Programming Research Group

DYNAMIC DATA STRUCTURES: THE BSP APPROACH

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

PRG-TR-11-97

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

Dynamic Data Structures: The BSP Approach*

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

Computing Laboratory, Oxford University, Oxford OX1 3QD, U.K.
{alexg,cs}@comlab.ox.ac.uk

March 1997

Abstract

In this work we present efficient and practical randomized data structures on the Bulk-Synchronous Parallel (BSP) model of computation along with an experimental study of their performance. In particular, we study data structures for the realization of Parallel Priority Queues (PPQs) and Parallel Dictionaries (PDs). We show that our algorithms are communication efficient and achieve optimality to within small multiplicative constant factors for a wide range of parallel machines. We also present an experimental study of our PPQ algorithms on a variety of parallel platforms, including the SGI Power Challenge and the Cray T3D.

Keywords BSP model - randomized algorithms - randomized data structures - hashing - parallel priority queues - parallel dictionaries.

*The work of the first author was supported in part by EPSRC(UK) under grant GR/K16999, and the second author was supported by a Bodossaki Foundation Graduate Scholarship.

1 Introduction

In this work we present an architecture independent study of the computation and communication requirements of an efficient Parallel Priority Queue (PPQ) and a Parallel Dictionary (PD) along with an experimental study of the former's data structure performance[1]. A PPQ is an abstract data structure that generalizes the sequential priority-queue data structure [29]. A p-processor PPQ allows the simultaneous insertion of n new items, n/p per processor, and the simultaneous deletion of the n items associated with the smallest values. As noted in [37], PPQs can be employed to efficiently implement several parallel algorithms, such as *branch-and-bound* [26, 27, 28], which require the solution of several sub-problems, each associated with a possibly different cost. By utilizing PPQs, processors can insert at each phase the newly generated sub-problems and efficiently select the n "global best" sub-problems to be solved in the next phase. Accordingly, a PD is an abstract data structure that generalizes the sequential dictionary data structure [29]. A p-processor PD allows the simultaneous insertion of n new items, n/p per processor, and the simultaneous deletion of n items. PDs can be employed to efficiently implement a multiplicity of parallel applications [24]. In this work we establish time bounds for PPQ and PD operations as a function of the available communication throughput of the system. The computational model adopted is the Bulk-Synchronous Parallel (BSP) model, proposed by L. G. Valiant [44], which deals explicitly with the notion of communication and synchronization among computational threads.

1.1 The BSP model

The main reason for the moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von-Neumann model has been for sequential computing. The introduction of the Bulk-Synchronous Parallel (BSP) model by L. G. Valiant comes to address these limitations and bridge the software and hardware worlds with respect to parallel computing. The model offers a high-level abstraction of hardware with the purpose of allowing parallel programs to be scalable and run efficiently on diverse hardware platforms. A BSP computer as introduced in [43, 44] and also examined in [11, 32, 33, 45] consists of three parts:

(1) a *collection of processor-memory components*,

(2) a *communication network* that can deliver messages point-to-point among the components, and

(3) a *facility for global synchronization*, in barrier style, of all or a subset of the components.

A BSP computer is parametrized by

(1) p – the number of processor-memory components,

(2) L – the minimal time, measured in terms of basic computational operations, between successive synchronization operations, and

(3) 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.

Computation on the BSP model proceeds in a succession of *supersteps*. A superstep may be thought of as a segment of computation during which each processor performs a given task using data already available there locally before the start of the superstep. The task may include (i) local computations, (ii) message transmissions, and (iii) message receipts. Each superstep is charged $\max\{L, x + gh\}$ time steps, where x is the maximum number of basic computational operations executed by any processor during a superstep and h is the maximum number of messages transmitted or received by any processor. In the remainder of this work we use $\log x$ and $\lg x$ to denote the logarithm of x to the base e and 2 respectively.

[1] A preliminary version of this work appeared in [14]

1.2 Related Work and New Results

The first approach on parallel priority queues is implied in the work of [27, 28], where a parallel randomized algorithm to approximate a priority queue is presented. The algorithm of [27, 28] is extended to handle exact priority queues in [39]. Deterministic algorithms for maintaining parallel priority queues are derived in [37]. Two new data structures are introduced, the n-Bandwidth-Heap and the n-Bandwidth-Leftist-Heap, that are obtained as extensions of the sequential binary-heap and leftist-heap. These approaches fail, however, to deliver optimal performance when directly implemented on the BSP model of computation. The only previously known results on BSP PPQ algorithms appear in [3, 14, 15]. In [14, 15], a new data structure is presented, the (n,d)-Bandwidth-Concurrent-Heap that is obtained as an extension of the n-Bandwidth-Heap [37]. In this work we improve upon the communication requirements of previous results on parallel priority queues [28, 37, 39, 40], and extend the BSP algorithms of [15] by presenting efficient and simple (non-pipelined) algorithms. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters. The randomized algorithms we present are based on the ideas of [27, 28] as extended in [39]. These results are summarized in tables 1 and 2. With reference to these tables, we assume an exclusive read exclusive write parallel model of computation. All algorithms operate on a data structure of size $m \geq n$ for sets of n items, where $h = \lg(m/n)$ and $\bar{h} = \lg(m/p)$ and $m \leq 2^{n^c}$, for any constant $c > 0$. The conventions underlying the τ notation are explained in section 2.1.

	Computation			
Reference	InsertQ	DeleteMinQ	FindMinQ	Max Processors
[37]	$O(\frac{n}{p}h + \frac{n}{p}\lg n)$	$O(\frac{n}{p}h + \lg p)^*$	$O(1)$	$O(n)$
[39]	$\tau_{IQ}((1+o(1))\frac{n}{p}, \frac{m}{p})$	$\tau_{DQ}((1+o(1))\frac{n}{p}, \frac{m}{p}) + O(\frac{n \lg n}{p})$	$\tau_{FQ}((1+o(1))\frac{n}{p}, \frac{m}{p}) + O(\frac{n \lg n}{p})$	$O(\frac{n}{\lg n})$
[15]	$O(\frac{n}{p}h + h\lg p)$	$O(\frac{n}{p}h + h\lg p)$	$O(1)$	$O(\frac{n}{\lg n})$
[15] (amortized)	$O(\frac{n}{p}h + \lg p)$	$O(\frac{n}{p}h + \lg p)$	$O(1)$	$O(\min\{n, \frac{n}{\lg n}h\})$
This work	$\tau_{IQ}((1+o(1))\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p})$	$\tau_{DQ}((1+o(1))\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p} + \lg p)$	$\tau_{FQ}((1+o(1))\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p} + \lg p)$	$O(\frac{n}{\lg n})$

Table 1: A * sign indicates that the associated result has been adapted to fit our model.

	Communication			
Reference	InsertQ	DeleteMinQ	FindMinQ	Max Processors
[37]	$O(\frac{n}{p}h + \frac{n}{p}\lg n)$	$O(\frac{n}{p}h + \lg p)^*$	$O(1)$	$O(n)$
[39]	$(1+o(1))\frac{n}{p}$	$O(\frac{n}{p}\lg n)$	$O(\frac{n}{p}\lg n)^*$	$O(\frac{n}{\lg n})$
[15]	$O(\frac{n}{p} + h\lg p) + o(\frac{n}{p}h)$	$O(h\lg p) + o(\frac{n}{p}h)$	$O(1)$	$O(\frac{n}{\lg n})$
[15] (amortized)	$O(\frac{n}{p} + h + \lg p) + o(\frac{n}{p}h)$	$O(\frac{n}{p} + h + \lg p) + o(\frac{n}{p}h)$	$O(1)$	$O(\min\{n, \frac{n}{\lg n}h\})$
This work	$(1+o(1))\frac{n}{p}$	$O(\lg p) + o(\frac{n}{p})$	$O(\lg p) + o(\frac{n}{p})$	$O(\frac{n}{\lg n})$

Table 2: A * sign indicates that the associated result has been adapted to fit our model.

Similar results to ours are obtained in [3] on an extension of the BSP model called BSP*. The BSP* model takes into explicit account the size of communicated messages. The original BSP model can also take message size into consideration by absorbing message size, for small messages, into one of its parameters, L, and for larger messages, into another parameter, g. Our results compare favorably to those in [3] in various ways. The selection algorithm presented in [3] requires $O(n/p)$ computation time as opposed to the $(n+C)/p + o(n/p)$, where C is the statistic to be found, obtained in [14, 15]. The communication time of the algorithm in [3] is also higher than the one in this work. As a consequence, the priority queue algorithms presented in [3] are slower in both computation and communication. In addition, our algorithms work for values of p much closer to n than those of [3] even when differences in the two models are taken into consideration.

The first approach on parallel dictionaries is implied in the work of [36], where a parallel implementation of 2-3 trees is presented. Randomized algorithms for maintaining parallel dictionaries are derived in [38]. These approaches fail, however, to deliver optimal performance when directly implemented on the BSP model of computation. The only previously known results related to BSP PD algorithms appear in [12, 16]. In [12, 16], a new static data structure is presented, the ordered h-level graph, that efficiently supports searching, but no inserting or deleting. Ordered h-level graphs can be extended (along the lines of [4]) to handle dynamic operations. Unfortunately, the resulting algorithms are unduly complicated and of little practical significance. In this work, we improve upon the communication requirements of previous results on parallel dictionaries [12, 16, 36, 38], and extend the BSP algorithms of [12, 16] by presenting efficient and simple algorithms. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters. The randomized algorithms we present are based on the ideas of [14, 15].

These results are summarized in tables 3 and 4. With reference to these tables, we assume an exclusive read exclusive write parallel model of computation. All algorithms operate on a data structure of size $m \geq n$ for sets of n items, where $h = \lg m$. The conventions underlying the τ notation are explained in section 4.1. [2]

	Computation			
Reference	InsertD	DeleteD	FindD	Max Processors
[36]	$O(\frac{n}{p}(h+\lg n))$	$O(\frac{n}{p}(h+\lg n))$	$O(\frac{n}{p}(h+\lg n))$	$O(n)$
[38]	$O(\frac{n}{p}(h+\lg n))^*$	$O(\frac{n}{p}(h+\lg n))^*$	$O(\frac{n}{p}(h+\lg n))^*$	$O(\frac{n}{\lg^2 n \lg\lg n})$
This work	$(1+o(1))\tau_{ID}(\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p})$	$(1+o(1))\tau_{DD}(\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p})$	$(1+o(1))\tau_{FD}(\frac{n}{p}, \frac{m}{p}) + O(\frac{n}{p})$	$O(\frac{n}{\lg n})$

Table 3: A * sign indicates that the associated result has been adapted to fit our model.

