A randomized sorting algorithm on the BSP model

Alexandros V. Gerbessiotis\textsuperscript{a}, Constantinos J. Siniolakis\textsuperscript{b}

\textsuperscript{a}CS Department, New Jersey Institute of Technology, Newark, NJ 07102, USA
\textsuperscript{b}The American College of Greece, 6 Gravias St., Aghia Paraskevi, Athens, 15342, Greece

Abstract

An oversampling-based randomized algorithm is introduced for sorting \( n \) keys of any abstract data type on the \( p \) processors of a latency-tolerant parallel system such as a bulk-synchronous parallel computer. The algorithm is asymptotically, for large \( n \), optimal in computation and communication compared to the best available sequential sorting algorithm, even when constant factors are taken into consideration. Its parallel time is within a \( (1+o(1))/p \) multiplicative factor of the corresponding sequential method for sorting, improving upon other approaches for a wider range of values of \( p \) relative to \( n \). It also improves upon other randomized sorting algorithms that have been developed for latency-tolerant models of computation on the amount of parallel slack (ratio \( n \) over \( p \)) required to achieve optimal speedup and also, on the associated failure probability. For values of \( p \) closer to \( n \) than other latency-tolerant randomized approaches, it can be turned into a PRAM-like algorithm but for such cases a speedup of \( O(p) \) rather than \( p/(1+o(1)) \) is then achievable. Although the general framework of the proposed algorithm relies on the well-established prior idea of sample-based sorting, there are some novel features in its design such as the choice of splitter size, the way keys are split, and the handling of the base case of the recursive sorting algorithm that contribute to its performance.

Keywords: Randomized sorting, latency-tolerant algorithms, random sampling, oversampling, bsp model.

1. Introduction and related work

New ideas in designing randomized oversampling-based parallel sorting algorithms are developed. A parallel sorting algorithm, RANDSORT, is proposed that is designed in an architecture independent latency-tolerant setting and its performance is analyzed under the bulk-synchronous parallel (BSP) model [44]. The abstraction of the BSP model as used in this work consists of three components: (a) a collection of \( p \) processor/memory components numbered \( 0, \ldots, p-1 \),

\textit{URL: alexg@cs.njit.edu} (Alexandros V. Gerbessiotis)
(b) a communication network that can deliver messages point to point among the components, and (c) a mechanism for efficient synchronization, in barrier style, of all or a subset of the components. A BSP computer can then be modeled by \( p \), the number of processor components, \( L \), the minimal time, measured in terms of basic computational operations, between successive synchronization operations, and \( g \) the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the router in terms of words of information delivered (i.e. inverse of router throughput). The segment of computation or communication between synchronization operations is defined as a superstep and thus a parallel program can be viewed as a sequence of supersteps. The definition of \( g \) relates to the routing of an \( h \)-relation in continuous message usage, that is, the situation where each processor sends or receives at most \( h \) messages (or words of information); \( g \) is the cost of communication so that an \( h \)-relation is realized within \( gh \) operations [44], for any \( h \) such that \( h \geq h_0 \), where \( h_0 \) is a machine dependent parameter. Otherwise, if \( h < h_0 \), the cost of communication is (absorbed by) \( L \). Although the model-based charge of \( \max\{L, x + gh\} \) time steps is assigned to a superstep, this cost can also be translated into a corresponding cost in other latency-tolerant models of parallel computation [13, 11, 7, 27]. Here, \( x \) is the maximum number of basic computational operations executed by any processor during a superstep. Along the lines of [15], and in order to simplify the analysis of the performance of our algorithms, it is assumed that each superstep is either pure computation or pure communication and the two kinds of costs are thus easily distinguished. The upper bound on \( L + x + gh \) is used for bounding \( \max\{L, x + gh\} \); this only affects low order terms of parallel time.

The problem of sorting \( n \) keys on a parallel computing model [36] has drawn considerable attention. First approaches were sorting networks introduced by Batcher [5]. Several results concerning parallel sorting have been reported in [35, 36]. The first sorting network to achieve, however, the asymptotically optimal \( O(\lg n) \) time bound to sort \( n \) keys is the AKS [2] network of parallelism \( O(n) \). The constant involved in the \( O(\lg n) \) time bound were high. Subsequently, Reif and Valiant [41] presented a randomized sorting algorithm that runs on a fixed-connection network and achieves the same asymptotic time bound as the AKS algorithm; the constant multipliers in the randomized algorithm are, however, considerably smaller. Cole [10] was the first to present asymptotically optimal \( O(\lg n) \) time sorting algorithms for \( n \)-processor CREW and EREW PRAMs. These methods fail, however, to deliver optimal performance, i.e. a speedup of \( p/(1 + o(1)) \), when directly implemented on the BSP model of computation.

Parallel sorting has been substantially studied under the BSP model. The resulting algorithms exhibit in many instances optimal performance, they are sometimes quite practical, and algorithm implementations yield performance with optimal or close to optimal speedup of \( p \) even for moderate problem sizes [20]. The observed performance of the implemented algorithms is also within the predictions of the model [24]. Previously known results on BSP sorting include deterministic [1, 7, 17, 22, 28] and randomized [15, 26, 21] algorithms.
An adaptation of the AKS sorting network [2] on the BSP model is shown in [7] to yield computation time $O((n/p) \lg n)$ and communication time $gO((n/p) \lg p) + LO(\lg p)$ respectively. The constant factors hidden however, remain high. It is shown in [1] that for all values of $n$ and $p$ such that $n \geq p$, a BSP version of column-sort [35] requires $O(\zeta^{3.42} n_p \lg n_p)$ and $O(g\zeta^{3.42} n_p) + O(L\zeta^{3.42})$ time for computation and communication respectively, where $\zeta = \lg n / \lg (n/p)$ and $n_p = \lceil n/p \rceil$.

In [7], it is shown that an adaptation on the BSP model of the cube-sort algorithm [12] requires computation time $O(25^{\lg^* n} - \lg^* (n_p) \zeta^2 n_p \lg n_p)$ and $O(g 25^{\lg^* n} - \lg^* (n_p) \zeta^2)$ for communication. This is worse than [1]. Furthermore, as noted in [28], a modification of cube-sort shown in [39] eliminates the term $25^{\lg^* n} - \lg^* (n_p) \zeta^2$ and the resulting algorithm improves upon the algorithm of [1] as in computation and communication $\zeta$ replaces $\zeta^{3.42}$.

In [15], a randomized BSP sorting algorithm that uses the technique of oversampling is introduced that for a sufficient range of the parameters $n$ and $p$, i.e., $p = O(n/\lg^{1+\alpha} n)$, for any constant $\alpha > 0$, requires – under realistic assumptions – computation time $(1 + o(1)) (n \lg n/p)$ and communication time $O(g \zeta n_p) + O(L \zeta)$, with high-probability. This is better than [1] and [12], since the $\zeta^2$ in the communication time of [1] and the $\zeta^{3.42}$ in [12] are replaced by a $\zeta$ in the communication time of the algorithm in [15]. In addition, the computation time of the latter algorithm has no dependence on $\zeta$. A similar algorithm for the case $p^2 < n$, but with a less tight analysis is discussed in [37] (but not in the context of the BSP model). Given any $O(n \lg n)$ worst-case running time sequential sorting algorithm, the randomized BSP algorithm utilizes the sequential algorithm for local sorting and exhibits optimal parallel efficiency implying a speedup of $p/(1 + o(1))$, provided that the BSP parameters are appropriately bounded (and these bounds accommodate most, if not all, currently built parallel systems). The constants hidden affect low order terms and are well defined. This has been one of the first attempts to establish in some general way parallel efficiency for an algorithm by taking into consideration communication and synchronization delays, and describing optimality criteria not in absolute terms, that may be meaningless, but by comparing the proposed parallel algorithm performance to that of the best sequential algorithm in a meaningful way. The bounds on processor imbalance during sorting are tighter than those of any other random sampling/oversampling algorithm [31, 14, 41, 42]. The randomized algorithm has been implemented on realistic systems (SGI Power Challenge, Cray T3D and IBM SP2) and exhibited performance well within the bounds indicated by the theoretical analysis ([3, 20, 24]). It is noted that sample-based sorting algorithms are extensively studied and implemented in the literature (eg. [8, 30]). For example, the sample-sort algorithm in [8] is computationally equivalent to the one round case of [15]. The analysis of [15] is however tighter. It allows better control of oversampling with the end result of achieving smaller bucket expansion (key imbalance during the routing operation). The work of [30] applied to $d$-dimensional meshes is similar to the general algorithm of [15].
Under the same realistic assumptions of [15], a new deterministic sorting algorithm is introduced in [17], with computation time \((1 + \lceil \frac{\zeta - 1}{(1 - \zeta^{-1})/2} \rceil) \cdot (n \lg n/p)\), which is approximately \((1 + \zeta/2) \cdot (n \lg n/p)\) for \(\zeta > 1\), and communication time \(O(g \zeta n_p) + O(L \zeta)\), thus improving upon the upper bounds of [1, 2, 7, 12, 35]. The bound on computation is subsequently improved in [22, 23]. In particular, for \(p = n/\lg^{2+\alpha} n\), \(\alpha = \Omega(1)\), the improved deterministic sorting algorithm in [22, 23] requires computation time \((1 + 2/\alpha + o(1)) \cdot n \lg n/p\) and communication time \(O(g \zeta n_p) + O(L \zeta)\) respectively. The algorithm performs deterministic sorting by extending regular sampling, a technique introduced in [43], to perform deterministic regular oversampling. Past results that use regular sampling have been available for cases with \(p^2 < n\). The BSP algorithm in [17, 22, 23] further extends the processor range (thus further minimizing slack) and achieves asymptotically optimal efficiency for a slack \(n/p\) that is very close to that ratio of the randomized BSP algorithms in [15, 21]. This was made possible by a detailed and precise quantification of the input key imbalance among the processors during the phases of deterministic sorting thus contributing to a better understanding of regular oversampling. By using regular oversampling in a deterministic setting, it is possible to regulate the oversampling ratio to bound the maximum key imbalance among the processors. As in the case of randomized sorting, the insistence, under an architecture independent algorithm design, of satisfying the criterion of 1-optimality, i.e. asymptotical 100% efficiency implying a speedup of \(p/(1 + o(1))\), led to these improvements.

In practice, implementations of the deterministic algorithm [17, 22, 23] were shown to be consistently faster than the randomized algorithm [15] due to certain algorithmic choices that may be asymptotically irrelevant yet highly efficient in practice. In particular, the randomized algorithm [15] realizes a routing step where many messages of potentially small size are transmitted resulting in a fine-grain communication pattern. For high-latency systems, this is a potentially very expensive communication step. This high-cost can be somewhat reduced by performing a sequential integer sort algorithm that packs together all messages destined to the same processor before the communication. The extra cost for that is linear and though theoretically (or asymptotically) insignificant, its contribution to the observed running time is still significant enough to override other potential benefits of using in certain cases the randomized algorithm as it appears in [15]. The BSP model assumes that for such fine-grain communication a combining of messages will be performed and the cost model assumes that indeed, this is performed at no extra cost; the test platforms were not, however, typical BSP machines. The implemented deterministic algorithm on the other hand, is resilient to this because by design it uses coarse-grain communication. By observing the differences between the two approaches, it was then possible to derive a variant of the randomized algorithm in [15] based on the deterministic one described in [17] that would perform better than either algorithm. The new algorithm is introduced in [24].

In [28] a BSP adaptation of parallel merge-sort [10] is presented that for all values of \(n\) and \(p\) such that \(p \leq n\), requires computation and communication time \(O(n \lg n/p)\) and \(O(g \zeta^{2/p}) + O(L \zeta)\) respectively. As it is a BSP adaptation of
parallel merge-sort, it is at least as involved as the original parallel merge-sort (EREW PRAM variant) algorithm [10], and the constant factors involved in the analysis of this BSP adaptation are considerably large. Thus, the algorithm seems to be of little practical use as opposed to the algorithms in [15, 17, 23].

Other studies [32, 33] of practical parallel algorithms with optimal observed performance of $O(p)$ speedup contain an analysis of their performance that is architecture dependent. Thus they cannot be used in a systematic way for the study of the performance of other algorithms or to predict the performance of those algorithms on other platforms.

2. Contributions

Algorithm RandSort introduced in this work extends the algorithm of [15] by demonstrating that the same asymptotic time bounds as those of [15] can be achieved for $p$ as high as $p = o(n \log \log n / \log n)$ thus reducing the amount of slack required for an optimal algorithm to less than logarithmic. This is better than the $p = n / \log^{1+\alpha} n$, for any constant $\alpha > 0$, of [15]. The failure probability for given $p$ and $n$ is also significantly lower in RandSort. These improvements are achieved by using new approaches in determining splitter size, deciding the splitting of input keys during the stages of RandSort by using parallel static tree-search methods, and handling the base-case key sorting. The depth of recursion $m$ in RandSort is thus no more than $\log \log n$ and this is superior to other approaches that for $p$ close to $n$ have recursion depth proportional to $\log n$ [28]. Our results can be summarized in the form of Theorems 1, 2, 3, that describe the performance characteristics of RandSort for three overlapping cases of $p$.

Theorem 1 describes the performance of RandSort for $p = n^{1-\epsilon}$ for any constant $0 < \epsilon < 1$, i.e. polynomial slack is available. Parallel performance-wise, RandSort is asymptotically as good as the algorithm in [15]. A benefit of using our approach is an improved (i.e. smaller) failure probability. From a practical point of view, RandSort is not very desirable. The splitting of the keys around the splitters using a parallel tree-search method is too complex compared to the naive and traditional approach of [15] and [8, 29] of using binary search on the array of at most $p$ splitters that has been replicated to all processors.

Theorem 2 highlights the benefits of RandSort when splitter size or processor number grows closer to $n$. In Theorem 2, $p$ can be as high as $p = n / \log^\alpha n$ for any constant $\alpha \geq 1$. A speedup of $p/(1 + o(1))$, i.e. 1-optimality, is still possible. The range of $p$ is higher than the algorithm in [15] that requires an $\alpha > 1$. An improved failure probability bound over [15] is also achievable.

