Blocking in Parallel Multisearch Problems
(Extended Abstract)

Wolfgang Dittrich *
Bosch Telecom GmbH, UC-ON/ERS
Gerberstraβe 33
71522 Backnang, Germany
Wolfgang.Dittrich@bk.bosch.de

David Hutchinson †
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
hutchins@scs.carleton.ca

Anil Maheshwari ‡
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
maheshwa@scs.carleton.ca

Abstract

External memory (EM) algorithms are designed for computational problems in which the size of the internal memory of the computer is only a small fraction of the problem size. Block-wise access to data is a central theme in the design of efficient EM algorithms. A similar requirement arises in the transmission of data between processors in certain parallel computation models such as BSP* and CGM, for which block-wise communication is a crucial issue.

We consider parallel problems where a large number of queries are to be simultaneously processed and satisfied by navigating through large data structures on parallel computers. The examples used originate as BSP* algorithms and we adapt them to the EM situation where the queries and data structure are considered to be much larger than the size of the available internal memory.

This paper presents techniques to achieve blocking for I/O as well as for communication in multisearch on the BSP* and EM-BSP* models.

In the area of EM algorithms new algorithms for multisearch in balanced trees are described. For search trees up to size \(O(n\log n)\) where \(n\) is the number of queries, we obtain work-optimal, parallel, EM multisearch algorithms whose I/O and communication time are the same, asymptotically, as the computation time. These algorithms are obtained via the simulation technique of [15].

For larger trees we describe a parallel, EM algorithm which is simultaneously c-optimal and I/O-optimal. We give a lower bound to the number of I/O operations required for filtering \(n\) queries through a binary or multiway search tree of size \(m\) when \(m \geq n^{2\epsilon}\), constant \(\epsilon > 0\).

In the domain of parallel, non-EM algorithms we describe a new I-optimal algorithm for BSP* multisearch in optimal trees which communicates in a block-wise fashion.

1 Introduction

1.1 Motivation

Accessing the main memory of a computer can be orders of magnitude faster than accessing an element of data in secondary memory such as a hard disk. This large difference is typically made less significant in practice by carefully ensuring that data on disk is accessed in a block-wise fashion. Thus the overheads of rotational delay and disk arm movement are amortized over the number of items in a disk block. Ensuring that I/O is fully blocked is therefore an important issue in reducing the running time of an algorithm. A second important issue, when more than one disk is present, is fully parallelizing disk I/O. If there are \(D\) disks present, and the disk block size is \(B\), \(DB\) data items can be transferred at the cost of a single I/O operation. These two issues are fundamental to our approach, since if I/O is not fully blocked, the running time can typically be up to a factor of \(10^3\) (the blocking factor) too high, and if parallel disks are not properly utilized, the running time can be a factor of \(D\) too high. More discussion of the traditional EM system model and related issues can be found in [32].

Algorithms designed for internal memory (e.g. RAM and PRAM models) have proven to be generally difficult to adapt to the important EM requirements of blocking and parallel disk I/O. However, a blocking requirement also arises in the transmission of data between processors for certain coarse grained parallel computation models, such as BSP* [7] and CGM (Coarse Grained Multicomputer) [16]. Blocking in communication is also important for efficiency. In this paper we examine algorithms in both domains because of this important, common issue.

In [15] a simulation technique was described which permits a large class of parallel BSP-like algorithms to be transformed to EM without losing asymptotic efficiency.

For suitable BSP-like algorithms the simulation was shown to preserve c-optimality in the conventional sense [20] (i.e. with respect to computation and communication) and simultaneously with respect to computation and I/O time. For optimality they required an algorithm's running time due to I/O to be asymptotically less than the computation time of the parallel algorithm. By comparison, the traditional cost measure of an EM algorithm has been the number of I/O operations performed, and internal computation time is ignored (see [32]). We will call an algorithm I/O-optimal if the number of I/O operations matches the lower bound for the number of I/Os required to solve the problem.

In multisearch problems [17, 5, 27] a large number of queries are simultaneously processed and satisfied by navigating through a large data structure on a parallel computer. Multisearch has been shown to be very useful for solving a...
number of data structuring problems on a parallel processor, e.g., trapezoidal decomposition, next element search, etc. on hypercubes [17], multi-planar point location, interval trees, hierarchical representations of polyhedra on meshes [5], dictionaries on 2-3 trees on a EREW-PRAM [27], maintaining binary search trees on a BSP* [7, 6], multi-point location in a class of hierarchical DAGs on a BSP [19] and a BSP* [18]. Achieving blocking for multiread is especially difficult.

Batch filtering was introduced in [21] as a sequential EM paradigm for executing a collection of simultaneous queries in search structures modeled as planar layered directed acyclic graphs (DAGs), and it is therefore an EM multiread technique. In this paper we restrict our attention to multiread in balanced binary trees and optimal trees [23, 24]. While batch filtering is I/O-optimal for sequential versions of these problems when \( m = O(n) \) where \( n \) is the number of queries and \( m \) is the size of the data structure to be queried, we consider data structures larger than \( O(n) \) and also parallel processing models of computation.

We also consider larger data structures than can be accommodated optimally by the techniques of [15]. The work of the algorithm remains an important aspect of our complexity measure for external memory algorithms, but achieving c-optimality in I/O may not be possible due to the blocking factor of the disks.

### 1.2 The Model

The BSP* model [7, 8, 6] is an extension of BSP designed to encourage blockwise communication between processors. An instance of the \( p \) processor BSP* model is characterized by the parameters \( g, b \) and \( L \). The parameter \( g \) is the time (in number of operations) the router needs to deliver a packet of size \( b \) (in number of machine words) when in continuous use, \( b \) is the minimum packet size that has to be sent in order to achieve the throughput potential of the router. \( L \) is the minimum time (in number of operations) between successive synchronization operations.

The EM-BSP* model [15] is an extension of BSP* to model secondary memories on each processor (see Figure 1). The BSP and CGM models can be extended in a similar way. We use the following additional parameters: \( M \) is the local memory size of each processor, \( D \) is the number of disk drives of each processor, \( B \) is the transfer block size of a local disk drive, and \( G \) is the ratio of local computational capacity (number of local computation operations) divided by the local I/O capacity (number of blocks of size \( B \) that can be transferred between the local disks and memory) per unit time.

There are several variations of the EM-BSP* model, including those for secondary memories on each processor and those for secondary memories on each processor. The EM-BSP* model is an extension of BSP* designed to encourage blockwise communication between processors. An instance of the \( p \) processor EM-BSP* model is characterized by the parameters \( g, b \) and \( L \). The parameter \( g \) is the time (in number of operations) the router needs to deliver a packet of size \( b \) (in number of machine words) when in continuous use, \( b \) is the minimum packet size that has to be sent in order to achieve the throughput potential of the router. \( L \) is the minimum time (in number of operations) between successive synchronization operations.

