Programming Research Group

COMMUNICATION EFFICIENT DATA STRUCTURES ON THE BSP MODEL WITH APPLICATIONS

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

PRG-TR-13-96

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Communication Efficient Data Structures on the BSP model with Applications*

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

Computing Laboratory
Oxford University
Wolfson Building
Oxford, OX1 3QD
United Kingdom

May 1996

Abstract

The implementation of data structures on distributed memory models such as the Bulk-Synchronous Parallel (BSP) model, rather than shared memory ones such as the Parallel Random Access Machine (PRAM), offers a serious challenge. In this work we undertake the architecture independent study of the computation and communication requirements of searching *ordered h-level graphs*, which include many of the standard data structures. We propose multi-way search as a general tool for the design, analysis and implementation of BSP algorithms. This technique allows elegant high-level design and analysis of algorithms, using data structures similar to those of sequential models. Applications to computational geometry and sorting are also presented. In particular, our new randomized sorting algorithm improves previously known BSP randomized sorting algorithms upon the amount of parallel slackness required to achieve optimality. Moreover, our methods are within a $1 + o(1)$ multiplicative factor of the respective sequential methods.

While our methods are fairly simple themselves, description of their performance in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables software to be written once and for all that can be migrated efficiently among a variety of parallel machines.

*The work of the first author was supported in part by EPSRC(UK) under grant GR/K16999 and the second author was supported in part by a Bodossaki Foundation Graduate Scholarship. A short abstract describing some of the results related to ordered h-level graphs appeared in EUROPAR '96, Lyons, France. The result related to randomized sorting is to appear in the 11th International Parallel Processing Symposium (IPPS '97), Geneva, Switzerland.

1 Introduction

Since the early days of sequential computing it has been apparent that at some time in the future sequential computers would be superseded by parallel ones. The main reason for the moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von Neumann model has been for sequential computing. The introduction of the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant [32, 33] comes to address these limitations of parallel computing.

One of the earliest attempts to model a parallel computer has been the Parallel Random Access Machine (PRAM) [12] which is one of the most widely studied abstract models of a parallel computer. The advantage of the PRAM model is its simplicity in capturing parallelism and abstracting away the communication requirements of parallel computing such as communication latency, block transfers, memory and network conflicts during routing, bandwidth of interconnection networks, interprocessor communication, memory management and synchronization. It is these advantages that make the PRAM easy to program and widely used for algorithm development.

The implementation of data structures on distributed memory models such as the BSP rather than shared memory ones like the PRAM offers a more serious challenge.

The problem of realizing such distributed data structures has been addressed in the past and several results have been obtained for particular network architectures, i.e., hypercubes, meshes [5, 11]. Architecture dependent features involved in these implementations, make the migration of these results to the more general setting of the BSP model non-trivial. The only previously known result on searching [7] involves a restricted class of the graphs we present in this work. In particular, in [7] searching on static balanced d-ary trees is realized optimally on a variant of the BSP model. In this work we generalize and extend this result in several ways, as described in later sections. Applications to computational geometry and sorting are also presented. In particular, our new randomized sorting algorithm improves all previously known BSP randomized sorting algorithms upon both the high-probability bounds and the amount of parallel slackness required to achieve optimality. Moreover, our methods are within a $1 + o(1)$ multiplicative factor of the respective sequential methods.

The computational model adopted is the Bulk-Synchronous Parallel (BSP) model of computation, which deals explicitly with the notion of communication and synchronization among computational threads. The BSP model was originally proposed by L.G. Valiant [32, 33] and subsequently addressed in [14, 24, 26] as a bridge between parallel software and hardware.

1.1 The BSP Model

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant [32, 33], as a unified framework for the design, analysis and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a precise way. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems (a global clock is utilized to achieve synchronization on every instruction), and (ii) fully asynchronous systems (synchronization is realized pairwise whenever communication is required). Processing units in the BSP model advance jointly through the program, with the required remote communication occurring between *supersteps*. The BSP computer as described in [14, 24, 25, 26, 32, 33, 34] consists of the following three components.

(1) A *collection of p processor/memory components* numbered $0, \ldots, p-1$. It is assumed that each processing unit consists of a sequential processor with a block of local memory.

(2) A *communication network* that can deliver messages point to point among the units.

(3) A *mechanism* for efficient *barrier synchronization* of all, or a subset, of the processing units.

The performance of any BSP computer can be characterized by the following three parameters.

(1) p - the number of processing units.

(2) g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. More formally, g is related to the time required to realize h-relations in continuous message usage in gh time steps [25].

(3) L - the minimum elapsed time, measured in basic computational operations, between successive synchronization operations.

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each processing unit is given a task to perform using data already available there locally before the start of the superstep. The task may include (i) computation steps on values held locally at the start of the superstep, (ii) message transmissions, and (iii) message receipts. A superstep can complete at any time after the first L time units. It should be noted that, although the model stresses global barrier mode synchronization, pairs of processing units may synchronize pairwise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules [14].

The time complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x+gh\}$ basic time steps, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor. Note that, if h_t is the maximum number of messages transmitted by any processor during S, and h_r is the maximum number of messages received by any processor during S, then $h = \max\{h_t, h_r\}$. Along the lines of [14], and in order to simplify the description of the performance of our algorithms, we assume that each superstep is either pure computation or pure communication. This way, the two kinds of costs are easily distinguished.

The minimum size for a superstep is bounded below by the frequency with which barrier synchronization can be realized. As it was noted in [14], any particular BSP machine may impose a lower bound on L on a twofold basis.

(1) There is a minimum time requirement for doing synchronization.

(2) The theoretical bound on g may be realizable only for large enough L, so that the communication network can achieve the stated capacity.

On the other hand, the maximum size for a superstep is a property of the application, since by definition a superstep can accommodate as much work as can be done without a remote data exchange. The actual values of L and g for any BSP machine are likely to depend on p. In particular, the parameters L and g are considered to be non-decreasing functions of the number of processors used by an algorithm.

1.2 Specifying the Performance of BSP Algorithms

We specify the performance of a BSP algorithm $\mathcal{A}$ in three parts. First, we specify a sequential algorithm $\mathcal{A}^*$ with which we are comparing it and make explicit how we charge for the basic operations of both $\mathcal{A}$ and $\mathcal{A}^*$. Second, we specify two ratios π and μ. The former, π, is the ratio between the computation time $C_{\mathcal{A}}$, of the BSP algorithm, over the time $C_{\mathcal{A}^*}$ of the comparing sequential algorithm divided by p, i.e., $\pi = pC_{\mathcal{A}}/C_{\mathcal{A}^*}$. The latter, μ, is the ratio between the parallel time $M_{\mathcal{A}}$ required by the communication supersteps of the BSP algorithm and the computation time of $\mathcal{A}^*$ divided by p, i.e., $\mu = pM_{\mathcal{A}}/C_{\mathcal{A}^*}$. When we specify μ it is necessary to be explicit about the amount of information that can be conveyed in a single message. Finally, we specify conditions on n, p, L and g that are sufficient for the algorithm to make sense and the claimed bounds on π and μ to hold. We also claim as corollaries some sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to the problem size as $n \to \infty$. The other parameters may also tend to ∞ if they are expressed in terms of n, though it is not assumed so.

In the remainder of this section we specify how to charge for local operations on the BSP model. Since the mechanisms of the BSP model are similar to those used by sequential models and we are interested in performance ratios of these two we can define operations in a higher level than machine level instructions. Throughout this work, lg and log denote logarithms to the base 2 and e respectively. A dominant term in an integer sorting BSP algorithm to be claimed later is contributed by sequential count-sort performed independently at each processor. We charge $O(n)$ time units for stable sorting n integers in the range $[0, 1, \ldots, n-1]$ by sequential count-sort algorithm [21]. We call such an algorithm COUNT_SORT. We also apply radix-sort techniques to sort n integers in the range $[1, \ldots, n^\alpha]$, for some positive constant α, within $\lceil \lg \lceil n^\alpha \rceil / \lg n \rceil$ iterations of COUNT_SORT and call such an algorithm RADIX_SORT. We also assume that there exists a sequential sub-quadratic time sorting algorithm [21], which we call SUB_QUADRATIC_SORT, that can sort $\lceil \sqrt{n} \rceil$ keys in $o(n)$ time units. For other operations, we charge $O(1)$ time for operations over an associative operator in parallel-prefix computations and $O(1)$ time for each comparison performed. We charge $n \lg n$ operations for sorting n keys sequentially. Finally, we charge $\lceil \lg n \rceil$ for performing binary search in a sorted sequence of length n. As noted in [14], had we charged $cn \lg n$ and $b \lg n$ operations for sorting and binary search respectively, for some constants $1 < b \le c$, then neither π nor μ would have increased in the analysis of our algorithms.

1.3 Related Work

Since the advent of parallel computing, considerable work has been done towards solving the problem of realizing distributed data structures and several results have been obtained for particular network architectures, i.e., hypercubes, meshes [5, 11]. Architecture dependent features involved in these implementations, make the migration of these results to the more general setting of the BSP model non-trivial. The only previously known result on searching [7] involves a restricted class of the graphs we present in this work. In particular, in [7] searching on static balanced d-ary trees is realized optimally on a variant of the BSP model. In this work we generalize and extend this result in several ways, as described in later sections.

The problem of sorting n keys on a parallel computing model [23] has also drawn considerable attention. First approaches were sorting networks introduced by Batcher [6]. Since then, a wealth of literature concerning parallel sorting has been published [22, 23]. The first sorting network to achieve, however, the optimal $O(\lg n)$ time bound to sort n keys is the AKS [2] sorting network of width $O(n)$. Subsequently, Reif and Valiant [30] presented the first randomized sorting algorithm that runs on a fixed-connection network and achieves the same time bound as the AKS algorithm; the constant multipliers in the randomized algorithm are, however, considerably smaller. Cole [10] was the first to present optimal $O(\lg n)$ time sorting algorithms for n-processor CREW and EREW PRAMs. These methods fail, however, to deliver optimal performance when implemented on the BSP model of computation.

Previously known results on BSP sorting include deterministic [1, 15] and randomized [14] algorithms. Our sorting algorithm improves upon all these approaches. In [1], it is shown that for all values of n and p such that $p = O(n^{1-\epsilon})$, for any constant $0 < \epsilon < 1$, a version of column-sort [22] provides with an upper bound of $\lceil \epsilon^{-3.42} \rceil O(n \lg n/p)$ and $\lceil \epsilon^{-3.42} \rceil O(gn \lg n/p)$ for computation and communication respectively. In [15], an improved upper bound of $(1 + \lfloor \epsilon^{-1}(1-\epsilon) \rfloor((1-\epsilon)/2) + o(1))(T_{seq}/p)$ and $O(gn/p))$ is shown for computation and communication respectively, where T_{seq} is the time required by sequential sorting. Finally, the algorithm of [14] requires, with high probability, computation and communication time $(1+o(1))(T_{seq}/p)$ and $O(gn \lg n/(p \lg (n/p)))$ respectively, for all values of n and p such that $p = O(n/\lg^{1+\alpha} n)$, for any constant $\alpha > 0$. In this work we derive $(1 + o(1))T_{seq}/p$ and $O(gn \lg n/(p \lg (n/p)))$ computation and communication time bounds for all values of n and p such that $p = \omega(n \lg \lg n / \lg n)$, which improves upon the slack of all previous approaches. We also improve, in case of polylogarithmic slack, the success probability of previously known randomized BSP algorithms [14].

Along the lines of [14, 15, 17, 18], our methods provide one-optimal BSP algorithms by transportable software that can be applied for a wide range of values of the parameters p, L and g. While the algorithms are fairly simple themselves, description of their performance in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables software to be written once and for all that can be migrated efficiently among a variety of parallel machines.

2 Primitive Operations

In this section we describe and analyze BSP algorithms for several fundamental operations, namely *broadcast*, *parallel-prefix* and *parallel radix-sort*. All these primitives are auxiliary routines for the algorithms described in latter sections. We assume that these primitives can be executed by all, or a subset, of the available p processors. A fully detailed account of such operations along with additional results can be found in [19].

Lemma 2.1 *There exists a BSP algorithm for broadcasting an n-word message that requires time at most $M^n_{brd}(p) = (\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, gt\lceil n/h\rceil\}$, for any integer $2 \le t \le p$, where $h = \lceil \log_t((t-1)p+1)\rceil - 1$.*

Proof: The underlying algorithm employs a pipelined t-ary tree, consisting of p vertices, for some appropriate t. The depth of the tree is $h = \lceil \log_t((t-1)p+1)\rceil - 1$. In each superstep, the root processor sends t copies of a separate m-word segment of the original message to its children, where $m = \lceil n/h\rceil$. Similarly, each internal processor sends t copies of the message that it received in the previous superstep to its own children. The algorithm completes within at most $\lceil n/m\rceil + h - 1$ supersteps. Each superstep requires gtm communication time, and the lemma follows. ■

Similarly, we obtain the following result.

Lemma 2.2 *There exists a BSP algorithm for computing n independent parallel-prefix operations that requires time at most $C^n_{ppf}(p) = 2(\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, t\lceil n/h\rceil\}$ and $M^n_{ppf}(p) = 2(\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, g2t\lceil n/h\rceil\}$, for computation and communication respectively, for any integer $2 \le t \le p$, where $h = \lceil \log_t p\rceil$, for a total time of $T^n_{ppf}(p) = C^n_{ppf}(p) + M^n_{ppf}(p)$.*

Proof: The underlying algorithm consists of two passes of the algorithm implied by lemma 2.1, on a pipelined t-ary tree, consisting of p leaf vertices and at most p internal vertices, for some appropriate t. The depth of the tree is $h = \lceil \log_t p\rceil$. The lemma follows by way of arguments similar to that of lemma 2.1. ■

For $n = 1$, we write $T_{ppf}(p)$ (respectively $C_{ppf}(p)$, $M_{ppf}(p)$) for $T^1_{ppf}(p)$ (respectively $C^1_{ppf}(p)$, $M^1_{ppf}(p)$). The same notational convention applies to all pipelined operations. The following integer sorting algorithm extends an idea of [29].

Lemma 2.3 *There exists a BSP algorithm for stable sorting n integers in the range $[0, 1, \ldots, R-1]$, that for any integer $2 \le t \le p$, and using the charging policy of section 1.2, requires time*

$$r\left(3\max\left\{L, O\left(\max\left\{\left\lceil\frac{n}{p}\right\rceil, t\lceil\log_t p\rceil\right\}\right)\right\} + C_{ppf}(p)\right),$$

and

$$gr\left(2\max\left\{\frac{L}{g}, \max\left\{\left\lceil\frac{n}{p}\right\rceil, t\lceil\log_t p\rceil\right\}\right\} + \max\left\{\frac{L}{g}, \left\lceil\frac{n}{p}\right\rceil\right\} + M_{ppf}(p)\right),$$

for computation and communication respectively, where $r = \lceil \lg R/\lg\max\{\lceil n/p\rceil, t\lceil\log_t p\rceil\}\rceil$.

Proof: Let $m = \max\{\lceil n/p\rceil, t\lceil\log_t p\rceil\}$. We show how to stably sort n keys in the range $[0, 1, \ldots, m-1]$, for any positive integers n and p such that $n \ge p$. We can then apply radix-sort techniques to prove that we can sort n keys in the range $[0, 1, \ldots, R-1]$, within $r = \lceil \lg R/\lg m\rceil$ iterations of the basic procedure. The input is assumed to be evenly distributed among the p processors before the beginning of the execution of the algorithm.

Program PARALLEL RADIX_SORT

begin
Stage 1 :
Every processor, in parallel with all the other processors, counts the number of its keys that fall into the m intervals induced by $[0, 1, \ldots, m-1]$. Let $S_{k,i}$ denote the number of keys with value k in processor i, $k \in \{0, 1, \ldots, m-1\}$ and $i \in \{0, 1, \ldots, p-1\}$.

[**Analysis of Stage 1:** When this step is executed, every processor holds at most $\lceil n/p \rceil$ keys. The cost of this stage is therefore bounded above by $\max\{L, m\}$.]

Stage 2 :
The position in the sorted array of the first item with value $\bar{k}$ that is assigned to processor $\bar{\imath}$ is given by

$$S_{\bar{k},\bar{\imath}} = 1 + \sum_{k,i} \{S_{k,i} \mid (k < \bar{k}) \text{ or } (k = \bar{k} \text{ and } i < \bar{\imath})\}.$$

Hence, if each processor $\bar{\imath}$ knows the value of $S_{\bar{k},\bar{\imath}}$ for all $\bar{k}$, then it can store, in sorted order, all of its keys in the appropriate output position. To realize this computation it is sufficient to order all the $S_{k,i}$'s in the direction of increasing k, perform a parallel-prefix sum operation, and finally restore the original order of the $S_{k,i}$'s.
[**Analysis of Stage 2:** This stage requires the realization of two m-relations and a parallel-prefix sum operation, for a total time of $2\max\{L, m\} + C_{ppf}(p)$ and $2\max\{L, gm\} + M_{ppf}(p)$ for computation and communication respectively.]

Stage 3 :
Every processor knows where to route every key in its possession, and performs this routing.
[**Analysis of Stage 3:** This stage requires time $\max\{L, g\lceil n/p \rceil\}$.]
end

Remark. The algorithm requires $O(\max\{n/p, t\log_t p\})$ space per processor; by replacing, however, in the above proposition $\lceil n/p \rceil$ for $\max\{\lceil n/p \rceil, t\lceil \log_t p \rceil\}$, the space can be reduced to $O(n/p)$ per processor. ■

3 Ordered h-level Graphs

In this section, we study the communication and synchronization requirements of searching on *ordered h-level graphs* [11], which include many of the standard data structures. We propose multi-way search as a general tool for the design, analysis, and implementation of BSP algorithms. As noted in [11], multi-way search allows elegant high-level design and analysis of algorithms, using data structures similar to those of sequential models. In later sections, applications of multi-way search to several fundamental computational problems are also presented.