Finally, Theorem 3 allows for larger $p$ that is, $p = O(n \log \log n / \log n)$. For the upper bound on $p$, a speedup of $O(p)$ can only be attained, i.e. 1-optimality is not possible. RandSort behaves like a PRAM algorithm, a behavior shared by the approach of [28]. Despite such a behavior, RandSort is more scalable than the algorithm in [15] which is not optimal any more for such high $p$. Even if RandSort is not fully scalable compared to the deterministic algorithm of [28] that works for $p \leq n$, compared to it, RandSort does not suffer from the
complications of [28] originating from those of [10] that require an adaptation of concurrent-read-based methods to an exclusive-read-based model that the traditional BSP model is.

Note that Theorems 1, 2, 3, only describe the performance of RandSort for some interesting cases that accommodate comparisons to other approaches. Other conclusions can also be drawn. For example if the value of $p$ is higher than $n/\lg n$, the upper bound implied by Theorem 2, but lower than that of Theorem 3, i.e. $p = o(n \lg \lg n / \lg n)$, 1-optimality can still be attained. However this will require that parameters $L$ and $g$ are bounded above by quantities that are closer to those described in Theorem 3 rather than those of Theorem 2.

3. Performance comparison and primitive operations

The performance of a BSP algorithm $A$ is specified by comparing its performance to a sequential algorithm $A^*$. Two ratios $\pi$ and $\mu$ are then computed that compare the parallel computation and communication performance of $A$ to that of $A^*$: $\pi$, is the ratio $\pi = pC_A/C_{A^*}$ and $\mu$, is the ratio $\mu = pM_A/M_{A^*}$, where $C_A$ is the computation time and $M_A$ is the communication time of $A$ and $C_{A^*}$ is the computation time of $A^*$. When communication time is described, it is necessary that the amount of information that can be transmitted in a single message be made explicit. The objective of optimal algorithm design under the BSP model is to design parallel algorithms whose $\pi$ is as close to one as possible, and $\mu$ as small as possible. Conditions on $n$, $p$, $L$ and $g$ are specified that are sufficient for the algorithm to be realistic and the claimed bounds on $\pi$ and $\mu$ to be valid. Corollaries describe sufficient conditions for the most interesting optimality criteria, such as $c$-optimality for which $\pi = c + o(1)$ as well as $\mu = o(1)$. All asymptotic bounds are for problem size $n \to \infty$.

The charge for local operations on the BSP model is as follows. Since the mechanisms of the BSP model are similar to those used by sequential models and one is interested in performance ratios of these two, one can define operations at a higher level than machine level instructions. Throughout this work, $\lg$ and $\log$ denote logarithms to the base two and $e$ respectively. A dominant term in a BSP integer sorting algorithm to be claimed later is contributed by sequential count-sort performed independently at each processor. A charge of $O(n)$ time units is used for stable sorting $n$ integers in the range $[0,1,\ldots,n-1]$ by sequential COUNT_SORT [34]. By applying radix-sort techniques algorithm RADIX_SORT sorts $n$ integers in the range $[1,\ldots,n^\alpha]$, for some positive constant $\alpha$, in $[\lg \lfloor n^\alpha \rfloor / \lg n]$ iterations of COUNT_SORT. For other operations, a charge of one time unit is used for operations over an associative operator in parallel-prefix computations and one time unit for each comparison performed. The charge for sorting $n$ keys sequentially is $n \lg n$, and for performing binary search in a sorted sequence of length $n-1$ is $\lceil \lg n \rceil$. As noted in [15], had one charged $cn \lg n$ and $b \lg n$ operations for sorting and binary search respectively, for some constants $1 < b \leq c$, then neither $\pi$ nor $\mu$ would have increased in the analysis of the proposed algorithms.
In the remainder of this section BSP algorithms for several fundamental operations are described, namely broadcast, parallel-prefix and parallel radix-sort. All these primitives are auxiliary routines for the algorithms described in later sections. 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, 26, 25]. Towards the end of this section the problem of parallel searching \( d \)-ary trees is discussed. The similarities between the sequential and parallel version of searching \( d \)-ary or traditional binary \((d = 2)\) search trees are obvious. The difference between a sequential and parallel approach relates to the fact that an entire tree structure is distributed among multiple processors and the input consists not of a single query but a set of queries.

The following two lemmas extend broadcast and parallel-prefix (also called, a scan) operations on arbitrary \( t \)-ary trees. This is an extension of PRAM-based approaches on binary trees and uses pipelining to achieve the claimed bounds.

**Lemma 1 ([26]).** There exists a BSP algorithm for broadcasting an \( n \)-word message that requires time at most \( M^n_{brd}(p) = \left\lceil \frac{n}{\lceil \frac{n}{h} \rceil} \right\rceil + h - 1 \) \( \max \{L, g_t(n/h)\} \), for any integer \( 2 \leq t \leq p \), where \( h = \lceil \log_t ((t - 1)p + 1) \rceil - 1 \).

**Lemma 2 ([26]).** There exists a BSP algorithm for computing \( n \) independent parallel-prefix operations that requires time at most \( C^n_{ppf}(p) = 2\left\lceil \frac{n}{\lceil \frac{n}{h} \rceil} \right\rceil + h - 1 \) \( \max \{L, g_t(n/h)\} \) and \( M^n_{ppf}(p) = 2\left\lceil \frac{n}{\lceil \frac{n}{h} \rceil} \right\rceil + h - 1 \) \( \max \{L, g_t(n/h)\} \), for computation and communication respectively, for any integer \( 2 \leq t \leq p \), where \( h = \lceil \log_t p \rceil \), for a total time of \( T^n_{ppf}(p) = C^n_{ppf}(p) + M^n_{ppf}(p) \).

For \( n = 1 \), \( T_{ppf}(p) \) (respectively \( C_{ppf}(p) \), \( M_{ppf}(p) \)) stands for \( T^1_{ppf}(p) \) (respectively \( C^1_{ppf}(p) \), \( M^1_{ppf}(p) \)). The following integer sorting algorithm extends the algorithm of [40].

**Lemma 3.** Algorithm PARRDXSORT stable sorts \( n \) integers in the range \([0, 1, \ldots, R - 1]\) such that for any integer \( t \), \( 2 \leq t \leq p \), and using the charging policy of this section, requires time

\[
T = r \left( 3 \max \{L, O(\max \{n_p, t \lceil \log_t p \rceil\})\} + C_{ppf}(p) \right),
\]

and

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

for computation and communication respectively, where \( r = \lceil \log R / \log \max \{n_p, t \lceil \log_t p \rceil\} \rfloor \) and \( n_p = \lceil n/p \rceil \).

**Proof:** Let \( m = \max \{n_p, t \lceil \log_t p \rceil\} \). The base step of the radix-sort algorithm is a stable count-sort algorithm that sorts \( n \) keys in the range \([0, 1, \ldots, m - 1]\),
Algorithm 1 ParRdxSort \((n, p, R)\) \{Sort \(n\) integer keys in range \([0, \ldots R-1]\) on \(p\) processors\}

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 \(i\), \(0 \leq k \leq m-1\) and \(0 \leq i \leq p-1\).

2: The position in the sorted array of the first item with value \(\bar{k}\) that is assigned to processor \(\bar{i}\) is given by \(S_{\bar{k},\bar{i}} = 1 + \sum_{k,i} \{S_{k,i} \mid (k < \bar{k}) \text{ or } (k = \bar{k}\text{ and } i < \bar{i})\}\). Order all the \(S_{k,i}\)s in the direction of increasing \(k\), perform a parallel-prefix sum operation, and then restore the original order of the \(S_{k,i}\)s.

3: Every processor knows where to route every key in its possession, and performs this routing.

for any positive integers \(n\) and \(p\) such that \(n \geq p\). Then sorting of \(n\) keys in the range \([0, 1, \ldots, R-1]\) is achieved with \(r = \lceil \log R / \log m \rceil\) iterations of count-sort. For the algorithm to work as claimed the input is assumed to be evenly distributed among the \(p\) processors.

In step 1 of Algorithm 1 implementing ParRdxSort every processor holds at most \(\lceil n/p \rceil\) keys. The cost of step 1 is therefore bounded above by \(\max\{L, m\}\). Hence, if each processor \(i\) knows the value of \(S_{k,i}\) for all \(k\), 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 \(S_{k,i}\) values in the direction of increasing \(k\), perform a parallel-prefix sum operation, and finally restore the original order of the \(S_{k,i}\) values. In step 2 each processor sends/receives at most \(m\) words twice, once for reordering and once for restoring the order of the \(S_{k,i}\) values, and performs a parallel-prefix sum operation, for a total time of \(2 \max\{L, m\} + C_{pf}(p)\) and \(2 \max\{L, gm\} + M_{pf}(p)\) for computation and communication respectively. The execution of step 3 requires time \(\max\{L, gn_p\}\).

The algorithm requires \(O(\max\{n/p, t[\log_p p]\})\) space per processor; by replacing, however, in the above lemma \(n_p\) for \(\max\{n_p, t[\log_p p]\}\), the space can be reduced to \(O(n/p)\) per processor.

A latency-tolerant algorithm for efficiently searching a \(d\)-ary tree is presented below. A more general algorithm for searching \(d\)-ary trees appeared in [6]. In [16, 18] a static-tree search algorithm (that fully traces the search path of a query) is presented for a wider class of graphs, called ordered \(h\)-level graphs that includes \(d\)-ary trees.

Let \(G = (V, E)\) be a \(d\)-ary search tree of height \(h\). A search path for a query \(q \in Q\), where \(Q\) is a set of queries, is a sequence \(Path(q) = (v_1, v_2, \ldots, v_h)\) of \(h\) nodes of \(G\), prescribed by a next function \(f: Q \times V \to V\) as follows: (1) \(v_1 = \text{root}(G)\), and (2) \(f(q, v_i) = v_{i+1}, i \leq h-1\), where \((v_i, v_{i+1}) \in E\). In other words, \(f(q, v)\) evaluates to (points to) the next node to be visited. For the sake of an example, if \(G\) is a binary search tree with \(d = 2\), \(q\) is a query (key value)
and $v$ is a node of that binary search tree, then $f(q, v)$ determines based on $q$ and $v$ whether the left or the right subtree of $v$ will be visited next. The root of the appropriate subtree that satisfies the query is then returned as the result of evaluating $f(q, v)$. Note that for this simple case, it suffices that every node $v$ stores one key value to evaluate $f(q, v)$. For $d$-ary trees every node $v$ might have to store up to $d - 1$ values; however the use of a generic function $f$ hides such considerations. The search process for a query $q$ is a process divided into $h$ phases such that by phase $l$, $1 \leq l \leq h$, the search path $(v_1, v_2, \ldots, v_l)$ of $q$ has been determined for level $l$ of $G$ and there exists a processor which contains a description of both $q$ and $v_l$. Then the evaluation $f(q, v_l)$ is realized to determine the next node of the search path. The search process for a query set $Q$ is the realization of the search process for every $q \in Q$. If parallelism is available both $Q$ and $G$ are evenly distributed among the $p$ available processors.

**Problem.** Given a set of $n$ queries $Q = \{q_1, q_2, \ldots, q_n\}$ evenly distributed among $p$ processors, and a $d$-ary tree $G = (V, E)$ with $|V| = k$ also evenly distributed among the $p$ processors, determine all $n$ search processes induced by the $n$ queries.

The $n$ search processes for all $n$ queries are realized in a sequence of phases where in phase $l$ the queries access and search the nodes of level $l$ of the tree. Let $v_l$ be a node of $V$. In each phase (i.e. level) the following two steps are executed: (1) evaluate $f(q, v_l)$, and (2) get the next node $v_{l+1} = f(q, v_l)$ in the search path of $q$. The set of queries visiting $v_l$ is defined to be the thread at or associated with node $v_l$. A thread might be distributed over a number of consecutive processors; such a thread is called shared, as opposed to an exclusive thread that is stored in a single processor. 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, the following distribution (load-balancing properties) of the graph $G$ among the processors is maintained.

1. Each node $u \in V$, has $d$ children. For each node $u$ the information `next_info` pertaining to its children is stored in the form of edges, that is for each child $v$ of $u$ information about edge $e = (u, v)$ is maintained. For each such edge a pointer to the `next_info` of $v$ is maintained at $u$. Information on $d$ edges thus needs to be stored at $u$, the number of children of $u$. Besides the aforementioned pointers, $d - 1$ key values are stored in the form of a sorted array so that the evaluation of $f$ at $u$ for a query $q$ will require time $\lceil \log d \rceil$ through binary search.

2. $G$ is orderly but otherwise evenly distributed among the processors so that consecutive memory locations of a single processor store information for any node such as $u$. This generates a slight imbalance in the distribution since for example the $d$ pointers of the `next_info` of $u$ can not span more than one processor to induce a fully even
distribution; they must be kept within a single processor thus causing an imbalance that involves no more than $d$ edges per processor. Overall $|E|$ is the total number of edges for which information needs to be maintained. Thus each processor stores information for no more than $\lceil |E|/p \rceil + d$ edges, with the additive term in the latter expression reflecting the excess storage that might be needed in any processor to maintain information in an orderly manner. In addition, $G$ is stored level-by-level top (starting from the root) to bottom (ending with the leaves), and for a given node $u$ the pointers of its children are stored left-to-right.

The procedure to manage the computation, for an arbitrary phase $l$, is summarized below.

**Lemma 4.** Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of $n$ queries and let $G = (V, E)$ be a $d$-ary tree, where $|V| = k \leq n$ and $d = o(n/p)$. Let the number of queries visiting any leaf (and thus, any node) of the tree be at least $d$. If $f$ is the next function and $\lceil \lg d \rceil$ is the time required to evaluate $f$ for any $q \in Q$ and $v \in V$, then, algorithm `MULTIWAYSEARCH` for computing the next node in the search path of $q_i$, $1 \leq i \leq n$, for any level $l$ of $G$, requires computation and communication time that are given by the following expressions.

