdot B \cdot d + L + M_{ppr}(d))$ and computation time $O((n/p) + L + C_{ppr}(d))$.

Step (ii): This step needs communication time $M_{br}(d)$ and computation time $C_{br}(d)$.

Step (iii): This computational superstep needs time $O(d + L)$.

Step (iv): This step needs communication time $M_{ppr}(d)$ and computation time $C_{ppr}(d)$. Further this step needs computation time $O(n/p)$ for local numbering.

Step (v): After the call of *Combine* at most one processor which stores queries of a successor job J_k has less than B queries of J_k. As the processors hold $O(n/p)$ queries each sends $O(n/p)$ queries to $O(n/pB)$ different processors. Each processor receives at most (n/p) queries from $O(n/pB)$ different processors. Thus this communication superstep needs time $O(g(n/Bp) + L)$.

Result 5 can be concluded by adding up the time bounds for Step 1 to Step 5. □

4. Multisearch with many queries

In this section we show how to do Multisearch for the case $p \leqslant n \geqslant m$ such that the internal work is almost the same as for the sequential algorithm and the ratio of communication time to computation time is in $o(1)$, i.e. we give an algorithm that is 1-optimal for a large bandwidth of parameter constellations.

We consider only the case $m = n$ but the algorithm also works for the case $n > m$. The main idea is the following. In a preprocessing step we construct a balanced search tree over S that has high degree and small depth. After that we let the queries "flow" through the search tree, along their search paths, level by level from the root to the leaves. The main problem arises from the fact that some nodes may be visited by only few queries and other nodes may be visited by many queries.

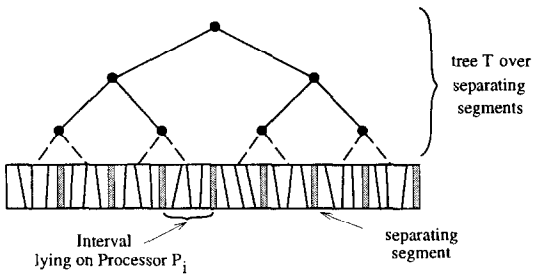

Fig. 5. Tree T built upon the set of separating segments. Here the degree d is 2.

4.1. Preprocessing

In the preprocessing phase a search tree is constructed over the set of input segments S. This is done as follows: As input we have at most $\lceil n/p \rceil$ consecutive segments on each processor, they form *intervals*. For each processor P_i we denote the largest segment held by P_i as a *separating segment*.

Now we build a balanced search tree T where the leaves are formed by the intervals and the internal nodes are formed by the separating segments (see Fig. 5). Let d be the degree of T. Later we will see how d has to be chosen in order to get an 1-optimal algorithm. Note that T has at most p internal nodes. The preprocessing proceeds in two steps.

(i) Allocate a processor for each internal node of T. In one communication superstep each processor fetches the appropriate separating segments that it needs in order to build the node for which it is responsible. Thus a d-relation with messages of size 1 is routed since each node is made from d separating segments and initially each processor holds only one separating segment.

(ii) In one computation superstep each processor builds a tree node from the separating segments it has fetched in the previous step.

Thus the following can easily be concluded:

Fact 3. *The preprocessing needs computation time* $O(\max\{d, L\})$ *and communication time* $\max\{g \cdot d, L\}$. *The constructed tree T has at most p internal nodes each of degree d. T has* $O(\log p / \log d)$ *levels and each processor stores the segments of at most one interval and one internal node of T. The data structure needs* $O((n/p) + d)$ *storage per processor. The root of T is stored on processor* P_1.

4.2. Executing queries

Now we let the queries flow through the search tree T. This is done by the algorithm *ManyQueries* that consists of three phases:

We distinguish small jobs and large jobs. We choose small jobs to be of size $< t$ and large jobs to be of size $\geqslant t$. Later we will see how t has to be chosen in order to get an 1-optimal algorithm. The first phase is subdivided into rounds. In the ith 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 large 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. No two of these jobs 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 large jobs on level i of T will be considered and the following redistribution problem has to be solved.

Definition 4. *Input for Round i*: Input jobs are the large jobs on level i of the search tree T (size at least t). They are distributed in a balanced, ordered and compact way among the processors (For notations compare Section 2).

Output of Round i: Output jobs are the jobs on level $i+1$ of T. Large output jobs are distributed in a balanced, ordered and compact way among the processors. Small output jobs are directly sent to the processors which hold the appropriate nodes of T.

Note that the input jobs for Round i may be *shared* or *exclusive* (see Section 2). The redistribution problem above can be solved by redistributing the queries of each shared input job among the processors of the respective processor group and by redistributing the queries of each exclusive job locally on the processor that holds this job. This will be done as follows:

- Each processor P redistributes the queries of exclusive input jobs which are mapped to it as follows: For each of its exclusive input jobs it fetches the nodes from the processors that store them. These can only be $(n/p \cdot t)$ many nodes, since input jobs are only large jobs (size $>t$). Thus, this step is not too expensive if the value of t is large enough. This is crucial for the analysis. After that each processor determines the successor jobs by means of binary search and stores the large successor jobs in an ordered way in its memory. Small successor jobs (size smaller than t) are directly sent to the processors that hold the appropriate node for that job.
- A group of processors $P_i,\ldots,P_j$ redistributes a shared input 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 among $P_i,\ldots,P_j$ such that afterwards the

successor jobs of J are distributed in a balanced, ordered and compact 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 redistribution of one shared job or the redistribution of some exclusive jobs. In fact it can be involved in the redistribution of up to two shared jobs *and* up to (n/pt) 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 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.