### 1.3 The Problem

The need for a model which addresses computation, communication, and I/O costs was identified as an open problem in the Position Statement, ACM Strategic Directions in Computing Research: Working Group on Storage I/O for Large-Scale Computing [12]. The EM-BSP, EM-BSP* and EM-CGM models, introduced in [15], were a step in this direction. In the current paper, we focus primarily on the EM-BSP* version of this model, which is sketched below.

Each processor can use all of its \( D \) disk drives concurrently, and transfers \( D \times B \) items from the local disks to its local memory in a single I/O operation and at cost \( G \). Only one track per disk can be accessed without posing any restriction on which track is accessed on each disk. It takes roughly the same amount of time to access and transfer one block or one word. This reflects the fact that the seek time for a record dominates the time to transmit a record, the transfer delay. A processor can store in its local memory at least one block from each local disk at the same time, i.e., \( M \geq DB \).

Like a computation on BSP*, a computation on the EM-BSP* model proceeds in a succession of supersteps. Communication and computation supersteps occur as in the BSP* model and multiple I/O-operations are permitted during a single computation superstep. The computation cost and communication cost are the same as for the BSP* model. For each local operation the RAM uniform cost measure is used. For an \( h \)-relation, i.e., a routing request where each processor sends and receives at most \( h \) messages of size \( b \), \( g \times h + L \) time units are charged per communication superstep. The \( I/O\)-cost (or \( I/O\)-time) of a computation superstep is \( t_{I/O} \) where \( t_{I/O} = \max_{j} t_{I/O,j} \) where \( t_{I/O,j} \) is the I/O-cost incurred by processor \( j \). Each I/O operation costs \( G \) time steps. For a computation superstep with at most \( t_{I/O} \) local operations on each processor, \( t_{I/O} + t_{I/O} + L \) time units are charged. We assume \( G \geq DB \) and \( g \geq b \) as reasonable lower bounds for the parameters. As in the BSP* model, it is worthwhile to send messages of size at least \( b \) and the model gives incentives to access all disks in using block transfers. For instance, a single processor EM-BSP* with \( D \) disks is capable of transferring a block of \( B \) items to or from each disk in a single I/O operation. An operation involving fewer elements incurs the same cost.

The goodness criterion used by the EM-BSP* model was adapted from the optimality criterion proposed in [20].

**Definition 1** [15] Let \( A \) be the optimal sequential algorithm on the RAM for the problem under consideration, and let \( T(A) \) be its worst case running time. Let \( c \) be a constant with \( c \geq 1 \). A c-optimal EM algorithm \( A^* \) meets the following criteria:

- The ratio \( \phi \) between the communication times of \( A^* \) and \( T(A)/p \) is \( c + o(1) \).
- The ratio \( \xi \) between the communication time of \( A^* \) and the computation time \( T(A)/p \) is \( o(1) \).
- The ratio \( \eta \) between the \( I/O\)-time of \( A^* \) and the computation time \( T(A)/p \) is \( o(1) \).

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

We say that a EM algorithm is **one-optimal** if it is c-optimal for \( c = 1 \). The constraint on \( \phi \) is another way of saying that \( A^* \) must be work optimal. The constraints on \( \xi \) and \( \eta \) ensure that the communication and I/O time do not affect the asymptotic running time, even by a constant factor.

In the current paper, we will use the terms **communication-efficient** and **I/O-efficient** to describe an algorithm for which \( \xi \) and \( \eta \), respectively, are \( O(1) \). An algorithm which is work-optimal, communication-efficient, and
I/O-efficient, therefore, is one whose running time complexity is no worse than the complexity $T(A)/p$. Constant factors are ignored. We will call an algorithm $I/O$-optimal if the number of I/O operations matches the lower bound for the number of I/Os required to solve the problem.

This definition applies equally well to EM-BSP, EM-BSP* and EM-CGM algorithms. In the interests of brevity and ease of exposition, we restrict ourselves to the EM-BSP* case. Conditions on the input size $n$ and the EM-BSP* parameters, $p$, $b$, $g$, $B$, $G$, $L$ and $M$ are specified that are sufficient for the algorithm to make sense and the bounds on $\phi$, $\xi$ and $\eta$ to hold. The restrictions on the parameters $p$, $b$, $g$, $B$, $G$, $L$ and $M$ are functions that grow with the input size $n$. This guarantees that the algorithms run efficiently on real parallel machines, from those with large parameter values, i.e., large bandwidth and large latency which require large messages to operate efficiently, to those with small network bandwidth and small latency.

### 1.3 Selected Previous Work

There have been many papers on blocking-related issues, as it is a very practical concern to obtain efficiency in practice. However, our focus is on algorithms which are provably optimal according to some stated model.

Sorting, permutation and related problems in EM have been extensively studied [1, 11, 32, 26]. I/O-optimal approaches to many computational geometry problems [21] and graph problems [10] have also been described. Data structures [3] and a number of applications [4] have been examined in this context. Some implementation work has also been done [29, 9, 13, 22] including some parallel processing EM implementations [14]. Blocking of graphs for external memory searching was studied in [25]. A recent survey appears in [30].

The classical EM two layer model is described in [32]. More complex models have been proposed as well, incorporating a hierarchy of memory layers. One such model is described in [5], and sorting for this model is studied in [31]. Such models are interesting because modern computers typically have several layers of memory which include main memory and caches as well as disks. In this paper, we restrict ourselves to a two-level I/O model, although we also consider block transfers between processors.

Blocking also arises in parallel computing models such as EM and CGM. The EM-CGM model [7] is an extension of BSP designed to encourage blockwise communication between processors. The CGM model [16] encourages the use of extremely large messages. Simulation of PRAM algorithms as a source of new EM techniques was described in [10]. Techniques for simulating BSP-like algorithms to produce efficient EM algorithms were explored in [15, 26].

### 1.4 New Results

We describe techniques for designing efficient, blocked multisearch algorithms, which are algorithms that filter a batch of requests through a layered data structure. The examples shown are multisearch in balanced trees and optimal trees. All of our multisearch algorithms involve a single pre-processing phase, after which multiple search phases can be performed.

- New algorithms for EM multisearch in balanced trees are described. For search trees up to size $O(n \log n)$ where $n$ is the number of queries, we obtain EM multisearch algorithms which are work-optimal, I/O-efficient, and in the case of multiple processors, communication-efficient. These algorithms are obtained via the simulation technique of [15].
- For larger trees we describe an EM algorithm which is simultaneously work-optimal and and I/O-optimal. When multiple processors are present, the algorithm is also communication-efficient.
- We give a lower bound for the number of I/O operations required for filtering $n$ queries through a binary or multiway search tree of size $m$ when $m \geq n^{2^{\varepsilon}}$, constant $\varepsilon > 0$.
- In the domain of parallel, non-EM algorithms this paper describes a new 1-optimal algorithm for EM multisearch in optimal trees. Like all BSP* algorithms, it communicates in a block-wise manner between processors.