	Communication			
Reference	InsertQ	DeleteMinQ	FindMinQ	Max Processors
[36]	$O(\frac{n}{p}(h+\lg p))$	$O(\frac{n}{p}(h+\lg p))$	$O(\frac{n}{p}(h+\lg p))$	$O(n)$
[38]	$O(\frac{n}{p}\frac{h+\lg n}{\lg(n/p)})^*$	$O(\frac{n}{p}\frac{h+\lg n}{\lg(n/p)})^*$	$O(\frac{n}{p}\frac{h+\lg n}{\lg(n/p)})^*$	$O(\frac{n}{\lg^2 n \lg\lg n})$
This work	$O(\frac{n}{p})$	$O(\frac{n}{p})$	$O(\frac{n}{p})$	$O(\frac{n}{\lg n})$

Table 4: A * sign indicates that the associated result has been adapted to fit our model.

We also note that results on PD algorithms are obtained in [4] on the BSP* model. A new data structure is presented, the BB*-tree, that is obtained as a mixture of B-trees and weight balanced trees. Our results compare favorably to those in [4] in various ways. The algorithm presented in [4] requires $O(\frac{n}{p}(h+\lg n))$ computation time. The communication time of the algorithm in [4] is also considerably higher than the one in this work. As a consequence, the dictionary algorithms presented in [4] are slower in both computation and communication. In addition, our algorithms work for values of p much closer to n than those of [4] even when differences in the two models are taken into consideration.

1.3 Primitive Operations

In this section we state results on BSP algorithms for the primitive operations of *broadcast*, *parallel-prefix*, *integer sorting* and *selection*. Proofs for these results can be found in [14, 18]. These primitive operations are auxiliary routines for the algorithms presented in the following sections.

Lemma 1.1 ([18]) *There exists a p-processor BSP algorithm for broadcasting an n-word vector, that for any integer $2 \leq t \leq p$, requires time at most*[3]

$$M^n_{brd}(p) = 2\max\{L, 2gn\} \cdot [n > 1] + h\max\{L, 2gt\} \cdot [n < p],$$

[2] The underlying assumption in our PD algorithms is that the items belong to a finite universe $U = \{0, 1, \ldots, Q-1\}$, and therefore, we are able to employ hashing in order to improve the performance of our algorithms.

[3] In general, if S is any statement that can be true or false, the bracketed notation [S] evaluates to 1 if S is true, and 0 otherwise.

for communication, where $h = \lceil \log_t ((t-1)\lceil p/n \rceil + 1) \rceil - 1$. *The overall time complexity of the algorithm is denoted by* $T^n_{brd}(p) = M^n_{brd}(p)$.

Lemma 1.2 ([18]) *There exists a* p*-processor BSP algorithm for realizing* n *disjoint parallel-prefix operations, each of size* p*, over an associative operator* $\otimes$*, that for any integer* $2 \le t \le p$*, requires time at most*

$$C^n_{ppf}(p,\otimes) = 2\max\{L, 2c_{\otimes} n\} \cdot [n > 1] + 2h \max\{L, 2c_{\otimes}(t-1)\} \cdot [n < p]$$

and

$$M^n_{ppf}(p,\otimes) = 2\max\{L, 2gn\} \cdot [n > 1] + 2h \max\{L, 2gt\} \cdot [n < p],$$

for computation and communication respectively, where $h = \lceil \log_t \lceil p/n \rceil \rceil$ *and* $c_{\otimes}$ *is the time required to evaluate* $\otimes$. *The overall time complexity of the algorithm is denoted by* $T^n_{ppf}(p,\otimes) = C^n_{ppf}(p,\otimes) + M^n_{ppf}(p,\otimes)$.

For $c_{\otimes} = O(1)$, we write $T^n_{ppf}(p)$ (respectively $C^n_{ppf}(p)$ and $M^n_{ppf}(p)$) for $T^n_{ppf}(p,\otimes)$ (respectively $C^n_{ppf}(p,\otimes)$ and $M^n_{ppf}(p,\otimes)$). We also write $T_{ppf}(p,\otimes)$ (respectively $C_{ppf}(p,\otimes)$ and $M_{ppf}(p,\otimes)$) for $T^1_{ppf}(p,\otimes)$ (respectively $C^1_{ppf}(p,\otimes)$ and $M^1_{ppf}(p,\otimes)$). The same notational convention applies to all pipelined operations.

Lemma 1.3 ([18]) *There exists a* p*-processor BSP algorithm for stable sorting a sequence consisting of* n *integers in the range* $[0,\ldots,R-1]$*, that for all integers* n*,* p *and* t*, such that* $n \ge p$ *and* $2 \le t \le p$*, requires time at most*

$$C_{rdx\_srt}(n,R,p) = r\left(3\max\left\{L, O\left(\max\left\{\frac{n}{p}, t\log_t p\right\}\right)\right\} + C_{ppf}(p)\right),$$

and

$$M_{rdx\_srt}(n,R,p) = r\left(2\max\left\{L, g\max\left\{\left\lceil\frac{n}{p}\right\rceil, t\lceil\log_t p\rceil\right\}\right\} + \max\left\{L, g\left\lceil\frac{n}{p}\right\rceil\right\} + M_{ppf}(p)\right),$$

for computation and communication respectively, where $r = \lceil \lg R / \lg \max\{\lceil n/p \rceil, t\lceil \log_t p \rceil\}\rceil$. *The overall time complexity of the algorithm is denoted by* $T_{rdx\_srt}(n,R,p) = C_{rdx\_srt}(n,R,p) + M_{rdx\_srt}(n,R,p)$.

Lemma 1.4 ([14, 15]) *For any constant* $\rho > 0$*, there exists a randomized* p*-processor BSP selection algorithm that, with probability* $1 - O(n^{-\rho})$*, determines the* C*-th smallest,* $C \le \lceil n/2 \rceil$*, of* n *keys that for any integers* n *and* $p \le n$*, requires time*

$$C_{sel}(n,C,p) = \begin{cases} \frac{n+C}{p} + O\left(L + \frac{\sqrt{Cp\lg n} + n^{\frac{2}{3}+\delta}\lg^{\frac{4}{3}} n}{p}\right) + O(C_{ppf}(p)) & \text{if } p \le n^{\frac{2}{3}+\zeta}, \\ \frac{n+C}{p} + O\left(L + \frac{\sqrt{Cp\lg n} + p\lg n}{p}\right) + O(C_{ppf}(p)) & \text{otherwise,} \end{cases}$$

and

$$M_{sel}(n,C,p) = \begin{cases} O\left(L + g\frac{n^{\frac{2}{3}+\delta}\lg^{\frac{1}{3}} n}{p}\right) + O(M_{ppf}(p)) & \text{if } p \le n^{\frac{2}{3}+\zeta}, \\ O(L) + gO(1) + O(M_{ppf}(p)) & \text{otherwise,} \end{cases}$$

for computation and communication respectively, where ζ *and* δ *are any arbitrarily small constants such that* $0 < \zeta < \delta < 1/3$. *The overall time complexity of the algorithm is denoted by* $T_{sel}(n,C,p) = C_{sel}(n,C,p) + M_{sel}(n,C,p)$.

Definition 1.1 *Let* $\hat{g}$ *be the value of* g *in lemma 1.4 such that* $M_{sel}(n,C,p) = o(C_{sel}(n,C,p))$. *In particular,*

$$\hat{g} = \begin{cases} o\left(\frac{n^{\frac{1}{3}-\delta}}{\lg^{\frac{1}{3}} n}\right) & \text{if } p \le n^{\frac{2}{3}+\zeta}, \\ o\left(\frac{n}{p\lg p}\right) & \text{otherwise,} \end{cases}$$

for any arbitrarily small constants ζ *and* δ *such that* $0 < \zeta < \delta < 1/3$.

2 BSP Priority Queues

In this section we study the *Parallel Priority Queue* (PPQ) [37] data structure, and formulate an implementation on the BSP model. We present results on PPQs that improve upon the communication requirements of previous results on parallel priority queues [28, 37, 39, 40], and extend the BSP algorithms of [15] by presenting efficient and simple (non-pipelined) algorithms. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters.

2.1 Preliminaries

A PPQ is an abstract data structure for maintaining a collection of (possibly duplicate) items and selecting the n items associated with the smallest values. We assume that the item values belong to a universe U equipped with a total order $<$ that can be evaluated in unit time by one processor for any given pair of arguments. The following operations are defined on a PPQ Q.

(1) $\textsc{InsertQ}^*(\{y_1, y_2, \ldots, y_n\}, Q)$ for the insertion of $\{y_1, y_2, \ldots, y_n\}$ into Q.

(2) $\textsc{FindMinQ}^*(n, Q)$ for the determination of the n smallest items of Q.

(3) $\textsc{DeleteMinQ}^*(n, Q)$ for the deletion of the n smallest items of Q.

The implicit assumption is that there exists a $\textsc{ConstructQ}^*(Q)$ operation for the construction of an empty queue Q. In contrast to [15, 37], our implementation has the advantage of supporting PPQ operations on data sets of varying size, i.e., in [15, 37], insert, delete-min and find-min operations have to be performed on data sets of predetermined size, by contrast, in this work we support data sets of varying size.

The randomized algorithms we present in this work for supporting the prementioned PPQ operations are based on an efficient processor-mapping of sequential ordinary (and leftist) heaps, even though, other sequential data structures might be supported as well, e.g., relaxed heaps, fibonacci heaps, search trees [7, 29, 42]. To this end, in the forthcoming analysis of the PPQ algorithms we do not make any assumptions about the underlying local (sequential) priority queues. In order to simplify the exposition, the following notational conventions are employed.

- $\tau_{IQ}(n, m)$ – denotes the time required to sequentially insert n items into a local priority queue of size m,
- $\tau_{FQ}(n, m)$ – denotes the time required to sequentially determine the n smallest items of a local priority queue of size m,
- $\tau_{DQ}(n, m)$ – denotes the time required to sequentially delete the n smallest items of a local priority queue of size m,
- $\tau_{CQ}(m)$ – denotes the time required to sequentially construct a local priority queue of size m (we note that in general $\tau_{CQ}(m) \leq \sum_{i=0}^{m-1} \tau_{IQ}(1, i)$),
- $\tau_{MQ}(m_1, m_2)$ – denotes the time required to sequentially meld two local priority queues of size m_1 and m_2 into one of size $m_1 + m_2$.

2.2 The PPQ Algorithms

In [27, 28], a parallel randomized technique to approximate a priority queue is presented. This technique is extended in [39] to handle exact priority queues. In particular, in [39] a randomized variant of the approximate PPQs of [28] is presented that delivers optimal performance for the insert operation but sub-optimal performance for the find-min and delete-min operations. The main inefficiency of the delete-min and find-min operations of [39] derives from a global sorting that is inherent in the particular implementation of these operations. Parallel *sorting* (in the setting of PPQs) has two major drawbacks, (i) it is communication intensive, and (ii) it has a super-linear computation and communication time complexity. In this work, along the lines of [3, 15, 40], we propose parallel *selection*, in order to alleviate the limitations imposed by sorting.