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

$$M(n) = O \left( \frac{g}{p} + L \right) + M_{ppf}^d (p) + O \left( M_{ppf} (p) \right).$$

The algorithm requires $O(n/p + k/p)$ space per processor.

**Proof:** Procedure `MULTIWAYSEARCH` evaluates each query $q_i$ starting from the root (i.e. $l = 1$) and moving downwards level-by-level from level $l$ to the next level $l + 1$. The code for realizing `MULTIWAYSEARCH` in Algorithm 2 is described as a sequence of four stages, each one consisting of a number of operations. The $n$ queries $Q = \{q_1, q_2, \ldots, q_n\}$ are initially evenly distributed among the processors so that each one gets up to $\lceil n/p \rceil$ of them. Graph $G = (V, E)$ also satisfies load-balancing properties 1-2. Note that since $G$ is a tree, $|E| = |V| - 1 = k - 1$. The following *invariants* are assumed to hold for phase $l$.

1. The queries are sorted with respect to a left-to-right ordering of the children of their associated node of level $l - 1$.
2. All shared threads at level $l - 1$ are identified and characterized by group and group leader.

Initially there is only one thread $Q$ of size $n$. Obviously this thread is shared among all processors and the group leader is the first of $p$ processors. After the
first call to MultiWaySearch with $l = 1$ this shared thread might be split into multiple smaller threads; some of them will be shared but it is possible that several of them are exclusive. A processor $r$ can have no more than two shared threads: a shared thread that overflowed a previous processor and ends in $r$, and another thread that starts in $r$ and might overflow $r$ and following processors; in the latter case $r$ becomes the group leader.

**Algorithm 2 Procedure MultiWaySearch ($l$)**

1. Every processor obtains, for each thread in its possession, the associated `next_info` information.
2. Every processor computes, for each query $q \in \bar{Q}$ in its possession, the next node $v_{l+1}$ in the search path of $q$ by evaluating $f(q, v_l)$, where $v_l$ is the node associated with $q$.
3. Processors sort the queries within each thread using as key the computed $f(q, v_l)$ value. All queries evaluating to the same child $f(q, v_l)$ of $v_l$ are thus grouped together to form the new threads of the following phase $l+1$. Invariant 1 is maintained for phase $l+1$.
4. Every processor determines and characterizes by group and group leader the newly formed threads at level $l$, thus updating Invariant 2 for phase $l+1$.

In phase $l = 1$ a single thread of size $n$ is associated with node $\text{root}(G)$. The procedure for level $l$ ensures that, for each query $q \in Q$, with search path $\langle v_1, v_2, \ldots, v_l \rangle$, there exists a processor which contains a description of both the query $q$ and node $v_l$. Finally, in the remainder of this proof, the term `next_info` is used, for a particular thread $\bar{Q}$, to describe the edges emanating from the node associated with $\bar{Q}$. Informally, `next_info` relates to the information required to evaluate the $f$ function $f$.

For phase $l$ of MultiWaySearch, in stage 1, let $\bar{s} \geq d$ (by way of the statement of Lemma 4 there are at least $d$ queries visiting a node) be the size of some thread $\bar{Q}$ that was formed in the previous phase $l-1$ or the single thread associated with node $\text{root}(G)$ for $l = 1$, and let $\bar{d}$ be the number of children of the node say $v_l$ associated with $\bar{Q}$ (if $l = 1$, then $\bar{s} = n$). It is noted that $\bar{d}$ is known by load-balancing property 1 and it is $\bar{d} = d$ but the remaining discussion works even if $\bar{d} \leq d$. Stage 1 is realized by one of several methods depending on the distribution of $\bar{Q}$ that has size $\bar{s} \geq d$.

(1) read `next_info` if $\bar{Q}$ is exclusive residing at $\bar{p}$,
(2) read `next_info` if $\bar{Q}$ is shared, and $\bar{p}$ is the group leader,
(3) remain idle otherwise,

When this stage is initiated every processor holds at most $\lceil n/p \rceil$ queries of one or more threads, and also maintains `next_info` information of at most $\lceil |E|/p \rceil + d = \lceil (k-1)/p \rceil + d$ edges, $k \leq n$. By way of cases 1-3 above, each processor reads information (`next_info`) of no more than $n/p + 2d$ edges. This is because thread $\bar{Q}$ has size $\bar{s} \geq d$, and the node $v_l$ associated with $\bar{Q}$ has only $d$ children whose edges need to be retrieved by $\bar{Q}$ to compute $f(q, v_l)$ for every
$q \in \tilde{Q}$. The second term of $n/p + 2d$ deals with the fact that each processor might be associated with a shared thread and as a group leader will read a next_info of size possibly larger than the portion of the shared thread residing in that processor. Hence, this stage 1 takes asymptotically $\max \{L, g O(n/p)\}$ since $d = o(n/p)$.

In stage 2, a preprocessing step is required if a thread is shared. If $\tilde{Q}$ is shared and $\tilde{p}$ is the group leader for $\tilde{Q}$, then $\tilde{p}$ broadcasts the next_info information it read to all the processors in its group. By Lemma 1, this takes communication time at most $M^d_{\text{ppf}}(p)$ if implemented in terms of a segmented parallel-prefix operation [36]. This completes the preprocessing. At the end of stage 1 each processor still holds at most $\lceil n/p \rceil$ queries, and therefore, the time required by stage 2 to evaluate $f(q, v_l)$ is bounded above by $\max \{L, [\lg d] \lceil n/p \rceil\}$, the cost of a binary search per query.

If $l < h$, stages 3 and 4 maintain (update) invariants 1 and 2 for the next phase $l + 1$. The sorting operation of stage 3 is performed by one of several methods depending on the size and distribution of $\tilde{Q}$. Particularly, radix-sort (sequential or parallel) is used.

(1) RADIX_SORT if $\tilde{Q}$ is exclusive,
(2) PARRDXSORT if $\tilde{Q}$ is shared.

Key values for sorting are the $f(q, v_l)$ computed in stage 2. Since each such value corresponds to one of the $d$ children of $v_l$, indexes $0$ to $d - 1$ are used to uniquely identify the children of $v_l$ in a left-to-right listing. Thus, when stage 3 is executed each thread by definition is of size at least $d$ and the overall size of all threads is at most $\lceil n/p \rceil$. There are thus more keys in a thread than key values and one round of radix-sort (i.e. count-sort) suffices for sorting. By way of algorithms RADIX_SORT and PARRDXSORT, each such thread can be sorted in one iteration of radix-based sorting, i.e., sorting of at least $d$ integers in the range $[0, \ldots, d - 1]$. Then, by way of Lemma 3, the time required for sorting all such threads is $O(\max \{L, n/p\}) + O(C_{\text{ppf}}(p))$ and $O(\max \{L, g(n/p)\}) + O(M_{\text{ppf}}(p))$ for computation and communication respectively.

At the conclusion of stage 2, for every query $q \in \tilde{Q}$, the node of level $l + 1$, say $v_{l+1} = f(q, v_l)$ has been identified to which this query will be evaluated next, as part of the following phase $l + 1$ involving nodes of the following level $l + 1$ of $G$. So far at stage 3, all queries of $\tilde{Q}$ evaluated to the same node of level $l + 1$ have been grouped together; this was realized through the sorting operation of stage 3. What remains to be done is to identify the newly formed threads derived from $\tilde{Q}$, and characterize them as shared or exclusive, and in the former case assign a group leader to each shared thread. This is done in stage 4. A simple scan operation of the sorted threads suffices. Exclusive threads are dealt with immediately in local memory, since there is only one processor storing them. 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 $(\text{shared\_thread}_i, \text{processor}_i)$ and $(\text{shared\_thread}_j, \text{processor}_j)$, and returns
\[(\text{shared\_thread}_i, \text{processor}_i') \quad \text{if} \quad \text{shared\_thread}_i = \text{shared\_thread}_j,\]
\[(\text{shared\_thread}_j, \text{processor}_j') \quad \text{if} \quad \text{shared\_thread}_i \neq \text{shared\_thread}_j,\]

where \(i < j\), in other words the prefix operation is a segmented prefix with the segments being the shared threads themselves. 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 value returned by the parallel-prefix operation.

When stage 4 is executed, every processor holds at most \(\lceil n/p \rceil\) queries and at most two shared threads. The cost of this stage is thus bounded above by \(\max \{L, O(n/p) + O(C_{ppf}(p))\}\) for computation and \(\max \{L, O(g)\} + O(M_{ppf}(p))\) for communication respectively.

Algorithm MultiWaySearch is repeated \(h\) times, once for each level of \(G\). Therefore the whole search requires time which is \(h\) times larger than the one indicated in Lemma 4. In the following section, Lemma 4 will be claimed for a \(d\) that is a power of two and therefore \(\lceil \lg d \rceil = \lg d\). Furthermore, given that \(G\) has height \(h\), the number of leaves will be \(k = d^h\), a power of \(d\) as well as two.

An issue that needs to be resolved later is the case where \(k\) is a power of two but not necessarily a power of \(d\), i.e. \(k = d^ht\), where \(t < d\) is still a power of two. One needs to address the applicability of Lemma 4 to such a case. Instead of building a \(d\)-ary tree on these \(k\) leaves of height \(h + 1\), a tree that is \(d\)-ary up to height \(h\) is built instead, and in the last level every node has degree exactly \(t\).

Lemma 4 applied to the first \(h\) levels gives a high order term for computation time that is proportional to \(hn \lg d/p\). For the last level, MultiWaySearch will be used with \(t\) substituting for \(d\) in Lemma 4. Given that \(t \leq d\), the computation time for that extra level will be proportional to \(n \lg t/p \leq n \lg d/p\). The total computational time over all levels has a high order term \((h \lg d + \lg t)n/p = \lg k \,(n/p)\) since \(k = d^ht\), and \(k, d, t\) are all powers of two. Thus the following Lemma is derived if Lemma 4 is applied \(h + 1\) times, of which \(h\) times apply to levels with nodes having \(d\) children and once to the penultimate level with nodes having only \(t\) children each.

**Lemma 5.** Let \(Q = \{q_1, q_2, \ldots, q_n\}\) be a set of \(n\) queries and let \(G = (V, E)\) be a \(d\)-ary tree, where \(|V| = k \leq n, k = d^ht, t \geq d, d, t\) are powers of two and \(d = o(n/p)\). Let the number of queries visiting any leaf (and thus, any node) of the tree be at least \(d\). If \(f\) is the next function and \(\lceil \lg d \rceil\) is the time required to evaluate \(f\) for any \(q \in Q\) and \(v \in V\), then, algorithm MultiWaySearch requires overall computation and communication time that are given by the following expressions.

\[C(n) = \lg k \left\lceil \frac{n}{p} \right\rceil + (h + 1)O \left( \frac{n}{p} + L \right) + (h + 1)O \left( C_{ppf}(p) \right),\]

and
\[ M(n) = (h + 1)O \left( \frac{n}{p} + L \right) + (h + 1)M_{ppf}^d(p) + (h + 1)O \left( M_{ppf}(p) \right). \]

The algorithm requires \( O(n/p + k/p) \) space per processor.

4. BSP Randomized Sorting

An efficient BSP algorithm is derived for the problem of sorting \( n \) keys, that achieves 1-optimality for a wide range of the parameters \( n, p, L \) and \( g \). The algorithm derives from the ideas of [14, 42] and the technique of over-sampling [41, 15], and utilizes MultiWaySearch of Lemma 5, a special case of more general graphs discussed in [18, 16]. An outline of some results on random sampling is first presented and then randomized algorithm RandSort is introduced.

4.1. Random Sampling

In the discussion and proofs to follow the notation of [15] is used. 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 \( 1 \leq i \leq N - 1 \). The implicit assumption is that keys are unique. Let \( Y = \{y_1, y_2, \ldots, y_{ks-1}\} \) be a randomly chosen subset of \( ks - 1 \leq N \) keys of \( X \) also indexed such that \( y_i < y_{i+1} \), for all \( 1 \leq i \leq ks - 2 \), for some positive integers \( k \) and \( s \). Parameter \( s \) is the oversampling factor and \( k \) is the splitter size for the given splitting. Having randomly selected set \( Y \), it is then used to partition set \( X - Y \) into \( k \) subsets, \( X_0, X_1, \ldots, X_{k-1} \), where

\[
X_0 = \{x \mid x \in X \text{ and } x < y_s\}, \\
X_i = \{x \mid x \in X \text{ and } y_{is} < x < y_{(i+1)s}\}, \text{ for all } 1 \leq i \leq k - 2, \\
X_{k-1} = \{x \mid x \in X \text{ and } y_{(k-1)s} < x\}.
\]

Let \( Q_i(j) \) be the probability that \( y_i = x_j \). Then, for all \( 1 \leq i \leq ks - 1 \) and \( 1 \leq j \leq N \) the following equality holds [15].

\[
Q_i(j) = \binom{j-1}{i-1} \binom{N-j}{ks-1-i},
\]

where for the binomial coefficients, \( \binom{j}{i} = 0 \) if \( j < i \). Let \( N_i \) denote the size of \( X_i \) and \( P_i(j) \) denote the probability that \( N_i = j \), where \( 0 \leq i \leq k - 1 \) and \( s - 1 \leq j \leq N - (k - 1)s \). The following result is shown in [15].

**Lemma 6.** For all \( i \),

\[
P_i(j) = P(j) = \binom{j}{s-1} \binom{N-j-1}{(k-1)s-1}. 
\]

A lower bound on the size of the \( X_i \)'s is derived 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. The following result is shown first.
Lemma 7. Let $k \geq 2$, $s \geq 1$, $ks < N/2$, $n \geq 1$, $0 < \varepsilon < 1$, $\rho > 0$, and

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

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

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

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

$P(j)$ is approximated by a binomial term and bounds on the right tail of the binomial distribution are then claimed. The proof utilizes an argument similar to that of [15]. Let $B = \left\lfloor (1 - \varepsilon)(N - k + 1)/k \right\rfloor - 1$ and $q = (ks - 1)/N$. The probability that in $N$ Bernoulli trials, each with a probability $q$ of success there are $s - 1$ successes among the first $j$, a success at the $(j + 1)$-st trial, and $(k - 1)s - 1$ successes in the remaining trials is

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

Let $A$ be the ratio of $R(j)$ to $P(j)$. Then, by way of Lemma 6,

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

The following fact is deducible from Stirling’s approximation formula [9] and holds for $a = c/d$, $b = 1 - a$, $c < d/2$.