Our EM multisearch techniques can be viewed as extending the EM paradigm of batch filtering[21] for balanced trees from the $m = O(n)$ case to include also $m = o(n)$, and from the RAM model to BSP-like parallel computation models. We will refer to the extended technique as multis-filtering, while recognizing that it may itself invite extensions to other data structures such as hierarchical DAGs.

While our results are stated in terms of the BSP* model, many of the techniques apply equally well to the BSP and CGM models. Due to lack of space we omit proofs in most cases.

### 2 EM Multisearch in Balanced Binary Trees

As input for the multisearch problem, we consider a set of $n$ queries and a balanced binary search tree $T$ of size $m$. The search tree is distributed among the processors and stored on their local disk drives.

To simplify the description in the following, we assume without loss of generality that $m = 2^k - 1$ and $n = 2^k - 1$ for suitable $i$ and $j$. All logarithms are to the base 2 unless otherwise indicated. For simplicity we omit ceilings/floors around log terms.

#### 2.1 Building a Supernode Graph

From $T$ an equivalent graph $\hat{T}$ is built. $T$ is viewed as a collection of mutually exclusive substructures, called supernodes. The supernodes of $T$ are the vertices of $T$, and two vertices of $\hat{T}$ are connected by an edge if the corresponding supernodes are connected by one or more edges in $T$. The vertices of $\hat{T}$ have out-degree $O(d)$.

For large balanced binary trees, we partition $T$ into two parts which we refer to as the upper part and the lower part, respectively. The upper part contains nodes of all levels $L_i$ for all $0 \leq i \leq \log \frac{m}{2}$ and the lower part contains the nodes of levels $L_i$ for $\log \frac{m}{2} < i < \log m$. The root is on level $L_0$ and at height $\log m$, while the leaves are on level $L_{\log m}$, and at height 0. Thus, the upper part contains $O(n)$ nodes and the lower part contains $O(m - n)$ nodes.

The supernode graph $\hat{T}$ is formed as follows: The edges between the nodes belonging to the $(j\log d - 1)^{th}$ and $(j\log d)^{th}$ levels of $T$ are cut, for $0 < j < \log m$. Thus $\hat{T}$ is divided into subtrees of height $\leq \log d - 1$. Each subtree which correspond to a vertex of the new graph $\hat{T}$. For each pair of vertices $t$, $t'$ the edge $(t, t')$ is created if the subtrees which correspond to $t$ and $t'$ are connected by an edge in $T$. The degree of a vertex of $\hat{T}$ is $O(d)$. $\hat{T}$ has $O(m|d|)$ vertices and height $h = \log m$. Bounds on $d$ are given in Theorems 1-3, depending on the circumstances.
2.2 Mapping Supernodes to Processors

Our approach is constrained by the requirement of the BSP* model that communication between processors be done in a blockwise manner to achieve valid results in our calculation of communication complexity. The details and analysis of the techniques we use for managing the computational and communication complexity were first presented in [7] and later refined in [18]. For completeness, we sketch the approach here.

The supernodes are assigned to processors using a random mapping. Because we require inter-processor communication to be done in a blockwise manner, however, we place a limit on the fanout from the processor storing a supernode to the processors storing the children of that supernode. The child nodes of a supernode are stored on processors chosen randomly from a restricted set of processors. This mapping is referred to as the plain block mapping with parameter z (see [7] for details). The advantage of the mapping is that even small jobs can be sent between processors in a blockwise fashion, since more than one of them can share a communication block.

For supernode graphs of many levels, this approach is not sufficient, however. To handle this situation, a layered block mapping with parameters z and h_d was introduced in [18]. The levels of T are partitioned into strips of height h_d, where the supernodes of each strip are distributed as follows:

1. The nodes of the first level of each strip are distributed among all processors by choosing at random a processor for every supernode.
2. The other levels are distributed by means of the plain block mapping with parameter z.

For a balanced binary tree, we assign the supernodes of T to the processors using the layered block mapping with parameters h_d = log(n/(p*B)) and z = O(dlog^1.0 n). We do not allow redundancy here, because we assume the search tree to be very large.

2.3 Dynamic Work Balancing

The requests to be processed are either balanced among the processors once in terms of their numbers or balanced dynamically considering the actual work performed. The former is applicable if the supernodes are asymptotically all of the same degree and the latter if the degree of a supernode varies widely. A number of techniques for ensuring work balance were presented in [7]. Due to lack of space we omit the details here.

We say a query q visits a node t of T if t belongs to the search path of q in T. We refer to a subset of the queries which visit node t as a job at supernode t which is denoted by J(t). We distinguish between large and small jobs, where a small job is of size at most (n/p^b)^a and a large job^2 is larger than (n/p^b)^a for a suitable constant a, 0 < a < 1. Let P(t) denote the processor that stores the node t of T after the preprocessing. Given k jobs J_1, J_2, . . . , J_k with \sum_{i=1}^{k} |J_i| = n, we say the jobs are distributed in compact format among a group of p processors P, if for each job J_i, 1 \leq i \leq k, all keys of J_i are either in the possession of a single processor or they are spread among group of processors (a subset of P) such that each processor is in possession of at most (1 + o(1)) \frac{p}{v} keys.

Large jobs are those for which it is efficient to move a supernode from a remote processor to the processor containing the large job. Small jobs are sent to the processor storing the required supernode. Within a strip (see the layered block mapping of Section 2.2) this is performed directly.

On the first level of a strip the small jobs are distributed by Algorithm 5 RouteSmallJobs, see Section 6.

We present two different kinds of results in the following. The first results (see Section 2.4) are applicable to smaller trees, and are obtained by applying the general BSP*-simulation technique of [15] to the BSP* algorithm Tree-Search presented in [7]. Refined versions of the BSP* multisearch algorithms presented in [7] can be found in [18], and so we base the current work on the latter descriptions. Section 2.5 describes a new multisearch algorithm for large trees, tailored to the EM-BSP*.

2.4 EM-BSP* Multisearch on a Small Tree

For completeness, we paraphrase the following result from [18].

Theorem 1 [18] Let \hat{T} be a d-ary balanced search tree of size m, distributed among v processors according to the layered block mapping. Let Q be a set of q queries distributed evenly among the processors. Let \alpha, \epsilon > 0 and \epsilon \geq 1 be constants, and b is the BSP* block size parameter. Let \frac{n}{v} = O(blog^{1+\epsilon} n), d = (\frac{n}{v})^{a}, and \alpha \leq \frac{n}{v^{2+\epsilon}}. Multisearch of Q against \hat{T} using BSP* algorithm Tree-Search (below) uses O(n\alpha q) memory per processor and (1 + o(1)) \frac{v}{\alpha} log m + O(q \cdot \frac{v}{\alpha} log_{d} m + L \cdot \log_{v} log_{d} m) running time with probability 1 - n^{-c}. In particular, Tree-Search is 1-\epsilon optimal for L = o(\frac{\log^{2} q}{\epsilon \alpha}) and \epsilon = o(blog d).