We formally define ordered h-level graphs and discuss some of their properties. The definitions and notation of this section derive from [11]. We extend, however, some of the notions presented in [11] to include a more general class of ordered h-level graphs. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. An *ordered h-level partitioning* of G is a partitioning of V into h ordered sequences of disjoint sets $V_1, V_2, \ldots, V_h$, together with an ordering of the keys in each subset V_i, $i \in \{1, 2, \ldots, h\}$. For every h-level partitioning of G we define, $\forall\, v \in V$, two parameters, namely *Level*(v) and *Level_Index*(v) as follows.

(1) *Level*$(v) = l$ if and only if $v \in V_l$, $l \in \{1, 2, \ldots, h\}$,

(2) *Level_Index*(v) is the rank of v with respect to the ordering of the vertices in $V_{Level(v)}$.

A directed graph $G = (V, E)$ is called an *ordered h-level graph* if it has the following properties (see figure 1 for an illustration).

(1) Every vertex v in V has out degree of at most d_{max}. In contrast to [11], we do not insist on constant degree graphs.

(2) The number of sources d_s of G is at most d_{max}.

(3) There exists an ordered h-level partitioning $V_1, V_2, \ldots, V_h$, of G such that

- every source of G is contained in V_1,
- if (u, v) is an edge of G, then *Level*$(v) =$ *Level*$(u) + 1$,

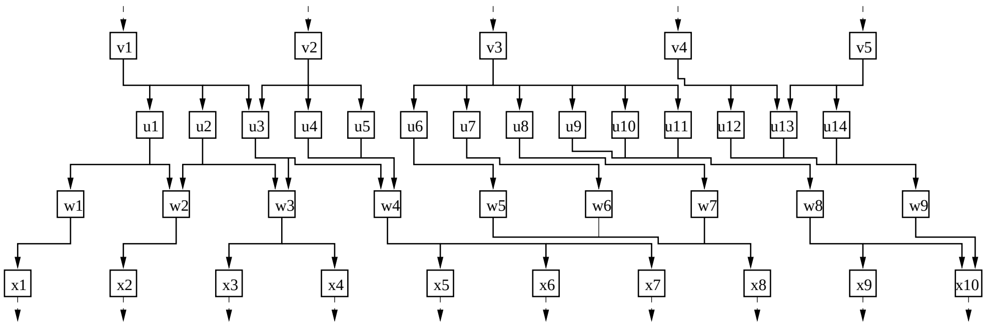

Figure 1: An ordered h-level graph.

- if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G such that $Level(u) = Level(\bar{u})$ and $Level\_Index(u) < Level\_Index(\bar{u})$, then, we have that $Level(v) = Level(\bar{v})$ and $Level\_Index(v) \leq Level\_Index(\bar{v})$.

(4) The graph has no double edges.

As it is noticed in [11], any d-ary tree can be expressed as an ordered h-level graph. The following lemma captures many important facts about ordered h-level graphs.

Lemma 3.1 *Let $G = (V, E)$ be an ordered h-level graph. The following statements are true.*

(1) G is planar and acyclic.

(2) $|E| \leq |V_1| + 2|V_2 \cup \ldots \cup V_{h-1}| + |V_h| - (h-1)$.

Proof: The planar and acyclic properties of G follow directly from the definition of ordered h-level graphs. In particular, the planarity of G derives from property 3 (case 3) of ordered h-level graphs. Similarly, the acyclic property of G derives from property 3 (case 2) of such graphs.

We use induction on the level number l of G to prove that $|E| \leq |V_1| + 2|V_2 \cup \ldots \cup V_{h-1}| + |V_h| - (h-1)$, The claim is initially true, for when $l = 1$, E is empty, and therefore, $|E| = 0$. Assume that the induction hypothesis holds up to level $l-1$, where $l \in \{1, 2, \ldots, h-1\}$. We next show that the claim holds for level l. We prove this by coloring the vertices of sets V_{l-1}, V_l and the edges between these two sets, and claiming that the number of colored edges is at most one less than the sum of the cardinalities of sets V_{l-1} and V_l. We use an inductive argument on the number of uncolored vertices in V_{l-1}. Let $u \in V_{l-1}$ be the vertex with the lowest *Level_Index* among the yet uncolored vertices of V_{l-1}. If there is no edge $(u, v) \in E$, we color u and our claim holds, since we have colored one vertex and no edges. Otherwise let $v_{j+1}, \ldots, v_{j+a} \in V_l$ be the vertices connected to u. By property 3, and the coloring so far, only the "rightmost" vertex v_{j+a} may have additional incoming edges from other uncolored vertices of V_{l-1}. Let the number of such additional edges be b. Denote by u_r the vertex with the highest *Level_Index* among the vertices of V_{l-1} pointing to v_{j+a}. By property 3, only vertex u_r among the b vertices of V_{l-1} may be connected to other vertices in V_l. Let the out-degree of u_r be $c \geq 1$, and let $v_{j+a}, v_{j+a+1}, \ldots, v_{j+a+c-1} \in V_l$ be the children of u_r. We color vertices $u, v_{j+1}, \ldots, v_{j+a+c-2}$ (all but the "rightmost" children of u_r), and the b vertices of V_{l-1} connected to v_{j+a} (u_r included). We also color the a outgoing edges of u, the b incoming edges of v_{j+a}, and the remaining $c-1$ outgoing edges of u_r. This process colors $a+b+c-1$ vertices and $a+b+c-1$ edges. If the "rightmost" child of u_r is not connected to some other yet uncolored vertex of V_{l-1}, then, we color it and our claim holds, since we have colored $a+b+c$ vertices and $a+b+c-1$ edges. We proceed as above until all vertices of V_{l-1} and V_l are colored. We can deduce that at the end of the coloring process, the number of colored vertices will be at least one more than the number of colored edges.

We thus conclude that the number of edges pointing to vertices in V_l is at most $|V_{l-1} \cup V_l| - 1$. By the induction hypothesis, the number of edges up to level $l-1$ is at most $|V_1| + 2|V_2 \cup \ldots \cup V_{l-2}| + |V_{l-1}| - (l-2)$, and thus, the number of edges up to level l is at most $|V_1| + 2|V_2 \cup \ldots \cup V_{l-1}| + |V_l| - (l-1)$, and the lemma follows. ■

4 Multi-Way Search on Ordered h-Level Graphs

In this section we present a BSP algorithm for efficiently searching an ordered h-level graph. We first introduce some formal definitions. Let $G = (V, E)$ be an ordered h-level graph of maximum out-degree d_{max}, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Moreover, let U be a universe of possible search queries on G. A *search path* for a query $q \in U$ is a sequence $Path(q) = \langle v_1, v_2, \ldots, v_h \rangle$ of h vertices of G, $v_i \in V_i$, prescribed by a *next* function $f : U \times (V \cup \{start\}) \to \{1, 2, \ldots, d_{max}\}$ as follows.

(1) $f(q, start) = Index(v_1)$,

(2) $f(q, v_i) = Index(v_{i+1})$, $i \in \{1, 2, \ldots, h-1\}$,

where $(v_i, v_{i+1}) \in E$ and for every $v \in V$, $Index(v) = \sum_{i=1}^{Level(v)-1} |V_i| + Level\_Index(v)$. For example, if G is a d-ary search tree, then for every query q, $f(q, start)$ evaluates to the index of (or pointer to) the root of the tree, and for every vertex v, $f(q, v)$ evaluates to the index of (or pointer to) the next vertex to be visited.

Given an ordered h-level graph G, we define the *search process* for a query q to be a process divided into h phases such that in phase l, $l \in \{1, 2, \ldots, h\}$, the search path $\langle v_1, v_2, \ldots, v_l \rangle$ of q has been determined and there exists a processor which contains a description of both the query q and the vertex v_l. We do not assume that the search path is given in advance, only that it is constructed during the search process by successive applications of the *next* function f. We note that, in contrast to [7], our method has the advantage of constructing the entire search path for each query q, and therefore, can be applied to the solution of a class of geometric problems such as those presented in [11]. The *multi-way search problem* is the following one.

Problem. Given a set of queries $Q = \{q_1, q_2, \ldots, q_n\} \subseteq U$ and an ordered h-level graph G with $|V| = m$, determine all n search processes induced by the n queries.

4.1 An Algorithm for Multi-Way Search

Assume we are given an ordered h-level graph $G = (V, E)$, of maximum out-degree d_{max}, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Let U be a universe of possible search queries on G and let $f : U \times (V \cup \{start\}) \to \{1, 2, \ldots, d_{max}\}$ be the *next* function. In this section, we consider a BSP algorithm for the solution of the multi-way search problem for a set $Q = \{q_1, q_2, \ldots, q_n\} \subseteq U$ of n queries.

In the implementation, the n search processes for all n queries are realized in a sequence of phases. The number of phases corresponds to the number of levels of the graph. At each phase l, $l \in \{1, 2, \ldots, h\}$, the queries advance through the graph along their search paths from level $l-1$ to level l. We observe that the first phase is different from subsequent phases, since it has to manage an arbitrary permutation of the queries, whereas each subsequent phase may utilize the processing performed in the previous stage to improve the performance of the algorithm. At each phase l, for each query $q \in Q$, the following two steps are executed.

(1) Let v_{l-1} be the $(l-1)$-th vertex in the search path of q. If $l = 1$, then $v_0 = start$. Evaluate $f(q, v_{l-1})$.

(2) Get the next vertex v_l in the search path of q.

The algorithm terminates within h phases. Some additional notation, describing the allocation of the queries among the processors, is required before proceeding to the details of the algorithm. Let v be a vertex of V. We define the set of queries visiting v to be the *thread* at vertex v. We refer to this thread as a thread of level $Level(v)$. A thread might be distributed over a number of consecutive processors; such a thread is called *shared*, as opposed to *exclusive*. The processors that hold a shared thread form a *group*. The first processor of each group is called the *group leader*.

In order to achieve communication efficiency, on a BSP machine, we maintain the following distribution (*load-balancing properties*) of the graph G among the processors.

(1) Each vertex $v \in V$, has out-degree of at most d_{max}, where $d_{max} = o(n/p)$. For each vertex v we store the information (*next_info*) pertaining to its children in the form of edges.

(2) Each edge $e \in E$, maintains (i) a pointer to the vertex v that e points to, (ii) the key associated with v, (iii) the out-degree of v.

(3) *start* is maintained as a vertex with children the sources of G.

(4) G is *orderly* and *evenly* distributed among the processors. That is, all outgoing edges of a given vertex are stored in consecutive memory locations of a single processor, and furthermore, no processor maintains more than $\lceil |E|/p \rceil + d_{max}$ edges.

The procedure to manage the computation, for an arbitrary phase l, is summarized in the following proposition.

Proposition 4.1 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Let the maximum out-degree of such a graph be $d_{max} = o(n/p)$, the minimum out-degree be d_{min}, and the number of source vertices be $d_s \leq d_{max}$. If f is the next function and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1, 2, \ldots, n\}$, that for any level l, and using the charging policy of section 1.2, has,*

$$\pi = \frac{p}{\tau_l(f)n} \left(\tau_l(f) \left(\left\lceil \frac{n}{p} \right\rceil + \frac{1}{\delta} \left\lceil \frac{2m}{p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p} + \frac{m}{\delta p} + L \right) + O\left(C_{ppf}(p)\right) \right),$$

and

$$\mu = \frac{p}{\tau_l(f)\, n} \left(O\left(g\delta \frac{n}{p} + g\frac{m}{\delta p} + L \right) + M_{brd}^{o(n/p)}(p) + O\left(M_{ppf}(p)\right) \right),$$

where $\delta = O(\sqrt{d_{min}})$ is some positive integer that expresses a trade-off between computation and communication time. The expression for π allows for all arithmetic as well as comparison operations. The algorithm requires $O(\delta n/p + m/p)$ space per processor.

Proof: The procedure realizing the algorithm is described as a sequence of stages, each consisting of a number of operations. The following *invariants* are assumed to hold for the procedure at phase l, $l \in \{1, 2, \ldots, h\}$.

(1) The queries are sorted with respect to the *Level_Index* of their associated vertex of level $l-1$.

(2) All shared threads at level $l-1$ are identified and characterized by their group and group leader.

In phase 1 we have a single thread associated with vertex *start* of out-degree $d_s \leq d_{max}$. The procedure for level l ensures that, for each query $q \in Q$, with search path $\langle v_1, v_2, \ldots, v_h \rangle$, there exists a processor which contains a description of both the query q and the vertex v_l. We assume that the n queries $Q = \{q_1, q_2, \ldots, q_n\}$ are initially evenly distributed among the processors. We also assume that the ordered h-level graph $G = (V, E)$, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$, satisfies load-balancing properties 1-4. Finally, we note that, in the remainder of this proof, we use the term *next_info*, for a particular thread $\bar{Q}$, to describe the edges emanating from the vertex associated with $\bar{Q}$. Informally, *next_info* relates to the information required to evaluate the *next* function.

Procedure MULTI_WAY SEARCH (l)

begin
Stage 1 :
Each processor obtains, for each thread in its possession, the associated *next_info* information.

[**Analysis of Stage 1:** Let $\bar{s}$ be the size of some thread $\bar{Q}$ formed in the previous phase or the single thread associated with vertex *start*, and let $\bar{d}$ be the out-degree of the vertex associated with $\bar{Q}$ (if $l=1$, then $\bar{s}=n$). We note that $\bar{d}$ is known by load-balancing property 1.

This stage is realized by one of several methods depending on the size and distribution of $\bar{Q}$. The motivation behind realizing different methods for obtaining *next_info* is that we aim at minimizing communication requirements. Hence, when a thread size is larger than the size of the associated *next_info* (cases 1-3 below), the latter information is read from the appropriate processor. Otherwise (case 4 below), the thread is communicated to the processor maintaining the associated *next_info*. In particular, if $\bar{Q}$ resides on some processor $\bar{p}$ and $\delta = O(\sqrt{\bar{d}})$, we proceed as follows.

(1) read *next_info*	if $\bar{s} \geq d_{max}/\delta$ and $\bar{Q}$ is exclusive,
(2) read *next_info*	if $\bar{s} \geq d_{max}/\delta$, $\bar{Q}$ is shared, and $\bar{p}$ is the group leader,
(3) read *next_info*	if $\bar{s} < d_{max}/\delta$ and $\bar{s} \geq \bar{d}/\delta$,
(4) send $\bar{Q}$ to $\hat{p}$	if $\bar{s} < \bar{d}/\delta$,
(5) remain idle	otherwise,

where $\hat{p}$ is the single processor maintaining *next_info*. When this stage is initiated every processor holds at most $\lceil n/p \rceil$ queries, and by lemma 3.1 and load-balancing property 4 it also maintains at most $\lceil 2m/p \rceil + d_{max}$ edges. By way of cases 1-3 above, each processor reads information (*next_info*) of size at most $\delta n/p + 2d_{max}$, where the $2d_{max}$ term is attributed to shared threads. Additionally, by way of case 4 above, each processor receives a set Q^* of at most $s^* = (\lceil 2m/p \rceil + d_{max})/\delta$ additional queries. Hence, this step takes $\max\{L, g\, O(\delta n/p + (m/p + d_{max})/\delta)\}$.

If $\bar{Q}$ is shared and $\bar{p}$ is the group leader for $\bar{Q}$, then $\bar{p}$ broadcasts the *next_info* information it read in the previous step to all the processors in its group. By lemma 2.1, we have that this step takes communication time at most $M_{brd}^{d_{max}}(p)$. At the end of this stage each processor holds at most $\lceil n/p \rceil + s^*$ queries.]

Stage 2 :
Each processor computes, for each query $q \in Q$ in its possession, the *Index* of the next vertex v_l in the search path of q by evaluating the function

$$\begin{array}{ll} f(q, start) & \text{if } l = 1, \\ f(q, v_{l-1}) & \text{otherwise .} \end{array}$$

[**Analysis of Stage 2:** When this stage is executed every processor holds at most $\lceil n/p \rceil + s^*$ queries, and therefore, the time required by this stage is bounded above by $\max\{L, \tau_l(f)(\lceil n/p \rceil + (\lceil 2m/p \rceil + d_{max})/\delta)\}$, where $\tau_l(f)$ is the time required to evaluate the function f at level l.]

If $l < h$, stages 3-5 maintain (update) invariants 1 and 2 for phase $l+1$.

Stage 3 :
Processors sort the queries within each thread with respect to the *Level_Index* of the associated vertex of the current level. This stage maintains invariant 1 for phase $l+1$.
[**Analysis of Stage 3:** Let $\bar{s}$ be the size of some thread $\bar{Q}$ formed in the previous phase or the single thread in the beginning of the algorithm, and let $\bar{d}$ be the out-degree of the vertex associated with $\bar{Q}$ (if $l=1$, then $\bar{s}=n$). The sorting operation is performed by one of several methods depending on the size and distribution of $\bar{Q}$. Particularly, we employ

(1) RADIX_SORT	if $\bar{s} \geq \sqrt{\bar{d}/\delta}$ and $\bar{Q}$ is exclusive,
(2) PARALLEL RADIX_SORT	if $\bar{s} \geq \sqrt{\bar{d}/\delta}$ and $\bar{Q}$ is shared,
(3) SUB_QUADRATIC_SORT	if $\bar{s} < \sqrt{\bar{d}/\delta}$.