(i) For each input job the corresponding node of T has to be fetched. For each shared input job the respective group leader fetches the appropriate node of T. Each processor that holds exclusive input jobs fetches the appropriate nodes of T for each of its exclusive jobs. Step 1 can be done in two communication supersteps. Note that input jobs are of size at least t and that each processor holds at most (n/pt) exclusive input jobs.

(ii) For each shared input job the groupleader of the respective group broadcasts the node it has fetched in the previous step to the other processors of the group.

(iii) 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. Step 2 can be done in one computation superstep.

(iv) For each shared input job the processors of the respective group compute the size of the new successor jobs by means of parallel prefix on vectors of size d, where the ith component of the vector corresponds to the number of queries which visit the ith child node of the current node of T. The group leader knows the size of each of these successor jobs afterwards.

(v) For each shared input job the successor jobs are redistributed in a balanced, ordered and compact way among the processors of the respective group by executing procedure Load-Balance. The processors holding exclusive jobs compute the redistribution sequentially (by a bucket sort approach).

(vi) Each processor sends the queries of each small successor job (size smaller than t) it holds to the processor which holds the node of T that the successor job wants to visit next. If the leaves of T are reached, Phase 2 is entered, else goto 1.

Description of Phase 2:

In this phase the large jobs which have reached the leaves of T after Phase 1 are processed. Remember that T has p leaves, therefore the queries are partitioned into at most p jobs. Unfortunately a processor can hold queries of up to (n/pt) large jobs, since a large job can be as small as t. Thus for Phase 2 we have to solve the following problem:

- *Input*: Input jobs are the up to p large jobs on the bottom level of T (size at least t). They are distributed in a balanced, ordered and compact way among the processors.
- *Output*: For each query of the input jobs its segment is determined.

Next we give the details of Phase 2:

(i) The processors redistribute the queries of the at most p input jobs as follows: Input jobs of size at most n/p are placed on the processor that holds the interval that the respective job has to visit. Thus different jobs of size at most n/p are placed on different processors. This is done in one communication superstep.

(ii) Let r_i be the size of the job which is placed on processor P_i in the last step. The value $(n/p)-r_i$ is called the gap of P_i. Input jobs of size at least n/p are distributed in an ordered way among the processors such that they fill up the gaps. This is a redistribution problem for jobs of size at least n/p which can be solved by the algorithm *Distribute*. After that each processor holds queries of at most three jobs and it holds at most n/p queries.

(iii) From Step 1 we get exclusive jobs which are stored in one processor and from step 2 we get shared jobs which are stored by a processor group. For each shared job the corresponding interval is fetched by the leader of the group that holds this job.

(iv) Each group leader broadcasts the interval it has fetched in the previous step to the processors of its group.

(v) 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 have been sent (during Phase 1) to the processors that hold the nodes of T they have to visit next. Thus for Phase 3 we have to solve the following problem:

- *Input*: Each processor holds at most one small job (size smaller than t). The jobs belong to different subtrees of T.
- *Output*: For each query of the input jobs its segment is determined.

Next we give the details of Phase 3:

(i) Each processor which has received a small job broadcasts it to the processors storing the intervals reachable from the corresponding node of T. Note that the small jobs that are left to phase 3 have to inspect disjoint subtrees.

(ii) Each processor determines for the received queries the correct segment by means of binary search on the segments of its interval.

Theorem 5. *Phase* 1 *of the algorithm ManyQueries needs*
- *communication time*

$$O\left(\frac{\log p}{\log d}\cdot\left(g\cdot\left(\frac{n}{p\cdot B}+\frac{n\cdot d}{p\cdot t}+B\cdot d+\frac{d}{B}+\log p\right)+L\cdot\log p\right)\right),$$

– *computation time*

$$\frac{n}{p}\log p + \mathrm{O}\left(\frac{\log p}{\log d}\cdot\left(d + \log p + \frac{n}{p} + L\cdot\log p\right)\right).$$

Phase 2 of the algorithm ManyQueries needs
– *communication time*

$$\mathrm{O}\left(g\cdot\left(\frac{n}{p\cdot B} + \frac{n}{p\cdot t} + \log p\right) + L\cdot\log p\right),$$

– *computation time*

$$\frac{n}{p}\log\frac{n}{p} + \mathrm{O}\left(\frac{n}{p} + \log p + L\cdot\log p\right).$$

Phase 3 of the algorithm ManyQueries needs
– *communication time*

$$\mathrm{O}\left(g\cdot\left(\frac{t}{B} + \log p\right) + L\cdot\log p\right),$$

– *computation time*

$$t\cdot\log\frac{n}{p} + \mathrm{O}(t + \log p + L\cdot\log p).$$

Proof. *Analysis of Phase* 1: Let h be the number of levels of T, i.e., $h = \mathrm{O}(\log p/\log d)$. The Steps 1–6 are repeated at most h times.