The randomized algorithms we present are based on the ideas of [27, 28] as extended in [39]. Let us outline the BSP derivations of these algorithms.

The PPQ is maintained as a collection of disjoint local (sequential) priority queues. For the insertion of n new items (at most $\lceil n/p \rceil$ per processor), each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at random chosen processors, where they are inserted into their local priority queues. The procedure to manage the insert operation on a PPQ Q is InsertQ*($\{y_1, y_2, \ldots, y_n\}$, Q) and is outlined in figure 1.

```
begin InsertQ* ({y_1, y_2, ..., y_n}, Q)
1.    denote by Y set {y_1, y_2, ..., y_n} ;
2.    distribute randomly the items of Y among the p processors ;
3.    for each processor p^<k>, 0 <= k < p, in parallel
4.        do InsertQ_Seq(Y^<k>, Q^<k>) ;
end InsertQ*
```

Figure 1: Procedure InsertQ*.

The correctness of procedure InsertQ* is immediate; it therefore remains to bound the resources required by the aforementioned procedure. The following lemma bounds the number of items received by any processor as a result of the random distribution of items to processors (line 2).

Lemma 2.1 *Let in procedure* InsertQ* $Y = \bigcup_k Y^{\langle k \rangle}$ *and* $Q = \bigcup_k Q^{\langle k \rangle}$, $0 \leq k < p$. *Then, for all* $0 \leq k < p$,

$$|Y^{\langle k \rangle}| = \begin{cases} \min\left\{O\left(\frac{n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \textit{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p\lg n}{n}}\right)\right)\frac{n}{p} & \textit{otherwise,} \end{cases}$$

and

$$|Q^{\langle k \rangle}| = \begin{cases} \min\left\{O\left(\frac{m}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \textit{if } \frac{m}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p\lg n}{m}}\right)\right)\frac{m}{p} & \textit{otherwise,} \end{cases}$$

with probability $1 - n^{-\Omega(1)}$, *where* m *denotes the total number of items in* Q *prior to the insertion.*

Proof In procedure InsertQ*, the random distribution of items to processors implies that the probability of a processor receiving an item is $1/p$. Therefore, the expected number of items received per processor is n/p. Invoking theorem A.2, for $n/p = 1$, it follows that

$$\Pr\left[|Y^{\langle k \rangle}| = \Omega\left(\frac{\lg n}{\lg\lg n}\right)\right] = n^{-\Omega(1)}$$

Similarly, by invoking theorem A.3, for $n/p = \omega(\lg n)$, it follows that

$$\Pr\left[|Y^{\langle k \rangle}| = \left(1 + \Omega\left(\sqrt{\frac{p\lg n}{n}}\right)\right)\frac{n}{p}\right] = n^{-\Omega(1)}.$$

The analysis of the remaining cases (i.e., $n/p = O(\lg n)$) follows a line of arguments similar to that of the aforementioned cases and is therefore omitted. The bound on $|Q^{\langle k \rangle}|$ follows similarly. ■

The resources required for procedure InsertQ* are bounded as follows. The random distribution of items to processors (line 2) takes time $\max\{L, O(|Y^{\langle k \rangle}|)\}$ and $\max\{L, g|Y^{\langle k \rangle}|\}$, for computation and communication respectively, with probability $1 - n^{-\Omega(1)}$. The insertion of the $|Y^{\langle k \rangle}|$ items in the local priority queues (line 4) requires time $\max\{L, \tau_{IQ}(|Y^{\langle k \rangle}|, |Q^{\langle k \rangle}|)\}$. Thus, insertion can be implemented efficiently by employing this simple randomized scheme in time

$$\max\left\{L, O(|Y^{\langle k\rangle}|)\right\} + \max\left\{L, g|Y^{\langle k\rangle}|\right\} + \max\left\{L, \tau_{IQ}(|Y^{\langle k\rangle}|, |Q^{\langle k\rangle}|)\right\}.$$

For the deletion of the n smallest items, the following (rather more complicated) procedure is employed (see also [39]). The $2\lceil n/p\rceil$ smallest items in each processor are duplicated. Let X denote the set of these $2n$ items. By employing selection, the n-th smallest item e of X is determined and broadcast. Let $R^{\langle k\rangle}$ denote the set of items in processor $p^{\langle k\rangle}$, $0 \le k < p$, that are not larger than e. By employing selection, the n-th smallest item $e^{'}$ from $\bigcup_k R^{\langle k\rangle}$ is determined and broadcast to all processors. Finally, the set S of the n smallest items (the items smaller than $e^{'}$) is removed from the local priority queues and returned as the result of the delete-min operation. The procedure to manage the deletion operation on a PPQ Q is DELETEMINQ$^*(n, Q)$ and is outlined in figure 2. A find-min operation is implemented similarly.

begin DELETEMINQ* (n, Q)
1. **if** $|Q| = n$ **then**
2.   **return** $Q^{\langle k\rangle}$;
3. **else**
4.   **for** each processor $p^{\langle k\rangle}$, $0 \le k < p$, in parallel
5.   **do** $X^{\langle k\rangle} = \{$FINDMINQ_SEQ$(2\lceil n/p\rceil, Q^{\langle k\rangle})\}$;
6.   **let** $R^{\langle k\rangle} = \{\}$;
7.   **let** $e =$ SELECT$(\bigcup_k X^{\langle k\rangle}, n)$;
8.   **while** FINDMINQ_SEQ$(1, Q^{\langle k\rangle}) \le e$
9.   **let** $x =$ DELETEMINQ_SEQ$(1, Q^{\langle k\rangle})$;
10.   **let** $R^{\langle k\rangle} = R^{\langle k\rangle} \bigcup \{x\}$;
11.   **let** $S^{\langle k\rangle} = \{\}$;
12.   **let** $e^{'} =$ SELECT$(\bigcup_k R^{\langle k\rangle}, n)$;
13.   **while** FINDMINQ_SEQ$(1, Q^{\langle k\rangle}) \le e^{'}$
14.   **let** $x =$ DELETEMINQ_SEQ$(1, Q^{\langle k\rangle})$;
15.   **let** $S^{\langle k\rangle} = S^{\langle k\rangle} \bigcup \{x\}$;
16.   **return** $S^{\langle k\rangle}$;

end DELETEMINQ*

Figure 2: Procedure DELETEMINQ*.

The correctness of procedure DELETEMINQ* is immediate. It therefore remains to bound the resources required by the aforementioned scheme. The following lemma bounds the total number of items (as well as the number of items per processor) that are smaller than e (lines 6-10).

Lemma 2.2 *Let in procedure* DELETEMINQ* $R = \bigcup_k R^{\langle k\rangle}$, $0 \le k < p$. *Then,* $|R| < 1.2n$ *with probability* $1 - 2^{-\Omega(n)}$. *Moreover, for all* $0 \le k < p$,

$$|R^{\langle k\rangle}| = \begin{cases} \min\left\{O\left(\frac{\lg n}{p \lg\lg n}\right), O(\lg n)\right\} & \text{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p\lg n}{n}}\right)\right)\frac{1.2n}{p} & \text{otherwise,} \end{cases}$$

with probability $1 - n^{-\Omega(1)}$.

Proof We shall prove the bound on R for the case $n = p$; the other cases follow similarly. Let $R_{1.2n}$ denote the smallest $\lceil 1.2n\rceil$ items (overall) at the beginning of procedure DELETEMINQ*. Along the lines of [39], we shall establish that with probability $1 - 2^{-\Omega(n)}$, set $X = \bigcup_k X^{\langle k\rangle}$, $0 \le k < p$, of procedure DELETEMINQ* contains at least n items from $R_{1.2n}$. Therefore, since $R \subseteq R_{1.2n}$ we have that $|R| < 1.2n$, with the corresponding probability.

Let Z be the random variable denoting the number of processors containing no items when $1.2n$ items are placed randomly into n processors. Similarly, let W be the random variable denoting the number of processors containing exactly one item when $1.2n$ items are placed randomly into n processors. Substituting in lemmas A.2 and A.1, $1.2n$ for m, n for k, and cn for λ (for some positive constant c), we obtain that

$$\Pr[|Z-\mu_0| \geq cn] = 2^{-\Omega(n)} \quad \text{and} \quad \Pr[|W-\mu_1| \geq cn] = 2^{-\Omega(n)},$$

where $\mu_0 = n(1-(1/n))^{1.2n}$ and $\mu_1 = 1.2n(1-(1/n))^{1.2n-1}$. By definition, X contains the two smallest items from each processor at the beginning of procedure DELETEMINQ*. Thus, X contains at least

$$W + 2(n-Z-W) \geq 2n - 2(\mu_0 + cn) - (\mu_1 + cn) \geq n,$$

items from $R_{1.2n}$, for some appropriately small constant c. Therefore, we obtain that $R \subseteq R_{1.2n}$, or, equivalently that $|R| < 1.2n$, with probability $1 - 2^{-\Omega(n)}$.

The analysis of the size of the $R^{\langle k \rangle}$'s follows a line of arguments similar to that of lemma 2.1 and is therefore omitted. ■

The resources required for procedure DELETEMINQ* are bounded as follows. The determination of the $2\lceil n/p \rceil$ smallest items per local priority queue (line 5) requires $\max\{L, \tau_{FQ}(2\lceil n/p \rceil, |Q^{\langle k \rangle}|)\}$ time. The median selection process (line 7) takes time $T_{sel}(2\lceil n/p \rceil p, n, p)$. The sequential find-min and delete-min operations per local priority queue (lines 8-10) involve $|R^{\langle k \rangle}|$ items, with probability $1 - n^{-\Omega(1)}$, and therefore require time $\max\{L, \tau_{FQ}(1, |Q^{\langle k \rangle}|) + (\tau_{FQ}(1, |Q^{\langle k \rangle}|) + \tau_{DQ}(1, |Q^{\langle k \rangle}|))|R^{\langle k \rangle}|\}$. Accordingly, the selection process (line 12) takes time at most $T_{sel}(p|R^{\langle k \rangle}|, n, p)$. By employing probabilistic arguments similar to those of lemma 2.1 we can establish that the sequential find-min and delete-min operations per local priority queue (lines 13-15) involve $|Y^{\langle k \rangle}|$ items, with probability $1 - n^{-\Omega(1)}$, and thus take $\max\{L, \tau_{FQ}(1, |Q^{\langle k \rangle}|) + (\tau_{FQ}(1, |Q^{\langle k \rangle}|) + \tau_{DQ}(1, |Q^{\langle k \rangle}|))|Y^{\langle k \rangle}|\}$ time. Finally, the local operations (lines 6, 11, 16) take time $3\max\{L, O(1)\}$. Thus, deletion can be implemented efficiently by employing this simple randomized scheme in time

$$\begin{gathered}\max\left\{L, \tau_{FQ}(2\lceil \tfrac{n}{p} \rceil, |Q^{\langle k \rangle}|)\right\} + \max\{L, \tau_{FQ}(1, |Q^{\langle k \rangle}|) + (\tau_{FQ}(1, |Q^{\langle k \rangle}|) + \tau_{DQ}(1, |Q^{\langle k \rangle}|))|R^{\langle k \rangle}|\} + \\ \max\{L, \tau_{FQ}(1, |Q^{\langle k \rangle}|) + (\tau_{FQ}(1, |Q^{\langle k \rangle}|) + \tau_{DQ}(1, |Q^{\langle k \rangle}|))|Y^{\langle k \rangle}|\} + T_{sel}(2n, n, p) + \\ T_{sel}(p|R^{\langle k \rangle}|, n, p) + 3\max\{L, O(1)\}.\end{gathered}$$

We therefore obtain the following result.