In case 3, if $\bar{Q}$ is shared, then the lower-indexed of the two processors sharing $\bar{Q}$ performs procedure SUB_QUADRATIC_SORT. When this stage is executed the overall size of all threads of size greater than $\sqrt{\bar{d}/\delta}$ on any one processor is at most $\lceil n/p \rceil + s^*$. By way of algorithms RADIX_SORT and PARALLEL_RADIX_SORT (sections 1.2 and 2), and the fact that $\delta = O(\sqrt{\bar{d}})$, we have that each such thread can be sorted in

$$\left\lceil \frac{\lg \bar{d}}{\lg \sqrt{\bar{d}/\delta}} \right\rceil = O(1)$$

iterations of the underlying linear-time radix-based sorting algorithm, i.e., sorting of at least $\sqrt{\bar{d}/\delta}$ integers in the range $[0, \ldots, \bar{d}-1]$. Thus, by way of lemma 2.3, the time required for sorting all such threads is $O(\max\{L, n/p + (m/p + d_{max})/\delta\}) + O(C_{ppf}(p))$ and $O(\max\{L, g(n/p + (m/p + d_{max})/\delta)\}) + O(M_{ppf}(p))$ for computation and communication respectively.

By the definition of SUB_QUADRATIC_SORT (section 1.2), each thread of size less than $\sqrt{\bar{d}/\delta}$ can be sorted in $o(\bar{d}/\delta)$ time. The total time required to sort all such threads $\bar{Q}_j$ of size $\bar{s}_j < \sqrt{\bar{d}_j/\delta}$ on any particular processor is $\max\{L, o(\sum_j \bar{s}_j^2)\} \leq \max\{L, o(\sum_j \bar{d}_j/\delta)\}$, where $\bar{d}_j$ is the out-degree of the vertex associated with thread $\bar{Q}_j$. By lemma 3.1 and load-balancing property 4, we have that $\sum_j \bar{d}_j \leq \lceil 2m/p \rceil + d_{max}$, and therefore, $\sum_j \bar{s}_j^2 \leq (\lceil 2m/p \rceil + d_{max})/\delta$. Thus, the total time required to sort all such threads is bounded is bounded above by $\max\{L, o((m/p + d_{max})/\delta)\}$ and $\max\{L, o(g(m/p + d_{max})/\delta)\}$ for computation and communication respectively.
]

Stage 4 :
Each processor returns to the original processor the threads that it received in stage 1.
[**Analysis of Stage 4:** Since the total size of all the threads that any single processor sent or received in stage 1 is at most $\lceil n/p \rceil$ and s^* respectively, this stage take time $O(\max\{L, g\max\{n/p, (m/p + d_{max})/\delta\}\})$.
]

Stage 5 :
Each processor determines and characterizes, by group and group leader, the threads at level l. This stage updates invariant 2 for phase $l+1$.
[**Analysis of Stage 5:** Since the queries are sorted with respect to their *Level_Index*, each processor can determine all of its exclusive threads by simply scanning the queries in its local memory. Moreover, if there are shared threads, all of them can be identified and characterized by communicating a constant number of messages and by performing a parallel-prefix operation. In particular, the prefix operator takes two operands, namely $(shared\_thread_i, processor_{i'})$ and $(shared\_thread_j, processor_{j'})$, and returns

$$\begin{array}{ll} (shared\_thread_i, processor_{i'}) & \text{if } shared\_thread_i = shared\_thread_j, \\ (shared\_thread_j, processor_{j'}) & \text{if } shared\_thread_i \neq shared\_thread_j, \end{array}$$

where $i < j$. After completion of the parallel prefix operation each processor knows for each of its shared threads its group and group leader, i.e., the group leader of some shared thread i is determined by the second argument of the pair returned by the parallel-prefix operation.

When this stage is initiated every processor holds at most $\lceil n/p \rceil$ queries and at most two shared threads. Thus, the cost of this stage is bounded above by $\max\{L, O(n/p)\} + O(C_{ppf}(p))$ and $\max\{L, O(g)\} + O(M_{ppf}(p))$ for computation and communication respectively.]
end

We therefore obtain that the time complexity, over all stages, of procedure MULTI_WAY SEARCH is

$$\tau_l(f)\left(\left\lceil \frac{n}{p} \right\rceil + \frac{1}{\delta}\left\lceil \frac{2m}{p} \right\rceil + o\left(\frac{n}{p}\right)\right) + O\left(\frac{n}{p} + \frac{m}{\delta p} + L\right) + O\left(C_{ppf}(p)\right),$$

and

$$O\left(g\delta\frac{n}{p} + g\frac{m}{\delta p} + L\right) + M_{brd}^{o(n/p)}(p) + O\left(M_{ppf}(p)\right)$$

for computation and communication respectively, and the proposition follows. ■

By substituting appropriate values for the parameters in proposition 4.1 a multiplicity of bounds on the run-time of algorithm MULTI_WAY SEARCH can be obtained. As noted in [14], the best time bounds are obtained by selecting these parameters with regard to the actual values of n, p, g and L. By making, however, more generic selections for the parameters and worst case bounds for p, g and L, more general results in terms of n can be obtained. The following is one such bound.

Theorem 4.1 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$, edge set E, maximum out-degree $d_{max} = o(n/p)$, minimum out-degree d_{min} and $d_s \leq d_{max}$ source vertices such that $|V| \leq n$. If f is the next function and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing all n search processes induced by the n queries, that for all positive integers n, p, L, δ and t satisfying $t = \min\{O(L/g), p\}$, $L = o(\min_l \{\tau_l(f)\} n/(p \log_t p))$ and $\delta = O(\sqrt{d_{min}})$, has, $\pi = 1 + 2/\delta + o(1) + O(h/(\sum_l \tau_l(f)))$, and if $g = o((\sum_l \tau_l(f))/(h\delta))$ also, $\mu = o(1)$. The algorithm requires $O(\delta n/p + m/p)$ space per processor.*

Remark. In the aforementioned result, the constraints $L = o(.)$ and $g = o(.)$ can be removed if we relax the one-optimality requirement. In that case, $L = O(.)$ and $g = O(.)$ would suffice to obtain $\pi = O(1)$ and $\mu = O(1)$.

4.2 Load-Balancing Multi-Way Search

In the description of algorithm MULTI_WAY SEARCH of section 4.1, we used the parameter δ to express a trade-off between computation and communication time. Particularly, we obtained a multiplicative constant factor of $1 + 2/\delta + o(1) + O(1/\tau(f))$ in computation at the expense of a δ-wise increase in the communication and space requirements of the algorithm. One-optimality in computation can thus be achieved for $\delta = \omega(1)$.

In the remainder of this section we show how to achieve one-optimality in computation and yet maintain $\delta < 3$. The main inefficiency of algorithm MULTI_WAY SEARCH derives from case 4 of stage 1, where a set Q_i^*, of size $s_i^* \leq (\lceil 2m/p \rceil + d_{max})/\delta$, is received by processor p_i. We assume that $s_i^* \leq n/p$, which is always true (asymptotically) for some $\delta < 3$, $m \leq n$ and $d_{max} = o(n/p)$. We next outline a scheme for load-balancing the queries in stage 1 so that each processor in stages 2 through 5 operates on a set of queries of total size at most $\lceil n/p \rceil$.

Recall that the queries in Q_i^* need to be evaluated by the recipient processor and therefore cannot be reallocated in order to achieve load-balance. The remaining queries, say, Q_i, are distributed among the processors so that each processor receives $\lceil n/p \rceil - s_i^*$ queries for a total of at most $\lceil n/p \rceil$. The load-balancing scheme is realized by a series of parallel-prefix sum and regular routing operations. Initially, the queries in $\cup_i Q_i$ are evenly distributed among the processors. This step can be achieved by realizing a parallel-prefix sum operation on the size of the Q_i's followed by a routing operation of size at most $\lceil n/p \rceil$. Let $\lceil n/p \rceil - s_i^*$ denote the *capacity* of processor p_i. We perform a parallel-prefix sum operation on the capacities to determine the final destination of the queries in $\cup_i Q_i$. The load-balancing operation is completed by realizing a routing operation of size at most $\lceil n/p \rceil$. The total time complexity of the aforementioned load-balancing scheme is bounded above by $2\max\{L, O(n/p)\} + O(C_{ppf}(p))$ and $2\max\{L, g\lceil n/p \rceil\} + O(M_{ppf}(p))$ for computation and communication respectively. We can therefore derive the following analogue of proposition 4.1.

Proposition 4.2 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m \leq n$. Let the maximum out-degree of such a graph be $d_{max} = o(n/p)$, and the number of source vertices be $d_s \leq d_{max}$. If f is the next function, and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1, \ldots, n\}$, that for any level l, and using the charging policy of section 1.2, has,*

$$\pi = \frac{p}{\tau_l(f) n} \left(\tau_l(f) \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{n}{p} + L\right) + O\left(C_{ppf}(p)\right) \right),$$

and

$$\mu = \frac{p}{\tau_l(f)\, n} \left(O\left(g\frac{n}{p} + L \right) + M_{brd}^{o(n/p)}(p) + O\left(M_{ppf}(p)\right) \right).$$

The expression for π allows for all arithmetic as well as comparison operations. The algorithm requires $O(n/p + m/p)$ space per processor.

Similarly, we obtain the following analogue of theorem 4.1.

Theorem 4.2 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$, edge set E, maximum out-degree $d_{max} = o(n/p)$ and $d_s \leq d_{max}$ source vertices such that $|V| \leq n$. If f is the next function and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing all n search processes induced by the n queries, that for all positive integers n, p, L and t satisfying $t = \min\{O(L/g), p\}$ and $L = o(\min_l \{\tau_l(f)\} n/(p \log_t p))$, has, $\pi = 1 + O(h/(\sum_l \tau_l(f)))$, and if $g = o((\sum_l \tau_l(f))/h)$ also, $\mu = o(1)$. The algorithm requires $O(n/p + m/p)$ space per processor.*

Remark. In the aforementioned result, the constraints $L = o(.)$ and $g = o(.)$ can be removed if we relax the one-optimality requirement. In that case, $L = O(.)$ and $g = O(.)$ would suffice to obtain $\pi = O(1)$ and $\mu = O(1)$. We also note that even though the stated load-balancing scheme improves asymptotically the performance of algorithm MULTI_WAY SEARCH, in most practical situations an appropriately chosen δ would result in superior performance.

4.3 Improving the Performance of Algorithm MULTI_WAY SEARCH

We note that not only can the bounds given in the previous sections be expressed in a variety of ways, but also algorithm MULTI_WAY SEARCH leaves room for several improvements. As stated, the algorithm makes the implicit assumption that the time $\tau_i(f)$ required to evaluate the *next* function f at some level i, $i \in \{1, \ldots, h\}$, is uniform over all vertices of level i. Even though this assumption is valid in most practical situations (e.g., fixed degree ordered h-level graphs, d-ary trees), there might be cases where the time required to evaluate f at level i varies considerably among the vertices of level i. We address this problem in section 4.3.1, by suggesting a simple vertex-decomposition scheme that normalizes the time required to evaluate f over all vertices of the same level.

We also note that algorithm MULTI_WAY SEARCH is limited by the fact that it requires g to be upper bounded by $o((\sum_i \tau_i(f))/h)$ in order to achieve *one*-optimality (theorem 4.2). This bound implies that the performance of algorithm MULTI_WAY SEARCH is closely linked to the actual time $\tau_i(f)$ required to evaluate the *next* function f at level i. For example, for $\tau_i(f) = O(1)$ (e.g., constant degree ordered h-level graphs), the algorithm requires $g = O(1)$ in order to obtain $\mu = O(1)$. Intuitively, the higher $\tau_i(f)$ is, the better the performance of the algorithm. As $\tau_i(f)$ is in general some non-decreasing function of the out-degree of the vertices at level i, the performance of algorithm MULTI_WAY SEARCH can be improved by increasing the out-degree of the vertices. To this end, in section 4.3.2 we present a rather natural level-composition scheme that coalesces a number of consecutive levels into a single level, and consequently, increases the out-degree of vertices.

4.3.1 Modular Decomposition of Nodes

In this section we outline a simple vertex-decomposition scheme that normalizes the time $\tau_i(f)$ required to evaluate the *next* function f over all vertices of the same level. Let $G = (V, E)$ be an ordered h-level graph with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Consider an arbitrary level i of G and let $\tau_{i_{min}}(f)$ and $\tau_{i_{max}}(f)$ denote respectively the minimum and maximum time required to evaluate f over all vertices in V_i. Similarly, let $d_{i_{min}}$ and $d_{i_{max}}$ denote respectively the minimum and maximum out-degree over all vertices in V_i. In practical situations, $\tau_{i_{min}}(f)$ and $\tau_{i_{max}}(f)$ are some non-decreasing functions of $d_{i_{min}}$ and $d_{i_{max}}$ respectively. For example, $\tau_i(f) = O(d_i)$ if the *next* function is realized as a linear-search routine, and similarly, $\tau_i(f) = O(\lg d_i)$ if the *next* function is realized as a binary-search routine.

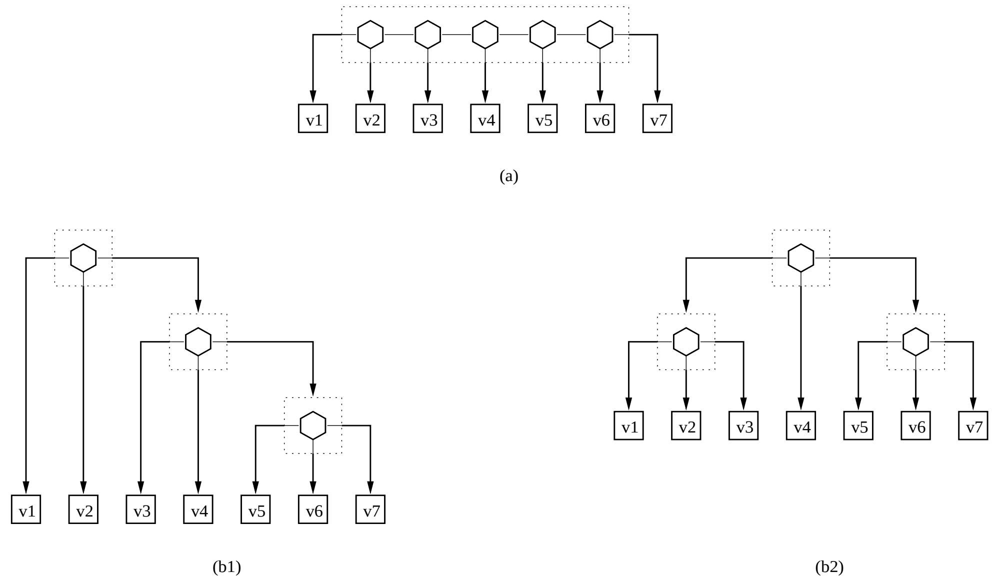

Figure 2: Modular decomposition of the vertex of out-degree 7 (case (a)) into the sub-vertices of out-degree 3 (cases (b1) and (b2)). The multi-level sub-vertex structure (of case (b1)) supports a linear-search *next* function and the multi-level vertex structure of (case (b2)) supports a binary-search *next* function. *Solid-line squares* and *dotted-line rectangles* represent *vertices*, solid-line arrows represent *edges*, and *solid-line hexagons* represent the *data* associated with edges.

The time required to evaluate the *next* function f over all vertices of level i can thus be normalized by *modularly decomposing* each vertex v_j in V_i into an r^*_{ij}-level structure, consisting of sub-vertices of minimum out-degree $d^*_{i_{min}} = d_{i_{min}}$ and maximum out-degree $d^*_{i_{max}}$, such that $\tau^*_{i_{max}}(f) = (1 + o(1))\tau^*_{i_{min}}(f)$. The aforementioned decomposition scheme guarantees that the time $\tau^*_i(f) = \tau^*_{k_{max}}(f)$ required to evaluate the *next* function f is $(1 + o(1))\tau^*_{i_{min}}(f)$, and therefore, the time normalization process is completed. If a vertex is decomposed into an r^*-level sub-vertex structure, then we say that the *decomposition factor* is $r^* \leq d_{max}$, e.g., the decomposition factor of cases (b1) and (b2) of figure 2 is 3 and 2 respectively. We note that the decomposition procedure is application specific and closely related to the *next* function f. We do not make any claims on how such a decomposition can be derived in general; in the remainder of this section we assume that such a procedure is available.

The adoption of such a vertex-decomposition scheme does not affect asymptotically the computational time complexity of our algorithm, it does, however, introduce a fundamental complication: by decomposing vertices into multi-level structures we have violated property 3 (cases 2 and 3) of ordered h-level graphs. We could remedy the situation by introducing fictitious null vertices in the multi-level structure in order to preserve the structural properties of ordered h-level graphs. This approach is however self-defeating since by introducing null vertices we further reduce the minimum out degree $d_{i_{min}}$ to zero.

In order to overcome these complications we extend the notion of ordered h-level graphs to that of r^*_{max}-ordered h^*-level graphs that can accommodate the multi-level structures we introduce by modularly decomposing the vertices of the underlying ordered h-level graph. In particular, a directed graph $G^* = (V^*, E^*)$ is called an *r^*_{max}-ordered h^*-level graph* if it has the following properties.

(1) Every vertex v in V^* has out-degree of at most d^*_{max}.

(2) The number of sources d^*_s of G^* is at most d^*_{max}.

(3) The maximum decomposition factor r^*_{max} of G^* is at most d^*_{max}.

(4) There exists an ordered h^*-level partitioning $V^*_1, V^*_2, \ldots, V^*_{h^*}$ of G^* such that

- every source of G^* is contained in V^*_1,
- if (u, v) is an edge of G^*, then $Level(u) < Level(v) \leq Level(u) + r^*_{max}$,

- if (u,v) and $(\bar{u},\bar{v})$ are two edges of G^*, such that both $Level(u) = Level(\bar{u})$ and $Level\_Index(u) < Level\_Index(\bar{u})$, then $Level(\bar{v}) - (r^*_{max} - 1) \leq Level(v) \leq Level(\bar{v}) + (r^*_{max} - 1)$, and if $Level(v) = Level(\bar{v})$, then also $Level\_Index(v) \leq Level\_Index(\bar{v})$.

(5) Graph G^* has no double edges.