Step (i): Each processor holds queries of at most $\mathrm{O}(n/pt)$ large input jobs. Therefore each processor has to fetch $\mathrm{O}(n/pt)$ 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 $\mathrm{O}(n/pt)$-relation with messages of size 1 is realized, i.e., each processor sends and receives $\mathrm{O}(n/pt)$ messages of size 1. In the second superstep the nodes will be sent to the requesting processors. Thus an $\mathrm{O}(n/pt)$-relation with messages of size $\mathrm{O}(d)$ is routed. The first superstep needs time $\max\{g\cdot(n/pt), L\}$ the second needs time $\max\{g\cdot\frac{n}{pt}\cdot\lceil\frac{d}{B}\rceil, L\}$. Together Step 1 needs communication time $\mathrm{O}(\max\{g\cdot(n/pt)\cdot\lceil d/B\rceil, L\})$.

Step (ii): This step needs computation time $C_{br}(d)$ and communication time $M_{br}(d)$.

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

Step (iv): This step needs communication time $M_{ppr}(d)$ and computation time $C_{ppr}(d)$.

Step (v): This step needs communication time $M_{lb}(d)$ and computation time $C_{lb}(d)$. For exclusive jobs the reorganisation is done sequentially in time $\mathrm{O}(n/p + d)$.

Step (vi): The queries of small successor jobs are directly sent to the nodes they have to visit next. A processor holds at most n/pt exclusive input jobs where each one has up to d successor jobs. Therefore each processor sends at most n/p queries to at most

$(n \cdot d)/(p \cdot t)$ processors and each processor receives at most t queries from at most 1 processor. Thus step 6 needs communication time $O(g((n/Bp)+(n \cdot d/p \cdot t))+L)$.

Analysis of Phase 2:

Step (i): Each processor sends at most n/p queries to at most n/pt different processors and each processor receives at most n/p queries from at most 2 different processors. This communication superstep needs time $O(g((n/pB)+(n/pt))+L)$.

Step (ii): This step requires time $M_{\text{dist}}(n/p)$ for communication and $C_{\text{dist}}(n/p)$ for computation.

Step (iii): Each processor fetches up to three intervals of size n/p. This can be done in two communication supersteps which need time $O(g \cdot (n/Bp)+L)$.

Step (iv): This step requires communication time $M_{br}(n/p)$ and computation time $C_{br}(n/p)$.

Step (v): It requires computation time $\lceil n/p \rceil \log \lceil n/p \rceil \leqslant (n/p)\log n-(n/p)\log p+O(n/p)$.

Analysis of Phase 3:

Step (i): This step needs communication time $M_{br}(t)$ and computation time $C_{br}(t)$.

Step (ii): This step needs computation time $t \cdot \log \lceil n/p \rceil$. □

The next corollary can easily be concluded from Theorem 5.

Corollary 6. *If* $n/p=\Omega(\log p \cdot B)$ *and* $d,t=O(n/p)$ *then the algorithm ManyQueries needs*

– *computation time*

$$\frac{n}{p}\log n+t \cdot \log \frac{n}{p}+O\left(\frac{\log p}{\log d} \cdot \left(\frac{n}{p}+L \cdot \log p\right)\right),$$

– *communication time*

$$O\left(\frac{\log p}{\log d} \cdot g \cdot \left(B \cdot d+\frac{n}{pB}+\frac{n \cdot d}{p \cdot t}\right)+L \cdot \log p\right)$$

If we further have $B=O((n/p)^{\eta})$, *with* $0<\eta<1/2$, $L=O(n/p)$ *and* $g=o(\log n \cdot B/\log_d p)$, *then we can choose* $t=(n/p)^{\alpha}$, *with* $1/2<\alpha<1$, *and* $d=(n/p)^{\alpha'}$, *with* $\alpha'=\min\{\alpha-\eta, 1-2\eta\}$. *In this case the algorithm ManyQueries is* 1*-optimal.*

In particular, for $t=(n/p)^{0.6}$, $d=(n/p)^{0.2}$, $B \leqslant (n/p)^{0.4}$ *and* $L \leqslant (n/p)^{0.2}$ *the algorithm ManyQueries is* 1*-optimal for*

– $n/p \geqslant n^{\varepsilon}, \varepsilon>0$, *if* $g=o(B \cdot \log n)$

– $n/p \geqslant \log^2 n$, *if* $g=o(B \cdot \log \log n)$.

Note that if we would not have exploited blockwise communication, then in the particular cases in the corollary we would need $g=o(\log n)$ and $g=o(\log \log n)$, respectively, in order to achieve 1-optimality.

5. Multisearch with few queries

In the sequel only few queries are asked: we allow now $p \leqslant n < m \leqslant 2^n$, rather than $n \geqslant m$. Recall that the sequential time needed is $n \log m$ in the worst case. It can easily be seen that algorithm ManyQueries is far from optimal in this case: If we would use the preprocessing step of algorithm ManyQueries, the leaves of the search tree would be of size m/p. Thus contention would be caused if all queries happen to belong to segments stored in the same leave. Therefore we need a slightly different search tree and a new way to distribute its nodes among the processors.