Algorithm 1 Tree-Search(\hat{T}, Q):

Input: the search tree \hat{T} with degree d distributed by the block mapping with parameters h_d and z. \frac{q}{v} queries of Q are in the possession of each processor.

Output: The target node for each query. Remark: The input for the first round is a large job located at the root node of \hat{T} containing all queries. t refers to a node of \hat{T}.

For i = height(\hat{T}) down to 0 do

Comparison Phase:
1. Fetch the node of \hat{T} for each large job. The small jobs are already located at the processor which stores the node of \hat{T} which they require.
2. Execute all jobs of level i: Each processor determines the successor node for each query in its possession. Queries visiting a common node on the next level will belong to the same successor job. Each job can break up into at most d pieces. P_i performs binary search for at most (1 + o(1)) \frac{v}{\alpha} queries.

Reorganization Phase:
3. Condense the successor jobs with a common destination node of \hat{T} and send each small successor job J(t) to processor P(t). Each processor now contains up to (2 + o(1)) \frac{q}{v} queries, (1 + o(1)) \frac{q}{v} queries belonging to small successor jobs and \frac{q}{v} belonging to large successor jobs.
4. Perform a global rebalancing of the large successor jobs among the processors. The number of comparisons is used as a criterion. For supernodes of identical degree, it is sufficient to balance the number of queries over the v processors.

— End of Algorithm —
When the search tree is small, i.e., about the size of the query set, we can obtain a work-optimal and I/O efficient algorithm by applying the simulation techniques of [15] to the BSP* multisearch algorithm of Theorem 1. Theorem 3 and Corollary 1 give the resulting running time performance for multisearch on multiple processors, and a single processor, respectively.

Theorem 2 is adapted from [18]. An earlier version appeared in [15].

**Theorem 2** [15] A v-processor BSP* algorithm $A$ with running time $\tau = g(b) \log b + \lambda$, and local memory of size $\mu$ can be simulated on a p-processor EM-BSP* with internal memory $M = O(\mu + BD)$ in time $\frac{n}{p} \tau = O\left(\frac{n}{p} \mu \log \frac{n}{p} \right) + O\left(\frac{n}{p} \lambda \log \frac{n}{p} \right) + O\left(\frac{n}{p} \frac{\log p}{\log M} + \frac{L}{\log M} \right)$ with probability $1 - \exp\left(-O\left(\log (1 - \log \frac{n}{p})\right)\right)$ for $p = O(\frac{n}{p} \mu \log \frac{n}{p})$ and $\frac{m}{p} \log \frac{n}{p} \geq \delta^2$, where $\delta, \lambda$ are positive constants and $k \geq \log(M/B)$.

If $A$ is c-optimal on the BSP* for the conditions $g \leq g(n)$, $b \leq b(n)$, $L \leq L(n)$ and $v \leq v(n)$, then the simulation results in an EM-BSP* algorithm $A'$ which is w.h.p. work-optimal, communication and I/O efficient for the conditions $\tau = O(\mu)$, $\lambda \log \frac{n}{p} = O\left(\frac{n}{p} \log \frac{n}{p} \right)$, $g \leq g(n)$, $b \leq b(n)$, $G = O\left(\frac{n}{p} \frac{\log p}{\log M} \right)$ and $L \leq L(n)$. If $\tau = O(\mu)$ instead, algorithm $A'$ is w.h.p. c-optimal and I/O efficient for the same conditions.

Combining Theorems 1 and 2, we obtain the following result:

**Theorem 3** Let $\alpha, \beta, \psi, \gamma > 0$ be constants. Given a $d$-ary tree $T$ of size $m \leq n^{(1)}$ distributed by a suitable preprocessing. Then, if $\psi = (n^{1-\beta})^{1/2}$, $\psi \leq \frac{1}{4}$, $D = O(n^{3-\gamma} (\log n)^3)$, $B = b = O(n^{1-\alpha})$, and $1 > \alpha > \beta > \gamma > 0$, the following holds w.h.p.:

a) multisearch for $n$ hub-queries against $T$ can be done in time $1 + O\left(\frac{n^{\alpha \log n}}{pM} + O\left(\frac{n}{p} \log \frac{n}{p} + g \cdot \frac{n}{p} + G \cdot \frac{m}{p} \log \frac{n}{p} \right) + L \cdot \log(M/B)\right)$ on a p-processor EM-BSP*, $p = n^3$, with $D = O(n^{1-\gamma} (\log n)^3)$.

b) internal memory of size $O\left(\frac{n^{1-\alpha \log n}}{pM} + BD\right)$ per processor and external memory of size $O(\frac{n}{p})$ per disk.

In particular, for $m = O(n \log n)$, $G = O(B D \frac{n^{1-\alpha \log n}}{pM})$, $g = O(b \log n)$ and $L = O\left(\frac{n^{1-\alpha \log n}}{p \log n} \log \frac{M}{BD}\right)$ the algorithm is work-optimal, communication and I/O efficient.

**Proof.** Using the multisearch algorithm Tree-Search for the case $m = \Omega(Bp \log^2 n)$, $c = 1$ the corollary can be concluded from a simulation of Tree-Search by means of Theorem 2 with $v = n^{1/4}$, $k = \log n$, and $\log d = O(\log n)$. $\square$

The overhead of the simulation with regard to work is too small to influence the asymptotic running time. Like the single processor external memory multisearch, the multi-processor EM-BSP* multisearch is work-optimal for sufficiently small trees, i.e., $m = O(n \log n)$. Compared to Tree-Search, the communication time scales according to the number of EM-BSP* processors, i.e., as good as possible and is optimal for trees of size $n^{O(1)}$.

For a single processing on multiple processors, we get a special case of Theorem 3 by substituting the values $b = B$, $n = \Omega(Bp \log^2 n)$ and $p = 1$.