The parameter r^*_{max} in the above definition reflects the maximum decomposition factor over all vertices of the underlying ordered h-level graph; if $r^*_{max} = 1$, then the definitions of ordered h-level graphs and r^*_{max}-ordered h^*-level graphs coincide.

The procedure to manage multi-way search on r^*_{max}-ordered h^*-level graphs is similar to MULTI_WAY SEARCH of section 4.1. The n search processes for all n queries are realized in a sequence of phases. The number of phases is upper bounded by the number of levels h^* of the underlying r^*_{max}-ordered h^*-level graph. At each phase l, the queries advance jointly through the graph along their search paths. We observe that, r^*_{max}-ordered h^*-level graphs, in contrast to ordered h-level graphs, allow queries to proceed to different levels of the graph. Therefore, the multi-way search algorithm for r^*_{max}-ordered h^*-level graphs has to cope with the requirement of queries being processed on different levels of the underlying graph.

To this end, we extend the invariants assumed to hold for algorithm MULTI_WAY SEARCH to accommodate the notion of queries being sorted with respect to the *Level* of their associated vertex (and still preserve within the *Level* the ordering with respect to *Level_Index*). Since queries on different levels can proceed independently we can employ algorithm MULTI_WAY SEARCH to realize searching on r^*_{max}-ordered h^*-level graphs.

It remains to show how we can effect the ordering of queries with respect to the *Level* of their associated vertex. We observe that, by way of property 4 (case 3) of r^*_{max}-ordered h^*-level graphs, the queries at some level i, $i \in \{1, \ldots, h^*\}$, can proceed to at most r^*_{max} distinct consecutive levels of the underlying graph. Similarly, by way of property 4 (case 2), the queries from at most r^*_{max} distinct consecutive levels can converge to the same level i. Thus, no two queries that are r^*_{max} levels apart at phase $l-1$ of the search procedure can be on the same level at phase l. We next sketch the algorithmic scheme utilized to obtain the queries at phase l sorted with respect to the *Level* of their associated vertex.

Let the queries at phase $l-1$ be sorted with respect to the *Level* of their associated vertex. Denote by $k_{(l-1)_{min}}$ and $k_{(l-1)_{max}}$ the minimum and maximum *Level* of all vertices visited by queries at phase $l-1$. We partition the queries into $k = \lceil (k_{(l-1)_{max}} - k_{(l-1)_{min}})/r^*_{max} \rceil$ buckets $H_1, \ldots, H_k$, such that H_j, $j \in \{1, \ldots, k\}$, consists of the queries that at phase $l-1$ visit some vertex at $Level \in \{i, \ldots, i'\}$, where

$$\{i, \ldots, i'\} = \begin{cases} \{k_{(l-1)_{min}}, \ldots, k_{(l-1)_{min}} + r^*_{max}\} & \text{if } j = 1, \\ \{k_{(l-1)_{min}} + (j-1)r^*_{max} + 1, \ldots, k_{(l-1)_{min}} + jr^*_{max}\} & \text{if } 1 < j < k, \\ \{k_{(l-1)_{min}} + (k-1)r^*_{max} + 1, \ldots, k_{(l-1)_{max}}\} & \text{if } j = k. \end{cases}$$

The determination of $k_{(l-1)_{min}}$ and $k_{(l-1)_{max}}$ takes time $O(T_{ppf}(p))$. The partioning of queries into buckets can be performed by way of a procedure similar to that of stage 5 of algorithm MULTI_WAY SEARCH, and therefore, requires time $\max\{L, O(n/p)\} + O(C_{ppf}(p))$ and $\max\{L, O(g)\} + O(M_{ppf}(p))$ for computation and communication respectively.

The queries within $H_j \cup H_{j+1}$, for all odd j in $\{1, \ldots, k-1\}$, are sorted independently with respect to the *Level* of their associated vertex at phase l. The sorting operation is realized by one of several methods depending on the size s of $H_j \cup H_{j+1}$. In particular, we employ

(1) RADIX_SORT if $s \geq (r^*_{max})^\alpha$ and $H_j \cup H_{j+1}$ is exclusive,
(2) PARALLEL RADIX_SORT if $s \geq (r^*_{max})^\alpha$ and $H_j \cup H_{j+1}$ is shared,
(3) SUB_QUADRATIC_SORT if $s < (r^*_{max})^\alpha$,

for any constant $0 < \alpha \leq 1/2$. In case 3, if $H_j \cup H_{j+1}$ is shared, then the lower-indexed of the two processors sharing $H_j \cup H_{j+1}$ performs procedure SUB_QUADRATIC_SORT.

Recall that the queries at some level i can proceed to at most r^*_{max} distinct consecutive levels of the underlying graph, and the queries from at most r^*_{max} distinct consecutive levels can converge to the same level i. Thus, by way of algorithms RADIX_SORT and PARALLEL RADIX_SORT (sections 1.2 and 2), we have that the queries within $H_j \cup H_{j+1}$, where $s \geq (r^*_{max})^\alpha$, can be sorted in

$$\left\lceil \frac{\lg(2r^*_{max})}{\lg((r^*_{max})^\alpha)} \right\rceil = O(1)$$

iterations of the underlying linear-time radix-based sorting algorithm, i.e., sorting of at least $(r^*_{max})^\alpha$ integers in the range $[0,\ldots,2r^*_{max}-1]$. Thus, by way of lemma 2.3, the time required for sorting all such buckets is $O(\max\{L, n/p\}) + O(C_{ppf}(p))$ and $O(\max\{L, gn/p\}) + O(M_{ppf}(p))$ for computation and communication respectively.

By the definition of SUB_QUADRATIC_SORT (section 1.2), each bucket of size less than $(r^*_{max})^\alpha$ can be sorted in $o((r^*_{max})^{2\alpha})$ time. The total number of such buckets is at most

$$\left\lceil \frac{k_{(l-1)_{max}} - k_{(l-1)_{min}}}{r^*_{max}} \right\rceil \leq \left\lceil \frac{h^*}{r^*_{max}} \right\rceil \leq \frac{m^*}{r^*_{max}} + 1.$$

We evenly distribute these buckets among the processors by employing the load-balancing scheme of section 4.2. The time required for this operation is $2\max\{L, O(n/p + (r^*_{max})^\alpha)\} + O(C_{ppf}(p))$ and $2\max\{L, g(\lceil n/p\rceil + (r^*_{max})^\alpha)\} + O(M_{ppf}(p))$ for computation and communication respectively. Then, the time required to sort all such buckets of size $s < (r^*_{max})^\alpha$ is $\max\{L, \lceil (m^*/r^*_{max}+1)/p)\rceil o((r^*_{max})^{2\alpha})\} = \max\{L, o((m^*(r^*_{max})^{2\alpha-1})/p + (r^*_{max})^{2\alpha})\}$.

It should be clear that the queries in buckets H_j and H_{j+2} are ordered with respect to each other. Thus, by sorting the queries within $H_j \cup H_{j+1}$, for all even $j \in \{2,\ldots,k-1\}$, we obtain the queries at phase l globally sorted with respect to the *Level* of their associated vertex. The overall time required by this sorting scheme is $O(n/p+L)+o((m^*(r^*_{max})^{2\alpha-1})/p+(r^*_{max})^{2\alpha})+O(C_{ppf}(p))$ and $O(gn/p+g(r^*_{max})^\alpha + L) + O(M_{ppf}(p))$ for computation and communication respectively. By substituting $1/2$ for α, we therefore obtain the following analogue of proposition 4.2.

Proposition 4.3 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G^* = (V^*, E^*)$ be an r^*_{max}-ordered h^*-level graph, with vertex set $V^* = V^*_1 \cup V^*_2 \cup \ldots \cup V^*_{h^*}$ and edge set E^*, where $|V^*| = m^* \leq n$. Let the maximum out-degree of such a graph be $d^*_{max} = o(n/p)$, the number of source vertices be $d^*_s \leq d^*_{max}$ and the maximum decomposition factor be $r^*_{max} \leq d^*_{max}$. If f is the next function and $\tau^*_l(f)$ is the time required to evaluate f at phase l, then, there exists a BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1,\ldots,n\}$, that for any phase l, and using the charging policy of section 1.2, has,*

$$\pi = \frac{p}{\tau^*_l(f)\, n}\left(\tau^*_l(f)\left\lceil \frac{n}{p}\right\rceil + O\left(\frac{n}{p}+L\right) + O\left(C_{ppf}(p)\right)\right),$$

and

$$\mu = \frac{p}{\tau^*_l(f)\, n}\left(O\left(g\frac{n}{p}+L\right) + M^{o(n/p)}_{brd}(p) + O\left(M_{ppf}(p)\right)\right).$$

The expression for π allows for all arithmetic as well as comparison operations. The algorithm requires $O(n/p + m^*/p)$ space per processor.

Remark. In proposition 4.3, by abuse of notation, we refer to $\tau^*_l(f)$ as the time required to evaluate f at *phase* l, rather than at *level* l. This is to be interpreted as $\tau^*_l(f) = \max_i\{\tau^*_i(f)\}$, where i ranges over all levels participating at phase l.

4.3.2 Structural Composition of Levels

In this section we present a simple level-composition scheme that coalesces a number of consecutive levels into a single level, and consequently, increases the out-degree of vertices. Let $G = (V, E)$ be an ordered h-level graph with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Consider an arbitrary level i of G and let $\tau^*_i(f)$ denote the time required to evaluate the *next* function f at level i. As noted in

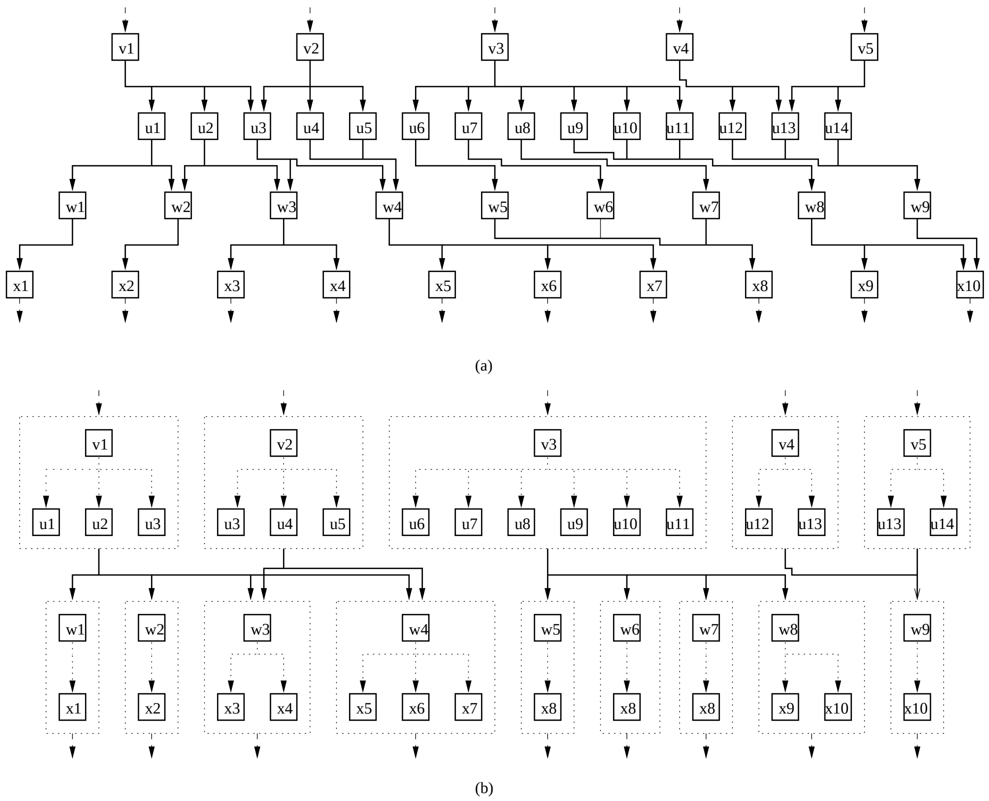

Figure 3: Structural composition of the one-level vertices (of case (a)) into the two-level vertices (of case (b)). *Solid-line squares* and *dotted-line rectangles* represent respectively *vertices* and *multi-level vertices*, *solid-line arrows* and *dotted-line arrows* represent respectively *edges* and *internal multi-level vertex edges*.

section 4.3, the performance of algorithm MULTI_WAY SEARCH is closely linked to the actual time $\tau_i^*(f)$ required to evaluate f; the higher $\tau_i^*(f)$ is, the better the performance of the algorithm.

As $\tau_i^*(f)$ is in general some non-decreasing function of the out-degree of the vertices at some level i, the performance of algorithm MULTI_WAY SEARCH can be improved by increasing the out-degree of the vertices at level i. To this end, we *structurally compose* the descendants of each vertex in V_i, up to some level $i' \geq i$, into a single multi-level vertex structure. If the vertices and their descendants are composed into R^*-level vertex structures, then we say that the *composition factor* is R^*, e.g., the composition factor of figure 3 is 2. Even though the composition procedure is application independent, we do not make any claims on how such a composition can be derived in general; in the remainder of this section we assume that such a procedure is available.

Remark. Since each vertex in levels $i+1$ through i' is replicated as many times as its in-degree, we obtain that the resulting vertex structure corresponding to levels i through i' has size at most $\sum_j \sum_{k=i}^{j} |V_k|$, $j \in \{i, \ldots, i'\}$, which is at most $R^* \sum_k |V_k|$, where $R^* = i' - i + 1$. We thus obtain (in the worst case) an R^*-fold increase in memory requirements. We note, however, that in most practical situations the increase in memory requirements is negligible, e.g., d-ary trees cause a one-fold increment in space requirements. In addition, the out-degree of any vertex of the composed graph is upper bounded by d^{R^*}, where d is its out-degree in the original graph.

The aforementioned composition scheme guarantees that the *next* function f at level i takes time $\sum_k \tau_k^*(f)$, $k \in \{i, \ldots, i+R^*-1\}$. We have thus increased the time required to evaluate the *next* function f at level i, and simultaneously, decreased the number of levels in the underlying graph. It is straightforward to verify that the adoption of such a level-composition scheme does not affect asymptotically the computational time complexity of our algorithm, it does, however, introduce a fundamental complication: by composing a vertex and its descendants into a single multi-level vertex structure we have violated property 3 (case 3) of ordered h-level graphs.

In order to overcome this complication we extend the notion of ordered h-level graphs to that of R^*_{max}-ordered h^*-level graphs that can accommodate the multi-level vertex structures we introduced by structurally composing the levels of the underlying ordered h-level graph. In particular, a directed graph $G^* = (V^*, E^*)$ is called an *R^*_{max}-ordered h^*-level graph* if it has the following properties.

(1) Every vertex v in V^* has out-degree of at most d^*_{max} and size at most s^*_{max}, where the size of a vertex is the size of its internal structure (vertices and edges).

(2) The number of sources d^*_s of G^* is at most d^*_{max}.

(3) The maximum composition factor of G^* is R^*_{max}.

(4) There exists an ordered h^*-level partitioning $V^*_1, V^*_2, \ldots, V^*_{h^*}$ of G^*, such that

- every source of G^* is contained in V^*_1,
- if (u, v) is an edge of G^*, then $Level(v) = Level(u) + 1$,
- if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G^*, such that both $Level(u) = Level(\bar{u})$ and $Level\_Index(u) < Level\_Index(\bar{u})$, then $Level(v) = Level(\bar{v})$ and $Level\_Index(v) \leq Level\_Index(\bar{v}) + (d^*_{max} - 1) \min\{1, R^*_{max} - 1\}$.

(5) Graph G^* has no double edges.

The parameter R^*_{max} in the above definition reflects the maximum composition factor over all levels of the underlying ordered h-level graph; if $R^*_{max} = 1$, then the definitions of ordered h-level graphs and R^*_{max}-ordered h^*-level graphs coincide.

The procedure to manage multi-way search on R^*_{max}-ordered h^*-level graphs is similar to MULTI_WAY SEARCH of section 4.1. The n search processes for all n queries are realized in a sequence of phases. The number of phases corresponds to the number of levels of the underlying R^*_{max}-ordered h^*-level graph. At each phase l, $l \in \{1, \ldots, h^*\}$, the queries advance through the graph along their search paths from level $l-1$ to level l. We observe that, for R^*_{max}-ordered h^*-level graphs, in contrast to ordered h-level graphs, sorting the queries within each thread with respect to the *Level_Index* of their associated vertex is not sufficient to maintain the queries globally sorted. Therefore, the multi-way search algorithm for R^*_{max}-ordered h^*-level graphs has to cope with this complication.

To this end, we extend the scope of stage 3 of algorithm MULTI_WAY SEARCH to handle sorting of queries with respect to the *Level_Index* of their associated vertex in the underlying R^*_{max}-ordered h^*-level graph. We next sketch the algorithmic scheme utilized to obtain the queries at phase l sorted with respect to the *Level_index* of their associated vertex.

Let the queries at phase $l-1$ be ordered with respect to the *Level_Index* of their associated vertex. Denote by $K_{l_{min}}$ and $K_{l_{max}}$ the minimum and maximum *Level_Index* over all vertices visited by queries at phase l. We partition the queries into $K = \lceil (K_{l_{max}} - K_{l_{min}})/d^*_{max} \rceil$ buckets $H_1, \ldots, H_K$, such that H_j, $j \in \{1, \ldots, K\}$, consists of queries that at phase l visit some vertex with *Level_Index* $\in \{i, \ldots, i'\}$, where

$$\{i, \ldots, i'\} = \begin{cases} \{K_{l_{min}} + (j-1)d^*_{max}, \ldots, K_{l_{min}} + (j+1)d^*_{max}\} & \text{if } 1 \leq j < K, \\ \{K_{l_{min}} + (K-2)d^*_{max}, \ldots, K_{l_{max}}\} & \text{if } j = K. \end{cases}$$

We note that the bucketing scheme used in the previous section (modular decomposition) does not work in this setting as the assignment of queries to buckets is realized with respect to the ordering of queries at phase l and not $l-1$ (at phase $l-1$ the queries are sorted with respect to *Level_Index*). By way of property 4 (case 2), the *Level_Index* of any two queries that are not ordered with respect to *Level_Index* at phase l may differ by at most d^*_{max}. This fact is exploited in the assignment of queries to the aforementioned "overlapped" buckets as follows.