We now organize the m segments in a search tree T of degree d with the leave nodes of size at most d. A suitable value for d will be given later. T has m/d nodes and depth $\log m/\log d$. This is in contrast to the algorithm ManyQueries where we have a search tree with at most $2p$ nodes, with leaf nodes of size m/p. 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 and therefore much more than p jobs may be generated. We can have up to n jobs of size 1. As before we denote jobs of size smaller than t as *small jobs* and jobs of size at least t as *large jobs*. A suitable value for t will be given later. Next we describe how the jobs are executed by the algorithm FewQueries:

In order to execute the jobs the queries of the jobs have to be brought together with the nodes they want to visit. We have different strategies for large and small jobs: For large jobs we proceed as in the algorithm ManyQueries: Tree nodes are sent to the processors that hold the appropriate job. For small jobs we cannot proceed like that. If a processor holds n/p small jobs of size 1 one would have to send n/p nodes to this processor which is too expensive if the nodes are big. Therefore for small jobs we employ another strategy: Jobs are sent to the processors that hold the appropriate node. There the small jobs are executed, i.e., they are partitioned in successor jobs which are then sent to the processors holding the nodes to be visited.

For each level of the search tree our algorithm performs a round. In Round i all jobs of level i are executed. In order to reach 1-optimality large jobs and small jobs are executed in an interleaving fashion.

If we proceed like that, we encounter the following problem: For every mapping of nodes of T to the processors we can find an input that generates many jobs that want to access only nodes mapped to the same processor. Thus we have high worst case contention for each mapping. A standard technique to reduce contention is randomization: If the nodes are randomly distributed among the processors, high contention becomes very unlikely for arbitrary sets of queries. For our purposes, however, a naive random distribution is not suitable, because the children nodes of an arbitrary node 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.

In the following we first present the algorithm that maps the nodes of T to the processors. After that we prove a lemma that states crucial properties of the mapping. Then we present and analyse the algorithm that executes the queries.

5.1. Preprocessing

The algorithm Build-Up(z) maps the nodes of T to the processors such that children of nodes of level i which are mapped to the same processor are distributed among only z, processors, for $z < p$.

Algorithm Build-Up(z).

$i = 0$: The root is 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 z, $z \leqslant p$, processors (neighbour processors) and distributes the children of nodes of R randomly among these neighbour processors.

The following is easy to check:

Fact 7. *The algorithm Build-Up needs time* $O(Lh + g(m/p))$ *and* $O(m/p)$ *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 8. (a) *Consider a tree T with degree d and depth h distributed among p processors by algorithm Build-Up(z). Fix $l, l \leqslant n$, nodes $v_1, \ldots, v_l$ on some level of T and give them non-negative weights $g_1, \ldots, g_l$, each at most, such that the total weight of $v_1, \ldots, v_l$ is at most n. Then, for arbitrary $c > 0$, there is a $c' > 0$ such that the following holds: If $z \geqslant c' \cdot \max\{d \cdot t \cdot \log n, d \cdot \log^2 n\}$ then the weight of fixed nodes that are placed on the same processor by Build-Up(z) does not exceed $(1 + o(1))\frac{n}{p}$ with probability at least $1 - n^{-c}$, if $n/p = \omega(\log n \cdot t)$.*

(b) *For each processor P_i, the parents of the nodes stored in P_i are distributed among $O(z)$ processors, with probability at least $1 - n^{-c}$ for an arbitrary $c > 0$.*

Proof of Lemma 8. (a) We need the following estimates. The first one is a tail estimate for sums of independent random variables due to Hoeffding [8]. The second estimate is elementary.

Fact 9. (a) *Let $X_1, \ldots, X_n$ be independent random variables with $X_i \in [0, \ldots, k]$ and $m = E(\sum_{i=1}^{n} X_i)$. Then for any $\delta > 0$,*

$$Prob\left(\sum_{i=1}^{n} X_i \geqslant (1+\delta)\cdot m\right) \leqslant \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{m/k}.$$

(b) *Consider q balls, randomly thrown in r bins. The probability that any bin gets more than $v \cdot (q/r)$ balls is less than $(e/v)^{vq/r} \cdot r$.*

Proof (Fact 9 (b)).

$$\begin{aligned}
&Prob(\text{at least } v \cdot (q/r) \text{ balls are in any bin}) \\
&\leqslant r \cdot Prob(\text{at least } v \cdot (q/r) \text{ balls are in bin } 1) \\
&\leqslant r \cdot \binom{q}{v \cdot \frac{q}{r}} \cdot \left(\frac{1}{r}\right)^{v \cdot \frac{q}{r}} \leqslant r \cdot \left(\frac{q \cdot e}{v \cdot \frac{q}{r}}\right)^{v \cdot \frac{q}{r}} \cdot \left(\frac{1}{r}\right)^{v \cdot \frac{q}{r}} \leqslant \left(\frac{e}{v}\right)^{vq/r} \cdot r. \qquad \square
\end{aligned}$$

Consider the nodes $v_1, \ldots, v_l$. If they are on a level of T with at most p nodes, then $v_1, \ldots, v_l$ are placed on different processors and therefore the assertion of Lemma 8 holds.

Let T_a (tree of active nodes) denote the subgraph of T which is spanned by the nodes $v_1, \ldots, v_l$ and its (direct as well as indirect) predecessors. Let h_a be the height of T_a. With L_i we denote the set of nodes of level i of T_a, $l_i := |L_i|$.

Let R be a subset of nodes of level i of T_a that are placed on processor P.

- The set of children of nodes of R in T_a is called $Ch(P,i)$. Note, that $Ch(P,i) \leqslant |R| \cdot d$.
- The set of d processors which P chooses in the ith iteration as neighbour processors is called $Succ(P,i)$.
- The set of processors which choose processor P as a neighbour processor in the ith iteration is called $Pred(P,i)$.
- We say: In the ith iteration of the algorithm Build-Up, processor P distributes the nodes of $Ch(P,i-1)$ among the neighbour processors $Succ(P,i)$.