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

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

$$e^{-1/(6(ks-1))} \frac{1}{(2\pi(ks - 1)(1 - ks - 1)/(N))^{1/2}} q^{ks+1} (1-q)^{-N+ks-1} \leq \left(\frac{N}{ks - 1}\right).$$

By way of Equation (1), the following inequality holds.

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

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

By setting \(s = (1+\beta)(B+1)q\) in this bound
\[e^{-(s-(B+1)q)^2/(3(B+1)q)}.\]

A lower bound for the term \(s-(B+1)q\), for \(0 < \varepsilon < 1\) and \(ks \leq N/2\), is first derived.
\[
s - (B+1)q = s - \left(\left\lfloor (1-\varepsilon)\frac{N-k+1}{k} \right\rfloor \right) \frac{ks-1}{N} \\
\geq s - s(1-\varepsilon) + \frac{(1-\varepsilon)(k-1)sk + (N-k+1)(1-\varepsilon)}{kN} \\
\geq s(1-\varepsilon) + \frac{-ks(k-1)(1-\varepsilon) - (1-\varepsilon)(N-k+1)}{kN}.
\]

An upper bound for \((B+1)q\) is derived similarly.
\[
(B+1)q \leq s(1-\varepsilon) + \frac{-ks(k-1)(1-\varepsilon) - (1-\varepsilon)(N-k+1)}{kN} \\
\leq (1-\varepsilon)s.
\]
The combination of these two bounds gives
\[ e^{-\beta^2 (B+1)q/3} \leq e^{-\varepsilon^2 s/(3(1-\varepsilon))}. \]

The following inequality for the sum of interest is obtained.

\[
\sum_{j \leq B} \frac{\binom{j}{s-1} q^{s-1}(1-q)^{j-s+1}}{B+1} = \frac{1}{q} \sum_{i=s}^{B+1} \binom{B+1}{i} q^i (1-q)^{B+1-i} \leq \frac{1}{q} e^{-\varepsilon^2 s/(3(1-\varepsilon))}. \tag{4}
\]

The desired bound follows from Equations (2), (3) and (4), and is given below.

\[
\sum_{j \leq B} P(j) \leq (2\pi(ks-1))^{1/2} e^{1/(6(ks-1))} e^{-\varepsilon^2 s/(3(1-\varepsilon))}. \tag{5}
\]

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

\[
(2\pi(ks-1))^{1/2} e^{1/(6(ks-1))} e^{-\varepsilon^2 s/(3(1-\varepsilon))}.
\]

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

The following is also proved in [15].

**Lemma 8.** Let \(k \geq 2\), \(s \geq 1\), \(ks < N/2\), \(n \geq 1\), \(0 < \varepsilon < 1\), \(\rho > 0\), and

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

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

### 4.2. Overview of RANDSORT

The randomized BSP sorting algorithm, **RANDSORT**, achieves 1-optimality (as defined in Section 3) for a wide range of the parameters \(L\) and \(g\) and for \(p\) as high as \(n/p = o(\log n / \log \log n)\). Furthermore, it only achieves constant-optimality for higher values of \(p\), i.e. for \(n/p = \Theta(\log n / \log \log n)\) and \(L\) and \(g\) appropriately bounded. Theorems 1, 2, 3 to follow describe the properties of RANDSORT as derived from Proposition 1 that describes its performance.

Algorithm **RANDSORT** is based on an efficient partitioning scheme and is recursive in nature. The features that distinguish it from other approaches to randomized oversampling-based sorting are the following.
1. Splitter size. In [8, 29, 15] splitter size is chosen to be \( p \). In other BSP or non-BSP sample-based sorting algorithms [15, 30] splitter size is fixed to \( p^{1/l} \) for some constant \( l \leq 1 \) that is dependent on \( \lg n / \lg(n/p) \). In \textsc{RandSort} splitter size is chosen as large as possible for the intended optimality condition to be still true.

2. Splitter processing. A follow-up step to splitter selection is usually a splitter broadcast step, and an operation that determines through sequential binary search the position of each input key relative to the chosen splitters. In \textsc{RandSort} the splitters are uniformly distributed among the processors to form a \( d \)-ary tree, and the determination of the position of an input key relative to the splitters becomes a parallel tree-search process that utilizes \textsc{MultiWaySearch}.

3. Final stages of recursion. A traditional approach for the base case in parallel sample-sorting is sequential sorting of the keys local to a processor. In \textsc{RandSort} an alternative is also available: parallel sorting with a suboptimal algorithm (e.g. Batcher’s method). Such an approach is chosen if it is not possible to further split the input keys.

The depth \( m \) of recursion in \textsc{RandSort} will be at most \( \lg \lg n \). The number of splitters \( k_i - 1 \) chosen at any depth \( i \) recursive call will vary with \( i \). Very roughly, this number will be proportional to \( n^{1/2}/s \). Thus at depth 1, the \( n \) keys will form \( n^{1/2}/s \) buckets, at depth 2, each such bucket will form \( n^{1/2^2}/s \) new ones, and so on. This completes when the overall number of buckets \( n^{1/2+1/2^2+...+1/2^i}/s^i = n^{1-1/2^i}/s^i \) approaches \( p \). Ignoring the contribution of \( s \) and equating \( n^{1-1/2^i} \) to \( p \), we obtain that \( i = \lg \lg n - \lg (\lg n - \lg p) \). Thus \( m \) is roughly doubly logarithmic in \( n \), a substantial improvement over traditional approaches including [28] in which \( m \) can grow up to \( \lg n \).

Specifically, in a depth \( i \) recursive call, \( k_i - 1 \) splitters are selected from a set of keys of size \( N_i \), where \( k_i \) is chosen so that \( k_i = \lfloor (\lfloor N_i^{1/2} \rfloor + 1)/s \rfloor \). The splitters split the \( N_i \) keys into \( k_i \) “buckets” of approximately the same size. The keys in each bucket are distributed over a number of processors. The choice of \( k_i \) maximizes the number of buckets into which the input keys are split; the oversampling factor \( s \) controls how imbalanced the buckets will be in that or subsequent recursive calls. The next recursive call performs a similar process of sub-partitioning each one of these buckets into \( k_{i+1} \) further buckets, and so on. Initially, \( N_0 = n \) with all the keys in a single “bucket” and distributed over the \( p \) processors. Then \( k_0 - 1 \) splitters are selected that split the \( N_0 = n \) keys into \( k_0 \) buckets. Each bucket is thus spread over \( p/k_0 \) processors. By way of the probabilistic considerations of Lemma 7, the size of any bucket thus formed is within \( (1 + \varepsilon) \) of the mean \( N_0/k_0 \) with high probability. Denoting this by \( N_1 \), \textsc{RandSort} proceeds to the next recursion level. Similarly, the size of any bucket at depth \( i \leq m \) is within a factor of \( (1 + \varepsilon) \) of the corresponding mean \( N_i/k_i \) with high probability. By choosing \( \varepsilon = 1/(m \omega_n) \) so that \( \omega_n \) is any nondecreasing function of \( n \), one concludes that the size of any bucket formed at depth \( m \) of \textsc{RandSort} is within a factor of \( (1 + \varepsilon)^m = (1 + 1/(m \omega_n))^m \leq e^{1/\omega_n} \) of the mean \( N_0/p = n/p \).
4.3. Algorithm RandSort

While RandSort conforms to the pattern of oversampling-based randomized sorting algorithms, the choice of the various controlling parameters and analysis of its performance in terms of the BSP parameters is somewhat involved. The remainder of this section is devoted to the derivation of the following proposition.

Proposition 1. Let $\omega_n$ be any nondecreasing function of $n$ such that $\omega_n > 2$ for sufficiently large $n$. Then, for any positive integers $n, p$ such that $n \geq 65536$ and $p$ a power of two, and $n/p = \Omega(\log n/\log \log n)$, RandSort for any $\delta$ such that $1 < \delta < (\log \log n - 1)/2$, has, with probability $1 - o(1)$, computation and communication time given by Equations (18) and (19) (resp. Equations (20) and (21)) depending on whether the slack $n/p$ is sufficiently large $n/p = \Omega(\log \log \log n)$ or not.

Remark. The success probability implied by $1 - o(1)$ in Proposition 1 is bounded below by

$$1 - n^{-\Theta((n/p) \log \log^2 n)} - n^{-\Theta(\log^{s-1} n/(m \omega n)^2)}.$$

Proof: Algorithm RandSort is recursive, and a recursive call is depicted in Algorithm 3, where the $N$ keys assigned to processors $l, \ldots, r$ are sorted. Initially a call to RandSort $(n, n, 0, p - 1)$ is issued. The $n$ keys are assumed to be evenly distributed among the $p$ available processors during this initial step. Moreover, the keys are distinct since otherwise, appending to each key the address/index of the memory location in which each key is stored can make them so. This is a standard method used in dealing with duplicate keys. In practice, such a method adds a constant-size tag to each key. At worse, the number of comparisons is doubled, thus leading to a 2-optimal rather than an 1-optimal algorithm. In several experimental settings keys can be long strings. If the keys are indeed strings longer than few bytes, this tag overhead may be negligible if the cost of a single key comparison becomes dependent on the length of the comparing key strings. If this is not the case in RandSort, and the tag attachment doubles the number of comparisons when comparing two keys, the performance of RandSort will change. The contribution to the running time of stage 6 described by Lemma 15 will double to roughly $2(1 - 1/2^m)(1 + 2/\omega n)n \log n/p$. The contribution of stage 1 summarized by Lemma 13 will still be $1/2^m(1 + 2/\omega n)n \log n/p$ since a sequential merge-sort is stable and the merging step in Batcher’s algorithm is also (or can be made) stable. By adding the contributions of stages 1 and 6, it is derived that RandSort will be roughly $(2 - 1/2^m)$-optimal. This is unavoidable in randomized sorting since the binary search in traditional parallel randomized sorting or in MultiWaySearch potentially doubles the number of comparisons performed.

In any recursive call of RandSort, variables $N, l, r$ give respectively the number of keys, and the index of the leftmost and rightmost processor participating in that call. The oversampling factor $s$ is set to $s = \lceil \log^\delta n \rceil$, for any $\delta$ such that $1 < \delta < (\log \log n - 1)/2$. 

19
Algorithm 3 RandSort \((n, N, l, r)\) \{ Sort the \(N \leq n\) keys currently assigned to processors \(l, \ldots, r\) \}

1: Set \(s = \lceil \log^\delta n \rceil\) for any \(\delta\) such that \(1 < \delta < (\log \log n - 1)/2\) (the same \(\delta\) is used in all calls).

2: **Stage 1.** \{Lines 3 to 10\}
3: if \((l == r)\) then
4:  SeqSort\(N\); \{Use an \(O(N \log N)\) sequential algorithm eg. mergesort.\}
5:  return;
6: end if
7: if \((l \neq r\) and \(N \leq \log \log \log n\)) then
8:  OddEvenMergeSort\((N, l, r)\); \{ Sort the \(N\) keys in parallel with \(q = r - l + 1\) processors\}
9:  return;
10: end if

11: **Stage 2.** \{Line 12\}
12: The \(q = r - l + 1\) processors collectively select at random without replacement a sample of \(ks - 1\) keys so that \(k\) is a power of two and divides \(q\) (for \(q > k\)). It suffices that \(ks - 1\) is between \(Ks - 1 \leq \lceil N^{1/2} \rceil\) and \(2Ks - 1\) keys, where \(K = \lfloor (\lfloor N^{1/2} \rfloor + 1)/s \rfloor\), in other words \(k\) is chosen to be the power of two between \(K\) and \(2K\).

13: **Stage 3.** \{Line 14\}
14: The sample is sorted using a BSP adaptation of an algorithm in [38].

15: **Stage 4.** \{Line 16\}
16: A balanced \(d\)-ary tree \(T\) is constructed over the \(k\) buckets induced by the \(k - 1\) sorted splitters that evenly partition the sample, where \(d = 2\lceil (N/q)^\alpha \rceil\), for any \(\alpha < 1\). The leaves in the tree are \(k\) values: the \(k - 1\) splitters plus an \(\infty\) sentinel value (indicating a maximum key value).

17: **Stage 5.** \{Line 18\}
18: Every processor decides the bucket destination of every key it holds with respect to the \(k - 1\) splitters using MultiWaySearch on \(T\).

19: **Stage 6.** \{Line 20\}
20: The \(N\) input keys are redistributed according to their bucket destination. If \(q/k \geq 1\), then \(q/k\) processors per bucket, else \(k/q\) buckets per processor are assigned. This is integer sorting, where the \(N\) integers are in the range \([0, 1, \ldots, k - 1]\).

21: **Stage 7.** \{Line 22\}
22: RandSort is called on each bucket to recursively sort its keys.
Description of \textsc{RandSort}.

Throughout \textsc{RandSort} two invariants are maintained: (i) the number of processors $q$ assigned to a recursive call is a power of two, and (ii) the number of splitter $k - 1$ will be such that $k$ is also a power of two. If the number of available processors $p$ is a power of two, (i) will be easy to maintain as long as (ii) is maintained. The sample selection procedure always chooses the appropriate $k$.

In the remainder, every time $N, k$ or $q$ refers to a recursive call of a specific depth $i$, the notation $N_i, k_i, q_i$ will be used instead.