The queries are scanned from "left" to "right", and query q is assigned to H_j or H_{j+1} depending on whether some query q' that belongs to bucket H_{j+1} but not H_j has already been encountered; if such a query q' has been encountered, then q is assigned to bucket H_{j+1}, otherwise, to bucket H_j.

The determination of $K_{l_{min}}$ and $K_{l_{max}}$ takes time $O(T_{ppf}(p))$. The partioning of queries into buckets can be performed by way of a procedure similar to that of stage 5 of algorithm MULTI_WAY SEARCH, and therefore, requires time $\max\{L, O(n/p)\} + O(C_{ppf}(p))$ and $\max\{L, O(g)\} + O(M_{ppf}(p))$ for computation and communication respectively.

The queries within $H_j \cup H_{j+1}$, for all odd $j \in \{1, \ldots, K-1\}$, are sorted independently with respect to the *Level_Index* of their associated vertex at phase l, and subsequently, the queries within $H_j \cup H_{j+1}$, for all even $j \in \{2, \ldots, K-1\}$, are sorted independently. The queries in buckets H_j and H_{j+2} are sorted with respect to each other. By way of the aforementioned sorting operations, we therefore obtain the queries at phase l globally sorted by the *Level_Index* of their associated vertex. These sorting operations can be realized by employing an algorithmic scheme similar to that of section 4.3.1. Hence, the overall time required is $O(n/p+L) + o((m^*(d^*_{max})^{2\alpha-1})/p + (d^*_{max})^{2\alpha}) + O(C_{ppf}(p))$ and $O(gn/p + g(d^*_{max})^{\alpha} + L) + O(M_{ppf}(p))$ for computation and communication respectively. We have therefore established the following analogue of proposition 4.2.

Proposition 4.4 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G^* = (V^*, E^*)$ be an R^*_{max}-ordered h^*-level graph, with vertex set $V^* = V^*_1 \cup V^*_2 \cup \ldots \cup V^*_{h^*}$ and edge set E^*, where $|V^*| = m^* \leq n$. Let the maximum out-degree of such a graph be $d^*_{max} = o(n/p)$, the maximum vertex size be $s^*_{max} = o(n/p)$ and the number of source vertices be $d^*_s \leq d^*_{max}$. If f is the next function and $\tau^*_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1, \ldots, n\}$, that for any level l, and using the charging policy of section 1.2, has,*

$$\pi = \frac{p}{\tau^*_l(f)\, n} \left(\tau^*_l(f) \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{n}{p} + L\right) + O\left(C_{ppf}(p)\right) \right),$$

and

$$\mu = \frac{p}{\tau^*_l(f)\, n} \left(O\left(g\frac{n}{p} + L\right) + M^{o(n/p)}_{brd}(p) + O\left(M_{ppf}(p)\right) \right).$$

The expression for π allows for all arithmetic as well as comparison operations. The algorithm requires $O(n/p + ((s^*_{max} + d^*_{max})m^*)/p)$ space per processor.

4.4 The Case $m > n$ through Randomization

Assume we are given an ordered h-level graph $G = (V, E)$, of maximum out-degree d_{max}, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. For the case $m > n$, the deterministic algorithm described in section 4.1 proves to be inefficient, since its running time depends on m. We remove all dependencies on m in the time bound of algorithm MULTI_WAY SEARCH by introducing randomization. In the remainder of this section, we sketch the details of the randomized algorithm.

In the randomized version, the data structure representing the ordered h-level graph is maintained in such a way that a vertex and its associated information (number of edges and edge pointers) is stored at some processor chosen uniformly at random. All information dependent on the storage address of this vertex is modified accordingly. Given this random distribution of vertices to processors, the implementation details of procedure MULTI_WAY SEARCH remain almost identical. We assume that $n/p = \omega_n r d^2_{max} \log n$, where ω_n is any function of n such that $\omega_n > 1$ for sufficiently large n, and r is the number of possible out-degree values of the underlying graph, i.e., for a balanced d-ary tree $r = 1$, since the only possible out-degree of an internal vertex is d. We note that in general, $r \leq d_{max}$.

The notable change is in the implementation of stage 1 of algorithm MULTI_WAY SEARCH; a minor modification is also required in the implementation of stage 3. Stages 2, 4-5 of algorithm MULTI_WAY SEARCH remain unmodified. In particular, we examine two cases in the analysis of stage 1 as described below. As the parameter δ of section 4.1 is irrelevant in our analysis, we fix $\delta = 1$.

[**Case 1**] If a thread $\bar{Q}$ is of size $\bar{s}$ such that $\bar{d} \leq \bar{s}$, where $\bar{d}$ is the out-degree of the vertex associated with $\bar{Q}$, a "read *next_info*" operation is realized, similar to cases 1, 2 and 3 of algorithm MULTI_WAY SEARCH. Each such thread $\bar{Q}$ receives information of size $\bar{s}$. Let the total size of all such threads, over all processors, be denoted by $S_1 \leq n$.

We next bound the number of queries that are sent to and received from any processor. Clearly, a processor can receive information of size at most $\lceil n/p \rceil$. In the remainder of the analysis of this case, we derive an upper bound on the size of information sent by any processor. Let r_{ik} be the number of threads of processor p_i such that the vertex associated with each such thread has out-degree $k = \bar{d}$, where $k \leq \bar{s}$. Then, $\sum_i r_{ik}$ is the number of such threads, over all p processors, whose associated vertices have out-degree k. Since such vertices are uniformly at random distributed among the p processors, we claim, by way of Chernoff bounds [4], that with probability $1 - n^{-\omega_n^{1/3}}$, a processor receives at most $(1 + \sqrt{3}\omega_n^{-1/3}) \max\{\sum_i r_{ik}/p, \omega_n \log n\}$ "read *next_info*" requests. For each such request, a processor needs to route information of size k. Summing over all r possible values of k, we claim that with probability $1 - rpn^{-\omega_n^{1/3}}$, which is $1 - n^{-\Omega(\omega_n^{1/3})}$, each processor routes information of size at most $(1+\sqrt{3}\omega_n^{-1/3}) \max\{\sum_k k(\sum_i r_{ik})/p, \sum_k (k\omega_n \log n)\}$. By definition we have that $\sum_{i,k}(kr_{ik}) \leq S_1$. Moreover, since $\bar{d} \leq d_{max}$, we have that $\sum_k k \leq rd_{max}$, and therefore, we obtain that with probability $1 - n^{-\Omega(\omega_n^{1/3})}$ a relation of size at most $(1+\sqrt{3}\omega_n^{-1/3}) \max\{(S_1/p), \omega_n r d_{max} \log n\}$ is routed. Thus, since $n/p = \omega_n r d_{max}^2 \log n$ and $S_1 \leq n$, we have that with high probability a relation of size at most $(1 + \sqrt{3}\omega_n^{-1/3})(n/p)$ is realized in stage 1 (cases 1, 2 and 3).

[**Case 2**] If a thread $\bar{Q}$ is of size $\bar{s}$ such that $\bar{s} < \bar{d}$, where $\bar{d}$ is the out-degree of the vertex associated with thread $\bar{Q}$, a "send $\bar{Q}$ to $\hat{p}$" operation is realized, similar to case 4 of algorithm MULTI_WAY SEARCH. Let the total size of all such threads, over all processors, be denoted by $S_2 \leq n$. It is clear that $S_2 = n - S_1$.

Each such thread $\bar{Q}$ sends information of size $\bar{s}$. We next bound the number of queries that are sent to and received from any processor. Clearly, a processor can send information of size at most $\lceil n/p \rceil$ (all the queries in its possession). In the remainder of the analysis of this case, we derive an upper bound on the size of information received by any processor. Let r_{ijk} be the number of threads of processor p_i of size $j = \bar{s}$ such that the vertex associated with each such thread has out-degree $k = \bar{d}$, where $j < k$. Then, $\sum_i r_{ijk}$ is the number of such threads, over all p processors, of size j, whose associated vertices are of out-degree k. Since such vertices are uniformly at random distributed among the p processors, we claim, by way of Chernoff bounds [4], that with probability $1 - n^{-\omega_n^{1/3}}$, a processor receives at most $(1 + \sqrt{3}\omega_n^{-1/3}) \max\{j \sum_i r_{ijk}/p, j\omega_n \log n\}$ queries. Summing over all r possible values of k and all d_{max} possible thread sizes j, we claim that with probability $1 - prdn^{-\omega_n^{1/3}}$, which is $1 - n^{-\Omega(\omega_n^{1/3})}$, each processor routes information of size at most $(1 + \sqrt{3}\omega_n^{-1/3}) \max\{\sum_{j,k}(j \sum_i r_{ijk}/p), \sum_{j,k} j\omega_n \log n\}$. By a simple combinatorial argument, we get that $\sum_{i,j,k} jr_{ijk} = S_2$. Moreover, since $\bar{s} < \bar{d} \leq d_{max}$, we have that $\sum_{j,k} j \leq rd_{max}^2$, and therefore, we obtain that with probability $1 - n^{-\Omega(\omega_n^{1/3})}$ a relation of size at most $(1 + \sqrt{3}\omega_n^{-1/3}) \max\{(S_2/p), \omega_n r d_{max}^2 \log n\}$ is routed. Thus, since $n/p = \omega_n r d_{max}^2 \log n$ and $S_2 \leq n$, we have that with high probability a relation of size at most $(1 + \sqrt{3}\omega_n^{-1/3})(n/p)$ is realized in stage 1 (case 4).

By way of the analysis of case 2, at the end of stage 1 of algorithm MULTI_WAY SEARCH, each processor maintains at most $\lceil n/p \rceil + (1 + \sqrt{3}\omega_n^{-1/3})(n/p)$ queries. We can therefore load-balance these queries, using a scheme similar to that of section 4.2, and subsequently, proceed as in stages 2-5 of algorithm MULTI_WAY SEARCH. It remains to show how to modify the sorting operation of stage 3 to cope with the fact that $m > n$.

Whereas algorithms RADIX_SORT and PARALLEL RADIX_SORT (as utilized in algorithm MULTI_WAY SEARCH) are oblivious to the size m of the underlying graph, the analysis of algorithm SUB_QUADRATIC_SORT heavily depends on m. To remove the dependency of the latter algorithm on m we substitute algorithm SUB_QUADRATIC_SORT by two applications of RADIX_SORT.

Let $\bar{Q}$ be a thread formed in the previous level and let $Index_{min}$ be the minimum *Level_Index* over all queries in $\bar{Q}$ in the current level. We define the *Index* of a query q in $\bar{Q}$ to be the *Level_Index* of the vertex associated with q minus the $Index_{min}$ of $\bar{Q}$. Clearly, *Index* is in the range $[0, \ldots, d_{max} - 1]$. Let $\bar{S}$ be the total size of all threads $\bar{Q}_j$ of size $\bar{s}_j < \sqrt{d_j}$ on any particular processor, i.e., $\bar{S} = \sum_j \bar{s}_j \leq \lceil n/p \rceil$. Each processor employs algorithm RADIX_SORT to sort the queries in $\cup_j \bar{Q}_j$ with respect to their *Index*, maintaining for each query its thread number. Thus, by way of algorithm RADIX_SORT (section 1.2), we have that the queries in all threads of size $\bar{s}_j < \sqrt{d_j}$ can be sorted in at most

$$\left\lceil \frac{\lg d_{max}}{\lg \bar{S}} \right\rceil$$

iterations of the underlying linear-time radix-based sorting algorithm, i.e., sorting of a set of $\bar{S}$ integers in the range $[0, \ldots, d_{max} - 1]$. Thus, the time required for sorting all such queries is $\max\{L, O(\bar{S} + d_{max})\}$. Finally, each processor employs once more algorithm RADIX_SORT to sort the queries with respect to their thread number. This latter operation results in globally sorting the queries in $\cup_j \bar{Q}_j$ with respect to the *Level_Index* of their associated vertex. It remains to establish the time complexity of the latter sorting operation. By way of algorithm RADIX_SORT and the fact that the number of queries is greater than the number of threads, we have that the queries can be sorted in $O(1)$ iterations of the underlying linear-time sorting algorithm. Hence, the time required for sorting all such queries is $\max\{L, O(\bar{S})\}$. The overall time required by this step is therefore bounded above by $\max\{L, O(n/p + d_{max})\}$.

These result completes the analysis of algorithm MULTI_WAY SEARCH. We have therefore established the following analogue of proposition 4.2.

Proposition 4.5 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m$. Let the maximum out-degree of such a graph be $d_{max} = o(n/p)$, the number of possible out-degree values be r, and the number of source vertices be $d_s \le d_{max}$. If ω_n is such that $\omega_n > 1$ for sufficiently large n, f is the next function, and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a randomized BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1, 2, \ldots, n\}$, that for any level l and positive integers n and p such that $n = \omega_n r d_{max}^2 p \lg n$, and using the charging policy of section 1.2, has, with probability $1 - n^{-\Omega(\omega_n^{1/3})}$, has,*

$$\pi = \frac{p}{\tau_l(f) n} \left(\tau_l(f) \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{n}{p} + L \lg p \right) + O\left(C_{ppf}(p) \right) \right),$$

and

$$\mu = \frac{p}{\tau_l(f)\, n} \left(O\left(g\frac{n}{p} + L \right) + M_{brd}^{o(n/p)}(p) + O\left(M_{ppf}(p) \right) \right).$$

The expression for π allows for all arithmetic as well as comparison operations. The algorithm requires $O(n/p + m/p)$ space per processor.

As long as h, the depth of the underlying ordered h-level graph, is a polynomial in n of degree $o(\omega_n^{1/3})$, our claims hold for all levels with high probability. We therefore obtain the following analogue of theorem 4.2.

Theorem 4.3 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$, edge set E, maximum out-degree $d_{max} = o(n/p)$, number of possible out-degree values r, and $d_s \le d_{max}$ source vertices. If f is the next function and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a randomized BSP algorithm for computing all n search processes induced by the n queries, that for all positive integers n, p, L, h and t satisfying $n = \omega(r d_{max}^2 p \lg n)$, $t = \min\{O(L/g), p\}$, $L = o(\min_l \{\tau_l(f)\} n/(p \log_t p))$ and $h = n^{O(1)}$, has, with probability $1 - n^{-\omega(1)}$, $\pi = 1 + O(h/(\sum_l \tau_l(f)))$, and if $g = o((\sum_l \tau_l(f))/h)$ also, $\mu = o(1)$. The algorithm requires $O(n/p + m/p)$ space per processor.*

Remark In the aforementioned result, the constraints $n = \omega(.)$, $L = o(.)$ and $g = o(.)$ can be removed if we relax the one-optimality requirement. In that case, $n = \Omega(.)$ $L = O(.)$ and $g = O(.)$ would suffice to obtain $\pi = O(1)$ and $\mu = O(1)$.

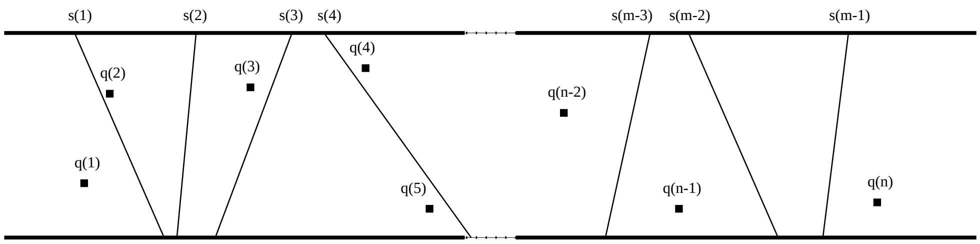

Figure 4: The multi-way point-location problem.

5 Multi-Way Point-Location through Multi-Way Search

We now apply the methods we developed in the previous sections to indicate an efficient BSP solution to an important computational geometry problem.

Problem. Given a set of n points (queries) $Q = \{q_1, q_2, \ldots, q_n\}$ and an ordered partitioning Y of a two-dimensional horizontal strip V by $m-1$ non-intersecting line segments $S = \langle s_1, s_2, \ldots, s_{m-1} \rangle$ into m slabs, determine, for each $q \in Q$, the vertical slab it belongs to.

The solution to this problem, which we call *multi-way point-location* (see figure 4 for an illustration), is central to several other geometric problems [28], i.e., planar point-location, plane sweep, and segment trees.

A direct sequential solution to the multi-way point-location problem is to perform, for each $q \in Q$, *binary-search* on the m slabs. Clearly, this method requires time proportional to $n \lg m$.

In the remainder of this section we assume $m = n$, the case $n > m$ can be derived in a similar way. The algorithm for solving the multi-way point-location problem consists of two phases. In the first phase we construct an ordered h-level graph G based on an ordered partitioning Y of the two-dimensional horizontal strip V. In particular, we construct a balanced d-ary tree on the n vertical slabs, where $d = \lceil (n/p)^\epsilon \rceil$ for any positive constant $\epsilon < 1$. In order to simplify the exposition, we assume that n is an integral power of $d \geq 2$; this assumption can be removed without affecting the asymptotic performance of our algorithms. In the second phase we utilize algorithm MULTI_WAY SEARCH, to determine, for each $q \in Q$, the slab it belongs to. The following claim shows how to efficiently implement the first phase.