Claim 10. *For arbitrary $c > 0$ there is a $c' > 0$ such that the following holds: Assume that algorithm Build-Up(z), $z \geqslant c' \cdot d \cdot \log^2 n$, has placed $O(\log n + \frac{l_{i-1}}{p})$ nodes of L_{i-1}, $0 < i \leqslant h_a$ on each processor. Then for each processor P the following holds with probability $1 - n^{-c}$: $\sum_{P' \in Pred(P,i)} |Ch(P', i-1)| \leqslant 3 \cdot \lceil l_i/p \rceil \cdot z$.*

Proof (*Claim* 10). Fix processor P. Let $X_1, \ldots, X_p$ be the random variables with $X_j = |Ch(P_j, i-1)|$ if $P_j \in Pred(P,i)$ and $X_j = 0$ otherwise. We have:

- $\sum_{j=1}^{p} |Ch(P_j, i-1)| = l_i$,
- $Prob(P_j \in Pred(P,i)) = z/p$,
- $E(\sum_{j=1}^{p} X_j) = \frac{l_i}{p} z$,
- $X_j \in [0, \ldots, k]$ where $k = O((\log n + \frac{l_{i-1}}{p})d)$,
- $X_1, \ldots, X_p$ are independent.

Then, for $u > 0$, the following holds with Fact 9(a):

$$Prob\left(\sum_{P'\in Pred(P,i)}|Ch(P',i-1)|\geqslant u\frac{l_i\cdot z}{p}\right)$$
$$=Prob\left(\sum_{j=1}^{p}X_j\geqslant u\frac{l_i\cdot z}{p}\right)\leqslant\left(\frac{e^{u-1}}{u^u}\right)^{\frac{l_i}{p}z/k}.$$

With $u=\max\{3,p/l_i\}$ and $z\geqslant c'\cdot d\cdot\log^2 n$, c' sufficiently large, Claim 10 follows. □

Claim 11. *For arbitrary $c>0$ there is a $c'>0$ such that the following holds: Assume that algorithm Build-Up(z), $z\geqslant c'\cdot\max\{d\cdot t\cdot\log n, d\cdot\log^2 n\}$, has places the nodes on the processors such that for each processor P and for an i, $1\leqslant i\leqslant h_a$,*

$$\sum_{P'\in Pred(P,i)}|Ch(P',i-1)|=O\left(\left\lceil\frac{l_i}{p}\right\rceil z\right).$$

Then the following holds with probability $1-n^{-c}$: $O(\log n+(l_i/p))$ nodes of level i of T_a will be placed on each processor.

Proof (*Claim* 11). Fix a processor P. We know from the assumption that $O(\lceil l_i/p\rceil z)$ many nodes from processors of $Pred(P,i)$ will be distributed among their neighbour processors. For each of these nodes the probability that it will be placed on processor P is $1/z$. Thus for processor P the situation is like being one of z bins in which $O(\lceil l_i/p\rceil z)$ balls (= nodes) are randomly thrown. From Fact 9 (b) we conclude that $O(\log n+l_i/p)$ of these nodes will be placed on processor P with high probability. This proves Claim 11. □

Now we can prove inductively that the above two claims hold for the level h_a-1 of T_a.

Claim 12. *For arbitrary $c>0$ there is a $c'>0$ such that for level h_a-1 the following holds: $O(\log n+(l_{h_a-1}/p))$ nodes will be placed on each processor by algorithm Build-Up(z), $z\geqslant c'\cdot\max\{d\cdot t\cdot\log n, d\cdot\log^2 n\}$, with probability $1-n^{-c}$.*

Proof (*Claim* 12). In order to prove the claim we need to multiply the conditional probabilities of Claims 10 and 11 for each level of T_a. Thus the probability can be bounded from above by $(1-n^c)^h\geqslant 1-h/n^{-c}$. As $m\leqslant 2^n$ then we have $h\leqslant n$. Since $c>0$ can be chosen arbitrarily, Claim 12 is proven. □

The leaves of T_a have weights $g_1,\ldots,g_l$. If R is a set of weighted nodes, we denote the total weight of nodes in R by $W(R)$. Let $w:=\sum_{j=1}^{l}g_j$. With the following two claims we have proven Lemma 8(a).

Claim 13. *For arbitrary $c>0$ there is a $c'>0$ such that the following holds: Assume that $O((l_{h_a-1}/p)+\log n)$ nodes of level h_a-1 are placed on the same processor by*

algorithm Build-Up(z), $z \geqslant c' \cdot \max\{d \cdot t \cdot \log n, d \cdot \log^2 n\}$. *Then for each processor* P *the following holds with probability* $1 - n^{-c}$:

$$\sum_{P' \in Pred(P,h_a)} W(Ch(P', h_a - 1)) = (1 + o(1))wz/p.$$

Proof (*Claim* 13). By assumption, $O((l_{h_a-1}/p) + \log n)$ nodes of L_{h_a-1} are placed on each processor P_j, and we conclude:

$$W(Ch(P_j, h_a - 1)) = O\left(\left(\frac{l_{h_a-1}}{p} + \log n\right) \cdot d \cdot t\right).$$

Fix a processor P. Let $X_1, \ldots, X_p$ be the random variables with $X_j = W(Ch(P_j, h_a - 1))$ if $P_j \in Pred(P, h_a)$ and $X_j = 0$ otherwise. We have
- $Prob(P_j \in Pred(P, h_a)) = z/p$,
- $E(\sum_{j=1}^{p} X_j) = \frac{w}{p} z$,
- $X_j \in [0, \ldots, k]$ where $k = O(((l_{h_a-1}/p) + \log n) \cdot d \cdot t)$,
- $X_1, \ldots, X_p$ are independent.