Theorem 2.1 *For any constant $\rho > 0$, there exists a randomized BSP algorithm for supporting the PPQ operations* INSERTQ*, FINDMINQ*, *and* DELETEMINQ*, *that for all integers n, p, L, and g such that $n/p = \Omega(\lg n)$, $L = O((n \lg(m/p))/(p \lg p))$, and $g = O(\lg(m/p) \min\{\hat{g}, 1\})$, with probability $1 - O(n^{-\rho})$, requires time, $O((n/p)\lg(m/p))$ per* INSERTQ*, *$O((n/p)\lg(m/p))$ per* FINDMINQ*, *and $O((n/p)\lg(m/p))$ per* DELETEMINQ* *operation, where m refers to the total number of items in the underlying PPQ and $\hat{g}$ is as in definition 1.1.*

Proof For any constant $\rho > 0$ and $n/p = \Omega(\lg n)$, it follows by way of lemmas 2.1 and 2.2 that $Y^{\langle k \rangle} = O(n/p)$ and $R^{\langle k \rangle} = O(n/p)$, with probability $1 - O(n^{-\rho})$. Similarly, $Q^{\langle k \rangle} = O(m/p)$, with probability $1 - O(n^{-\rho})$. Moreover, for $t = 2$, it follows by way of lemmas 1.2 and 1.4 that $T_{sel}(2n, n, p) + T_{sel}(p|R^{\langle k \rangle}|, n, p) = O(n/p)$. Substituting in the analysis of procedures INSERTQ* and DELETEMINQ*, $O((n/p)\lg(m/p))$ for $\tau_{IQ}(O(n/p), O(m/p))$, $\tau_{FQ}(O(n/p), O(m/p))$ and $\tau_{DQ}(O(n/p), O(m/p))$ we obtain the desired result, i.e., ordinary heaps [7, 42]. ■

We note that we can improve the performance of the INSERTQ* and DELETEMINQ* operations in several ways. For example, in case $n/p = \Omega(\lg n)$, a simplified version of procedure DELETEMINQ* can be employed to realize the delete-min operation. The procedure to manage the simplified deletion operation on a PPQ Q is DELETEMINQ_S*(n, Q) and is outlined in figure 3. A find-min operation is implemented similarly. To simplify the exposition, in the remainder of this section we assume that $n/p = \Omega(\lg n)$.

In theorem 2.1, the underlying local (sequential) data-structure maintained by each processor is assumed to be an ordinary sequential heap, and therefore, the insertion of $O(n/p)$ items per processor requires time $O((n/p)\lg(m/p))$. If instead of using ordinary heaps, we use leftist heaps [42], then an INSERTQ* operation

```
begin DeleteMinQ_S* (n, Q)
    if |Q| = n then
        return Q^⟨k⟩ ;
    else
        for each processor p^⟨k⟩, 0 ≤ k < p, in parallel
            do X^⟨k⟩ = {FindMinQ_Seq(⌈(1 + ε)n/p⌉, Q^⟨k⟩)} ;
                let R^⟨k⟩ = {} ;
                let e = Select(⋃_k X^⟨k⟩, n) ;
                while FindMinQ_Seq(1, Q^⟨k⟩) ≤ e
                    let x = DeleteMinQ_Seq(1, Q^⟨k⟩) ;
                    let R^⟨k⟩ = R^⟨k⟩ ⋃ {x} ;
                return R^⟨k⟩ ;
end DeleteMinQ_S*
```

Figure 3: Procedure DeleteMinQ_S*.

can be performed in $\tau_{IQ}(O(n/p), O(m/p)) = O(n/p + \lg(m/p))$ time as follows. Each processor constructs an auxiliary leftist heap on the $O(n/p)$ items it receives, as a result of the random distribution of items to processors (line 2), in time $\tau_{CQ}(O(n/p)) = O(n/p)$ [42], and subsequently melds this auxiliary leftist heap with its local heap that is part of Q. The latter operation requires time $\tau_{MQ}(O(n/p), O(m/p)) = O(\lg(n/p) + \lg(m/p))$ [42]. Overall, an improved $\tau_{IQ}(O(n/p), O(m/p)) = O(n/p + \lg(n/p) + \lg(m/p))$ computation time bound can be claimed for insertion.

The usage of leftist heaps can also be exploited to obtain $\tau_{FQ}(O(n/p), O(m/p)) = O(n/p)$ computation time for the find-min operation. The general idea is as follows (recall that the find-min operation is realized along the lines of the delete-min operation). Each processor maintains a copy of the $2\lceil n/p \rceil$ smallest items of its local leftist heap in a separate list. We observe that the $2\lceil n/p \rceil$ smallest items can be maintained in a separate local list in time $O(n/p)$ under the operations insert and delete-min by employing a linear time sequential selection algorithm [14, 15, 29]. As it is noted in [42], the heap structure of leftist heaps allows the determination of all items not exceeding a prespecified value $\bar{e}$ in time proportional to the number of such items, i.e., perform a preorder traversal of the underlying leftist heap starting from the root, listing every encountered item with value not exceeding $\bar{e}$ and immediately retreating from each item with value exceeding $\bar{e}$. Therefore, the determination of items with values not exceeding e and $e^{'}$ (lines 8-10, 13-15 of procedure DeleteMinQ*) requires time $O(n/p)$. The $\tau_{FQ}(O(n/p), O(m/p)) = O(n/p)$ time bound on the find-min operation follows. The following analogue of theorem 2.1 can then be derived (we assume an initial random distribution of items to processors, i.e., line 2 of algorithm InsertQ* is then redundant).

Theorem 2.2 *For any constant $\rho > 0$, there exists a randomized BSP algorithm for supporting the PPQ operations* InsertQ*, FindMinQ*, *and* DeleteMinQ*, *that for all integers n, p, L, and g such that $n/p = \Omega(\lg n)$, $L = O(n/(p \lg p))$, and $g = O(\hat{g})$, with probability $1 - O(n^{-\rho})$, requires time, $O(n/p + \lg(n/p) + \lg(m/p))$ per* InsertQ*, *$O(n/p)$ per* FindMinQ*, *and $O((n/p)\lg(m/p))$ per* DeleteMinQ* *operation, where m refers to the total number of items in the underlying PPQ and $\hat{g}$ is as in definition 1.1.*

We additionally note that on leftist heaps the delete-min operation, and consequently the contributing term $O((n/p)\lg(m/p))$, can be further optimized by using lazy evaluation techniques [42] for an increase in the time of insert operations. Alternatively, we can employ tree-based data structures [7, 29, 42] to improve the performance of the delete-min and find-min operations without a corresponding increase in the time required for the insert operation. For example, in [7, 42] it is shown for balanced binary trees (2-3-4 trees) that $\tau_{IQ}(1, m) = O(\lg m)$, $\tau_{DQ}(1, m) = O(\lg m)$, $\tau_{DQ}(n, m) = O(n + \lg m)$ and $\tau_{MQ}(m_1, m_2) = O(|\lg m_1 - \lg m_2|)$. The following analogue of theorem 2.1 can then be derived (we assume an initial random distribution of items to processors, i.e., line 2 of algorithm InsertQ* is then redundant).

Theorem 2.3 *For any constant $\rho > 0$, there exists a randomized BSP algorithm for supporting the PPQ operations* INSERTQ*, FINDMINQ*, *and* DELETEMINQ*, *that for all integers n, p, L and g such that $n/p = \Omega(\lg n)$, $L = O(n/(p \lg p))$, and $g = O(\hat{g})$, with probability $1 - O(n^{-\rho})$, requires time, $O((n/p)\lg(n/p) + |\lg(m/p) - \lg(n/p)|)$ per* INSERTQ*, *$O(n/p + \lg(m/p))$ per* FINDMINQ*, *and $O(n/p + \lg(m/p))$ per* DELETEMINQ* *operation, where m refers to the total number of items in the underlying PPQ and $\hat{g}$ is as in definition 1.1.*

3 Hashing

Given a set S of n elements from a finite universe $U = \{0, 1, \ldots, q-1\}$, the *hashing* problem is to determine a function $h : S \to \{0, 1, \ldots, s-1\}$, for some $s \leq q$ such that the following properties hold.

- h is "easy" to generate,
- h needs a "small" amount of additional space to store,
- h is "easy" to evaluate,
- h behaves "almost" as a random function.

In the remainder of this section we quantify the aforementioned properties.

3.1 A Universal Class of Hash Functions

For the scope of this work we describe the universal class [5] of hash functions introduced in [8] and further addressed in [9, 10].

Definition 3.1 *Let q be prime and $U = \{0, 1, \ldots, q-1\}$ be a finite universe. Let*

$$H_s^d = \{h | h : U \to \{0, 1, \ldots, s-1\}\},$$

where $d \geq 1$ and $s \geq 1$ are integers, $b_0, b_1, \ldots, b_d \in U^{d+1}$, and let h be defined by

$$h(x) = \left(\sum_{i=0}^{d} b_i x^i \bmod q \right) \bmod s, \quad \text{for } x \in U.$$

Definition 3.2 *Let q be prime and $U = \{0, 1, \ldots, q-1\}$ be a finite universe. Let*

$$H_{r,s}^{d_1,d_2} = \left\{ h_{f,\hat{f}} | h_{f,\hat{f}} : U \to \{0, 1, \ldots, s-1\} \quad \text{where} \quad f \in H_r^{d_1} \text{ and } \hat{f} \in H_s^{d_2} \right\},$$

where $d_1, d_2 \geq 1$ and $r, s \geq 1$ are integers, $a_0, a_1, \ldots, a_{r-1} \in \{0, 1, \ldots, s-1\}$, and $h_{f,\hat{f}}$ is defined by

$$h_{f,\hat{f}}(x) = \left(\hat{f}(x) + a_{f(x)} \right) \bmod s, \quad \text{for } x \in U.$$

In the remainder of this section we present some of the properties of the aforementioned classes of hash functions; the following basic notational conventions and facts [8, 10] apply. A hash function is chosen from a class of hash functions uniformly at random.