4.3.1. Stage 1

With reference to Algorithm 3, in stage 1, the recursion can terminate for one of two reasons: all relevant keys fall into one processor and thus a sequential algorithm can be used for sorting, or the keys to be sorted in this recursive call are too few even though they are distributed over multiple processors. In this latter case an inefficient parallel algorithm is used that incurs low overhead. In the former case, \textsc{RandSort} is terminated by using an $O(n \lg n)$ worst-case running time sequential sorting algorithm to sort the local keys assigned to processor $l$. If however, there are few keys left distributed over more than one processor, that is $N \leq \lg \lg \lg n$, Batcher’s [5] odd-even merge-sort is used to sort these keys. \textsc{RandSort} does not recur further and subsequently terminates. It is just noted at this point that the bound on the stopping size of $N$ can be lowered to $N \leq \lg \zeta n$, for any $\zeta > 2\delta + 1$.

Any further analysis of stage 1 will be postponed until the very end of this proof. Towards a full analysis, we need to first establish the depth $m$ of recursion and the maximum key imbalance per processor at the very end (depth $m$ recursive calls) so that the running time of lines 4 and 8 can be reliably established.

4.3.2. Stage 2

Sample selection takes place unless \textsc{RandSort} terminated in stage 1. There are two issues related to random sampling: (i) what the best sample and splitter size is, and (ii) how it can be performed in parallel. In \textsc{RandSort}, sample size is chosen as large as possible but no more than about $N^{1/2}$. By maximizing sample size, one can also maximize the available splitter size $k - 1$ as well. The larger the $k$ is the lower the depth $m$ of the recursion is going to be, and thus the lower the communication cost of \textsc{RandSort} will also be. Given than $N$ changes from one recursive call to another, sample and splitter size varies through the course of \textsc{RandSort} as opposed to other approaches [8, 29, 15, 30]. A secondary consideration with splitter selection has to do with the requirement that $k$ is a power of two. Given that $p$ is also a power of two, this makes splitting easier both from a practical and theoretical point of view. If a potentially optimal sample size $N^{1/2}$ does not provide a $k$ that is a power of two then one between $N^{1/2}$ and $2N^{1/2}$ will; the exact value will not affect the asymptotics in the analysis of the performance of this stage. Thus it suffices to adjust $ks - 1$
appropriately (eg. line 12 of Algorithm 3) so that this requirement is satisfied. Sampling in parallel is not dissimilar to sequential sampling. Instead of selecting randomly indexes that correspond to keys stored at the indexed location, one could randomly select global indexes of keys spread over multiple processors and translate a global index into a local index within a processor, then route to that processor these indexes (or equivalently, a count) and let that processor locally determine afterwards the desired sample. The steps of this parallelized sample selection are as follows.

1. Each processor $p_u$ selects uniformly at random with replacement $\lceil (ks - 1)/q \rceil$ (or $\lfloor (ks - 1)/q \rfloor$) processor identifiers, i.e. integers in the range $0, \ldots, p - 1$. The total number of identifiers selected by any processor is bounded above by $\lceil (ks - 1)/q \rceil$. Let $t_{uv}$ be the number of identifiers, a count, so selected identifying processor $p_v$.

2. The $t_{uv}$ count is routed from processor $p_u$ to $p_v$.

3. Processor $p_v$ that receives its own $t_{uv}$ counts for all $u = 0, \ldots, p - 1$, chooses uniformly at random without replacement a set of size $\sum_u t_{uv}$ among its keys.

4. A parallel-prefix sum operation is performed on the $\sum_u t_{uv}$ values.

5. Using the information of the parallel-prefix operation, processor $p_v$ along with every other processor distributes evenly its $\sum_u t_{uv}$ keys among the $q$ processors. Thus the sample of $ks - 1$ keys is now evenly distributed for the sample sorting operation to commence.

For a depth $i$ recursive call, step (1) takes linear time thus requiring $\max\{L, O(N_i^{1/2}/q_i)\}$, since $O(N_i^{1/2})$ is the total number of identifiers generated by line 12.

In order to upper bound the number of processor identifiers generated by any processor in step (3), the generalized Chernoff bound of [9] is claimed. According to [9], $\text{Prob}[S_{n,P} \geq euP/\log u] \leq e^{-unP}$, 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 $n = \lceil N_i^{1/2} \rceil$, $P = 1/q_i$, and $u = \Theta(N_i^{1/2} \log N_i^{1/2})$, with probability at least $1-q_i \exp(-\Theta(N_i \log N_i^{1/2}/q_i))$ no processor generates more than $O(N_i/q_i)$ processor identifiers. It is noted that this is a very rough bound; if a more refined Chernoff bound was used, an $O(N_i^{1/2}/q_i)$ bound could have been derived. The intent here is to maximize the confidence (probability) rather than refine the bound.

Step (2) then requires $\max\{L, O(gN_i/q_i)\}$ time with the corresponding high probability, whether a processor routes collective counts or raw identifiers, and thus step (3) requires time $\max\{L, O(N_i/q_i)\}$ for sample key selection. Since the term $N_i/q_i$ depends on the Chernoff bound considerations, it is again noted that it can be replaced by a $N_i^{1/2}/q_i$ term; the failure probability however increases, and the corresponding confidence decreases. The parallel-prefix sum operation of step (4), by lemma 2, takes computation time $C_{ppf}(q_i)$ and communication time $M_{ppf}(q_i)$. Step (5) requires time $\max\{L, \Theta(gN_i/q_i)\}$.
Overall the computation time for stage 2 is bounded above by \( L, \Theta(N_i/q_i) \) and the communication time by \( 2 \max \{L, \Theta(gN_i/q_i)\} + M_{ppf}(q_i) \) respectively, with probability at least \( 1 - q_i \exp(-\Theta(N_i \log N_i^{1/2}/q_i)) \). It is also noted that the use of asymptotic notation, make no difference between using a sample of size \( N_i^{1/2} \) or \( 2N_i^{1/2} \).

4.3.3. Stage 3

In stage 3, the sample is sorted using the sorting method of [38] which is suited for cases where processors are in abundance relative to the number of available keys. In such cases, all comparisons between any two keys can be performed in few parallel steps followed by parallel-prefix/scan operations that determine the rank of each input key in the (sorted) output sequence. This is further outlined below in a procedure that will be called \textsc{NassimiSahni}. Consider input set \( X = \{x_1, x_2, \ldots, x_{N_i^{1/2}}\} \) of size \( [N_i^{1/2}] \) and \( [N_i^{1/2}]^2 \) processors. Processors are numbered in a mesh-like fashion, so that one can refer to processor at row \( i \) and column \( j \), as \( p_{i,j}, 1 \leq i, j \leq [N_i^{1/2}] \). Input keys are assumed to be distributed evenly (one per processor) among the processors of the first row, i.e., key \( x_1 \) resides in the local memory of processor \( p_{1,j} \). Executing \( [N_i^{1/2}] \) distinct broadcast operations, processor \( p_{1,j} \), for all \( 1 \leq j \leq [N_i^{1/2}] \), broadcasts key \( x_j \) to all processors of column \( j \). This can be implemented in a single segmented parallel-prefix operation [36], where each segment is the set of processors of a given column. Similarly, processor \( p_{i,1} \), for all \( 1 \leq i \leq [N_i^{1/2}] \), broadcasts key \( x_i \) to all processors of row \( i \). This can also be realized by a single segmented parallel-prefix computation. In a single parallel step, processor \( p_{i,j}, 1 \leq i, j \leq [N_i^{1/2}] \), compares \( x_i \) and \( x_j \), and subsequently, by executing \( [N_i^{1/2}] \) distinct parallel sum operations the rank \( \text{rank}(x_i : X) \) of each \( x_i \) in \( X \) is determined. These parallel sums can also be realized by a single segmented parallel-prefix operation. The sorting operation is completed by routing each \( x_i \) to its final destination by processor \( p_{i,1} \). Overall the algorithm requires three segmented parallel-prefix operations for the two broadcast and parallel-sum operations, one parallel comparison step, and the routing of a permutation (each processor sends and receives exactly one key).

When \textsc{NassimiSahni} is applied to stage 3, there are at most \( [N_i^{1/2}] \) sample keys and \( q \) processors. Each one of these \( q \) processors is thus assigned to a \( [(N_i/q)^{1/2}] \times [(N_i/q)^{1/2}] \) segment of computation of \textsc{NassimiSahni}. Thus stage 3 requires computation time

\[
\max \left\{ L, O \left( \frac{N_i}{q_i} \right) \right\} + 3C_{ppf}^{((N_i/q_i)^{1/2})/2}(q_i^{1/2}),
\]

and communication time

\[
\max \left\{ L, O \left( \frac{N_i^{1/2}}{q_i} \right) \right\} + 3M_{ppf}^{((N_i/q_i)^{1/2})/2}(q_i^{1/2}).
\]

It is also noted that the use of asymptotic notation, make no difference between using a sample of size \( N_i^{1/2} \) or \( 2N_i^{1/2} \).
4.3.4. Stage 4

The sorted sample at the end of stage 3 induces \( k - 1 \) sorted splitters that are consequently used to build the static \( d \)-ary tree \( T \) in stage 4. This stage builds static \( d \)-ary tree \( T \) on \( k_i \) leaves that includes the \( k_i - 1 \) splitters plus a sentinel infinity value. Parameter \( d \) is chosen so that \( d_i = 2^\lceil (N_i/q_i)^\alpha \rceil \) is a power of two, for any \( \alpha < 1 \); the choice of \( d_i \) guarantees that the tree is shallow i.e. it has few levels. Throughout section 3 it is assumed that \( k_i \) is an integer power of \( d_i \), besides being a power of two, i.e. \( k_i = d_i^k \). At the end of section 3 it is briefly discussed how one can relax the former condition (integer power of \( d_i \)) and still maintain the claims related to Lemma 4 for MultiWaySearch, as expressed in Lemma 5. Therefore, in this discussion it is assumed that \( k_i \) is an integer power of \( d_i \), but Lemma 5 is claimed.

Let \( h \) be the height of the \( d \)-ary tree \( T \) that will be constructed. Since \( T \) is regular and has \( k_i \) leaves the process of constructing \( T \) is the process of determining the \( k_i - 1 \) internal nodes and the adjacency list (i.e. children) of these nodes. Obviously, \( h = \log_d k_i \) and the construction requires \( \log_d k_i \) phases, which is the number of levels of \( T \). The construction moves bottom up starting from the leaves; the first \( \lceil \log_d (k_i/q_i) \rceil \) of the phases do not involve any communication at all since all the related keys reside within the memory of a single processor. This is because the sample sorting step in stage 3 delivered the sample in sorted order and uniformly distributed among the \( q_i \) processors, and the extracted \( k_i - 1 \) splitters are also sorted and evenly distributed among those processors, \( k_i/q_i \) consecutive keys per processor.

The time required by the first \( \lceil \log_d (k_i/q_i) \rceil \) phases is \( \{ L, O(k_i/q_i) \} \), since \( k_i/q_i \) leaves involve \( k_i/q_i - 1 \) internal nodes. The remaining \( \log_d k_i - \lceil \log_d (k_i/q_i) \rceil \) phases require interprocessor communication and each such phase takes computation time \( \max \{ L, O(d_i) \} \) and communication time \( \max \{ L, O(gd_i) \} \). This is because in these latter phases only \( q_i - 1 \) internal nodes are constructed, and each participating processor receives information from \( d_i \) other processors that facilitate the construction of an internal node from its \( d_i \) children. In these phases several processor become idle after the completion of each phase.

The total computation and communication time complexity of stage 4 is thus bounded respectively by \( \max \{ L, O(k_i/q_i) \} + \lceil \log_d (k_i/q_i) \rceil \max \{ L, O(d_i) \} \) and \( \lceil \log_d (k_i/q_i) \rceil \max \{ L, O(gd_i) \} \), as \( k_i \geq q_i \).

4.3.5. Stage 5

In stage 5, MultiWaySearch is applied to \( T \) to determine the position of each one of the input keys relative to the splitters; the \( k - 1 \) splitters induce \( k \) buckets. The performance of stage 5 will be determined by Lemma 5. At the end of stage 5, the position of each one of the \( N \) keys relative to the \( k - 1 \) splitters is determined; thus a bucket index \( [0, \ldots, k-1] \) can be assigned to each input key.

The height of the balanced \( d \)-ary tree of stage 4 is at most \( \log_d k_i \) and after substituting in lemma 5, for \( n = N_i \), \( k = k_i \) and \( p = q_i \), stage 5 requires computation time \( \lg k_i \lceil N_i/q_i \rceil + \log_d k_i O(N_i/q_i + L) + \log_d k_i O(C_{ppf}(q_i)) \) and communication time \( \log_d k_i O(gN_i/q_i + L) + \log_d k_i O(M_{ppf}(q_i)) \).
4.3.6. Stage 6

Finally in stage 6, the \(N\) input keys are routed so that keys destined for the same bucket end up together. This stage is equivalent to one of integer sorting, where \(N\) integers in the range \([0, 1, \ldots, k-1]\) are sorted. The only other issue that needs to be addressed is whether a bucket spans a single or more than one processors. In this latter case keys destined for the same bucket need to be evenly distributed over the processors assigned to that bucket. If on the other hand, a bucket fits within a single processor’s memory, it might occur that more than one buckets are assigned to that processor. Thus, if \(q/k \geq 1\), \(q/k\) processors are assigned per bucket, otherwise \(k/q\) buckets are assigned per processor (since \(k, q\) are both power of two, the fraction \(k/q\) or \(q/k\) is always an integer).

For the analysis of stage 6 it is assumed 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, as we do, that \(p\) is a power of two and that the number of splitters selected in stage 4 is a power of two as well. Stage 6 is equivalent to one of integer sorting, where the \(N\) integers to be sorted belong to the range \([0, 1, \ldots, k-1]\). The keys can thus be redistributed according to their bucket destination by employing the radix-sort algorithm implied in lemma 3.

Substituting for \(n = N_i, p = q_i, R = k_i\) and \(r = \lceil \frac{\lg k_i}{\lg \max \{\lceil N_i/q_i \rceil, t \lceil \log_t q_i \rceil\} \rceil\) in lemma 3 one gets a computation time for stage 6 of \(3r \max \{L, O(\max \{\lceil N_i/q_i \rceil, t \lceil \log_t q_i \rceil\})\} + rC_{ppf}(q_i)\) and communication time \(2r \max \{L, g(\max \{\lceil N_i/q_i \rceil, t \lceil \log_t q_i \rceil\})\} + rM_{ppf}(q_i)\).