With Fact 9 (a) for $u > 1$ the following holds:

$$Prob\left(\sum_{j=1}^{p} X_j \geqslant u\frac{w}{p}z\right) \leqslant \left(\frac{e^{u-1}}{u^u}\right)^{\frac{w}{p}z/k}$$

Thus, if we choose $u = 1 + (1/\log n)$ and $z > c' \cdot \max\{d \cdot t \cdot \log n, d \cdot \log^2 n\}$ with c' big enough, the probability can be bounded by n^{-c} for arbitrary $c > 0$. Thus the claim follows. □

Claim 14. *Assume that for each processor* P *the assertion of Claim* 13 *holds, i.e.*

$$\sum_{P' \in Pred(P,b)} W(Ch(P', h_a - 1)) = (1 + o(1))wz/p.$$

Then for each processor P *the following holds with probability* $1 - n^{-c}$: *The total weight of nodes of* T_a *which are placed on processor* P *is at most* $(1 + o(1))(n/p)$.

Proof (*Claim* 14). Fix a processor P. By assumption the processors of $Pred(P, h_a)$ distribute the set of nodes $\bigcup_{P' \in Pred(P,h_a)}(Ch(P', h_a - 1))$ with total weight $(1 + o(1))(w/p)z$ among their neighbour processors. The number of these nodes is $l \leqslant n$, each of these nodes has weight at most t, g_j is the weight of such a node. Let $X_1, \ldots, X_l$ be the random variables with $X_j = g_j$ if the jth node of $\bigcup_{P' \in Pred(P,h_a)}(Ch(P', h_a - 1))$ is placed in processor P and $X_j = 0$ otherwise. We have:
- $Prob(j\text{th node is placed on } P) = 1/z$,
- $E(\sum_{j=1}^{p} X_j) = (1 + o(1))w/p$,
- $X_j \in [0, \ldots, t]$,
- $X_1, \ldots, X_l$ are independent.

With Fact 9 (a) the following holds for $u>1$:

$$Prob\left(\sum_{j=1}^{l} X_j \geqslant u\cdot(1+o(1))\frac{n}{p}\right) = \left(\frac{e^{u-1}}{u^u}\right)^{w\cdot(1+o(1))/pt}.$$

If we choose $u=1+1/\log n$ we can bound the probability from above by n^{-c} for arbitrary $c>0$. Note, that this only holds if w is big enough. If it is not we can easily introduce dummy weights. This proves Claim 14. □

From Claims 12–14 we can easily conclude Lemma 8 (a) since we took the assertion of Claim 12 as assumption of Claim 13 and the assertion of Claim 13 as assumption of Claim 14. Note that we proved conditional probabilities. Thus we have to multiply these probabilities to get the required bound and we have proved Lemma 8 (a). □

Proof of Lemma 8(b). We only have to show that $|Pred(P,i)|=O(z)$ for each processor P and each i, $1\leqslant i\leqslant h$. Fix processor P. Let $X_1,\ldots,X_p$ be the random variables with $X_j=1$ if $P_j\in Pred(P,i)$ and $X_j=0$ otherwise. We have

- $Prob(P_j\in Pred(P,i))=z/p$,
- $E(\sum_{j=1}^{p} X_j)=z$,
- $X_j\in[0,1]$,
- $X_1,\ldots,X_p$ are independent.

Then, for $u>0$, the following holds with Fact 9 (a):

$$Prob(|Pred(P,i)|\geqslant u\cdot z)=Prob\left(\sum_{j=1}^{p} X_j\geqslant u\cdot z\right)\leqslant\left(\frac{e^{u-1}}{u^u}\right)^z.$$

This proves Lemma 8(b). □

5.2. *Executing queries*

In this subsection we present the algorithm FewQueries that solves the Multisearch problem with $p\leqslant n<m\leqslant 2^n$. It works on the tree mapped to the processors by algorithm Build-Up(z). The algorithm performs h rounds. In the ith round we have the following input and output:

- *Input for Round i*: All jobs on level i distributed among the processors as follows:
  - Each small job at some node v on level i is stored by the processor to which v is mapped by the preprocessing. Let r_j be the total size of small jobs on level i held by processor P_j. Note that Lemma 8 guarantees that r_j is not larger than $(1+o(1))n/p$.
  - Large jobs are distributed in an ordered way among the processors such that processor P_j holds at most $(n/p)-r_j$ queries of large jobs.
- *Output for Round i*: All jobs on level $i+1$ distributed among the processors as follows:
  - Each small job at some node v on level $i+1$ is stored by the processor to which v is mapped by the preprocessing. Let r'_j be the total size of small jobs on level

$i+1$ held by processor P_j. Note that Lemma 8 guarantees that r'_j is not larger than $(1+o(1))n/p$.

- Large jobs are distributed in an ordered way among the processors such that processor P_j holds at most $(n/p)-r'_j$ queries of large jobs.