Lemma 5.1 *For any constant ϵ, $0 < \epsilon < 1$, there exists a BSP algorithm for constructing a balanced d-ary tree T on n ordered elements, that for all positive integers n, p and d satisfying $d = \lceil (n/p)^\epsilon \rceil$ requires time* $\max\{L, O(n/p)\} + \lceil \log_d p \rceil \max\{L, O(d)\}$ *and* $\lceil \log_d p \rceil \max\{L, O(gd)\})$ *for computation and communication respectively.*

Proof: Let l be an arbitrary level of T. We define every d-th key of level l of T to be a *pivot* key. The construction of the balanced tree proceeds from the leaves to the root as follows. For each level l in T we construct level $l-1$ by only considering the pivot keys of level l. Since the height of T is $\lceil \log_d n \rceil$ the construction takes $\lceil \log_d n \rceil$ phases, of which $\lceil \log_d p \rceil$ phases are non-local (require inter-processor communication). Each such phase takes time $\max\{L, O(d)\}$ and $\max\{L, O(gd)\}$ for computation and communication respectively. Since the overall cost of local computations is bounded above by $\max\{L, O(n/p)\}$, the lemma follows. ■

The so constructed balanced tree T has n leaves and out-degree $d = \lceil (n/p)^\epsilon \rceil$, and therefore, its height is at most $\lceil \lg n / (\epsilon \lg (n/p)) \rceil$. Moreover, since the in-degree of any vertex $v \in T$ is at most one and the number of internal vertices is at most $n-1$, the number of edges in T is bounded above by $2n-1$. Combining these observations with lemma 5.1 and theorem 4.2 we obtain the following result, which summarizes the discussion of this section.

Theorem 5.1 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n points and let Y be an ordered partitioning of a two-dimensional horizontal strip V, such that $|Y| \leq n$. If ϵ is a positive constant such that $0 < \epsilon < 1$, then,* *there exists a BSP algorithm for solving the multi-way point-location problem induced by the n queries, that for all positive integers n, p, L, d and t satisfying $t = \min\{O(L/g), p\}$ $L = o(n \lg d/(p \log_t p))$ and $d = (n/p)^\epsilon$, has, $\pi = 1 + o(1)$, and if $g = o(\lg d)$ also, $\mu = o(1)$. The algorithm requires $O(n/p)$ space per processor.*

Remark. In the aforementioned result, the constraints $L = o(.)$ and $g = o(.)$ can be removed if we relax the one-optimality requirement. In that case, $L = O(.)$ and $g = O(.)$ would suffice to obtain $\pi = O(1)$ and $\mu = O(1)$.

6 BSP Randomized Sorting

We now apply the methods we developed in the previous sections to obtain an efficient BSP algorithm for the problem of sorting n keys, that achieves one-optimality for a wide range of the parameters n, p, L and g. The algorithm derives from the ideas of [13, 31], and utilizes the notions of *ordered h-level graphs* [11, 16] and the technique of *over-sampling* [14, 30].

6.1 Random Sampling

In this section we present results related to random sampling. In the discussion and proofs to follow we use the notation of [14]. Let $X = \{x_1, x_2, \ldots, x_N\}$ be an ordered set of keys indexed such that $x_i < x_{i+1}$, for all $1 \le i \le N-1$. The implicit assumption is that keys are unique since we can always make them so by appending to them their index position. Let $Y = \{y_1, y_2, \ldots, y_{ks-1}\}$ be a randomly chosen subset of $ks - 1 \le N$ keys of X also indexed such that $y_i < y_{i+1}$, for all $1 \le i \le ks-2$, for some positive integers k and s. Having randomly selected set Y we employ it to partition the set $X - Y$ into k subsets, $X_0, X_1, \ldots, X_{k-1}$, where,

$$\begin{aligned} X_0 &= \{x \mid x \in X \text{ and } x < y_s\}, \\ X_i &= \left\{x \mid x \in X \text{ and } y_{is} < x < y_{(i+1)s}\right\}, \text{ for all } 1 \le i \le k-2, \\ X_{k-1} &= \left\{x \mid x \in X \text{ and } y_{(k-1)s} < x\right\}. \end{aligned}$$

Let $Q_i(j)$ be the probability that $y_i = x_j$. Then, for all $1 \le i \le ks-1$ and $1 \le j \le N$ the following equality holds [14].

$$Q_i(j) = \frac{\binom{j-1}{i-1}\binom{N-j}{ks-1-i}}{\binom{N}{ks-1}},$$

where for the binomial coefficients, we assume that $\binom{i}{j} = 0$ if $j < i$. Let N_i denote the size of X_i and $P_i(j)$ denote the probability that $N_i = j$, where $0 \le i \le k-1$ and $s - 1 \le j \le N - (k-1)s$. The following result is shown in [14].

Claim 6.1 *For all i,*

$$P_i(j) = P(j) = \frac{\binom{j}{s-1}\binom{N-j-1}{(k-1)s-1}}{\binom{N}{ks-1}}.$$

We now derive a lower bound on the size of the X_i's by proving that the probability of having a set of size less than $(1-\varepsilon)(N-k+1)/k$, $0 < \varepsilon < 1$, is very small. We first derive the following result.

Claim 6.2 *Let $k \ge 2$, $s \ge 1$, $ks < N/2$, $n \ge 1$, $0 < \varepsilon < 1$, $\rho > 0$, and*

$$s \ge \frac{1-\varepsilon}{\varepsilon^2}\left(3\rho \log n + \log\left(k^3 (2\pi(ks-1))^{3/2} e^{1/(2(ks-1))}\right)\right).$$

Then the probability that any one of the X_i, for all i, $0 \le i \le k-1$, is of size less than $\lfloor (1-\varepsilon)(N-k+1)/k \rfloor$ is at most $n^{-\rho}$.

Proof: In order to establish an upper bound for the term

$$\sum_{j \leq \lfloor (1-\varepsilon)(N-k+1)/k \rfloor - 1} P(j)$$

we approximate $P(j)$ by a binomial term and we use bounds on the right tail of the binomial distribution. The proof utilizes a line of arguments similar to that of [14]. Let $B = \lfloor (1-\varepsilon)(N-k+1)/k \rfloor - 1$ and $q = (ks-1)/N$. The probability that in N Bernoulli trials, each with a probability q of success we get $s-1$ successes among the first j, a success at the $(j+1)$-st trial, and $(k-1)s-1$ successes in the remaining trials is

$$R(j) = \binom{j}{s-1} q^{s-1}(1-q)^{j-s+1}\; q \binom{N-j-1}{(k-1)s-1} q^{(k-1)s-1}(1-q)^{N-j-(k-1)s}.$$

Let A be the ratio of $R(j)$ to $P(j)$. Then, by way of claim 6.1, we obtain that

$$A = \frac{R(j)}{P(j)} = \binom{N}{ks-1} q^{ks-1}(1-q)^{N-ks+1}. \tag{1}$$

We next state a fact deducible from Stirling's approximation formula [9] that holds for $a = c/d$, $b = 1-a$, $c < d/2$, and gives that

$$e^{-1/(6c)} \frac{1}{\sqrt{2\pi abd}} \left(a^a b^b\right)^{-d} \leq \binom{d}{c}.$$

Substituting for $d = N$, $c = ks-1$, $a = q = (ks-1)/N$ and $b = 1-(ks-1)/N$ we get the following inequality.

$$e^{-1/(6(ks-1))} \frac{1}{\sqrt{2\pi(ks-1)(1-\frac{ks-1}{N})}}\; q^{-ks+1}\,(1-q)^{-N+ks-1} \leq \binom{N}{ks-1}.$$

Then, we derive, by way of equation 1, that the following inequality holds.

$$A \geq e^{-1/(6(ks-1))} \frac{1}{\sqrt{2\pi(ks-1)(1-\frac{ks-1}{N})}}. \tag{2}$$

Since by definition $R(j) \leq \binom{j}{s-1} q^{s-1}(1-q)^{j-s+1} q$ and $P(j) = R(j)/A$ the following inequality also holds.

$$\sum_{j \leq \lfloor (1-\varepsilon)\frac{N-k+1}{k} \rfloor - 1} P(j) \leq \frac{q}{A} \sum_{j \leq \lfloor (1-\varepsilon)\frac{N-k+1}{k} \rfloor - 1} \binom{j}{s-1} q^{s-1}(1-q)^{j-s+1}. \tag{3}$$

An upper bound for the following term is to be established.

$$\sum_{j \leq B} \binom{j}{s-1} q^{s-1}(1-q)^{j-s+1}.$$

The technique to be described employs generating functions. The term $\binom{j}{s-1} q^{s-1}(1-q)^{j-s+1}$ is the coefficient of x^{s-1} in the polynomial $((1-q)+qx)^j$. The sum of these terms for $j \leq B$ is the coefficient of x^{s-1} in the following sum.

$$\sum_{j \leq B} ((1-q)+qx)^j = \frac{((1-q)+qx)^{B+1}}{-q(1-x)} + \frac{1}{q(1-x)}.$$

The coefficient of x^{s-1} in this expression is

$$\frac{1}{q}-\frac{1}{q}\sum_{i=0}^{s-1}\binom{B+1}{i}q^i(1-q)^{B+1-i}=\frac{1}{q}\sum_{i=s}^{B+1}\binom{B+1}{i}q^i(1-q)^{B+1-i}.$$

The sum in the term above gives the probability of having at least s successes in $B+1$ independent Bernoulli trials with probability of success q. Chernoff bounds [4] on the probability of having at least $(1+\beta)(B+1)q$ successes gives the upper bound of

$$e^{-(\beta^2(B+1)q)/3}.$$

By setting $s=(1+\beta)(B+1)q$ in this bound we obtain

$$e^{-(s-(B+1)q)^2/(3(B+1)q)}.$$

We first derive a lower bound for the term $s-(B+1)q$, for $0<\varepsilon<1$ and $ks\le N/2$.

$$\begin{aligned} s-(B+1)q &= s-\left(\left\lfloor(1-\varepsilon)\frac{N-k+1}{k}\right\rfloor\right)\frac{ks-1}{N} \\ &\ge s-s(1-\varepsilon)+\frac{(1-\varepsilon)(k-1)sk+(N-k+1)(1-\varepsilon)}{kN} \\ &\ge \varepsilon s. \end{aligned}$$

An upper bound for $(B+1)q$ is derived similarly.

$$\begin{aligned} (B+1)q &\le \left(\left\lfloor(1-\varepsilon)\frac{N-k+1}{k}\right\rfloor\right)\frac{ks-1}{N} \\ &\le s(1-\varepsilon)+\frac{-ks(k-1)(1-\varepsilon)-(1-\varepsilon)(N-k+1)}{kN} \\ &\le (1-\varepsilon)s. \end{aligned}$$

The combination of these two bounds gives

$$e^{-\beta^2(B+1)q/3}\le e^{-\varepsilon^2 s/(3(1-\varepsilon))}.$$

We thus obtain the following inequality for the sum of interest.

$$\begin{aligned} \sum_{j\le B}\binom{j}{s-1}q^{s-1}(1-q)^{j-s+1} &= \frac{1}{q}\sum_{i=s}^{B+1}\binom{B+1}{i}q^i(1-q)^{B+1-i} \\ &\le \frac{1}{q}e^{-\varepsilon^2 s/(3(1-\varepsilon))}. \end{aligned} \tag{4}$$

The desired bound follows from equations 2, 3 and 4, and is given below.

$$\sum_{j\le B}P(j) \le \sqrt{2\pi(ks-1)}\,e^{1/(6(ks-1))}\,e^{-\varepsilon^2 s/(3(1-\varepsilon))}. \tag{5}$$

The probability that a given set, among the k possible ones, is of size less than $\lfloor(1-\varepsilon)(N-k+1)/k\rfloor$, by equation 5, is bounded above by the following term.

$$\sqrt{2\pi(ks-1)}\,e^{1/(6(ks-1))}\,e^{-\varepsilon^2 s/(3(1-\varepsilon))}.$$

The probability that any set is so is at most k times this term. Substituting for the value of s, we get that the probability of having some set of size less than $\lfloor(1-\varepsilon)(N-k+1)/k\rfloor$ is at most $n^{-\rho}$, and the claim follows. ■

The following is also proved in [14].

Claim 6.3 *Let $k \geq 2$, $s \geq 1$, $ks < N/2$, $n \geq 1$, $0 < \varepsilon < 1$, $\rho > 0$, and*

$$s \geq \frac{1+\varepsilon}{\varepsilon^2}\left(2\rho \log n + \log\left(2\pi k^2 (ks-1) e^{1/(3(ks-1))}\right)\right).$$

Then the probability that any one of the X_i, for all i, $0 \leq i \leq k-1$, is of size more than $\lceil (1+\varepsilon)(N-k+1)/k \rceil$ is at most $n^{-\rho}$.

6.2 The Randomized Sorting Algorithm

We present in this section a new randomized BSP sorting algorithm that achieves one-optimality in computation for an ample range of the parameters n, p and L, and also achieves one-optimality in communication if g is appropriately bounded.

The algorithm is based on an efficient partitioning scheme and is recursive in nature. In particular, the algorithm realizes m recursive calls, where m may be equal to 1 or 2 in most practical situations. In each iteration i, $k_i - 1$ splitters are selected from the current set of size N_i, where $k_i = \lfloor (\lfloor \sqrt{N_i} \rfloor + 1)/s \rfloor$, and this set is partitioned into k_i buckets of approximately the same size. The next iteration performs a similar process of sub-partitioning each of these buckets into k_{i+1} further buckets, and so on. The size of every bucket produced in any of the m iterations is within a factor of $(1+\varepsilon)$ of the mean N_i/k_i with high probability. Therefore, for $\varepsilon = 1/(m\omega_n)$ and ω_n any function of n such that $\omega_n > 2$ for sufficiently large n, the size of any subset produced in the m iterations of the underlying sorting algorithm is within a factor of $(1 + 1/(m\omega_n))^m \leq e^{1/\omega_n}$ of the mean.

While the algorithm described in this section is not complicated itself, description of its performance in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables software to be written once and for all that can be migrated efficiently among a variety of parallel machines. The remainder of this section is devoted to the derivation of the following proposition.

Proposition 6.1 *Let ω_n be any function of n such that $\omega_n > 2$ for sufficiently large n. Then, for any positive integers n and p such that $n/p = \Omega(\lg n / \lg\lg n)$, there exists a randomized BSP sorting algorithm that for any δ such that $1 < \delta < (\lg\lg n - 1)/2$, and using the charging policy of section 1.2, has, with probability $1 - o(1)$,*

$$\pi \leq \frac{p}{n \lg n}\left(\left(1 + \frac{2}{\omega_n}\right)\frac{n}{p}\lg n + O\left(\frac{n}{p}\left(\frac{\lg n}{\lg \frac{n}{p}} + \lg\lg^4 n\right) + L\left(\frac{\lg^2 n}{\lg \frac{n}{p}} + \lg\lg^4 n\right)\right)\right),$$

and

$$\mu \leq \frac{gp}{n \lg n}\left(O\left(\frac{n}{p}\left(\frac{\lg n}{\lg \frac{n}{p}} + \lg\lg^4 n\right) + \max\left\{\frac{L}{g}, 1\right\}\left(\frac{\lg^2 n}{\lg \frac{n}{p}} + \lg\lg^4 n\right)\right)\right).$$

The expression for π allows for all arithmetic as well as comparison operations.

Remark. The success probability of the algorithm implied in the aforementioned proposition is at least

$$1 - m\left(n^{-\Omega\left((n/p)\lg\lg^2 n\right)} + n^{-\Omega\left(\lg^{\delta-1} n/(m\omega_n)^2\right)}\right).$$

Proof: The algorithm is recursive in nature, and each recursive call is divided into a sequence of stages. Initially, the variables N, l and r have the values $N = n$, $l = 0$ and $r = p-1$ respectively. In any recursive call of the algorithm, these variables give respectively the size of the problem, the leftmost processor and the rightmost processor participating in the call. The oversampling factor s is set to $\lceil \lg^{\delta} n \rceil$, for any δ such that $1 < \delta < (\lg\lg n - 1)/2$. The input is assumed to be evenly distributed among the p processors at the beginning of the execution of the algorithm. Moreover, we assume that the keys are distinct since we can always make them so by, for example, appending to them the code for the memory location in which they are stored. We note that the algorithm requires subset processor synchronization.

Program RAND_SORT (N, l, r)

begin

Stage 1 :

If $(l = r)$ or $(l \neq r$ and $N \leq \lg^{\lg\lg n} n)$, then the $q = r - l + 1$ processors participating in this recursive call of the algorithm sort the input by some sequential deterministic sorting method in the former case and some deterministic parallel sorting method in the latter case. Subsequently, the algorithm terminates.

[**Analysis of Stage 1:** The sorting procedure is realized in one of two ways depending on the values of l, r and N.

[**Case** $(l = r)$] Processor l sorts, by using a deterministic sequential sorting algorithm, the keys in its possession. The analysis of this stage is treated in detail in the discussion following the description of this algorithm.

[**Case** $(l \neq r$ **and** $N \leq \lg^{\lg\lg n} n)$] The $q = l - r + 1$ processors participating in this recursive call of the algorithm sort the keys in their possession, by using an adaptation on the BSP model of odd-even merge-sort [6].]

Stage 2 :

The q processors participating in this recursive call of the algorithm, choose at random without replacement a sample of size $ks - 1 = \lfloor(\lfloor\sqrt{N}\rfloor + 1)/s\rfloor s - 1 \leq \lfloor\sqrt{N}\rfloor$.

[**Analysis of Stage 2:** The sample selection method is as follows.

(1) Each processor selects uniformly at random with replacement $\lceil(ks-1)/q\rceil$ (or $\lfloor(ks-1)/q\rfloor$) processor identifiers for a total, over all processors, of $\lfloor\sqrt{N}\rfloor$ identifiers. The total number of identifiers selected by any processor is bounded above by $\lceil(ks-1)/q\rceil \leq \lceil\lfloor\sqrt{N}\rfloor/q\rceil$.

(2) The processor identifiers selected in the previous step are routed to the so identified processors.

(3) Processor p_i that received t_i identifiers in the previous step, chooses uniformly at random without replacement a set of size t_i among its keys.