4.3.7. Stage 7

After the keys are redistributed in the newly formed buckets, RANDSORT is called again in stage 7 and this completes the description of the algorithm.

Overall performance of RANDSORT.

A description of RANDSORT has been given and an analysis of the performance of individual stages has been completed. In the remainder, the individual performance metrics will be combined. Towards this, additional results need to be established as follows.

- An upper bound on \(N_i\), the number of keys in any depth \(i\) recursive call of RANDSORT will be derived. Given that RANDSORT is randomized, the corresponding confidence (or respectively, the failure probability) on the bound will also be established. This is Lemma 9.

- The depth \(m\) of recursion is determined in Lemma 10 as a consequence of Lemma 9. It is shown that \(m\) is at most doubly logarithmic.

- Lemma 11 quantifies the maximum key imbalance per processor by providing tight upper and lower bounds for \(N_i/q_i\), the number of keys per processor at any depth \(i\) recursive call. It is noted that because of the two
stopping conditions for stage 1 (lines 3 and 7 of RandSort), this bound might not be of full use for the first case (lines 3 through 6) of stage 1.

- Lemma 12 describes the maximum key imbalance per processor in the form of upper and lower bounds for the base case at depth $m$, for which Lemma 11 is not applicable. It summarizes both the case that one bucket is assigned to each processor as well as the case that multiple buckets are assigned to individual processors. Therefore the running time of stage 1, lines 3-6, can be resolved.

- Lemma 13 expresses the running time of stage 1.

- Lemma 14 expresses the running time of stages 2 through 4.

- Lemma 15 expresses the running time of stage 5 and Lemma 16 that of stage 6.

Combining all contributions as expressed in Lemma 13, Lemma 14, and Lemma 15, Proposition 1 is then derived.

4.3.8. Bound on $N_i$

The bound on $N_i$ in Lemma 9 is derived as follows. Initially there are $N_0 = n$ keys. Roughly $N_0^{1/2}/s$ splitters out of a sample of size $N_0^{1/2}$ split the $N_0$ keys into $N_0^{1/2}/s$ buckets of size approximately $N_1 \approx sN_0^{1/2} = n^{1/2}s$. Then, $N_1^{1/2}/s$ splitters split each bucket of size $N_1$ into buckets of size approximately $sN_1^{1/2} = n^{1/4}s^{1+1/2}$. Thus at depth $i$ one expects $N_i$ to be of size approximately $n^{1/2} s^{1+1/2+\ldots+1/2^{i-1}} = n^{1/2} s^{2^{-1/2^i}}$ with overwhelming probability.

**Lemma 9.** Let $\omega_n$ be any nondecreasing function of $n$ such that $\omega_n > 2$ for sufficiently large $n$. Let $m$ be the depth of the recursion of algorithm RandSort, $n > 65536$, $1 < \delta < (\log \log n - 1)/2$, and $N_i \geq \log \log n$. Then,

$$\Prob \left[ N_i \leq (1+\varepsilon) \frac{n}{2} - \Theta \left( \log^\varepsilon n \frac{n}{(m\omega_n)^2} \right) \right] \geq 1 - n^{-\Omega \left( \varepsilon \frac{n}{(m\omega_n)^2} \right)}$$

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

**Proof:** Consider an arbitrary depth $i$ recursive call of algorithm RandSort, such that $N_i \geq \log \log n$. The number of buckets $k_i$ formed in such a call will be a power of two between $K_i = \lceil (\log N_i^{1/2})/s \rceil$ and $2K_i$, i.e. $K_i \leq k_i \leq 2K_i$. By Lemma 8, for $\varepsilon = 1/(m\omega_n)$, $k = k_i$ and $s = \lceil \log^\varepsilon n \rceil$, with probability $1 - n^{-\Theta \left( \log^\varepsilon n \frac{n}{(m\omega_n)^2} \right)}$, in the $(i+1)$-st recursive call

$$N_{i+1} \leq \lceil (1+\varepsilon)(N_i - k_i + 1)/k_i \rceil + 1.$$
As \( K_i \leq k_i \leq 2K_i \), then \( k_i \leq 2(N_i^{1/2} + 1)/s \) and \( k_i \geq N_i^{1/2}/s - 1 \). By substituting the latter inequality in \( N_{i+1} \leq [(1 + \varepsilon)(N_i - k_i + 1)/k_i] + 1 \)

\[
N_{i+1} \leq [(1 + \varepsilon)(N_i - k_i + 1)/k_i] + 1 \\
\leq [s(1 + \varepsilon)(N_i + 2 - N_i^{1/2}/s)/(N_i^{1/2}/s - s)] + 1 \\
\leq (1 + \varepsilon)(sN_i + s)/(s - N_i^{1/2} - s) + (1 - \varepsilon) \\
\leq (1 + \varepsilon)(sN_i + s)/(N_i^{1/2} - s) + (1 + \varepsilon) \\
\leq (1 + \varepsilon)(sN_i^{1/2} + 2s^2)
\]

provided that \( 1 \leq s^2 - 2s^3/N_i^{1/2} \) is true so that \( (sN_i + s)/(N_i^{1/2} - s) + 1 \leq (sN_i + s)/(N_i^{1/2} + 2s^2) \) and the last inequality in the upper bound can be derived. Inequality \( 1 \leq s^2 - 2s^3/N_i^{1/2} \) is true since \( N_i \geq \log \log n \), \( s = \lfloor \log \delta n \rfloor \) and \( \delta < (\log \log n - 1)/2 \), thus making \( 2s/N_i^{1/2} < 1/2 \). The condition \( n > 65536 \) guarantees that \( n = N_0 \geq \log \log n \).

By using induction on \( i \), and the inequality \( x + y \leq x^{1/2} + y^{1/2} \), for all \( x, y > 0 \), so that one can upper bound expressions such as \( (sN_{i-1} + 2s^2)/(s^{1/2}N_{i-1}^{1/2} + 2^{1/2}s) \), the following is obtained.

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

where \( N_0 = n \). By substituting in this expression for \( s \) the stated bound is derived. ■

4.3.9. The depth \( m \) of recursion

By using Lemma 9 an upper bound on \( m \), the depth of the recursion of RANDSORT is derived.

**Lemma 10.** Algorithm RANDSORT involves

\[
1 \leq m \leq \left[ \log \log n - \log \left( \log \left( \max \left\{ \log \log n, \Theta \left( \frac{n}{p} \right) \right\} \right) - (2\delta + 1) \log \log n - 2 \right) \right]
\]

levels of recursion.

**Proof:** Let \( m \) be the depth of recursion. By Lemma 9 in a depth \( i \) recursive call, \( i \leq m \),

\[
K_i \leq (1 + \varepsilon)^{2 - 1/2^{i-1}} N_0^{1/2} s^{2 - 1/2^{i-1}} + 2(1 + \varepsilon)^2 s^2 i \leq (1 + \varepsilon)^2 n^{1/2} s^2 + 2(1 + \varepsilon)^2 s^2 i
\]

where \( N_0 = n \), \( \varepsilon = 1/(m\omega_n) \), \( \omega_n > 2 \), and \( s = \lfloor \log \delta n \rfloor \). For \( i \leq \log n, n > 65536, m \geq 1 \) and \( (1 + \varepsilon) \leq (1 + 1/2) \).

\[
N_i \leq (1 + \varepsilon)^2 n^{1/2} s^2 + 2(1 + \varepsilon)^2 s^2 i \leq 4n^{1/2} \log^{(2\delta+1)} n.
\]

(7)
Moreover, for all \( i \) such that \( i < m \), by way of stages 1 and 6 of algorithm RandSort, it is derived that

\[
N_i \geq \max \left\{ \lg \lg \lg n, \Theta \left( \frac{n}{p} \right) \right\}.
\]

By combining the previous observations and solving inequality

\[
4n^{1/2^i} \lg (2\delta + 1) n < \max \left\{ \lg \lg \lg n, \Theta \left( \frac{n}{p} \right) \right\}
\]

with respect to \( i \), the stated upper bound on the depth of the recursion is derived.

4.3.10. Upper and lower bounds of \( N_i/q_i \)

Upper and lower bounds are derived on the number of keys per processor at any depth \( i \) recursive call.

**Lemma 11.** An upper bound for \( N_i/q_i \) at depth \( i \) is given by

\[
\frac{N_i}{q_i} \leq (1 + \varepsilon)^i \frac{n}{p} + 4 \lfloor \lg \lg n \rfloor,
\]

and a lower bound is given by

\[
\frac{N_i}{q_i} \geq (1 - \varepsilon)^i \frac{n}{p} - \lfloor \lg \lg n \rfloor.
\]

**Proof:** Implicit in the analysis of stage 6 is that the number of splitters \( k_i \) divides \( q_i \) which is possible if \( p \) and \( k_i \) are powers of two as they are assumed to be. In a recursive call at depth \( i - 1 \) there are \( q_{i-1} \) processors assigned to a bucket. In the next recursive call (one level deeper), \( k_{i-1} - 1 \) splitters are selected that split such a bucket into \( k_{i-1} \) buckets and \( q_{i-1}/k_{i-1} \) processors are assigned to each such newly formed bucket. Although \( q_{i-1} > k_{i-1} \), it is possible that at the bottom of the recursion \( q_{i-1} \leq k_{i-1} \).

By way of Lemma 8, \( N_{i+1} \leq \lfloor (1 + \varepsilon)(N_i - k_i + 1)/k_i \rfloor \), with high probability, and therefore, \( N_{i+1} \leq (1 + \varepsilon)N_i/k_i + 2 \) since \( \varepsilon + (1 + \varepsilon)/k_i \leq 2 \) by way of \( \varepsilon = 1/(m\omega n) \), \( \omega n > 2 \), \( m \geq 1 \), \( k_i \geq 1 \). Similarly, by way of Lemma 7, \( N_{i+1} \geq \lceil (1 - \varepsilon)(N_i - k_i + 1)/k_i \rceil + 1 \), and thus, \( N_{i+1} \geq (1 - \varepsilon)N_i/k_i - 1 \). Upper and lower bounds for \( N_i/q_i \) are then derived.

Since \( N_{i+1} \leq (1+\varepsilon)N_i/k_i + 2 \) and \( q_i = q_{i-1}/k_{i-1} \), \( N_i/q_i \leq (1+\varepsilon)N_{i-1}/(q_{i-1}/k_{i-1}) + 2/q_i \), and therefore, \( N_i/q_i \leq (1+\varepsilon)N_{i-1}/q_{i-1} + 2k_{i-1}/q_{i-1} \). By unfolding the recurrence relation,

\[
\frac{N_i}{q_i} \leq (1 + \varepsilon)^i \frac{N_0}{q_0} + 2 \sum_{j=0}^{i-1} (1 + \varepsilon)^{i-1-j} \frac{k_j}{q_j}.
\]
By way of stages 1 and 6 of algorithm RandSort, it is true that $k_i < q_i$, except possibly at the bottom of the recursion. Then, for all $i \leq m$, $(1 + \varepsilon)^{i-1} \leq (1 + \varepsilon)^m = (1 + 1/(m\omega_n))^m \leq e^{1/\omega_n} < e^{1/2} < 2$, since $\omega_n > 2$ and $\varepsilon = 1/(m\omega_n)$. By Lemma 10, $m \leq \lfloor \lg \lg n \rfloor$ and thus for all $i$ such that $k_i < q_i$, the following inequality holds.

$$
\frac{N_i}{q_i} \leq (1 + \varepsilon)^i \frac{n}{p} + 4 \lfloor \lg \lg n \rfloor. \tag{9}
$$

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

$$
\frac{N_i}{q_i} \geq (1 - \varepsilon)^i \frac{n}{p} - \lfloor \lg \lg n \rfloor. \tag{10}
$$

This completes the proof of Lemma 11. ■

4.3.11. Key imbalance at depth $m$

The previous analysis applies to all calls except possibly the ones at the bottom of the recursion since in such a recursive call $k_{m-1} \geq q_{m-1}$ is a possibility. For that particular case the following claim holds.

**Lemma 12.** In RandSort, for $l = r$, the number of keys in any processor is $N_m[k_{m-1}/q_{m-1}]$ where

$$
N_m[k_{m-1}/q_{m-1}] \leq (1 + \varepsilon)^m \frac{n}{p} + 8 \lfloor \lg \lg n \rfloor + O \left( n/p \lg^{1/2} n + \left( \frac{n}{ps^2} \right)^{1/2} \right),
$$

and

$$
N_m[k_{m-1}/q_{m-1}] \geq (1 - \varepsilon)^m \frac{n}{p} - \lfloor \lg \lg n \rfloor - O \left( \left( \frac{n}{ps^2} \right)^{1/2} \right),
$$

for $n/p = \Omega(\lg n/\lg \lg n)$.

**Proof:** The following analysis handles the case $k_{m-1} \geq q_{m-1}$. At the bottom of the recursion the requirement that $k_{m-1}$ be a power of two is relaxed and thus $k_{m-1} = K_{m-1}$ (see proof of Lemma 9 for a definition of $K_{m-1}$) buckets are formed and allocated to $q_{m-1}$ processors. Every processor is then assigned $[k_{m-1}/q_{m-1}]$ buckets.