Next we give the algorithm FewQueries which in each round redistributes the queries as described above:

Algorithm FewQueries.

Let T be the search tree generated by Build-Up(z). If h is the depth of T then the algorithm makes h iterations. In the ith iteration it executes the following 7 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.)
2. Each group leader broadcasts the fetched node to the other group members.
3. 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 7 that P_i has to perform binary search for at most $(1+o(1))n/p$ queries.

4–6. These steps are the same as Steps 4–6 in Phase 1 of algorithm ManyQueries.

7. Let r'_j be the total size of small jobs on level $i+1$ that processors P_j holds, $1 \leqslant j \leqslant p$. In order to get the desired output for Round i the large jobs on level $i+1$ have to be distributed in an ordered way among the processors, such that P_j holds at most $\frac{n}{p}-r'_j$ queries of large jobs. This is a *redistribution task for jobs of size at most t* and can be performed by the algorithm *Distribute* (see Section 3).

Theorem 15. *Let* $m=O(2^n)$ *and* $n/p=\omega(\log n \cdot t)$. *Then for arbitrary* $c>0$ *there is a* $c'>0$ *such that the following holds: If the preprocessing is done by algorithm Build-Up*(z) *with* $z \geqslant c' \cdot \max\{d \cdot \log^2 n, d \cdot t \cdot \log n\}$ *the algorithm FewQueries needs*

- *communication time*

$$O\left(\frac{\log m}{\log d} \cdot g \cdot \left(B \cdot d + \frac{n}{pB} + \frac{n \cdot d}{p \cdot t} + \frac{d}{B} + \log p + z\right) + L \cdot \log p\right),$$

- *computation time*

$$(1+o(1)) \cdot \frac{n}{p} \log m + O\left(\frac{\log m}{\log d} \cdot \left(\frac{n}{p} + d + \log p + L \cdot \log p\right)\right),$$

- *space* $O(m/p)$ *per processor is needed for storing the tree and space* $O((n/p) \cdot (1+o(1)))$ *per processor is needed for the searching.*

with probability at least $1-n^{-c}$.

The following corollary can easily be concluded.

Corollary 16. *If in addition to the conditions in Theorem* 15 $d = \omega(1)$ *and* $B = O(\min\{n/p \cdot z, t/d, (n/p \cdot d)^{1/2}\})$, *then we have*

- *communication time*

$$O\left(\frac{\log m}{\log d} \cdot g \cdot \frac{n}{pB} + L \cdot \frac{\log m}{\log d} \cdot \log p\right)$$

- *computation time*

$$(1 + o(1)) \cdot \frac{n}{p} \log m + O\left(\frac{\log m}{\log d} \cdot \log p \cdot L\right).$$

If further $g = o(\log d \cdot B)$ *and* $L = O(n/p \cdot \log m / \log d)$ *then the algorithm is* 1*-optimal.*

In particular, if $d = (n/p)^{0.1}, t = (n/p)^{0.4}, B \leqslant (n/p)^{0.3}$, $L \leqslant (n/p)^{0.3}$ *and* $z = c' \cdot (n/p)^{0.7}$, *then algorithm FewQueries is* 1*-optimal with probability* $1 - n^{-c}$ *for*

- $n/p \geqslant n^{1+\varepsilon}$ *with* $\varepsilon > 0$, *if* $g = o(B \log n)$ *and*
- $n/p \geqslant \log^3 n$ *with* $\varepsilon > 0$, *if* $g = o(B \log \log n)$.

Note that if we would not have exploited blockwise communication, then in the particular cases in the corollary we would need $g = o(\log n)$ and $g = o(\log \log n)$, respectively, in order to achieve 1-optimality.

Proof of Theorem 15. In the following we analyse the time bounds for each round i:

Step 1: Each processor fetches at most $n/p \cdot t$ nodes. Mark each of these nodes with weight t. Lemma 8 (a) guarantees that Build-Up distributes these nodes such that the total weight of nodes placed on each processor is at most $(1 + o(1))(n/p)$ with probability $1 - n^{-c}$, for an arbitrary $c > 0$. Thus each processor gets $O(n/p \cdot t)$ 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 $O(n/p \cdot t)$-relation of messages of size 1, i.e., each processor sends and receives $O(n/p \cdot t)$ messages of size 1. The second superstep realizes a $O(n/p \cdot t)$-relation of messages of size d. Thus this step needs communication time $\max\{g \cdot n/p \cdot t \cdot \lceil d/B \rceil, L\}$.

Step 2: This step needs communication time $M_{br}(d)$ and computation time $C_{br}(d)$.

Step 3: At the start of each round each processor holds at most $(1 + o(1))(n/p)$ queries. This is guaranteed by Lemma 8 (a) and the execution of the routine Distribute at the end of each iteration. Therefore in Step 3 of each round every processor performs at most $(1 + o(1))n/p \log d$ comparisons.

Step 4–5: The analysis for these steps is the same as for Steps 3 and 4 of Phase 1 of algorithm ManyQueries.