**Corollary 1** Let $\psi, \alpha, \epsilon > 0$ be constants. Let $T$ be a $d$-ary search tree of size $m = n^{O(1)}$ distributed by the layered block mapping. Then, if $b = (n^{1-\epsilon})/B$, $\psi \leq \frac{1}{4}$, $D = O(n^{1-\epsilon} (\log(M/B))$, $B = O(n^{1-\epsilon})$, and $\alpha > \epsilon$, the following holds w.h.p.:

a) multissearch for $n$ queries against $T$ can be done in time $n \log n + O(n + G \frac{m}{\log m})$ on a single processor EM-BSP* with $D$ disks,

b) internal memory of size $M = O(\frac{n}{m})$ and external memory of size $O(\frac{n}{m}).$

In particular, the multisearch algorithm is work-optimal and I/O-efficient for $m = O(n \log n)$ and $G = O(BD \frac{n^{1-\epsilon \log n}}{pM}).$

The internal memory required by the EM-BSP* is of size $O(m \log m)$, i.e., it is a small fraction of the tree size. This permits the solution of very large problems, much larger than without external memory. The amount of external memory used per disk is $O(m/D)$, which is optimal. Due to the simulation technique the disk I/O is fully blocked and all available disks are used in parallel. The overhead of the simulation is small and does not influence the asymptotic running time of the EM-BSP* algorithm. The conditions posed on the blocksize $b$ and number of disks $D$ are functions of $n$. Since $n$ is usually very large these conditions can be fulfilled easily for typical values of $b$ and $D$. For $m = O(n \log n)$, the I/O-time is in Big-Oh of the computation time, so the algorithm is I/O-efficient and work-optimal.

### 2.5 EM-BSP* Multisearch on a Large Tree

For trees of size $\omega(n \log n)$ the I/O time of the above algorithms becomes the dominating term of their running time complexity. I/O-efficiency is not possible for large ratios $m/n$. Therefore, we adapt algorithm Tree-Search to the EM situation.

**EM-Tree-Search** depends on a preprocessing phase, which we only sketch here. In the preprocessing, a search tree $T$ with node degree $d$ is constructed from a binary tree $T$ and is distributed using the layered block mapping among the processors. Within a processor each supernode is distributed among the disk drives as described in Section 2.5.1. In the following, we focus on the running time of the search phase. The tree is much too large to be stored completely in the internal memory. We restrict our discussion to the case where each processor has enough internal memory to keep its share of the queries completely in its internal memory. However, with straightforward changes, larger sets of queries can be accommodated in an I/O optimal manner. We omit a detailed description of this case due to lack of space.

We will need the following definitions. A collection of records is in **block format** if its records are grouped into blocks of size $B$. A collection of records, stored on $D$ disks, is in **standard consecutive format** if (i) the records are in block format, and (ii) the number of records in each block differs by at most one, and (iii) on each disk, the blocks are stored in consecutive tracks. An **EM-pointer** contains the disk and track number of a block. A **global EM-pointer** is an EM-pointer plus the processor id on which the disk block is stored. When we refer to a pointer to a disk block or supernode we mean either an EM-pointer or a global EM-pointer. The meaning should be clear from the context.
2.5.1 Allocation of blocks to disks

Efficient I/O techniques are required on each processor to control the complexity of locally processing a collection of requests against a portion of the data structure. Each supernode of $\hat{T}$ is a binary search tree of height $O(\log d)$ containing nodes of $T$. Supernodes are split into $\log_d d$ layers for disk storage. Since the supernodes are complete binary trees, each layer consists of binary trees of height $\log B$, each one disk block in size. We have different strategies for the upper and lower parts of a balanced binary tree. The upper and lower parts were defined in Section 2.1. Supernodes are distributed among the local disk drives of a processor as follows.

Upper Part: Supernodes are distributed in standard consecutive format. For each processor, supernodes of size $O(\frac{n}{p^2})$ are stored using $O(n)$ tracks of each disk.

Lower Part: The blocks of a supernode are stored in random format, meaning that each of its blocks is assigned independently to one of the $D$ available disks.

2.5.2 I/O scheduling

The upper part of the tree is of size $O(n)$, so from a complexity point of view we can afford to read all of its supernodes completely into internal memory.\(^3\) We therefore use a batch filtering [21] approach. Each supernode of the upper part is decomposed into blocks layer by layer in breadth-first search order. The blocks are distributed across the disks in standard consecutive format. For each job $J(t)$ at processor $P(t)$ we read supernode $t$ from the disks in parallel and execute $J(t)$ by passing each of its queries through $t$ to determine its successor node $t'$ of $\hat{T}$.

We now present algorithm $EM$-$Tree$-$Search$ for performing multisearch on the lower part of large trees. Algorithm $EM$-$Tree$-$Search$ is very similar to algorithm $Tree$-$Search$ except that it performs the I/O operations required to bring the necessary parts of supernodes into internal memory in order to perform multisearch\(^4\).

**Algorithm 2 EM-$Tree$-$Search(\hat{T}, Q)$**

Input: The search tree $\hat{T}$ with degree $d$ distributed by the block mapping with parameters $h, d$ and $n$. All queries of $Q$ are in the possession of each processor. The queries for the first round form a set $\{J(t)\}$ is a node of the first layer of the lower part of $T$ of large and small jobs. For small jobs, $J(t)$ will be located at $P(t)$, the processor which stores $t$. For a large job, $J(t)$ may be located at a different processor than the one which stores $t$. See [18] for more details.

Output: The target node for each query.

For $i = \log_d n$ downto 0 do

Comparison Phase:

1. For each large job $J(t)$, fetch $t$ from the disks of the (possibly) remote processor $P(t)$ using algorithm Schedule-Read-Requests. Each small job is already located at the processor which stores the node of $\hat{T}$ which requires it.

2. Execute all jobs of level $i$: For each large job $J(t)$, supernode $t$ is already in internal memory of the processor containing $J(t)$. For the small jobs, each processor reads only the blocks of $t$ required to process the queries of $J(t)$ from its local disks using Schedule-Read-Requests.

Reorganization Phase:

3. As in algorithm Tree-Search

4. As in algorithm Tree-Search

--- End of Algorithm ---

For the lower part, we fetch the blocks of supernodes using Algorithm 3 (see below). Recall that a supernode is partitioned into $\log_d d$ layers and $\lceil \frac{d}{d} \rceil$ blocks of size $B$. These blocks are distributed in random format among the disks by a preprocessing phase.

Assume we are given $\ell$ sets $R_1, R_2, \ldots, R_\ell$, where $|R_i| \leq |R_{i+1}|$ for all $i$. Each set $R_i$, $1 \leq i \leq \ell$ contains requests for blocks, where answering a request means to read the disk which contains the specified block from a disk into internal memory. For each request, we know that the disk to be accessed is randomly chosen out of the $D$ disks. We only know $R_1$ to begin with and do not know any $R_i$ for $i > 1$ at the beginning. At any stage, after all blocks of $R_i$ are transferred into internal memory it is easy to generate the requests $R_{i+1}$ for blocks of the $(i+1)^{th}$ layer, because the blocks contain EM-pointers to the blocks of the next level. Consider the following algorithm for one EM-BSP* processor with $D$ disks.

**Algorithm 3 Schedule Read Requests**

Input: Given $R_1, R_2, \ldots, R_\ell$ containing requests, where $|R_i| \leq |R_{i+1}|$ for all $i$.

Output: All requests have been answered.