Definition 3.3 *Let $S \subseteq U$, $|S| \leq n$, and consider function $h : U \to \{0, 1, \ldots, s-1\}$ randomly chosen from some universal class of hash functions.*

- *The j-th bucket induced by h on S is $B_j^h = \{x \in S \mid h(x) = j\}$, $0 \leq j < s$.*
- *The size of the j-th bucket B_j^h is $b_j^h = |B_j^h|$, $0 \leq j < s$.*
- *h is called c-perfect for S if $b_j^h \leq c$, for all $0 \leq j < s$.*

Lemma 3.1 ([10]) *Let $S \subseteq U$, $|S| \leq n$, and consider functions $h, h_{f,\hat{f}}$ randomly chosen from H_s^d and $H_{r,s}^{d_1,d}$ respectively, where $s \leq q$. Then, for any $u \geq 1$,*

$$Pr\left[b_j^h \leq u\right], \quad Pr\left[b_j^{h_{f,\hat{f}}} \leq u\right] \geq \begin{cases} 1-\gamma\left(\frac{e^u}{(u+1)^{u+1}}\right)\left(\frac{n}{s}\right)^{u+1} & \text{if } 1 \leq u < d \\ 1-\gamma\left(\frac{e^d}{(u+1)^{d+1}}\right)\left(\frac{n}{s}\right)^{d+1} & \text{otherwise} \end{cases}$$

for any $0 \leq j < s$, where $\gamma \leq (1+s/q)^{d+1}$.

Lemma 3.2 ([10]) *Let $S \subseteq U$, $|S| \leq n$, and consider functions $h, h_{f,\hat{f}}$ randomly chosen from H_s^d and $H_{r,s}^{d_1,d}$ respectively, where $n \leq s \leq q$. Then,*

$$Pr[h \text{ is } d\text{-perfect for } S], \quad Pr\left[h_{f,\hat{f}} \text{ is } d\text{-perfect for } S\right] \geq 1 - \frac{\gamma}{(d+1)!} n \left(\frac{n}{s}\right)^d,$$

where $\gamma \leq (1+s/q)^{d+1}$.

Lemma 3.3 ([30, 8]) *Let $S \subseteq U$, $|S| \leq n$, and consider functions $h, h_{f,\hat{f}}$ randomly chosen from H_s^d and $H_{r,s}^{d_1,d}$ respectively. Let $0 < \delta < 1$ be fixed such that $s = n^{1-\delta}$ and $n \geq ds/2$. Then, for any $u > 0$ and $0 \leq j < s$,*

$$Pr\left[b_j^h \leq (1+u)n^\delta\right], \quad Pr\left[b_j^{h_{f,\hat{f}}} \leq (1+u)n^\delta\right] \geq 1 - \alpha\left(\frac{1}{u}\right)^d n^{-\delta d/2},$$

where $\alpha = (d/2)^{d+1}/(d/2)!$.

Proof In [30] it is shown that if $X_1, X_2, \ldots, X_n$ are 0-1 valued, d-independent, equidistributed random variables with $\mu = E(\sum_{i=1}^n X_i)$, then, for $2\mu \geq d$,

$$\Pr\left[\sum_{i=1}^n X_i \geq (1+u)\mu\right] \leq \alpha \frac{\mu^{d/2}}{(u\mu)^d},$$

where $\alpha = (d/2)^{d+1}/(d/2)!$. By way of arguments similar to those employed in the proof of fact 2.4 of [8] and substituting for $\mu = n/s = n^\delta$ we obtain the desired result. ∎

The following theorem is a generalization of a result due to [8].

Theorem 3.1 ([8]) *Let $S \subseteq U$, $|S| \leq n$, and consider function $h_{f,\hat{f}}$ randomly chosen from $H_{r,s}^{d_1,d_2}$. Let $0 < \delta < 1$ be fixed such that $r = n^{1-\delta}$ and $n \geq d_1 r/2$. Then, with probability $1 - O(\alpha n^{1-\delta(d_1/2+1)} + (\gamma/(d_2+1)!) n^{\delta d_2+1}(2/s)^{d_2})$, for any $u > 0$ and $0 < \varepsilon < 1$,*

$$Pr\left[b_j^{h_{f,\hat{f}}} \geq (1+u)\frac{n}{s}\right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{n/(d_2 s)} \tag{1}$$

and

$$Pr\left[b_j^{h_{f,\hat{f}}} \geq (1+\varepsilon)\frac{n}{s}\right] \leq e^{-(1/3)\varepsilon^2 n/(d_2 s)} \tag{2}$$

for any $0 \leq j < s$, where $\alpha = (d_1/2)^{d_1+1}/(d_1/2)!$ and $\gamma \leq (1+s/q)^{d_2+1}$.

Proof Let

$$\mathcal{R} = \left\{h_{f,\hat{f}} \in H_{r,s}^{d_1,d_2} \mid b_i^f \leq 2\frac{n}{r} \text{ and } \hat{f} \text{ is } d_2\text{-perfect for } B_i^f, \text{ for all } 0 \leq i < r\right\}.$$

Along the lines of [8], we show that the aforementioned class of hash functions has the desired properties. Assume $|S| = n$ (otherwise add some dummy elements). Consider fixed $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ such that $b_i^f \leq 2n/r$ and $\hat{f}$ is d_2-perfect for B_i^f, for all $0 \leq i < r$. It suffices to show that the assertion holds for the probability space generated by the random choice of $a_0, a_1, \ldots, a_{r-1}$. With respect to this probability space, denote the probability and expectation by $\Pr_{f,\hat{f}}[\ldots]$ and $\mathrm{E}_{f,\hat{f}}[\ldots]$ respectively. Fix j, $0 \le j < s$, and define the random variable Y_i as follows:

$$Y_i = \left|\left\{x \in B_i^f \mid \hat{f}(x) + a_i \equiv j \pmod{s}\right\}\right|, \quad 0 \le i < r.$$

Then,

$$\mathrm{E}_{f,\hat{f}}[Y_i] = \frac{1}{s}\sum_{a=0}^{s-1}\left|\left\{x \in B_i^f \mid \hat{f}(x) + a \equiv j \pmod{s}\right\}\right| = \frac{1}{s}b_i^f, \quad 0 \le i \le r.$$

The independence of the a_is implies the independence of the Y_is, and therefore, we obtain the following for the mean expected value μ of $\sum_i Y_i = b_j^{h_{f,\hat{f}}}$.

$$\mu = \mathrm{E}_{f,\hat{f}}\left[\sum_{i=0}^{r-1} Y_i\right] = \sum_{i=0}^{r-1}\mathrm{E}_{f,\hat{f}}[Y_i] = \sum_{i=0}^{r-1}\left(\frac{1}{s}b_i^f\right) = \frac{|S|}{s} = \frac{n}{s}.$$

The desired result is obtained by observing that $0 \le Y_i \le d_2$ and applying theorem A.1 (the former observation follows from the fact that $\hat{f}$ is d_2-perfect for B_i^f, for all $0 \le i < r$).

$$\Pr_{f,\hat{f}}\left[b_j^{h_{f,\hat{f}}} \ge (1+u)\frac{n}{s}\right] \le \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{n/(d_2 s)}, \quad 0 \le j < s.$$

By the right hand side of the following inequality [22],

$$-(1/2)\varepsilon^2 \le \varepsilon - (1+\varepsilon)\log(1+\varepsilon) \le -(1/3)\varepsilon^2, \quad 0 < \varepsilon < 1,$$

we further conclude that

$$\Pr_{f,\hat{f}}\left[b_j^{h_{f,\hat{f}}} \ge (1+\varepsilon)\frac{n}{s}\right] \le e^{-(1/3)\varepsilon^2 n/(d_2 s)}, \quad 0 \le j < s.$$

It remains to establish that for suitably chosen d_1, d_2, r and s, a randomly chosen $h_{f,\hat{f}} \in H_{r,s}^{d_1,d_2}$ satisfies $h_{f,\hat{f}} \in \mathcal{R}$, i.e., $b_i^f \le 2\frac{n}{r}$ and $\hat{f}$ is d_2-perfect for B_i^f, for all $0 \le i < r$. Fix $a_0, a_1, \ldots, a_{r-1}$ and assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \ge d_1 r/2$. Then, by way of lemma 3.3, it follows that a randomly chosen $f \in H_r^{d_1}$ satisfies $b_i^f \le 2n^\delta$, for all $0 \le i < r$, with probability

$$1 - O\left(n^{1-\delta}\alpha n^{-\delta d_1/2}\right) = 1 - O\left(\alpha n^{1-\delta(d_1/2+1)}\right),$$

where $\alpha = (d_1/2)^{d_1+1}/(d_1/2)!$. Given such an f, a randomly chosen $\hat{f} \in H_s^{d_2}$ is d_2-perfect for B_i^f, for all $0 \le i < r$, with probability

$$1 - O\left(n^{1-\delta}\left(\frac{\gamma}{(d_2+1)!}2n^\delta\left(\frac{2n^\delta}{s}\right)^{d_2}\right)\right) = 1 - O\left(\frac{\gamma}{(d_2+1)!}n^{\delta d_2+1}\left(\frac{2}{s}\right)^{d_2}\right),$$

by way of lemma 3.2 and subadditivity, where $\gamma \le (1+s/q)^{d_2+1}$. The desired result follows. ■

3.2 Construction and Evaluation of $h_{f,\hat{f}} \in H_{r,s}^{d_1,d_2}$

In the remainder of this section we concentrate on BSP algorithms for the construction and evaluation of random members of the aforementioned class of hash functions. The following two lemmas summarize our main results.

Lemma 3.4 *Let $h_{f,\hat{f}}$ be randomly chosen from $H_{r,s}^{d_1,d_2}$. Then, there exists a p-processor BSP algorithm that constructs $h_{f,\hat{f}}$ in $O(d_1 + d_2 + r/p)$ space per processor and time*

$$C_{c\_hsh}(n,p) = \max\left\{L, O\left(d_1 + d_2 + \frac{r}{p}\right)\right\} + C_{brd}^{d_1+d_2}(p) + C_{brd}^{\lceil r/p\rceil}\left(\left\lceil\frac{p}{r}\right\rceil\right)$$

and

$$M_{c\_hsh}(n,p) = M_{brd}^{d_1+d_2}(p) + M_{brd}^{\lceil r/p \rceil}\left(\left\lfloor \frac{p}{r} \right\rfloor\right),$$

for computation and communication respectively (we refer to this construction as construction CPAR*). The overall time complexity of the algorithm (for construction* CPAR*) is denoted by* $T_{c\_hsh}(n,p) = C_{c\_hsh}(n,p) + M_{c\_hsh}(n,p)$. *Alternatively, the algorithm constructs* $h_{f,\hat{f}}$ *in* $O(d_1 + d_2 + r)$ *space per processor and time*

$$C_{c\_hsh^*}(n,p) = \max\{L, O(d_1 + d_2 + r)\} + C_{brd}^{d_1+d_2}(p) + C_{brd}^{r}(p)$$

and

$$M_{c\_hsh^*}(n,p) = M_{brd}^{d_1+d_2}(p) + M_{brd}^{r}(p),$$

for computation and communication respectively (we refer to this construction as construction CSEQ*). The overall time complexity of the algorithm (for construction* CSEQ*) is denoted by* $T_{c\_hsh^*}(n,p) = C_{c\_hsh^*}(n,p) + M_{c\_hsh^*}(n,p)$.