Step 6: The small jobs on level $i + 1$ are sent to the processors that hold the nodes of level $i + 1$ the jobs want to visit. We distinguish two kinds of small jobs on level $i + 1$: Small jobs on level $i + 1$ that are successor jobs of large jobs on level i (we call them *A-jobs*) and small jobs on level i that are successor jobs of small jobs on level i (we call them *B-jobs*). First we analyse the cost for sending the A-jobs to the nodes they want to visit:

Each processor holds at most $n/p \cdot t$ large jobs on level i and can therefore generate at most $n \cdot d/p \cdot t$ A-jobs on level $i+1$. For the proof we mark the nodes that the A-jobs want to visit with weight t/d. From Lemma 8 (a) we know that each processor holds nodes of total weight at most $(1+o(1))(n/p)$ and it follows that each processor holds at most $(1+o(1))(n \cdot d/p \cdot t)$ of these nodes. Thus for A-jobs we have: Every processor sends at most n/p queries to at most $n \cdot d/p \cdot t$ processors and every processor receives at most $(1+o(1))(n/p)$ queries from at most $(1+o(1))(n \cdot d/p \cdot t)$ processors. It follows that for A-jobs we have communication time $O(g \cdot ((n \cdot d/p \cdot t) + (n/p \cdot B)) + L)$.

Next we analyse the communication cost for B-jobs: Each processor sends B-jobs with total weight at most $(1+o(1))(n/p)$ to at most z neighbour processors which hold the appropriate nodes. Mark these nodes with the size of the respective jobs. By Lemma 8 (a) and (b) we know that each processor receives jobs of total weight at most $(1+o(1))(n/p)$ from $O(z)$ neighbour processors with probability $1-n^{-c}$ for an arbitrary $c>0$. Each processor needs computation time $O(\frac{n}{p})$ to combine the queries to large packets. Each processor sends $O(n/p)$ queries to at most z processors and each processor receives $O(n/p)$ queries from at most $O(z)$ processors in one communication superstep. Thus the communication time for B-jobs is $O(g \cdot (n/p \cdot B + z) + L)$.

Step 7: This step needs communication time $M_{\text{dist}}(t)$ and computation time $C_{\text{dist}}(t)$.

Since the depth of T and therefore the number of iterations is $\log m / \log d$, we achieve the resource bounds stated by Theorem 15. □

6. Experiments

We have implemented the Algorithm ManyQueries for the case $n \geqslant m$ 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.

We measured the BSP* parameters: In order to do this we used random relations as communication patterns. The router reaches maximal throughput in terms of Bytes/s when we have messages of size 1 kByte. Therefore the value for parameter B is 1 kByte. The value for g is around 10 000 ops, where ops is the time for an integer arithmetic operation on the T800 Transputer. The value for L is 17 000 ops.

The experiments show, for $n/p = 6144$ and $m/p = 2048$, a speed-up of 31 with 64 processors, and for $n/p = 16384$ and $m/p = 8192$, a speed-up of 64 with 256 processors. These results are quite good if one takes into account that the sequential binary search algorithm, with which we compare our algorithm, can be implemented very efficiently.

References

[1] A. Aggarwal, A.K. Chandra, M. Snir, On communication latency in PRAM computations, in: Proc. ACM Symp. on Parallel Algorithms and Architectures, 1989, pp. 11–21.

[2] M.J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, J.-J. Tsay, Multisearch techniques for implementing data structures on a mesh-connected computer, in: Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 204–214.
[3] M.J. Atallah, A. Fabri, On the multisearching problem for hypercubes, in: Proceedings of PARLE '94 – Parallel Architectures and Languages Europe, Lecture Notes in Computer Science, 159–166, Springer-Verlag, July 4–8, 1994.
[4] R.H. Bisseling, W.F. McColl, Scientific computing on bulk synchronous parallel architectures, in: Proc. 13th IFIP World Computer Congress, vol. 1, 1994.
[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation, in: Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 1993.
[6] F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable parallel computational geometry for coarse grained multicomputers, in: Proc. ACM Conf. on Comp. Geometry (1993).
[7] A.V. Gerbessiotis, L. Valiant, Direct bulk-synchronous parallel algorithms, J. Parallel Distrib. Comput. (1994).
[8] W. Hoeffding, Probability inequalities for sums of bounded random variables, Amer. Statist. Assoc. J. (1963) 13–30.
[9] B.H.H. Juurlink, P.S. Rao, J.F. Sibeyn, Worm-hole gossiping on meshes, in: Proc. Euro–Par'96, Lecture Notes in Computer Science, vol. I, Springer, Berlin, 1996, pp. 361–369.
[10] C.P. Kruskal, L. Rudolph, M. Snir, A complexity theory of efficient parallel algorithms, in: Proc. 15th Int. Colloq. on Automata, Languages, and Programming, 1988, pp. 333–346.
[11] R.E. Ladner, M.J. Fischer, Parallel prefix computation, J. ACM 27(4) (1980) 831–838.
[12] W.F. McColl, The BSP approach to architecture independent parallel programming, Technical Report, Oxford University Computing Laboratory, 1994.
[13] R. Miller, Two approaches to architecture-independent parallel computation, D. Phil thesis, Oxford University Computing Laboratory, 1994.
[14] W. Paul, U. Vishkin, H. Wagener, Parallel dictionaries on 2–3 trees, in: Proc. 10th ICALP, 1983, pp. 597–609.
[15] A. Ranade, Maintaining dynamic ordered sets on processor networks, in: Proc. 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 127–137.
[16] J.H. Reif, S. Sen, Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications, SIAM J. Comput. 23(3) (1994) 633–651.
[17] L. Valiant, A Bridging Model for parallel Computation, Comm. ACM 33(8) (1994).