Remark: For disk $i$, $1 \leq i \leq D$, we maintain a FIFO queue $Q_i$ containing unanswered requests for disk $i$.

1. Insert all requests from $R_1$ into the corresponding queues.

2. For $i = 2$ to $\ell$ do:
   i. Repeat until all queues are empty:
      - Remove one request from each of the $D$ queues and implement them in one parallel read operation. If a queue is empty, do nothing for that disk.
   ii. Insert all requests from $R_i$ into the corresponding queues.

--- End of Algorithm ---

**Lemma 1** Algorithm Schedule Read Requests can answer all of the requests in time $O(N + \frac{n}{\alpha})$ with probability $1 - \exp(-\Omega(\frac{|R_i|}{\log^{\frac{|R_i|}{\log D}}}))$, for $N = \sum_{i=1}^\ell |R_i|$ and $|R_i| \geq D \log D$.

To fetch a supernode we transfer all its blocks into internal memory using Algorithm 3, where $R_i$ contains the blocks of the $i^{th}$ layer. We need $\log_d d$ iterations in total.

We have to fetch $N = \frac{n}{\alpha} \lceil \frac{d}{d} \rceil$ blocks, which results in the following theorem.

**Theorem 4** Let $\alpha, \epsilon > 0$ and $\epsilon > 0$ be constants. Given $n$ queries and a $d$-ary tree $\hat{T}$ of size $m$ distributed among $p$ processors by a preprocessing step. Then, if $n = \Omega(Dp\log^{\frac{1+\epsilon}{\epsilon}} n)$, $B < d = (\frac{d}{d})^\alpha$, $\alpha \leq \frac{1}{2}\frac{\log D}{\log n}$, the following holds with probability $1 - 1/n^2$:
a) Multisearch for \( n \) queries against \( T \) can be done on an EM-BSP* with \( p \) processors and \( D \) disks, in time \((1 + o(1)) \frac{n}{p} \log m + O \left( \frac{\log \log m}{\log B} + \right. \) + \left. G \cdot \left( \frac{\log m}{\log B} + \frac{\log \log m}{\log B} \right) \right) + L \cdot \left( \frac{\log p}{\log \left( n\log \frac{m}{(pD)} \right)} \right) \).

b) using \( O \left( \frac{n}{p} \right) \) internal memory per processor and \( O \left( \frac{n}{pD} \right) \) external memory per disk.

**Proof Sketch.** Algorithm EM-Tree-Search is a variant of algorithm Tree-Search hence it inherits its properties regarding communication and computation time and number of supersteps. Algorithm EM-Tree-Search extends the notion of scalability to the number of disks \( D \). External memory space consumption is optimal, as it requires \( O(m/pD) \) memory of each of the \( pD \) disks. Unfortunately, the I/O costs dominate the overall running time for large trees, so EM-Tree-Search is not I/O-efficient for large trees. However, in Section 3 we show a lower bound that proves that the I/O-time of EM-Tree-Search is asymptotically optimal. In other words, this lower bound shows that I/O-efficiency is not feasible for large trees. In the light of this lower bound EM-Tree-Search is work-optimal, communication efficient and I/O-optimal for suitable parameter constellations.

The results obtained by the BSP* simulation are the first (parallel) work-optimal and I/O-efficient external memory algorithms for multisearch. Algorithm EM-Tree-Search is the first multisearch algorithm for a realistic parallel external memory model that is efficient for large trees in terms of work, I/O and communication.

### 3 Multisearch Lower-Bound

Our model consists of a single processor which has access to an unlimited internal and external memory. The external memory can only be accessed in blocks of size \( b \) by block transfers. During a block transfer, \( b \) words contained in a single block are transferred from the external memory into internal memory. The processor can only process data that is stored in internal memory (Also, any block which is transferred to the internal memory stays there forever). We charge only the number of block transfers. We assume balanced binary search tree on \( m \) nodes is stored in external memory and the set of \( n \) queries in internal memory at the beginning of the search. An algorithm for the multisearch problem consists of two parts: An assignment of the search tree to the blocks, known as blocking and a paging algorithm to specify which blocks are brought into the internal memory. The blocking must specify which node is stored in which block prior to the search, having no pre-knowledge of the search paths. In particular, we rule out storing multiple copies of the same node.

We begin by looking at a special paging algorithm, Special-Search, with the following properties:

i. The queries are answered in turn, i.e., the search process for query \( i \) is finished before the search process for query \( i+1 \) starts.

ii. Special-Search is lazy, i.e., it only brings a block into internal memory that services an immediately preceding page fault. No blocks are read in advance.

iii. At any point of time its decision of which block to bring into memory is based only upon the previous history of the path. The algorithm does not know the search path of any query in advance.

For the blocking algorithm, we make the natural assumptions that the binary tree is stored in the external memory in a blocked fashion, we do not allow multiple copies of nodes, and node assignment to blocks is deterministic. An adversary tries to cause as many page faults as possible and the adversary knows the tree and the blocking of the tree and constructs the search paths on-line. The adversary has different strategies depending on the active level \( l \) and the active node \( v \). The adversary ensures that all \( n-1 \) nodes of the upper part of the tree (where \( l < \log n \)) have to be transferred into internal memory by the paging algorithm and that the search paths are disjoint in the lower part. Furthermore, the adversary ensures that for each new block that is transferred into the internal memory, serving a page fault caused by an active node belonging to the lower part, at most \( \log(B+1) \) nodes will be visited. It can be shown that

**Lemma 2** Algorithm Special-Search uses \( \Omega \left( \frac{n}{pD} \right) \) block accesses to perform multisearch for \( n \) queries on a complete binary tree of height \( \log m \).

The next step is to ease the requirements the paging algorithm must fulfill. We allow any on-line paging algorithm with the following properties:

A. The blocking is deterministic and each node is assigned to exactly one block.

B. The decisions on what block to bring in, at any point of time, depend only on the previous history of the search path.

The paging algorithm can consider the queries in any order, even interleaved. It can also fetch any block in advance.

Property A is reasonable because duplicates are undesirable for large trees. While it forbids combining nodes in a random fashion into blocks, it does not rule out that deterministically assembled blocks may be randomly distributed among the disks on the EM-BSP* Property B holds for searching in any reasonable search tree. Although the following lower bound is formulated for binary search trees, it also holds for \( d \)-ary search trees because searching in \( d \)-ary search trees can easily be reduced to searching in binary search trees.

**Lemma 3** No algorithm which obeys properties A and B requires fewer block transfers for performing multisearch than algorithm Special-Search.