Proof Let processor p_0 select at random hash functions $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ in time $\max\{L, O(d_1+d_2)\}$, and subsequently, broadcast f and $\hat{f}$ to all other processors in $T_{brd}^{d_1+d_2}(p)$ time. The construction of $h_{f,\hat{f}}$ requires the additional selection of random $a_0, a_1, \ldots, a_{r-1} \in \{0, 1, \ldots, s-1\}$. Processors $p_0, p_1, \ldots, p_{\min\{p-1,r\}}$ accomplish this selection in $\max\{L, O(\lceil r/p \rceil)\}$ time. Then, the processors replicate the r parameters $\lfloor p/r \rfloor$ times – in some predetermined structured way – in $T_{brd}^{\lceil r/p \rceil}(\lfloor p/r \rfloor)$ time. The overall representation (construction CPAR) requires $O(d_1 + d_2 + r/p)$ space per processor.

Alternatively, let processor p_0 select the r parameters $a_0, a_1, \ldots, a_{r-1}$ in $\max\{L, O(r)\}$ time, and subsequently, broadcast them to all other processors in $T_{brd}^{r}(p)$ time. The overall alternative representation (construction CSEQ) requires $O(d_1 + d_2 + r)$ space per processor and the lemma follows. ∎

Lemma 3.5 *Let* $S \subseteq U$, $|S| \le n$, *and consider function* $h_{f,\hat{f}}$ *randomly chosen from* $H_{r,s}^{d_1,d_2}$. *Let* $0 < \delta < 1$ *be fixed such that* $r = n^{1-\delta}$ *and* $n \ge d_1 r/2$. *Then, for any constant* $\rho > 0$, *there exists a randomized p-processor BSP algorithm that, with probability* $1 - O(\alpha(1/u)^{d_1} n^{1-\delta(d_1/2+1)} + n^{-\rho})$, *evaluates (for construction* CPAR *in lemma 3.4)* $h_{f,\hat{f}}(x)$, *for all* $x \in S$, *and any* $u > 0$, *in time*

$$C_{e\_hsh}(n,p) = \max\left\{L, \tau\left(h_{f,\hat{f}}\right)\left\lceil \frac{n}{p} \right\rceil\right\}$$

and

$$M_{e\_hsh}(n,p) = \begin{cases} \max\left\{L, g(1+u)n^{\delta}\left\lceil \frac{n^{1-\delta}}{p} \right\rceil\right\} & \text{if } p \le r, \\ \max\left\{L, g\min\left\{O\left(\frac{(1+u)n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\}\right\} & \text{if } p > r \text{ and } (1+u)\frac{n}{p} = O(\lg n), \\ \max\left\{L, g\left(1 + O\left(\sqrt{\frac{p\lg n}{(1+u)n}}\right)\right)\frac{(1+u)n^{\delta}}{\lfloor p/n^{1-\delta} \rfloor}\right\} & \text{otherwise,} \end{cases}$$

for computation and communication respectively, where $\alpha = (d_1/2)^{d_1+1}/(d_1/2)!$ *and* $\tau(h_{f,\hat{f}})$ *is the time required to evaluate* $h_{f,\hat{f}}$. *The overall time complexity of the algorithm (for construction* CPAR*) is denoted by* $T_{e\_hsh}(n,p) = C_{e\_hsh}(n,p) + M_{e\_hsh}(n,p)$. *Alternatively, the algorithm evaluates (for construction* CSEQ *in lemma 3.4)* $h_{f,\hat{f}}$, $x \in S$, *in time*

$$T_{e\_hsh^*}(n,p) = C_{e\_hsh^*}(n,p) = \max\left\{L, \tau\left(h_{f,\hat{f}}\right)\left\lceil \frac{n}{p} \right\rceil\right\},$$

for computation.

Proof The computation of $h_{f,\hat{f}}(x) = (\hat{f}(x) + a_{f(x)}) \bmod s$, $x \in S$, involves the parameters associated with f, $\hat{f}$ and $a_{f(x)}$. Clearly, for construction CSEQ in lemma 3.4, these parameters are local, and therefore, the evaluation of $h_{f,\hat{f}}(x)$, for all $x \in S$, requires $\max\{L, \tau(h_{f,\hat{f}})\lceil n/p \rceil\}$ time (the underlying assumption is that the input S is assumed to be evenly distributed among the p processors before the beginning of the execution of the evaluation process). For construction CPAR in lemma 3.4, only the parameters associated with f and $\hat{f}$ are local. In the remainder of this proof we propose a randomized scheme for the determination of $a_{f(x)}$ that results in low contention. Recall that by construction CPAR in lemma 3.4 there are $\lceil p/r \rceil$ copies of $a_{f(x)}$. Additionally, for any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq d_1 r/2$, by way of lemma 3.3, it follows that a randomly chosen $f \in H_r^{d_1}$ satisfies $b_i^f \leq (1+u)n^\delta$, for all $0 \leq i < r$ and any $u > 0$, with probability

$$1 - O\left(n^{1-\delta}\alpha\left(\frac{1}{u}\right)^{d_1} n^{-\delta d_1/2}\right) = 1 - O\left(\alpha\left(\frac{1}{u}\right)^{d_1} n^{1-\delta(d_1/2+1)}\right),$$

where $\alpha = (d_1/2)^{d_1+1}/(d_1/2)!$. We distinguish two cases. If $p \leq n^{1-\delta}$, then it is straightforward to verify that the maximum number of accesses X to any single processor is

$$X \leq (1+u)n^\delta \left\lceil \frac{n^{1-\delta}}{p} \right\rceil.$$

If $p > n^{1-\delta}$, then we assume that processors select uniformly at random one of the $\lfloor p/n^{1-\delta} \rfloor$ copies of $a_{f(x)}$. Then, invoking theorem A.2, it follows that the maximum number of accesses X to any single processor is upper bounded as follows:

$$X = \begin{cases} \min\left\{O\left(\frac{(1+u)n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \text{if } (1+u)\frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \lg n}{(1+u)n}}\right)\right)\frac{(1+u)n^\delta}{\lfloor p/n^{1-\delta} \rfloor} & \text{otherwise,} \end{cases}$$

with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$. The lemma follows. ■

Remark We note that the time $\tau(h_{f,\hat{f}})$ required to evaluate $h_{f,\hat{f}}$ is bounded by $O(d_1 + d_2)$.

3.3 Quasi-Random Functions

In the following sections we shall employ the aforementioned classes of hash functions to obtain simple and efficient BSP algorithms for dictionary operations. In order to simplify the exposition of these subsequent algorithms we assume the following corollaries that are immediate applications of theorem 3.1 and lemmas 3.4 and 3.5 respectively.

Definition 3.4 *Let $S \subseteq U$, $|S| \leq n$, and consider function $h_{f,\hat{f}}$ randomly chosen from $H_{\sqrt{n},s}^{d_1,d_2}$, such that $\sqrt{n} \geq d_1/2$. Then, we say that $h_{f,\hat{f}}$ is* quasi-random *for S.*

Corollary 3.1 *Let $S \subseteq U$, $|S| \leq n$, and consider a quasi-random function $h_{f,\hat{f}}$. Then, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, such that $d_1 \geq 4\rho + 2$ and $d_2 \geq (2\rho + 2)/(2\lg s/\lg n - 1)$, and any $u > 0$ and $0 < \varepsilon < 1$,*

$$Pr\left[b_j^{h_{f,\hat{f}}} \geq (1+u)\frac{n}{s}\right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{n/(d_2 s)}, \quad 0 \leq j < s \tag{3}$$

and

$$Pr\left[b_j^{h_{f,\hat{f}}} \geq (1+\varepsilon)\frac{n}{s}\right] \leq e^{-(1/3)\varepsilon^2 n/(d_2 s)}, \quad 0 \leq j < s. \tag{4}$$

Corollary 3.2 *Let $S \subseteq U$, $|S| \leq n$, and consider a quasi-random function $h_{f,\hat{f}}$ drawn from a universal class $H_{\sqrt{n},s}^{d_1,d_2}$ with $d_1 = \Theta(1)$ and $d_2 = \Theta(1)$. Then, there exists a p-processor BSP algorithm that constructs $h_{f,\hat{f}}$ in $O(1 + \sqrt{n}/p)$ space per processor and time*

$$C_{c\_qrnd}(n,p) = \max\left\{L, O\left(1+\frac{\sqrt{n}}{p}\right)\right\} + C_{brd}^{d_1+d_2}(p) + C_{brd}^{\lceil\sqrt{n}/p\rceil}\left(\left\lceil\frac{p}{\sqrt{n}}\right\rceil\right)$$

and

$$M_{c\_qrnd}(n,p) = M_{brd}^{d_1+d_2}(p) + M_{brd}^{\lceil\sqrt{n}/p\rceil}\left(\left\lceil\frac{p}{\sqrt{n}}\right\rceil\right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{c\_qrnd}(n,p) = C_{c\_qrnd}(n,p) + M_{c\_qrnd}(n,p)$. Additionally, there exists a randomized p-processor BSP algorithm that, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, such that $d_1 \geq (4\rho+2)/(1+(4\lg u)/\lg n)$, evaluates $h_{f,\hat{f}}(x)$, for all $x \in S$, in time

$$C_{e\_qrnd}(n,p) = \max\left\{L, \tau\left(h_{f,\hat{f}}\right)\left\lceil\frac{n}{p}\right\rceil\right\}$$

and

$$M_{e\_qrnd}(n,p) = \begin{cases} \max\left\{L, g(1+u)\sqrt{n}\left\lceil\frac{\sqrt{n}}{p}\right\rceil\right\} & \text{if } p \leq \sqrt{n}, \\ \max\left\{L, g\min\left\{O\left(\frac{(1+u)n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\}\right\} & \text{if } p > \sqrt{n} \text{ and } (1+u)\frac{n}{p} = O(\lg n), \\ \max\left\{L, g\left(1+O\left(\sqrt{\frac{p\lg n}{(1+u)n}}\right)\right)\frac{(1+u)\sqrt{n}}{\lfloor p/\sqrt{n}\rfloor}\right\} & \text{otherwise}, \end{cases}$$

for computation and communication respectively, where $\tau(h_{f,\hat{f}})$ is the time required to evaluate $h_{f,\hat{f}}$. The overall time complexity of the algorithm is denoted by $T_{e\_qrnd}(n,p) = C_{e\_qrnd}(n,p) + M_{e\_qrnd}(n,p)$.