Since $N_m \leq (1 + \varepsilon)N_{m-1}/k_{m-1} + 2$ (see proof of Lemma 11), the total size of all $[k_{m-1}/q_{m-1}]$ such buckets on any one processor by Equation (9) and the fact that $(1 + \varepsilon) < 2$ is bounded above as follows.

$$
N_m[k_{m-1}/q_{m-1}] \leq (1 + \varepsilon) \frac{N_{m-1}}{q_{m-1}} + (1 + \varepsilon) \frac{N_{m-1}}{k_{m-1}} + 2k_{m-1}/q_{m-1} + 2
$$

$$
\leq (1 + \varepsilon)^m \frac{n}{p} + 8 \lfloor \lg \lg n \rfloor + (1 + \varepsilon) \frac{N_{m-1}}{k_{m-1}} + 2k_{m-1}/q_{m-1} \tag{12}
$$
Since \( k_{m-1} = K_{m-1} = \lfloor (\lfloor N_{m-1}^{1/2} \rfloor + 1) / s \rfloor \), the following is derived.

\[
k_{m-1} \geq (N_{m-1}^{1/2} - 1 + 1) / s - 1 \geq N_{m-1}^{1/2} / s - 1 \geq N_{m-1}^{1/2} / 2s
\]

Therefore, from the latter inequality, \((1 + \varepsilon)N_{m-1}/k_{m-1} = O(N_{m-1}^{1/2})\), and this further bounds the third term in Equation (11).

By way of Equations (7) and (8) one obtains \( N_{m-1} \leq 4n^{1/2m-1} \log^{(2\delta+1)} n \), and similarly \( N_m \leq 4n^{1/2m} \log n = \Theta(n/p) \). Solving the latter for \( n^{1/2m} \), one obtains that \( n^{1/2m} \leq c_1((n/p)/(4\log^{(2\delta+1)} n)) \) for some constant \( c_1 > 0. \)

Substituting into the inequality involving \( N_{m-1} \) one derives that \( N_{m-1}^{1/2} s = O(n/(\log^{1/2} n)) \), and this completes the bound on the third term in Equation (11).

\[
N_m[k_{m-1}/q_{m-1}] \leq (1 + \varepsilon)^m n^m p + 8 \log \log n + O(n/(\log^{1/2} n)) + 2k_{m-1}/q_{m-1} + 2.
\]  

(12)

A corresponding lower bound can be derived similarly.

\[
N_m[k_{m-1}/q_{m-1}] \geq N_m k_{m-1}/q_{m-1} \geq (1 - \varepsilon)^m n^m p - \log \log n - k_{m-1}/q_{m-1}. \quad (13)
\]

An upper bound for the term \( k_{m-1}/q_{m-1} \) that appears in Equations (12) and (13) is then obtained. By definition, \( k_{m-1} \leq 2\lfloor (\lfloor N_{m-1}^{1/2} \rfloor + 1) / s \rfloor \), and consequently \( k_{m-1} \leq 2(N_{m-1}^{1/2} + 1)/s \). It is also \( N_m \leq (1 + \varepsilon)N_{m-1}/k_{m-1} + 2 \). Since \( N_{i+1} \leq (1 + \varepsilon)N_i/k_i + 2 \) and \( k_{m-2}/q_{m-2} \leq 1 \), by Equation (9) the following bound is obtained.

\[
\frac{k_{m-1}}{q_{m-1}} \leq \frac{2(N_{m-1}^{1/2} + 1)/s}{q_{m-2}/k_{m-2}} \leq \frac{((1 + \varepsilon)(N_{m-2}/k_{m-2}) + 2)^{1/2} + 1}{k_{m-2}} \frac{q_{m-2}}{s} \leq O\left(\frac{(n/p)^{1/2} + (\log \log n)^{1/2}}{s}\right) = O\left(\frac{(n/p)^{1/2}}{s}\right).
\]

By substituting in Equations (12) and (13) for \( k_{m-1}/q_{m-1} \) the following bounds are derived respectively.

\[
N_m[k_{m-1}/q_{m-1}] \leq (1 + \varepsilon)^m n^m p + 8 \log \log n + O\left(n/(\log^{1/2} n) + \left(\frac{n}{ps^2}\right)^{1/2}\right), \quad (14)
\]
and
\[ N_m[k_{m-1}/q_{m-1}] \geq (1 - \varepsilon)^m \frac{n}{p} - \lfloor \log \log n \rfloor - O\left( \left( \frac{n}{ps^2} \right)^{1/2} \right). \quad (15) \]

This analysis completes the separate examination of a call at the bottom of the recursion of RandSort and also completes the proof of Lemma 11.

A detailed analysis of the running time of RandSort based on the results obtained in Lemmas 9-12 follows.

4.3.12. Stage 1 running time

The running time of stage 1 is analyzed first.

**Lemma 13.** The time complexity of stage 1 of RandSort for \( n/p = \Omega(\log n/\log \log n) \), is
\[ \frac{1}{2m} \left( 1 + \frac{2}{\omega_n} \right) \frac{n}{p} \log n + O\left( \frac{n}{p} \delta \log \log n + \frac{n}{p} \log \left( \frac{n}{p} \right) \right). \]
for \( l = r \) and it is pure computation, and
\[ O\left( \frac{n}{p} (\log \log n)^2 \right) + O\left( (\log \log n)^4 (L + \frac{n}{p} + \frac{g n}{p}) \right), \]
for \( l \neq r \) and it includes computation and communication.

**Proof:** The time complexity of stage 1 of RandSort follows by distinguishing two cases.

[Ccase (l = r)] The \([k_{m-1}/q_{m-1}]\) buckets assigned to processor \( l \) are sorted locally (sequentially). Each bucket can be sorted in time \( N_m \log N_m \) for a total of \( N_m \log N_m \log [k_{m-1}/q_{m-1}] \). An upper bound on \( N_m[k_{m-1}/q_{m-1}] \) is given by Equation (14) of Lemma 12, and therefore, by Equation (7) for \( i = m \) the following bound on the time complexity of this step is derived.

\[
\begin{align*}
\frac{k_{m-1}}{q_{m-1}} N_m \log N_m & \leq \left( (1 + \varepsilon)^m \frac{n}{p} + 8 \lceil \log \log n \rceil + O\left( \frac{n}{p \log \log \log n} + \left( \frac{n}{ps^2} \right)^{1/2} \right) \right) \log N_m \\
& \leq \left( (1 + \varepsilon)^m \frac{n}{p} + 8 \lceil \log \log n \rceil + O\left( \frac{n}{p \log \log \log n} + \left( \frac{n}{ps^2} \right)^{1/2} \right) \right) \log \left( 4n^{1/2m} \log^{(2\delta+1)} n \right).
\end{align*}
\]

The last term \( \log \left( 4n^{1/2m} \log^{(2\delta+1)} n \right) \) above is bounded by \( \log n/2^m + O(\delta \log \log n) \).

The contribution of the first term \( \log n/2^m \) is important when multiplied with \((1+\varepsilon)^mn/p\). For the other products, the bound \( \log n/2^m = O(\log (n/p) + (\log \log n)^2) \)
is used, which is derived by taking logarithms of both sides of Equation (8) for \( i = m \). Since for \( \omega_n > 2 \),
\[
(1 + \varepsilon)^m \leq e^{1/\omega_n} \leq (1 + 2/\omega_n)
\]
and \( s = \lceil \lg^4 n \rceil \) the following bound for local sequential sorting holds.
\[
\frac{k_{m-1}}{q_{m-1}} N_m \lg N_m \leq \frac{1}{2^m} \left( 1 + \frac{2}{\omega_n} \right) \frac{n}{p} \lg n + O \left( \frac{n}{p} \delta \lg \lg n + \frac{n}{p} \frac{\lg (n/p)}{(\lg n)^{1/2}} \right).
\]

[Case \( l \neq r \) and \( N_m \leq \lg \lg n \)] 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 algorithm [5], where comparators are replaced by \( \lceil N_m/q_m \rceil \)-mergers [34]. Each processor is assigned an \( N_m/q_m \)-size segment of the single bucket. Each such segment is first sorted locally in time \( (N_m/q_m) \lg (N_m/q_m) \).

For \( N_m \leq \lg \lg n \) it is \( \lg N_m \leq \lg \lg^2 n \). As \( l \neq r \) it is \( k_m < q_m \), and therefore, by Equations (9) and (16), and \( N_m \leq \lg \lg n \), the following bound for the time complexity of this local sorting step is derived.
\[
\frac{N_m}{q_m} \lg \frac{N_m}{q_m} \leq \left( (1 + \varepsilon)^m \frac{n}{p} + 4 \lceil \lg \lg n \rceil \right) \lg N_m
\leq \left( (1 + \varepsilon)^m \frac{n}{p} + 4 \lceil \lg \lg n \rceil \right) (\lg \lg n)^2
= O \left( \frac{n}{p} (\lg \lg n)^2 \right).
\]

The analysis of this step concludes 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 \( q_m \leq N_m \), then \( \lg q_m = \lg N_m = O((\lg \lg n)^2) \). The odd-even merge-sort phase requires computation time (\([19]\))
\[
O \left( (\lg \lg n)^4 \max \{L, N_m/q_m\} \right) = O \left( (\lg \lg n)^4 \max \left\{ L, \frac{L}{p} \right\} \right),
\]
and communication time (\([19]\))
\[
O \left( (\lg \lg n)^4 \max \{L, gN_m/q_m\} \right) = O \left( (\lg \lg n)^4 \max \left\{ L, g \frac{n}{p} \right\} \right).
\]

4.3.13. Stage 2-4 running time

The running time of stages 2-4 combines the bounds derived from the description of RANDSORT and the previous derivations related to the number of keys per processor at various recursive calls.
Lemma 14. The computation time of stages 2-4 of RandSort is

\[ O \left( L \log n + \frac{n}{m} \log n \right), \]

and the communication time is

\[ O \left( L \log n + g \frac{m}{p} m + g \log n \right), \]

for computation and communication respectively.

Proof: The examination of the complexity of stages 2-4 of algorithm RandSort is completed by substituting in lemmas 1 and 2, for \( t = 2 \), and observing that

\[ \sum_{i=0}^{m-1} \log q_i = O(\log n + \delta m \log \log n) = O(\log n), \]  

(17)

which is a result of Lemma 9, Equation (7), and \( q_i \leq N_i \).

The time complexity of stage 2 is summarized below.

\[ \sum_{i=0}^{m-1} \left( \max \{ L, \Theta(N_i^{1/2}/q_i) \} + \max \{ L, \Theta(N_i/q_i) \} + T_{ppf}(q_i) + 2 \max \{ L, \Theta(gN_i/q_i) \} \right). \]

The time complexity of stage 3 is

\[ \sum_{i=0}^{m-1} \left( \max \{ L, O(gN_i^{1/2}/q_i) \} + \max \{ L, O(N_i/q_i) \} + T_{ppf}^{[1/(N_i/q_i)^{1/2}]}(q_i^{1/2}) + 2M_{brd}^{[1/(N_i/q_i)^{1/2}]}(q_i^{1/2}) \right). \]

The time complexity of stage 4 shown below is subsumed in those of stages 2-3.

\[ \sum_{i=0}^{m-1} \left( \max \{ L, O(k_i/q_i) \} + \lceil \log_{d_i} q_i \rceil \max \{ L, O(d_i) \} + \lceil \log_{d_i} q_i \rceil \max \{ L, O(gd_i) \} \right). \]

The time for parallel-prefix and broadcasting operations is \( \sum_i O(L \log q_i + \log q_i + g \log q_i + gn/p + n/p) \) as it involves \( (N_i/q_i)^{1/2} \) elements in the worst case of these operations and is therefore by Equation (17) \( O(L \log n + \log n + g \log n + gmn/p + mn/p) \).

The computation and communication time of stages 2-4 for all calls of RandSort is then \( O(L \log n + mn/p + \log n) \) and \( O(L \log n + gmn/p + g \log n) \) respectively.
4.3.14. Stage 5 running time

The following summarizes the contribution to the running time of the tree-search component of RANDSORT.

Lemma 15. The computation time of stage 5 is

\[(1 - \frac{1}{2^m}) \left(1 + \frac{2}{\omega_n}\right) \frac{n}{p} \lg n + O \left(\frac{n}{p} \left(\frac{\lg n}{\lg \frac{n}{p}} + m^2\right) + \frac{\lg n \lg p}{\lg \frac{n}{p}} + L \left(\frac{\lg n \lg p}{\lg \frac{n}{p}} + m\right)\right).\]

and the communication time is

\[O \left(\frac{g n}{p} \left(\frac{\lg n}{\lg \frac{n}{p}}\right) + \frac{\lg n \lg p}{\lg \frac{n}{p}} + L \left(\frac{\lg n \lg p}{\lg \frac{n}{p}}\right)\right).\]

Proof: Stage 5, for any depth \(i\) recursive call, requires time

\[\lg k_i[\frac{N_i}{q_i}] + \log_{d_i} k_i O(N_i/q_i + L) + \log_{d_i} k_i O(C_{ppf}(q_i))\]

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 up these contributions for all \(0 \leq i \leq m - 1\), and noting that \(k_i < q_i\) an expression for the running time of stage 5 is derived.

The sum \(\sum_{i=0}^{m-1} \lg k_i\) is first considered. By way of \(\ln(1 + x) \leq 1.5x\), [9] one obtains that \(\lg(x + y) \leq \lg x + 1.5(y/x)\) for any \(x, y > 0\). Then, \(\lg(x + y)^{1/2} \leq \lg x/2 + y/x\). By definition, \(k_i \leq 2(N_i^{1/2} + 1)/s\), where \(s = \lfloor \lg^4 n \rfloor\), and because of the analysis of stage 1, \(N_i \geq \lg^6 \lg^5 n\), for all \(i < m\). Then,

\[\sum_{i=0}^{m-1} \lg k_i \leq \sum_{i=0}^{m-1} \lg 2(N_i^{1/2} + 1) - m \lg s\]

\[\leq \sum_{i=0}^{m-1} (\lg N_i^{1/2}) + m + \frac{1.5m}{\lg \lg n/2} - m \lg s.\]

The first term can be further bounded by using Equality (6).

\[\sum_{i=0}^{m-1} \lg N_i^{1/2} \leq \sum_{i=0}^{m-1} \left(1 - 1/2^i\right) \lg (1 + \varepsilon) + (1/2^i) \lg n + (1 - 1/2^i) \lg s + 2i((1 + \varepsilon)s/n^{1/2})^{1/2^i} - 1\]

\[\leq \sum_{i=0}^{m-1} (1 - 1/2^i) \lg (1 + \varepsilon) + (1/2^i) \lg n + (1 - 1/2^i) \lg s + 2i\]

\[\leq m \lg (1 + \varepsilon) + (1 - \frac{1}{2^m}) \lg n + m \lg s + m(m - 1).\]
By combining the previous inequalities the following bound holds.

\[
\sum_{i=0}^{m-1} \lg k_i \leq m \lg (1 + \varepsilon) + \left(1 - \frac{1}{2^m}\right) \lg n + m^2 + \frac{1.5m}{\lg \lg n/2} \leq O(m^2) + (1 - \frac{1}{2^m}) \lg n
\]

The analysis of stage 5 is completed by observing that \(d_i = 2\lceil \frac{N_i}{q_i} \rceil\), for some constant \(\alpha < 1\), that the term \(N_i/q_i\), \(i < m\), is bounded by Lemma 11, and \((1 + \varepsilon)^i\) and \(\sum \lg q_i\) are bounded by Equations (16) and (17) respectively.