**Theorem 5** Let \( A \) be a multisearch algorithm for a \( p \)-processor EM-BSP* with \( D \) local disks per processor. Let \( c > 0 \) be a constant. Given \( n \) queries and a complete binary search tree \( T \) of size \( m \geq n^{2+c} \) and height \( \log m \). Assume the search tree \( T \) is distributed among the local disks such that Property A is fulfilled. Then, if \( A \) obeys Property B and uses \( O(n) \) local memory per processor its I/O-time is \( \Omega \left( \frac{n}{pD} + \frac{n \log m}{pD \log B} \right) \).

The lower bound holds for our randomized external memory algorithm EM-Tree-Search because it obeys the above properties. We use randomization only for the distribution of the nodes to the processors and the disks and not for the assignment of the nodes to the blocks.

### 4 BSP* Multisearch in Optimal Trees

In this section we consider the multisearch problem for optimal search trees [23, 24]. We describe a new 1-optimal BSP* multisearch algorithm which communicates in a block-wise fashion.
Let \( S = \{x_1 < x_2 < \ldots < x_m\} \) be an ordered set of size \( m \) and let \( \beta_j \geq 0 \) be the probability that item \( x_j \) will be accessed, for \( 1 \leq j \leq m \). A large probability indicates that the element is important and accessed frequently. It is therefore desirable for this element to be stored near the root of the tree so it can be accessed quickly.

Let \( \alpha_i \geq 0 \) be the probability that a value \( x \) is queried, where \( x_{i+1} < x < x_i \), and \( \sum_{i,j} (\beta_i + \alpha_j) = 1 \). The \((2m+1)\)-tuple \( D = (\alpha_0, \beta_1, \alpha_1, \ldots, \beta_m, \alpha_m) \) is called the access (probability) distribution. Optimal search trees exploit knowledge of the access probability distribution to speed-up the search process.

We adopt the definitions and notations of \[24\]. Let \( T \) be a search tree for set \( S \). \( T \) contains \( m \) internal nodes and \( m+1 \) leaves. The internal nodes correspond to the elements of \( S \). Hence, a successful search for a key with value \( x \) will terminate at the internal node denoted by \( x \). A leaf represents an unsuccessful search process. If a key with value \( x \) is queried, where \( x_i < x < x_{i+1} \), the search will terminate at the leaf which is denoted by \((x, x_{i+1})\).

Let \( b_i \) be the depth of node \( x_i \) and \( a_i \) be the depth of leaf \((x_i, x_{i+1})\). Consider a search tree for an element \( x \) from the universe. If \( x = x_i \) we must compare \( x \) with \( b_i + 1 \) elements in the tree. If \( x_i < x < x_{i+1} \), we compare \( x \) with \( a_i \), elements prior to reaching the leaf \((x_i, x_{i+1})\). Hence, \( P_T = \sum_{i=0}^{m-1} \beta_i (1+b_i) + \sum_{i=0}^{m} \alpha_i a_i / \) is the average number of comparisons performed during a search for a single element.

The value \( P_T \) is called the weighted path length of tree \( T \). A search tree \( T \) is called optimal if \( P_T \) is minimized.

Consider the search paths that are taken by queries which visit nodes at depth \( \ell \). For sufficiently large \( \ell \), the following lemma shows that these queries visit a large number of nodes at depth \( \ell \), rather than a few. In other words, their search paths are spread out over the whole search tree with high probability.

**Lemma 4** Let \( t \) be a node at depth \( \ell \) in an optimal search tree \( T \). Then, the sum of the access probabilities of the nodes belonging to the sub-tree rooted at \( T(t) \) is at most \( (\frac{1}{2})^{\ell - 1} \).

**Lemma 5** Let \( T \) be an optimal search tree on which \( n \) queries are to be performed, where the queries are issued from the same access probability distribution used for the construction of \( T \). A node of \( T \) at depth \( \delta \log n \) is visited by at most \( 4n^{1-\delta/2} \) queries with probability at least \( 1 - \exp(\frac{- n^{1-\delta/2}}{2} + \delta \ln n) \), where \( 0 \leq \delta \leq 1 \).

**Proof.** Let \( t \) be an arbitrary fixed node of \( T \) at depth \( h = \delta \log n \). By Lemma 4 the probability access of \( t \) and its successors is \( \leq (\frac{1}{2})^{\ell - 1} = 2n^{\delta - \ell/2} \). Let \( X_i \) be a random variable for query \( q_i \), where \( X_i = 1 \) if query \( q_i \) visits \( t \), otherwise \( X_i = 0 \). \( \Pr[X_i = 1] \leq 2n^{\delta - \ell/2} \). The queries are issued independently from the same access distribution, and so the random variables are independent. Let \( X = \sum_{i=1}^{m} X_i \), so \( E[X] = 2n^{\delta - \ell/2} \). From Chernoff bounds \( \Pr[X \geq 1, n^{1-\delta/2}] \leq \exp(-n^{\frac{1-\delta/2}{2}}) \). Since at depth \( h \) the tree \( T \) has at most \( n^{\delta \log n} \) such nodes the required probability is obtained.

We now sketch the techniques used for parallel multi-search on an optimal tree under the assumptions that \( \frac{n}{p} \geq n^{\frac{1}{2} + \epsilon}, n \leq m \leq n^{\gamma} \) for constants \( \epsilon \) and \( \gamma \), where \( 0 < 2\gamma < \epsilon < 1/3 \), and that both the queries and the search tree fit into the internal memories of the processors. For the case of a larger number of processors, refer to \[21\].

**Building supernode graph:** Let \( T \) be an optimal search tree. Cut the edges of \( T \) between all nodes at depth \( i-h_i + 1 \) and \( i-h_i + 1 + 1 \) for all integers \( i \geq 0 \), where \( h_i = \frac{n}{2^i} \log n \) and \( h_2 = \frac{n}{2^2} \log n \). This yields a partition of \( T \) into subtrees of height \( h_i \) and \( h_i - 1 \), respectively. Out of these subtrees we create a new tree \( T \) with the sub-trees as vertices. We refer to the subtrees which correspond to the vertices as supernodes.

**Mapping supernodes to the processors:** The root of \( T \) is stored on every processor. All other supernodes are distributed randomly among the processors. A processor may store more than one supernode of the same level. \( P(t) \) denotes the processor storing supernode \( t \).

**Dynamic Work Balance:** The load balancing problem is simpler than for the balanced binary tree case because we can show that the jobs visiting nodes at depth \( \ell \), \( \ell > 1 \), are small with high probability (Lemma 5).

**Algorithm 4 Algorithm OptTreeSearch:**

**Input:** Each processor holds \( n/p \) queries positioned at the root of \( T \).

**Output:** The results of all queries.

While unanswered queries remain
1. Each processor performs a search for each of its queries on the corresponding tree. Queries which are not yet finished will visit another supernode \( t \) of \( T \). They are collected in queue \( Q_t \).
2. Using \textit{Route-Small-Job} each processor sends any unanswered queries to the next level of \( T \).