(4) A parallel-prefix sum operation is performed on the number t_i of selected keys.

(5) Using the information of the previous step the keys are distributed evenly among the q participating processors.

Step 1 requires $\max\{L, O(\lceil\sqrt{N}/q\rceil)\}$ time for random processor identifier generation. In order to upper bound the number of processor identifiers received by any processor in step 2, we use the generalized Chernoff bound [9] $\text{Prob}[S_{n,P} \geq eunP/\log u] \leq e^{-unP}$, for $u \geq e$ and $u^2nP \geq \log u$, where $S_{n,P}$ is the number of successes in n independent Bernoulli trials with individual probability of success P. Substituting for $n = \lfloor\sqrt{N}\rfloor$, $P = 1/q$ and $u = \Theta(\sqrt{N}\lg\sqrt{N})$, we obtain that with probability at least

$$1 - qe^{-\Omega\left(N\lg\sqrt{N}/q\right)}$$

no processor receives more than $\Theta(N/q)$ processor identifiers. Therefore, step 2 requires time $\max\{L, \Theta(gN/q)\}$ with the corresponding high probability. Similarly, the random key selection in step 3 requires time $\max\{L, \Theta(N/q)\}$. The parallel-prefix sum operation of step 4, by lemma 2.2, takes time $C_{ppf}(q)$ and and $M_{ppf}(q)$ for computation and communication respectively. The final distribution of keys in step 5 requires time $\max\{L, \Theta(gN/q)\}$. Thus, the overall computation and communication time required for this stage is bounded above by, $\max\{L, \Theta(\lceil\sqrt{N}/q\rceil)\} + \max\{L, \Theta(N/q)\} + C_{ppf}(p)$ and $2\max\{L, \Theta(gN/q)\} + M_{ppf}(p)$ respectively, with the stated probability.]

Stage 3 :

The sample of size $\lfloor\sqrt{N}\rfloor$ is sorted using a BSP adaptation of an algorithm in [27].

[**Analysis of Stage 3:** The sorting method follows. Consider input set $X = \{x_1, x_2, \ldots, x_{\lfloor\sqrt{N}\rfloor}\}$ of size $\lfloor\sqrt{N}\rfloor$ and $\lfloor\sqrt{N}\rfloor^2$ processors. We number processors in a mesh-like fashion, so that we can refer to processor at row i and column j, as $p_{i,j}$, $i, j \in \{1, 2, \ldots, \lfloor\sqrt{N}\rfloor\}$. Input keys are assumed to be distributed among the processors in the first row, i.e., key x_j resides in the local memory of processor $p_{1,j}$. Executing

$\lfloor\sqrt{N}\rfloor$ distinct broadcast operations, processor $p_{1,j}$, for all $j \in \{1, 2, \ldots, \lfloor\sqrt{N}\rfloor\}$, broadcasts key x_j to all processors of column j. Similarly, processor $p_{i,i}$, for all $i \in \{1, 2, \ldots, \lfloor\sqrt{N}\rfloor\}$, broadcasts key x_i to all processors of row i. In a single superstep, processor $p_{i,j}$, $i, j \in \{1, 2, \ldots, \lfloor\sqrt{N}\rfloor\}$, compares x_i and x_j, and subsequently, by executing $\lfloor\sqrt{N}\rfloor$ distinct parallel sum operations determines the rank $\mathrm{rank}(x_i : X)$ of each x_i in X. The sorting operation is completed by routing each key to its final destination.

When this step is executed there are $\lfloor\sqrt{N}\rfloor$ sample keys and q processors. Therefore, each physical processor in the above routine is assigned a $\lceil\sqrt{N/q}\rceil \times \lceil\sqrt{N/q}\rceil$ segment of computation. We get that this stage takes

$$\max\left\{L, O\left(\frac{N}{q}\right)\right\} + C_{ppf}^{\lfloor\sqrt{N/q}\rfloor}(\sqrt{q}),$$

and

$$\max\left\{L, O\left(g\frac{\sqrt{N}}{q}\right)\right\} + 2M_{brd}^{\lfloor\sqrt{N/q}\rfloor}(\sqrt{q}) + M_{ppf}^{\lfloor\sqrt{N/q}\rfloor}(\sqrt{q}),$$

time for computation and communication respectively.]

Stage 4 :
A balanced d-ary tree T is constructed over the $k = \lfloor(\lfloor\sqrt{N}\rfloor + 1)/s\rfloor$ buckets induced by the $k-1$ sorted splitters that evenly partition the sample.
[**Analysis of Stage 4:** We set $d = 2\lceil(N/q)^\alpha\rceil$, for any $\alpha < 1$. In order to simplify the exposition we assume that k is an integral power of d. This assumption can be removed without affecting the asymptotic analysis of the algorithm. The balanced d-ary tree construction proceeds as follows. Let h be the height of the tree, and assume that the k leaves of T lie on level h. Since the height of T is $\log_d k$ the construction requires $\log_d k$ phases, of which $\lfloor\log_d (k/q)\rfloor$ can be performed by only considering keys in the local processor memories. The total time required by these local phases is bounded by $\max\{L, O(k/q)\}$. The remaining $\log_d k - \lfloor\log_d (k/q)\rfloor$ phases require interprocessor communication and each such phase takes time $\max\{L, O(d)\}$ and $\max\{L, O(gd)\}$ for computation and communication respectively. Thus, the total computation and communication time complexity of this stage is bounded respectively, as $k \le q$, by $\max\{L, O(k/q)\} + \lceil\log_d q\rceil \max\{L, O(d)\}$ and $\lceil\log_d q\rceil \max\{L, O(gd)\}$.]

Stage 5 :
Each processor decides the bucket destination of every key it holds with respect to the $k-1$ splitters.
[**Analysis of Stage 5:** This stage can be realized by employing the algorithm implied by proposition 4.2. Given that the height of the balanced d-ary tree of stage 4 is at most $\log_d k$ and by substituting in proposition 4.2, for $n = N$, $m = k$ and $p = q$ we get that this stage requires time $\max\{L, \lg k\lceil N/q\rceil\} + \log_d k\ O(N/q + L) + \log_d k O(C_{ppf}(q))$ and $\log_d k\ O(gN/q + L) + \log_d k O(M_{ppf}(q))$ for computation and communication respectively.
]

Stage 6 :
The N input keys are redistributed according to their bucket destination. If $q/k \ge 1$ we assign $\lfloor q/k\rfloor$ processors per bucket, otherwise we assign $\lceil k/q\rceil$ buckets per processor.
[**Analysis of Stage 6:** We assume that q (respectively, k) divides k (respectively, q). This assumption can be removed without affecting the asymptotic analysis of the algorithm, i.e., by assuming that p is a power of two and that the number of splitters selected in stage 4 is a power of two as well. This stage is equivalent to one of integer sorting, where the N integers to be sorted belong to the range $[0, 1, \ldots, k-1]$. Thus, the keys can be redistributed according to their bucket destination by employing the radix-sort algorithm implied in lemma 2.3. Substituting for $n = N$, $p = q$, $R = k$ and $r = \lceil\lg k / \lg\max\{\lceil N/q\rceil, t\lceil\log_t q\rceil\}\rceil$ in lemma 2.3 gives that this stage takes time $3r\max\{L, O(\max\{\lceil N/q\rceil, t\lceil\log_t q\rceil\})\} + rC_{ppf}(q)$ and $2r\max\{L, g\max\{\lceil N/q\rceil, t\lceil\log_t q\rceil\}\} + r\max\{L, g\lceil N/q\rceil\} + rM_{ppf}(q)$ for computation and communication respectively.]

Stage 7 :
Recursively sort the keys in each bucket, for all buckets concurrently.

end

Specification of π and μ. We start by showing the following bound on the size N_i of a bucket in the i-th recursive call of the algorithm. We assume $N_0 = n$.

Claim 6.4 *Let ω_n be any function of n such that $\omega_n > 2$ for sufficiently large n. Let m be the depth of the recursion of algorithm* RAND_SORT, *$n > 256$, $1 < \delta < (\lg\lg n - 1)/2$, and $N_i \geq \lg^{\lg\lg n} n$. Then,*

$$Prob\left[N_i \leq (1+\varepsilon)^{2-1/2^{i-1}} N_0^{1/2^i} \left\lceil \lg^\delta n \right\rceil^{2-1/2^{i-1}} + 2(1+\varepsilon)^2 \left\lceil \lg^\delta n \right\rceil^2 i\right] \geq 1 - n^{-\Omega\left(\lg^{\delta-1} n/(m\omega_n)^2\right)},$$

where $\varepsilon = 1/(m\omega_n)$.

Proof: Consider an arbitrary recursive call i of algorithm RAND_SORT, such that $N_i \geq \lg^{\lg\lg n} n$. By definition, the number of buckets generated in the i-th recursive call is $k_i = \lfloor(\lfloor\sqrt{N_i}\rfloor + 1)/s\rfloor$. Thus, by claim 6.3, for $\varepsilon = 1/(m\omega_n)$, $k = k_i$ and $s = \lceil \lg^\delta n\rceil$, with probability at least $1 - n^{-\Omega\left(\lg^{\delta-1} n/(m\omega_n)^2\right)}$, in the $(i+1)$-st recursive call, we have that $N_{i+1} \leq \lceil (1+\varepsilon)(N_i - k_i + 1)/k_i\rceil + 1$. By substituting in this inequality, for $k_i = \lfloor(\lfloor\sqrt{N_i}\rfloor + 1)/s\rfloor$, we obtain that $N_{i+1} \leq (1+\varepsilon)(s\sqrt{N_i} + 2s^2)$, for all n such that $n > 256$. An inductive argument on i, shows that in the i-th recursive call,

$$N_i \leq (1+\varepsilon)^{2-1/2^{i-1}} N_0^{1/2^i} s^{2-1/2^{i-1}} + 2(1+\varepsilon)^2 s^2,$$

where $N_0 = n$. By substituting in this expression for N_0, ε and s we get the stated bound. ■

Hence, we conclude the following on the depth of the recursion.

Claim 6.5 *Algorithm* RAND_SORT *realizes at most*

$$\left\lceil \lg\lg n - \lg\left(\lg\left(\max\left\{\lg^{\lg\lg n} n, \Omega\left(\frac{n}{p}\right)\right\}\right) - (2\delta+1)\lg\lg n - 1\right)\right\rceil$$

recursive calls.

Proof: Let m be the depth of the recursion. From the discussion of claim 6.4 we have that in the i-th recursive call,

$$N_i \leq (1+\varepsilon)^{2-1/2^{i-1}} N_0^{1/2^i} s^{2-1/2^{i-1}} + 2(1+\varepsilon)^2 s^2,$$

where $N_0 = n$, $\varepsilon = 1/(m\omega_n)$ and $s = \lceil \lg^\delta n\rceil$. For $i \leq \lg n$ and $n > 256$ the following inequalities hold.

$$N_i \leq (1+\varepsilon)^{2-1/2^{i-1}} N_0^{1/2^i} s^{2-1/2^{i-1}} + 2(1+\varepsilon)^2 s^2 < 2n^{1/2^i} \lg^{(2\delta+1)} n. \quad (6)$$

For all $i \leq m$, since $\omega_n > 2$, we have that $(1-\varepsilon)^i \geq (1-\varepsilon)^m = (1 - 1/(m\omega_n))^m > (1 - 1/(2m))^m \geq (1 - 1/(2m))^{2m-1} \geq e^{-1} > 1/3$. By employing arguments similar to those to be used later in this section to prove equation 8, we can show that, for all $i \leq \lg n$,

$$\frac{N_i}{q_i} = \Omega\left(\frac{n}{p} - \min\left\{\frac{n}{p}, \lg n\right\} + 1\right).$$

Moreover, for all i such that $i < m$, we have, by way of stages 1 and 6 of algorithm RAND_SORT, that

$$N_i \geq \max\left\{\lg^{\lg\lg n} n, \Omega\left(\frac{n}{p}\right)\right\}.$$

By combining the previous observations and solving inequality

$$2n^{1/2^i} \lg^{(2\delta+1)} n < \max\left\{\lg^{\lg\lg n} n, \Omega\left(\frac{n}{p}\right)\right\}$$

with respect to i, the stated upper bound on the number of recursive calls is derived. ■

We proceed to deriving upper and lower bounds on the number of keys per processor at the i-th recursive call, in order to establish upper bounds on the computation and communication time of the algorithm. Let parameters N_i, q_i and k_i denote, the number of keys, the number of processors, and the number of splitters in the i-th recursive call of algorithm RAND_SORT respectively. Clearly, we have that $N_0 = n$ and $q_0 = p$. In the analysis of stage 6 we assumed that we selected the number of splitters in such a way that k_i divides q_i by choosing p and k_i to be powers of two. We can choose such a k_i (number of splitters) by possibly selecting in stage 2 a sample of size at most twice the stated one in the analysis of the algorithm. This would only affect lower order terms in the analysis.

We first note that in the $(i-1)$-st recursive call we have q_{i-1} processors assigned to a bucket. In the next recursive call, we select $k_{i-1} - 1$ splitters that split such a bucket into k_{i-1} buckets and $q_i = q_{i-1}/k_{i-1}$ processors are assigned to each such newly formed bucket. Although $q_{i-1} > k_{i-1}$ in general, it is, however, possible that in the last recursive call $q_{i-1} \leq k_{i-1}$.

By way of claim 6.3, we have that $N_{i+1} \leq \lceil (1+\varepsilon)(N_i - k_i + 1)/k_i \rceil + 1$, with high probability, and therefore, we conclude that $N_{i+1} \leq (1+\varepsilon)N_i/k_i + 2$. Similarly, by way of claim 6.2, we have that $N_{i+1} \geq \lfloor (1-\varepsilon)(N_i - k_i + 1)/k_i \rfloor + 1$, and thus, we conclude that $N_{i+1} \geq (1-\varepsilon)N_i/k_i - 1$. We proceed to establishing upper and lower bounds on the ratio N_i/q_i.

Since $N_{i+1} \leq (1+\varepsilon)N_i/k_i + 2$ and $q_i = q_{i-1}/k_{i-1}$, we have that $N_i/q_i \leq (1+\varepsilon)N_{i-1}/(q_i k_{i-1}) + 2/q_i$, and therefore, $N_i/q_i \leq (1+\varepsilon)N_{i-1}/q_{i-1} + 2k_{i-1}/q_{i-1}$. This recurrence relation gives that

$$\frac{N_i}{q_i} \leq (1+\varepsilon)^i \frac{N_0}{q_0} + 2\sum_{j=0}^{i-1} (1+\varepsilon)^{i-1-j} \frac{k_j}{q_j}.$$

By way of stages 1 and 6 of algorithm RAND_SORT, it is true that $k_i < q_i$, except possibly for the last recursive call. Thus, for all $i \leq m$, we have that $(1+\varepsilon)^{i-1-j} \leq (1+\varepsilon)^m = (1 + 1/(m\omega_n))^m \leq e^{1/\omega_n} < e^{1/2} < 2$, since $\omega_n > 2$. By claim 6.5 we have that $m \leq \lceil \lg\lg n \rceil$ and we thus conclude that for all i such that $k_i < q_i$ the following inequality holds.

$$\frac{N_i}{q_i} \leq (1+\varepsilon)^i \frac{n}{p} + 4\lceil \lg\lg n \rceil . \tag{7}$$

Similarly, for all recursive calls, except possibly for the last one, we can prove a corresponding lower bound. Since $N_{i+1} \geq (1-\varepsilon)N_i/k_i - 1$ and $q_i = q_{i-1}/k_{-1}$, we obtain using arguments similar to those employed for the derivation of the upper bound, that for all i such that $k_i < q_i$,

$$\frac{N_i}{q_i} \geq (1-\varepsilon)^i \frac{n}{p} - \lceil \lg\lg n \rceil . \tag{8}$$

The former analysis applies to all but (possibly) the last recursive call, since $k_{m-1} \geq q_{m-1}$ is possible. The following analysis handles the case $k_{m-1} \geq q_{m-1}$. In the last recursive call, we have k_{m-1} buckets and q_{m-1} processors. Each processor is assigned to k_{m-1}/q_{m-1} buckets. Since $N_m \leq (1+\varepsilon)N_{m-1}/k_{m-1} + 2$, the size of all k_{m-1}/q_{m-1} such buckets on any one processor is at most $N_m k_{m-1}/q_{m-1} \leq (1+\varepsilon)N_{m-1}/q_{m-1} + 2k_{m-1}/q_{m-1}$, which by equation 7 and the fact that $(1+\varepsilon) < 2$ is bounded above as follows.

$$N_m k_{m-1}/q_{m-1} \leq (1+\varepsilon)^m \frac{n}{p} + 8\lceil \lg\lg n \rceil + 2k_{m-1}/q_{m-1}. \tag{9}$$

Similarly, a corresponding lower bound can be derived.

$$N_m k_{m-1}/q_{m-1} \geq (1-\varepsilon)^m \frac{n}{p} - \lceil \lg\lg n \rceil - k_{m-1}/q_{m-1}. \tag{10}$$

We next derive an upper bound for the term k_{m-1}/q_{m-1}. By definition, we have that $k_{m-1} = \lfloor (\lfloor \sqrt{N_{m-1}} \rfloor + 1)/s \rfloor$, and thus, $k_{m-1} \leq (\sqrt{N_{m-1}} + 1)/s$. We also have that $N_m \leq (1+\varepsilon)N_{m-1}/k_{m-1} + 2$. Since $N_{i+1} \leq (1+\varepsilon)N_i/k_i + 2$ and $k_{m-2}/q_{m-2} \leq 1$, we obtain by equation 7 the following bound.

$$
\begin{aligned}
\frac{k_{m-1}}{q_{m-1}} &\leq \frac{(\sqrt{N_{m-1}}+1)/s}{q_{m-2}/k_{m-2}} \\
&\leq \frac{\left(\sqrt{(1+\varepsilon)(N_{m-2}/k_{m-2})+2}+1\right)k_{m-2}}{sq_{m-2}} \\
&= O\left(\frac{\sqrt{n/p}+\sqrt{\lg\lg n}}{s}\right).
\end{aligned}
$$

By substituting in equations 9 and 10, for k_{m-1}/q_{m-1}, we obtain the following bounds respectively.

$$
N_m k_{m-1}/q_{m-1} \leq (1+\varepsilon)^m \frac{n}{p} + 8\lceil \lg\lg n\rceil + O\left(\sqrt{\frac{n}{ps^2}} + \frac{\sqrt{\lg\lg n}}{s}\right), \tag{11}
$$

and

$$
N_m k_{m-1}/q_{m-1} \geq (1-\varepsilon)^m \frac{n}{p} - \lceil \lg\lg n\rceil - O\left(\sqrt{\frac{n}{ps^2}} + \frac{\sqrt{\lg\lg n}}{s}\right). \tag{12}
$$

This analysis completes the separate examination of the last recursive call of the algorithm.