4 BSP Dictionaries

In this section we study the *Parallel Dictionary* (PD) [36] data structure, and formulate an implementation on the BSP model. In this work, we improve upon the communication requirements of previous results [12, 16, 36, 38], and extend the BSP algorithms of [12, 16] by presenting efficient and simple algorithms. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters.

4.1 Preliminaries

A PD is an abstract data structure for maintaining a collection of (unique) items and supporting efficient insert, delete, and find operations. We assume that item values belong to a finite universe $U = \{0, 1, \ldots, q-1\}$, where q is prime, equipped with a total order $<$ that can be evaluated in unit time by one processor for any given pair of arguments. The following operations are defined on a PD D.

(1) InsertD*($\{y_1, y_2, \ldots, y_n\}$, D) for the insertion of $\{y_1, y_2, \ldots, y_n\}$ into D.

(2) FindD*($\{y_1, y_2, \ldots, y_n\}$, D) for the determination of $\{e_1, e_2, \ldots, e_n\}$, where $e_i = y_i$ if $y_i \in D$, and $e_i = \varepsilon$ otherwise, $1 \leq i < n$.

(3) DeleteD*($\{y_1, y_2, \ldots, y_n\}$, D) for the deletion of $\{y_1, y_2, \ldots, y_n\}$ from D.

The implicit assumption is that there exists a ConstructD*(D) operation for the construction of an empty dictionary D. In contrast to [4], our implementation has the advantage of supporting PD operations on data sets of varying size, i.e., in [4], insert, delete and find operations have to be performed on data sets of predetermined size, by contrast, in this work we support data sets of varying size.

We would like to make the following remarks about our parallel dictionary model. We assume that a group of identical operations is acted on a dictionary at the same time (i.e. no mixing of various insert, delete, find operations is allowed) [38]. All item values of a single operation are distinct (implied by the set notation adapted above). The duplicate handling process outlined in a later section along with integer-sorting could eliminate this problem, however. No dynamic arrival of requests is allowed in the sense that all n requests become available simultaneously. The results stated below are for a single round of one of the operations. As we might have multiple waves of operations, the size of the dictionary grows, and the currently chosen hash function may fail to exhibit the necessary behavior, we need to reorganize the dictionary. This would involve strategies similar to the ones employed in [10]. Such details are left out of this report. Worst case randomized time bounds may become expected time ones in that case, if dictionary reorganization is taken into consideration. Note also, that our proposed sequential solutions are not the best available (i.e. non-linear time), as other more elaborate schemes could have been used [8]. Nevertheless, they show how one can utilize effectively sequential algorithms in a parallel setting (build parallel algorithms based on sequential components).

The randomized algorithms we present in this work for supporting the prementioned PD operations are based on an efficient processor-mapping of sequential search trees and hash tables, even though, other sequential data structures might be supported as well, e.g., treaps, skip lists [35]. To this end, in the forthcoming analysis of the PD algorithms we do not make any assumptions about the underlying local (sequential) dictionary. In order to simplify the exposition, the following notational conventions are employed.

- $\tau_{ID}(n,m)$ – denotes the time required to sequentially insert n items into a local dictionary of size m,
- $\tau_{FD}(n,m)$ – denotes the time required to sequentially find n items in a local dictionary of size m,
- $\tau_{DD}(n,m)$ – denotes the time required to sequentially delete n items from a local dictionary of size m,
- $\tau_{CD}(m)$ – denotes the time required to sequentially construct a dictionary of size m (we note that in general $\tau_{CD}(m) \leq \sum_{i=0}^{m-1} \tau_{ID}(1,i)$),
- $\tau_{JD}(m_1,m_2)$ – denotes the time required to sequentially join two dictionaries of size m_1 and m_2 into one of size m_1+m_2.

4.2 The PD Algorithms

The quasi randomized algorithms we present in this section derive from the ideas of [14, 15]. Let us outline the BSP derivations of these algorithms. We note that since our algorithms make use of the quasi-random functions of section 3, we assume an implicit preprocessing step for the construction of the underlying quasi-random function. By way of corollary 3.2, the time required for this preprocessing step is $T_{c\_qrnd}(n,p)$.

The PD is maintained as a collection of disjoint local (sequential) dictionaries. For the insertion of n new items (at most $\lceil n/p \rceil$ per processor), each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at quasi-random chosen processors, where they are inserted in their local dictionaries. The procedure to manage the insert operation on a PD D is InsertD*($\{y_1,y_2,\ldots,y_n\}$, D) and is outlined in figure 4.

```
begin InsertD* ({y1, y2, ..., yn}, D)
1.   denote by Y set {y1, y2, ..., yn} ;
2.   distribute quasi-randomly the items of Y among the p processors ;
3.   for each processor p<k>, 0 <= k < p, in parallel
4.        do InsertD_Seq(Y<k>, D<k>) ;
end InsertD*
```

Figure 4: Procedure InsertD*.

The correctness of procedure InsertD* is immediate; it therefore remains to bound the resources required by the aforementioned procedure. The following lemma bounds the number of items received by any processor as a result of the quasi-random distribution of items to processors (line 2).

Lemma 4.1 *Let in procedure* InsertD* $Y = \bigcup_k Y^{\langle k \rangle}$ *and* $D = \bigcup_k D^{\langle k \rangle}$, $0 \leq k < p$. *Then, for all* $0 \leq k < p$,

$$|Y^{\langle k \rangle}| = \begin{cases} \min\left\{O\left(\frac{n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \textit{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \lg n}{n}}\right)\right)\frac{n}{p} & \textit{otherwise,} \end{cases}$$

and

$$|D^{\langle k \rangle}| = \begin{cases} \min\left\{O\left(\frac{m}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \textit{if } \frac{m}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \lg n}{m}}\right)\right)\frac{m}{p} & \textit{otherwise,} \end{cases}$$

with probability $1 - n^{-\Omega(1)}$, *where* m *denotes the total number of items in* D *prior to the insertion.*

Proof In procedure InsertD*, the quasi-random distribution of items to processors implies that the results of section 3 can be employed to bound the number of items per processor. Hence, invoking equation (3) of corollary 3.1, for $n/p = 1$, it follows that

$$\Pr\left[|Y^{\langle k \rangle}| = \Omega\left(\frac{\lg n}{\lg\lg n}\right)\right] = n^{-\Omega(1)}.$$

Similarly, by invoking corollary 3.1 (4), for $n/p = \omega(\lg n)$, it follows that

$$\Pr\left[|Y^{\langle k \rangle}| = \left(1 + \Omega\left(\sqrt{\frac{p \lg n}{n}}\right)\right)\frac{n}{p}\right] = n^{-\Omega(1)}.$$

The analysis of the remaining cases (i.e., $n/p = O(\lg n)$) follows a line of arguments similar to that of the aforementioned cases and it is omitted. The bound on $|D^{\langle k \rangle}|$ follows similarly. ■

The resources required for procedure InsertD* are bounded as follows. By way of corollary 3.2, with probability $1 - n^{-\Omega(1)}$, the quasi-random distribution of items to processors (line 2) takes time $C_{e\_qrnd}(n,p)$ and $M_{e\_qrnd}(n,p) + \max\{L, g|Y^{\langle k \rangle}|\}$, for computation and communication respectively. The insertion of the $|Y^{\langle k \rangle}|$ items in the local dictionaries (line 4) takes $\max\{L, \tau_{ID}(|Y^{\langle k \rangle}|, |D^{\langle k \rangle}|)\}$ time. Thus, insertion can be implemented efficiently by employing this simple quasi-randomized scheme in time

$$T_{e\_qrnd}(n,p) + \max\left\{L, g|Y^{\langle k \rangle}|\right\} + \max\left\{L, \tau_{ID}(|Y^{\langle k \rangle}|, |D^{\langle k \rangle}|)\right\}.$$

For the deletion of n items (at most $\lceil n/p \rceil$ per processor), the following (rather similar to InsertD*) procedure is employed. Each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at quasi-random chosen processors, where they are deleted from their local dictionaries. The procedure to manage the delete operation on a PD D is DeleteD*($\{y_1, y_2, \ldots, y_n\}$, D) and is outlined in figure 5. A find operation is implemented similarly.

begin DeleteD* ($\{y_1, y_2, \ldots, y_n\}$, D)
1. denote by Y set $\{y_1, y_2, \ldots, y_n\}$;
2. *distribute* quasi-randomly the items of Y among the p processors ;
3. **for** each processor $p^{\langle k \rangle}$, $0 \le k < p$, in parallel
4. **do** DeleteD_Seq($Y^{\langle k \rangle}$, $D^{\langle k \rangle}$) ;

end DeleteD*

Figure 5: Procedure DeleteD*.

The correctness of procedure DeleteD* is immediate. i.e., the quasi-random distribution of items to processors results in identical distributions for both InsertD* and DeleteD* procedures. The analysis of the resources required for procedure DeleteD* is similar to that of InsertD* and it is omitted. Thus, deletion can be implemented efficiently by employing this simple quasi-randomized scheme in time

$$T_{e\_qrnd}(n,p) + \max\left\{L, g|Y^{\langle k \rangle}|\right\} + \max\left\{L, \tau_{DD}(|Y^{\langle k \rangle}|, |D^{\langle k \rangle}|)\right\}.$$

We therefore obtain the following result.

Theorem 4.1 *For any constant $\rho > 0$, there exists a randomized BSP algorithm for supporting the PD operations* INSERTD*, FINDD*, *and* DELETED*, *that for all integers n, p, L, and g such that $n/p = \Omega(\lg n)$, $L = O((n/p)\lg(m/p))$, and $g = O(\lg(m/p))$, with probability $1 - O(n^{-\rho})$, requires time, $O((n/p)\lg(m/p))$ per* INSERTD*, *$O((n/p)\lg(m/p))$ per* FINDD*, *and $O((n/p)\lg(m/p))$ per* DELETED* *operation, where m refers to the total number of items in the underlying PD. The algorithm requires $O(1 + \sqrt{n}/p) + T_{brd}(n,p)$ preprocessing time.*

Proof For any constant $\rho > 0$ and $n/p = \Omega(\lg n)$, it follows by way of lemma 4.1 that $Y^{\langle k\rangle} = O(n/p)$, with probability $1 - O(n^{-\rho})$. Similarly, $D^{\langle k\rangle} = O(m/p)$, with probability $1 - O(n^{-\rho})$. Hence, by substituting in the analysis of procedures INSERTD* and DELETED*, $O((n/p)\lg(m/p))$ for $\tau_{ID}(O(n/p), O(m/p))$, $\tau_{FD}(O(n/p), O(m/p))$ and $\tau_{DD}(O(n/p), O(m/p))$ we obtain the desired result, i.e., balanced binary trees (2-3-4 trees) [7, 42]. Note that, by way of corollary 3.2, the quasi-random evaluation time is bounded by $\max\{L, O(n/p)\}$ and $\max\{L, O(gn/p)\}$ for computation and communication respectively. The time required for constructing the underlying quasi-random function (preprocessing time) derives from corollary 3.2. ■

We note that we can improve the performance of the INSERTD* operation in several ways. For example, instead of implementing the insertion of $O(n/p)$ elements per processor as a sequence of $O(n/p)$ single insert operations, we use the following scheme on balanced binary trees (2-3-4 trees). To simplify the exposition, in the remainder of this section we assume that $n/p = \Omega(\lg n)$. Each processor constructs an auxiliary 2-3-4 tree on the $O(n/p)$ items it receives, as a result of the quasi-random distribution of items to processors (line 2), in time $\tau_{CD}(O(n/p)) = O((n/p)\lg(n/p))$ [7, 42], and subsequently joins this auxiliary 2-3-4 tree with its local tree that is part of D. The latter operation requires time $\tau_{JD}(O(n/p), O(m/p)) = O(\lg n/p + \lg m/p)$ [7, 42]. Overall, an improved $\tau_{ID}(O(n/p), O(m/p)) = O((n/p)\lg(n/p) + \lg(m/p))$ computation time bound can be claimed for insertion. The following analogue of theorem 4.2 can then be derived.