--- End of Algorithm ---

**Lemma 6 Algorithm OptTreeSearch uses at most \( j \) rounds to answer \( n \) queries with probability at least \( 1 - m^{-\frac{1}{2}2^j + \frac{3}{2}} \).**

**Proof Sketch.** After round \( j \), \( j \geq 1 \), the algorithm processes nodes at depth \( \ell = \frac{n}{2^j} (j-1) \log (m/p) + \frac{1}{2} \log n + 1 \) of \( T \). By Lemma 4 we know that a node at depth \( \ell \) is visited with probability at most \( (\frac{1}{2})^{\ell - 1} \). We can therefore calculate the probability that a particular query reaches a node at depth \( \ell \). The required probability that no query reaches a node at depth \( \ell \) or larger can be obtained by modelling the problem as \( n \) Bernoulli trials.

The total number of comparisons performed in answering the \( n \) queries is the total work. We can show that the total work is nearly equally distributed among the processors. In each round either a small amount of work is done or the work is evenly distributed among the processors. For example, in round 1 after we have processed the root of \( T \), we have only small jobs, i.e., asymptotically smaller than \( O(n^{2/3}) \).

Let \( n_i \) be the total number of queries that enter round \( i \), for \( i \geq 1 \). Each supernode at depth \( \ell \) is visited by at most \( O(n^{2/3}) \) queries. Let \( W_i \) be the work required for level \( i \). Note that \( W_i \geq n_i \).

**Lemma 7 Let \( \omega_n \) be any function \( \omega_n \rightarrow \infty \) as \( n \rightarrow \infty \). Given \( n_i \) queries which enter round \( i \), \( i > 1 \), where \( W_i \) work that has to be performed in round \( i \). Then, if \( m \leq n^{\gamma} \), \( n/p \geq n^{2/3 + \epsilon} \), and \( 2\gamma < \epsilon \), there is a constant \( \eta \), \( \eta < \epsilon - \gamma \) such that the following holds with probability \( 1 - 1/n^c \) for constant \( c \geq \frac{1}{2} \).**

--- See Section 6. ---
The complexity of parallel internal memory multisearch on optimal search trees is summarized in the following theorem.

**Theorem 6** Let \( \gamma, \epsilon, \lambda > 0 \) and \( c \geq 1 \) be constants. Let \( T \) be an optimal search tree of size \( m \), which is constructed for a particular access probability distribution \( D \) and is distributed by a suitable preprocessing among the processors. Let \( Q \) be a set of \( n \) queries distributed in compact format and conforming to \( D \), where \( n \leq m \leq n^\gamma \). Then

a) algorithm OptTreeSearch is \( 1 \)-optimal, and

b) each processor requires \( O(\frac{n}{p}) \) local memory and \( O(1) \) supersteps,

with probability \( 1 - 1/n^c \) for \( \frac{n}{p} \geq n^{\frac{\gamma}{c}} \), \( L = O(\frac{n}{p}) \), \( B = O(\frac{n^{\frac{\gamma}{c}}}{\log n}) \), \( g = o(B \cdot \log n) \), and \( 0 < 2\gamma < c < \frac{\lambda}{1} \).

**Proof Sketch.** The work in each round is w.h.p. either evenly distributed or asymptotically smaller than \( \frac{n}{p} \), but overall, at least \( \frac{n}{p} \) comparisons are required. Each “leftover” which has to be routed has produced a lot of work in the previous round. The idea is to amortize the routing costs by the work performed for the leftovers. We can show that the running time of the first round dominates the running time of later rounds, and so overall, asymptotically less time is spent in communication than is spent in computation.

The sequential search time for \( n \) queries against an optimal tree depends on the access probability distribution \( D \). The running time can be \( n \log m \) for an even distribution or \( O(n) \) for a suitably biased distribution. In other words, the running time of multisearch is a function of the access probability distribution \( D \). Hence, we do not give an explicit running time for OptTreeSearch. Instead, we compare its running time to the running time of an optimal sequential algorithm. OptTreeSearch is \( 1 \)-optimal, where the communication costs are smaller than the computation time by a factor of \( \log n \). It adapts to the blocksize \( b \) up to a certain maximum blocksize which depends on the input size and the number of processors employed. An interesting property of algorithm OptTreeSearch is that asymptotically, the number of supersteps is constant even for very large trees.

5 Conclusion

We have described new parallel EM multisearch algorithms for static balanced binary trees. We also described new \( 1 \)-optimal algorithms for BSP* multisearch in static optimal trees. A common requirement in our approach to these problems is blocking of the data.

We note that the simulation techniques of [15] can be used to create efficient parallel EM algorithms from a large class of BSP-like algorithms. In the current paper, we used these results in the application of multisearch, but we also considered larger data structures than can be accommodated optimally by the techniques of [15].

We gave a lower bound to the number of I/O operations required for filtering \( n \) queries through a binary or multiway search tree of size \( m \) when \( m \geq n^{2^x} \), constant \( x > 0 \).

While batch filtering is I/O-optimal for sequential versions of these multisearch problems when \( m = O(n \log n) \) where \( n \) is the number of queries and \( m \) is the size of the data structure to be queried, we consider data structures larger than \( O(n \log n) \) and also parallel processing models of computation.

Presently, we are working on extending the techniques presented in this paper to hierarchical directed acyclic graphs and to dynamic EM data structures.

**References**


Appendix

Algorithm 5 RouteSmallJobs:

Input: Jobs of total size \( n \), where at most \( c \frac{n}{p} \) queries are destined for and originate from each processor. Further for each job \( J(t) \) its destination processor number \( P(t) \) is known.

Output: Every small job \( J(t) \) is in the possession of processor \( P(t) \).

1. Each query of every small job \( J(t) \) is marked with the processor number of \( P(t) \), i.e., the destination of the small job \( J(t) \).
2. Each processor \( P_i \) adds \( b \) dummy queries with label \( i \) to the queries in its possession.
3. The processors sort the queries and dummy queries by means of Radix-Sort\(^7\) according to their labels. The labels are in the range \([0, \ldots, p - 1]\).
4. Afterwards, the queries of small successor jobs which are destined for the same processor are distributed among the processors in compact format\(^8\) with respect to the labels. Each processor sends the queries (not the dummies) in its possession directly to their final destinations.

--- End of Algorithm ---

Result 1 Given \( n \) queries, where \( O(n/p) \) originate from and are destined for each processor. Then, if \( \frac{n}{p} \geq \log^{1+\epsilon} p \), constant \( \epsilon > 0 \), the running time of algorithm RouteSmallJobs is

\[
O\left( g \cdot \frac{n}{bp} \log p \log(n/(p\delta)) + \frac{\log^2 p}{\log^2 (n/(p\delta))} \right)
\]

---

\(^{7}\)See [18] for details of the BSP\(^*\) algorithm RadixSort.

\(^{8}\)See Section 2.3