We proceed to establishing the time complexity of the first stage of algorithm RAND_SORT. We distinguish two cases.
[**Case** $(l=r)$] The k_{m-1}/q_{m-1} buckets assigned to processor indexed l are sorted locally. Each bucket can be sorted in time $N_m \lg N_m$ for a total time of $k_{m-1}N_m \lg N_m/q_{m-1}$. An upper bound on $N_m k_{m-1}/q_{m-1}$ is given by equation 11, and therefore, by equation 6 we obtain the following bound on the time complexity of this step.

$$
\begin{aligned}
\frac{k_{m-1}}{q_{m-1}} N_m \lg N_m &\leq \left((1+\varepsilon)^m \frac{n}{p} + 8\lceil \lg\lg n\rceil + O\left(\sqrt{\frac{n}{ps^2}} + \frac{\sqrt{\lg\lg n}}{s}\right)\right) \lg N_m \\
&\leq \left((1+\varepsilon)^m \frac{n}{p} + 8\lceil \lg\lg n\rceil + O\left(\sqrt{\frac{n}{ps^2}} + \frac{\sqrt{\lg\lg n}}{s}\right)\right) \lg\left(2n^{1/2^m} \lg^{(2\delta+1)} n\right).
\end{aligned}
$$

By way of claim 6.5, we have that $\lg n/2^m = O(\lg(n/p) + \lg\lg^2 n)$. Thus, since

$$
(1+\varepsilon)^m \leq e^{1/\omega_n} \leq (1+2/\omega_n) \tag{13}
$$

and $s = \lceil \lg^\delta n\rceil$ we obtain the following bound.

$$
\frac{k_{m-1}}{q_{m-1}} N_m \lg N_m \leq \frac{1}{2^m}\left(1+\frac{2}{\omega_n}\right)\frac{n}{p}\lg n + O\left(\delta \lg\lg n\left(\frac{n}{p} + \lg\lg n\right) + \lg\lg^3 n\right).
$$

[**Case** $(l \neq r$ **and** $N_m \leq \lg^{\lg\lg n} n]$ The single bucket assigned to processors indexed l through r is sorted by an adaptation on the BSP model of Batcher's odd-even merge-sort [6], where comparators are replaced by $\lceil N_m/q_m\rceil$-mergers [21]. Each processor is assigned a N_m/q_m-size segment of the single bucket. Each such segment can be locally sorted in time $(N_m/q_m)\lg(N_m/q_m)$. Since $l \neq r$ we have that $k_m < q_m$, and therefore, by equations 7 and 13, and the fact that $N_m \leq \lg^{\lg\lg n} n$, we obtain the following bound on the time complexity of this local sorting step.

$$
\begin{aligned}
\frac{N_m}{q_m}\lg\frac{N_m}{q_m} &\leq \left((1+\varepsilon)^m \frac{n}{p} + 4\lceil \lg\lg n\rceil\right)\lg N_m \\
&\leq \left((1+\varepsilon)^m \frac{n}{p} + 4\lceil \lg\lg n\rceil\right)\lg\lg^2 n \\
&= O\left(\lg\lg^2 n\left(\frac{n}{p} + \lg\lg n\right)\right).
\end{aligned}
$$

We conclude the analysis of this step by noting that the number of rounds in Batcher's merge-sort algorithm is $(1/2)(\lceil \log q_m \rceil + 1)\lceil \log q_m \rceil$. Since $q_m \leq N_m$, we get that $\lg q_m = O(\lg \lg^2 n)$. We can therefore verify that the odd-even merge-sort procedure requires time

$$O\left(\lg \lg^4 n \max\{L, N_m/q_m\}\right) = O\left(\lg \lg^4 n \max\left\{L, \frac{n}{p}\right\}\right),$$

and

$$O\left(\lg \lg^4 n \max\{L, gN_m/q_m\}\right) = O\left(\lg \lg^4 n \max\left\{L, g\frac{n}{p}\right\}\right),$$

for computation and communication respectively (see also [19]).

The examination of the complexity of stages 2-4 of algorithm RAND_SORT is completed, by substituting in lemmas 2.1 and 2.2, for $t = 2$, and observing that

$$\sum_{i=0}^{m-1} \lg q_i = O(\lg n + \delta m \lg \lg n). \tag{14}$$

This is so by way of claim 6.4, equation 6, and the fact that $q_i \leq N_i$.

It remains to establish the time requirements of stages 5 and 6. The former stage, for any recursive call i, requires time $\max\{L, \lg k_i \lceil N_i/q_i \rceil\} + \log_{d_i} k_i \ O(N_i/q_i + L) + \log_{d_i} k_i O(C_{ppf}(q_i))$ and $\log_{d_i} k_i \ O(gN_i/q_i + L) + \log_{d_i} k_i O(M_{ppf}(q_i)))$ for computation and communication respectively. Summing over all recursive calls i such that $0 \leq i \leq m-1$, and noting that $k_i < q_i$ we obtain the total time required by this stage over all iterations. We calculate this cost by first considering the sum $\sum_{i=0}^{m-1} \lg k_i$. We claim inequality $\lg(x+y) \leq \lg x + 1.5(y/x)$ which holds by way of $\log(1+x) \leq x$, for any $x, y > 0$ [9]. By definition, $k_i \leq (\sqrt{N_i} + 1)/s$, where $s = \lceil \lg^\delta n \rceil$, and because of the analysis of stage 1, $N_i \geq \lg^{\lg \lg n} n$, for all $i < m$. Thus, we obtain the following bound.

$$\begin{aligned}
\sum_{i=0}^{m-1} \lg k_i &\leq \sum_{i=0}^{m-1} \lg(\sqrt{N_i} + 1) - m \lg s \\
&\leq \sum_{i=0}^{m-1} \lg \sqrt{N_i} + \frac{1.5m}{\lg^{\lg \lg n/2} n} - m \lg s.
\end{aligned}$$

The first term can be further bounded, by claim 6.4, as follows.

$$\sum_{i=0}^{m-1} \lg \sqrt{N_i} \leq m \lg(1+\varepsilon) + (1 - \frac{1}{2^m}) \lg n + m \lg s + m^2.$$

By combining the previous inequalities we get the following bound.

$$\sum_{i=0}^{m-1} \lg k_i \quad \leq \quad m \lg(1+\varepsilon) + (1 - \frac{1}{2^m}) \lg n + m^2 + \frac{1.5m}{\lg^{\lg \lg n/2} n}.$$

We conclude the analysis of stage 5 by observing that $d_i = 2\lceil (N_i/q_i)^\alpha \rceil$, for some constant $\alpha < 1$, and that the term N_i/q_i, $i < m$, is bounded by equation 7. We therefore derive the following bound on the computation time of stage 5, by substituting in lemma 2.2, for $t = 2$.

$$\sum_{i=0}^{m-1}\left(\lg k_i\left\lceil\frac{N_i}{q_i}\right\rceil+\lg_{d_i}k_iO\left(\frac{N_i}{q_i}\right)+\lg_{d_i}k_iO\left(C_{ppf}(q_i)\right)\right)\leq\left(1-\frac{1}{2^m}\right)\left(1+\frac{2}{\omega_n}\right)\frac{n}{p}\lg n+$$
$$O\left(\frac{n}{p}\left(\frac{\lg n}{\lg\frac{n}{p}}+m^2\right)+L\left(\frac{\lg^2 n}{\lg\frac{n}{p}}+\frac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\frac{\delta m^2\lg n}{\lg\frac{n}{p}}+\frac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right).$$

where we bounded $(1+\varepsilon)^i$ and $\sum_i \lg q_i$ by equations 13 and 14 respectively. Similarly, we obtain the following bound on the communication time of stage 5.

$$O\left(g\frac{n}{p}\left(\frac{\lg n}{\lg\frac{n}{p}}+\frac{m^2}{\lg\frac{n}{p}}\right)+\max\{L,O(g)\}\left(\frac{\lg^2 n}{\lg\frac{n}{p}}+\frac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\frac{\delta m^2\lg n}{\lg\frac{n}{p}}+\frac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right).$$

By substituting in lemma 2.3, for $t=2$, we obtain, by a way of arguments similar to that employed in the analysis of stage 5, that the computation and communication time complexity of stage 6 is bounded above by

$$O\left(\frac{n}{p}\left(\frac{\lg n}{\lg\frac{n}{p}}+\frac{m^2}{\lg\frac{n}{p}}\right)+L\left(\frac{\lg^2 n}{\lg\frac{n}{p}}+\frac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\frac{\delta m^2\lg n}{\lg\frac{n}{p}}+\frac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right).$$

and

$$O\left(g\frac{n}{p}\left(\frac{\lg n}{\lg\frac{n}{p}}+\frac{m^2}{\lg\frac{n}{p}}\right)+\max\{L,O(g)\}\left(\frac{\lg^2 n}{\lg\frac{n}{p}}+\frac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\frac{\delta m^2\lg n}{\lg\frac{n}{p}}+\frac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right).$$

The total time for computation and communication over all stages is thus given by the following formulae.

$$\left(1-\tfrac{1}{2^m}\right)\left(1+\tfrac{2}{\omega_n}\right)\tfrac{n}{p}\lg n+O\left(\tfrac{n}{p}\left(m^2+\tfrac{\lg n}{\lg\frac{n}{p}}\right)+L\left(\tfrac{\lg^2 n}{\lg\frac{n}{p}}+\tfrac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\tfrac{\delta m^2\lg n}{\lg\frac{n}{p}}+\tfrac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right)+$$
$$\left\{\begin{array}{ll}\tfrac{1}{2^m}\left(1+\tfrac{2}{\omega_n}\right)\tfrac{n}{p}\lg n+O\left(\delta\lg\lg n\left(\tfrac{n}{p}+\lg\lg n\right)+\lg\lg^3 n\right) & \text{if } \tfrac{n}{p}>\lg^{\lg\lg n} n\\ O\left(\tfrac{n}{p}\lg\lg^4 n+L\lg\lg^4 n\right) & \text{otherwise}\end{array}\right\},$$

and

$$g\,O\left(\tfrac{n}{p}\left(m^2+\tfrac{\lg n}{\lg\frac{n}{p}}\right)+\max\left\{\tfrac{L}{g},O(1)\right\}\left(\tfrac{\lg^2 n}{\lg\frac{n}{p}}+\tfrac{\delta m\lg n\lg\lg n}{\lg\frac{n}{p}}+\tfrac{\delta m^2\lg n}{\lg\frac{n}{p}}+\tfrac{\delta m^3\lg\lg n}{\lg\frac{n}{p}}\right)\right)+$$
$$g\left\{\begin{array}{ll}O\left(\tfrac{n}{p}\lg\lg^4 n+\max\left\{\tfrac{L}{g},1\right\}\lg\lg^4 n\right) & \text{if } \tfrac{n}{p}>\lg^{\lg\lg n} n\\ O(1) & \text{otherwise}\end{array}\right\}.$$

The failure probability for each recursive call of stage 2 is $n^{-\Omega\left((n/p)\lg\lg^2 n\right)}$. We have also proved, by claims 6.2, that the analysis related to bucket sizes fails with probability $n^{-\Omega\left(\lg^{\delta-1} n/(m\omega_n)^2\right)}$ for each recursive call, and thus the proposition follows. ■

By substituting appropriate values for the parameters in proposition 6.1 a variety of bounds on the runtime of algorithm RAND_SORT can be derived. As noted in [14], the optimal time bounds are derived by selecting m, δ and ω_n with regard to the actual values of n, p, L and g. By making, however, more generic selections for these parameters and worst-case bounds for p, L and g more general (but less tight) results in terms of n can be obtained. The following is one such bound.

Theorem 6.1 *For any positive constants c and ϵ, $0<\epsilon<1$, there exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p=n^{1-\epsilon}$ and $L=o(n^{\epsilon})$ has, with probability $1-n^{-\Omega(\lg^c n)}$, $\pi=1+o(1)$, and if $g=o(\lg n)$ also, $\mu=o(1)$.*

Proof: Let in proposition 6.1 $\omega_n = \lg\lg n$ and $\delta > c+1$. ■

By allowing more than a constant number of iterations the algorithm achieves one-optimality for values of n much closer to p than these of theorem 6.1.

Theorem 6.2 *For any constant $\alpha \geq 1$, there exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p = n/\lg^{\alpha} n$ and $L = o(\lg^{\alpha-1} n \lg\lg n)$, has, with probability $1 - n^{-\Omega(\lg^{\alpha} n \lg\lg^2 n)}$, $\pi = 1 + o(1)$, and if $g = o(\lg\lg n)$ also, $\mu = o(1)$.*

Proof: Let in proposition 6.1 $\omega_n = \lg\lg n$ and $\delta > 1+\alpha$. ■
The following theorem can also be derived.

Theorem 6.3 *There exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p = n\lg\lg n/\lg n$ and $L = O(1)$, has, with probability $1 - n^{-\Omega(\lg n \lg\lg n)}$, $\pi = O(1)$, and if $g = O(1)$ also, $\mu = O(1)$.*

We note that not only can the bounds given above be expressed in a variety of ways, but also the algorithm itself leaves room for several improvements. As stated it sorts the sample of size $\sqrt{N}$ by employing a BSP adaptation of a sorting algorithm of [27]. In most circumstances its performance can be improved by any one of several methods. For example, the sorting could be performed by a more efficient randomized parallel algorithm [14]. We also note that the recursion may terminate for smaller values of N than $\lg^{\lg\lg n}$. Thus, if we terminate the recursion for all N such that $N \leq \lg^{\beta} n$, for any $\beta = o(\lg\lg n)$, the contribution to the lower order terms of Batcher's sorting algorithm is $O((n/p)(\beta^2 \lg\lg^2 n))$ rather than $O((n/p)\lg\lg^4 n)$. This improves the asymptotic performance of the algorithm whereas at the same time the failure probability is affected insignificantly. The random sample selection in stage 2 of algorithm RAND_SORT could also be realized with higher probability by utilizing integer sorting techniques. Furthermore, the analysis of the ordered h-level graph algorithm can be improved by taking into consideration that the underlying structure is a d-ary tree, and the queries progress in a balanced fashion through the tree.

7 Conclusion

In this work we have presented a general technique for implementing data structures on the BSP model. We have also proposed multi-way search as a general tool for the design, analysis and implementation of BSP algorithms. Our methods support elegant high-level design and analysis of algorithms, using data structures similar to those of sequential models. We have also shown that optimality in computation to within a multiplicative factor close to *one* can be achieved for our algorithms for a wide range of values of the parameters p, g and L. Applications to computational geometry and sorting have also been presented. While our methods have been fairly uncomplicated themselves, description of their performance in terms of the BSP parameters has been somewhat complicated. The main reward for quantifying these complications, is that it enables software to be written once and for all that can be migrated efficiently among a variety of parallel machines.

References

[1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures,* pp. 129-136, 1995.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(\log n)$ sorting network. In *Proceedings of the 15-th Annual ACM Symposium on Theory of Computing,* pp. 1-9, 1983.

[3] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Florida, 1985.

[4] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences,* 18:155-193, 1979.

[5] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J. Tsay. Multisearch techniques for implementing data structures on a mesh-connected computer. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures,* pp. 204-214, 1991.

[6] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference,* pp. 307-314, 1968.

[7] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, 1995.

[8] R. H. Bisseling and W. F. McColl. Scientific computing on bulk-Synchronous Parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[9] B. Bollobás. *Random Graphs.* Academic Press, New York, 1984.

[10] R. Cole. Parallel merge sort. *SIAM Journal on Computing,* 17(4):770-785, 1988.

[11] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing,* 8:367-375, 1990.

[12] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing,* pp. 114-118, 1978.

[13] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM,* 17(3):496-507, July 1970.

[14] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[15] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.

[16] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EURO-PAR'96*, Lyon, France, LNCS 1124, Springer-Verlag, August 1996.

[17] A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996.

[18] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996.

[19] A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.

[20] D. E. Knuth. *The Art of Computer Programming. Volume II: Seminumerical Algorithms.* Addison-Wesley, Reading, 1969.

[21] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1973.

[22] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers,* C-34(4):344-354, 1985.

[23] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, California, 1991.

[24] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation,* (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.

[25] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

[26] W. F. McColl. An architecture independent programming model for scalable parallel computing. *Portability and Performance for Parallel Processors,* (J. Ferrante and A. J. G. Hey, eds.), John Wiley and Sons, 1994.

[27] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM,* 29(3):642-667, 1982.

[28] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[29] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing,* 18:594-607, 1989.

[30] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM,* 34:60-76, January 1987.

[31] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing,* 14(2):396-409, 1985.

[32] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* 33(8):103-111, August 1990.

[33] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (J. van Leeuwen, ed.), North Holland, 1990.

[34] L. G. Valiant. A combining mechanism for parallel computers. In *Proceedings of the First Heinz Nixdorf Symposium*, November 1992.