The following bound on the computation time of stage 5, by also substituting in lemma 2, for \(t = 2\) is derived.

\[
\sum_{i=0}^{m-1} \left(\lg k_i \left\lceil \frac{N_i}{q_i} \right\rceil + \log_{d_i} k_i \max \{L, O\left(\frac{N_i}{q_i}\right)\} + \log_{d_i} k_i O(C_{\text{ppf}}(q_i))\right)
\leq \left(1 - \frac{1}{2^m}\right) \left(1 + \frac{2}{\omega_n}\right) \frac{n}{p} \lg n + O\left(\frac{n}{p} \left(\frac{\lg n}{\lg p} + m^2\right) + \frac{\lg n \lg p}{\lg p} + L \frac{\lg n \lg p}{\lg p}\right).
\]

Similarly, the following bound for the communication time of stage 5 is obtained.

\[
O\left(\frac{n}{p} \left(\frac{\lg n}{\lg p} + m^2\right) + g \frac{\lg n \lg p}{\lg p} + L \frac{\lg n \lg p}{\lg p}\right).
\]

4.3.15. Stage 6 running time

The following can then be established for stage 6.

**Lemma 16.** The computation time of stage 6 of **RandSort** is

\[
O\left(\frac{n}{p} \left(\frac{\lg n}{\lg p} + m^2\right) + \frac{\lg n \lg p}{\lg p} + L \left(\frac{\lg n \lg p}{\lg p} + m\right)\right).
\]

and the communication time is

\[
O\left(\frac{n}{p} \left(\frac{\lg n}{\lg p} + m\right) + g \frac{\lg n \lg p}{\lg p} + L \frac{\lg n \lg p}{\lg p}\right).
\]

**Proof:** By substituting in lemma 3, for \(t = 2\) and by a way of arguments similar to that employed in the analysis of stage 5, the computation and communication time complexity of stage 6 is subsumed in that of stage 5.
4.3.16. Computation and communication time of RandSort

The proof of Proposition 1 is now complete. The running time over all stages of RandSort is given by the following formulae where it is noted that $m \leq \lceil \lg \lg n \rceil$ and $\delta \leq (\lg \lg n - 1)/2$. By combining the results of Lemma 13, Lemma 14, Lemma 15, Lemma 16, the following is obtained.

**Case** $l = r$

The computation time $C(n, p)$ is given by the following expression.

$$C(n, p) = (1 + \frac{2}{\omega_n}) \frac{n \lg n}{p} + O\left(\frac{n}{p}(\delta \lg \lg n + m^2 + \frac{\lg n}{\lg \frac{n}{p}} + \frac{\lg (n/p)}{(\lg n)^{1/2}}) + \frac{\lg n \lg p}{\lg \frac{n}{p}} + L \lg n + L \frac{\lg n \lg p}{\lg \frac{n}{p}}\right).$$  \hspace{1cm} (18)

The communication time $M(n, p)$ is given by the following expression.

$$M(n, p) = O\left(\frac{L \frac{\lg n \lg p}{\lg \frac{n}{p}}}{\lg \frac{n}{p}} + L \lg n + g \frac{n \lg n}{\lg \frac{n}{p}} + g \frac{\lg n \lg p}{\lg \frac{n}{p}} + g \lg n + \frac{n}{p} m q\right).$$  \hspace{1cm} (19)

**Case** ($l \neq r$ and $N \leq \lg \frac{\lg n}{n}$)

The computation time $C(n, p)$ is given by the following expression.

$$C(n, p) = (1 - \frac{1}{2^m}) \left(1 + \frac{2}{\omega_n}\right) \frac{n \lg n}{p} + O\left(\frac{n}{p} \frac{\lg^4 n}{\lg \frac{n}{p}} + \frac{\lg n}{\lg \frac{n}{p}} + \frac{n}{p} \frac{\lg n \lg p}{\lg \frac{n}{p}} + L \lg n + L \frac{\lg n \lg p}{\lg \frac{n}{p}}\right).$$  \hspace{1cm} (20)

The communication time $M(n, p)$ is given by the following expression.

$$M(n, p) = O\left(g \frac{n}{p} \frac{\lg^4 n}{\lg \frac{n}{p}} + L \frac{\lg n \lg p}{\lg \frac{n}{p}} + L \lg n + g \frac{n \lg n}{\lg \frac{n}{p}} + g \frac{\lg n \lg p}{\lg \frac{n}{p}} + g \frac{n}{p} m q\right).$$  \hspace{1cm} (21)

The running time as expressed in the statement of Proposition 1 combines the terms of the two cases for computation and communication time.

The failure probability of stage 2 for a depth $i$ recursive call is $q_i \exp(-\Theta(N_i \lg N_i^{1/2}/q_i))$. Therefore the failure probability for stage 2 over all levels of recursive calls is at most the sum of individual failure probabilities

$$\sum_{i=0}^{m-1} p e^{-\Theta(N_i \lg N_i^{1/2}/q_i)} = O(n \lg \lg n) e^{-\Theta((n/p) \lg \lg^2 n)} = n^{-\Theta((n/p) \lg \lg^2 n)},$$

since by Lemma 11, $N_i/q_i = \Theta(n/p + \lg \lg n)$, the $N_i \geq \lg \frac{\lg n}{n}$, for all $i < m$, induces $\lg N_i \geq \lg \lg^2 n$.

By Lemma 9 the failure probability of a bucket having the claimed size $n^{-\Theta((\lg^2 n)/(m \omega_n^2))}$. As the total number of induced buckets is $n$, the failure probability for the claim to be observed remains asymptotically the same.
The failure probability of algorithm RandSort is thus the sum of the two individual failure probabilities \( n^{-\Theta\left(\frac{\lg^2\delta}{m\omega_n}\right)^2} \) and \( n^{-\Theta\left(\frac{n}{p}\lg\lg n\right)} \).

5. Three interesting cases for RandSort

By substituting appropriate values for the parameters in Proposition 1 a variety of bounds on the runtime of algorithm RandSort can be derived. As noted in [15], the optimal time bounds are derived by selecting \( m, \delta \) and \( \omega_n \) with regard to the actual values of \( n, p, L \) and \( g \). By making, however, more generic selections for \( m, \delta, \omega_n \), and considering worst-case bounds for \( p, L \) and \( g \), more general, but less tight results can be obtained. The following is one such bound.

**Theorem 1.** For any positive constants \( c \) and \( \epsilon \), \( 0 < \epsilon < 1 \), algorithm RandSort sorts \( n \) keys that for all positive integers \( n \geq 65536 \), \( p \) and \( L \) satisfying \( p = n^{1-\epsilon} \) and \( L = o(n^{\epsilon}) \) it has, with probability \( 1 - n^{-\Theta(\frac{n}{\lg n})} \), \( \pi = 1 + o(1) \), and if \( g = o(\lg n) \) also, \( \mu = o(1) \).

**Proof:** Let \( \omega_n = \lg \lg n \) and \( \delta \) a constant such that \( \delta > c + 1 \) in Proposition 1. Then, by Equation (8) it is obtained that \( m = \Theta(1) \). Algorithm RandSort always terminates for \( l = r \) and therefore by Equations (18) and (19) the values for \( g \) and \( L \) claimed in the statement are derived.

Compared to the stated result of [15, Theorem 5] the probability of failure \( n^{-\Theta(\frac{\lg^2\omega_n}{\lg n})} \) of RandSort as expressed in Theorem 1 is substantially lower than the one in [15] which is \( 1/n^\zeta \), for constant \( 0 < \zeta < 1 \).

By allowing the depth of the recursion to vary with \( n \) rather than being a constant, RandSort achieves 1-optimality for smaller slack, i.e. for values of \( p \) much closer to \( n \) than those of Theorem 1.

**Theorem 2.** For any constant \( \alpha \geq 1 \), algorithm RandSort sorts \( n \geq 65536 \) keys such that for all positive integers \( n, p \) and \( L \) satisfying \( p = n/\lg^\alpha n \) and \( L = o(\lg n) \), it has, with probability \( 1 - n^{-\Theta(\frac{n\lg \lg n}{\lg n})} \), \( \pi = 1 + o(1) \), and if \( g = o(\lg \lg n) \) also, \( \mu = o(1) \).

**Proof:** Let \( \omega_n = \lg \lg n \) and \( \delta > 1 + \alpha \) in Proposition 1.

Compared to the stated result of [15, Theorem 6] the probability of failure \( n^{-\Theta(\frac{n\lg \lg n}{\lg n})} \) of RandSort as expressed in Theorem 2 is substantially lower than the one in [15] which is \( 1/n^\zeta \), for constant \( 0 < \zeta < 1 \). The probability of failure of [15, Theorem 6] reflects a choice of \( \omega_n = (\lg n)^{1/2} \) as opposed to \( \omega_n = \lg \lg n \) in Theorem 2. A choice of \( \omega_n = \lg \lg n \) in [15, Theorem 6] could yield a probability of failure of \( n^{-o(\lg n)} \), which is still substantially worse than the one of Theorem 2. Slackwise, it seems that Theorem 2 is only marginally better than [15], since the former holds for \( a \geq 1 \) and the latter holds for \( a > 1 \).
However this is not true for Proposition 1, as it can yield a stronger theorem than the stated Theorem 2, where 1-optimality is possible for even smaller slack as long as \( p = o(n \lg \lg n/\lg n) \). Such a case however requires that \( L, g \) attain lower values than the bounds stated in Theorem 2.

The following theorem is also derived for \( p \) closer to \( n \), for which the available slack is as small as possible. In this case, RANDSORT resembles a PRAM algorithm and the only supported values for \( L \) and \( g \) are constant. 1-optimality is not achievable any more; a constant-optimality can only be claimed, with communication time contributing significantly to the overall running time.

**Theorem 3.** Algorithm RANDSORT sorts \( n \) keys such that for all positive integers \( n \geq 65536 \), \( p \) and \( L \) satisfying \( p = n \lg \lg n/\lg n \) and \( L = O(1) \), it has, with probability \( 1 - n^{-\Theta(\lg n \lg \lg n)} \), \( \pi = O(1) \), and if \( g = O(1) \) also, \( \mu = O(1) \).

**Proof:** A constant \( \delta > 1 \) suffices for the bound to hold. \( \blacksquare \)

Theorem 3 is of importance only when compared to PRAM-like results, i.e. cases where \( L, g \) are bounded by constant values. It is noted that if the simulation of a PRAM randomized sorting algorithm on the BSP model was to be undertaken, the constants involved in such a simulation, would be potentially higher than those implied by Theorem 3, because of various other factors, such as hash-function evaluation time and simulation overhead. The slack (i.e. \( n/p = \Omega(\lg n) \)) required for such a simulation would make such a simulation-based result worse than the one claimed through Theorem 3 that requires \( p = n \lg \lg n/\lg n \).

RANDSORT does not need to be as complicated as it appears to be. It is general enough so that it can work for all claimed values of \( p \) and the other parameters described in Proposition 1. If the values of \( p \) and \( n \) are known in advance, several optimizations can be undertaken. If for example \( p \) is approximately \( n^{1/2} \), the depth \( m \) of the underlying recursion is one or two, and the parallel \( d \)-ary tree can be replicated by a broadcast operation, rather than be built in a distributed fashion. Then MULTIWAYSEARCH becomes a sequential \( d \)-ary search method.

It is noted that not only can the bounds given above be expressed in a variety of ways, but also RANDSORT itself leaves room for several improvements. As stated, it sorts the sample of size \( N^{1/2} \) by employing a BSP adaptation of a sorting algorithm of [38]. 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 [15]. The recursion may terminate for smaller values of \( N \) than \( \lg \lg n \). Therefore, if it is terminated as soon as \( N \leq \lg \beta n \), for any \( \beta = o(\lg \lg n) \), the contribution to the lower order terms of Batcher’s sorting algorithm is \( O((n/p)(\beta^2 \lg^2 n)) \) 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 RANDSORT could also be realized with higher probability by utilizing integer sorting techniques.

38
6. Conclusion

In this work a BSP randomized oversampling-based algorithm RandSort is introduced that sorts $n$ keys on $p$ processors in time that is asymptotically optimal provided that $n/p = \Omega((\log n)/\log \log n)$. In such a case, a speedup of $O(p)$ is possible. If the slack $n/p$ is allowed to be larger that is, if $n/p > \log^{\alpha} n$, for any constant $\alpha \geq 1$, then the total number of comparisons performed is comparable to that of sequential sorting up to low order terms, and optimal speedup of $p/(1 + o(1))$ can be claimed instead of $O(p)$. RandSort combines and utilizes some well-known algorithmic techniques. The combination of these techniques in the various steps of the algorithm is our own contribution. The careful choice of a number of relevant parameters makes the underlying analysis complicated yet it yields 1-optimal results, thus advancing our understanding of randomized sorting on parallel systems, whether they are latency-tolerant or PRAM-like.

Early partial support for this work was provided by an EPSRC grant GR/K16999 (UK), and subsequent work of the first author was supported in part in part by NSF/EIA-9977508 and NSF/ITR IIS-0324816.

References