Theorem 4.2 *For any constant $\rho > 0$, there exists a randomized BSP algorithm for supporting the PD operations* INSERTD*, FINDD*, *and* DELETED*, *that for all integers n, p, L, and g such that $n/p = \Omega(\lg n)$, $L = O((n/p)\lg(n/p))$, and $g = O(\lg(n/p))$, with probability $1-O(n^{-\rho})$, requires time, $O((n/p)\lg(n/p)+\lg(m/p))$ per* INSERTD*, *$O((n/p)\lg(m/p))$ per* FINDD*, *and $O((n/p)\lg(m/p))$ per* DELETED* *operation, where m refers to the total number of items in the underlying PD.*

Remark We note that quasi-random functions can be employed to deliver optimal performance for PPQ operations (along the lines of PD operations). The significant advantage, however, of employing quasi-random hash functions is that the processor location of items can be effectively determined (in contrast to purely random functions). For example, there exist applications involving parallel computations that generate tasks that have to be stored and subsequently scheduled for execution according to their priority, e.g., branch-and-bound applications [26, 27, 28]. As noted in [25], tasks are usually not independent of one another, and hence, they have to be able to communicate. Employing quasi-random function allows the efficient determination of the location of a task.

4.3 Duplicate Item Handling

The explicit assumption in the analysis of the PD operations is that items are unique. In case of duplicate items, the upper bounds presented in the previous section do not apply. To remedy the situation, in the remainder of this section we present a simple randomized scheme for the elimination of duplicates.

Recall that we assume that the item values belong to a finite universe $U = \{0, 1, \ldots, q-1\}$, q prime, equipped with a total order $<$ that can be evaluated in unit time by one processor for any given pair of arguments. One technique for eliminating duplicates is to sort items according to their value so that items with the same value are adjacent in the sorted output. A multiplicity of BSP comparison-based sorting algorithms can be employed to realize the aforementioned scheme [11, 13, 17, 18, 19, 20, 41]. For the scope of this work, however, we employ the BSP integer sorting algorithm in [18]. The choice of integer sorting (rather than comparison sorting) is motivated by the fact that the item values are known to belong to some finite universe $U = \{0, 1, \ldots, q-1\}$. Unfortunately, no BSP integer sorting algorithm is known to achieve optimality for all values of q. Along the lines of [43], we define the weaker operation of *semisorting* as that of reordering the items so that any subset of identically valued keys is adjacent in the semisorted output (but these subsets need not be ordered).

Lemma 4.2 *For any constant $\rho > 0$, there exists a p-processor randomized BSP algorithm for semisorting a sequence consisting of n integers in the range $[0, \ldots, q-1]$, where q is prime, that for all integers n, p and t, such that $n \geq p$ and $2 \leq t \leq p$, with probability $1 - O(n^{-\rho})$, requires time at most*

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

and

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

for computation and communication respectively, where $r = \lceil \lg n^{2+\rho} / \lg\max\{\lceil n/p\rceil, t\lceil\log_t p\rceil\}\rceil$. The overall time complexity of the algorithm is denoted by $T_{semi\_srt}(n,p) = C_{semi\_srt}(n,p) + M_{semi\_srt}(n,p)$.

Proof Let q be prime, $X \subseteq U = \{0, 1, \ldots, q-1\}$, $|X| = n$, and consider function h to be chosen at random from $H^1_{n^{2+\rho}}$. Then, by way of lemma 3.2,

$$\Pr\left[h \text{ is 1-perfect for } X\right] = 1 - O\left(n^{-\rho}\right).$$

It is therefore straightforward to verify that h maps distinct items of $X \subseteq \{0, 1, \ldots, q-1\}$ to distinct integer values in $[0, 1, \ldots, n^{2+\rho} - 1]$, with probability $1 - O(n^{-\rho})$. These values can be subsequently sorted in time $T_{rdx\_srt}(n, n^{2+\rho}, p)$ by way of lemma 1.3. The desired result follows. ∎

Remark We note that the case $q \leq n^{2+\rho}$ requires no randomization, i.e., integer sorting can be directly applied to the input. The duplicate elimination scheme now follows.

5 Experimental Results for PPQ related operations

Although the BSP model is generally regarded as an architectural model, it can also be employed as a programming paradigm [33]. The underlying concept in the BSP model is the notion of the superstep and the abstraction that non-local communication takes place between supersteps as a global operation. Thus, a BSP program may be viewed as a succession of supersteps, with the required non-local communication occurring at the end of each superstep. Viewed this way, the BSP model can be realized as a library of functions with architecture independent semantics for process creation, remote data access and bulk synchronization.

The Oxford BSP Toolset, *BSPlib*, implements such a paradigm [21]. The library functions of *BSPlib* are callable from standard imperative languages such as, C and Fortran. In the tables to follow the following remarks apply.

- CPU times are given in seconds on a shared memory Silicon Graphics Power Challenge system and a distributed-memory Cray T3D.
- The implementations are written in ANSI C and linked with the Oxford BSP Toolset, *BSPlib*. The code is compiled using the native `cc` compiler with option `-O3`.
- The high-precision real-time clock function `bsp_time` [21] (wall-clock time) has been employed for timings.

We note that only recompilation is required to run the same code on different parallel machines.

In the remainder of this section we present experimental results on PPQs – PDs will be treated elsewhere. Ordinary and leftist heaps are both implemented as the underlying sequential data structure. In the experiments we perform a number of INSERTQ* and DELETEMINQ* operations on heaps of sizes 128K and 512K for the

Insertion (Heap Size: 128K)				
	Ordinary Heap		Leftist Heap	
n	$p = 1$	$p = 4$	$p = 1$	$p = 4$
$0.5K$	0.0012	0.0005	0.0030	0.0008
$2K$	0.0054	0.0022	0.0119	0.0039
$8K$	0.0262	0.0087	0.0507	0.0263
$32K$	–	–	–	–

Insertion (Heap Size: 512K)				
	Ordinary Heap		Leftist Heap	
n	$p = 1$	$p = 4$	$p = 1$	$p = 4$
$0.5K$	0.0013	0.0005	0.0029	0.0017
$2K$	0.0053	0.0021	0.0119	0.0046
$8K$	0.0240	0.0083	0.0483	0.0163
$32K$	0.1142	0.0337	0.2084	0.0573

Table 5: Execution time for INSERTQ* on an SGI Power Challenge.

Deletion (Heap Size: 128K)				
	Ordinary Heap		Leftist Heap	
n	$p = 1$	$p = 4$	$p = 1$	$p = 4$
$1K$	0.0191	0.0110	0.0336	0.0096
$4K$	0.0735	0.0278	0.1414	0.0257
$16K$	0.2707	0.0974	0.5736	0.0917
$64K$	–	–	–	–

Deletion (Heap Size: 512K)				
	Ordinary Heap		Leftist Heap	
n	$p = 1$	$p = 4$	$p = 1$	$p = 4$
$1K$	0.0300	0.0136	0.0571	0.0283
$4K$	0.1160	0.0603	0.2385	0.0640
$16K$	0.4538	0.1160	1.0473	0.1788
$64K$	1.5614	0.3810	4.3834	0.6512

Table 6: Execution time for DELETEMINQ* on an SGI Power Challenge.

SGI Power Challenge and 64K and 256K for the Cray T3D. Data elements are integers drawn from a uniform distribution. The *generic* nature of our implementations allows for any native or user-defined data element type. To this end, a comparison function `compare` is used to facilitate the comparison of two elements e_1 and e_2. In particular, the `compare` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2. The timing results mentioned in the following tables are averages over a number of experiments. Each table describes the results we obtained for the INSERTQ* and DELETEMINQ* operations. The first column gives the size of the INSERTQ* and DELETEMINQ* operations, over all processors. The remaining columns give the timing results for varying number of processors.

We note that as the trace of these operations differs for varying machine configurations (number of processors) one cannot derive reliable speedup conclusions from the timing results mentioned; this is attributed to the randomly distributed nature of the underlying data structures and their associated operations. Another reason speedup conclusions cannot be easily drawn, particularly for the SGI machine, is the effect of caching, through the presence of first- and second-level cache, on operations involving a relatively small number of elements. Therefore, the timing results should be interpreted with caution.

As evidenced from tables 5 and 6 the INSERTQ* operation on four processors of the SGI machine is approximately 2–3.5 times faster than on one processor. For the DELETEMINQ* operation even higher performance is achieved. The corresponding tables for the Cray T3D indicate that the INSERTQ* operation on thirty-two processors is approximately 5–18 times faster than on one processor, with typical values 10–15. For the DELETEMINQ* operation comparable performance is achieved. We note that the size of the heaps and the number of elements inserted/deleted per operation has been kept relatively small to point-out the efficiency and practicality of our methods. For large problem instances, higher and more consistent performance is achieved. The main reason for the high performance of our implementations on small problem instances derives from the efficiency of the underlying parallel selection operation (less than 10% of the total running time), which justifies the use of selection instead of sorting. Although it might have been expected, following the theoretical conclusions, that the leftist heap implementation would yield higher performance, particularly for INSERTQ* operations, this has not been the case in general. The explanation for this is that ordinary heap operations involve array indexing whereas leftist heap operations involve more complex (and expensive) pointer manipulations.

Insertion (Heap Size: 64K)						
	Ordinary Heap			*Leftist Heap*		
n	$p=1$	$p=8$	$p=32$	$p=1$	$p=8$	$p=32$
$1K$	0.0049	0.0013	0.0011	0.0098	0.0021	0.0012
$2K$	0.0106	0.0022	0.0017	0.0196	0.0036	0.0019
$4K$	0.0235	0.0044	0.0022	0.0391	0.0065	0.0027

Insertion (Heap Size: 256K)						
	Ordinary Heap			*Leftist Heap*		
n	$p=1$	$p=8$	$p=32$	$p=1$	$p=8$	$p=32$
$1K$	0.0044	0.0016	0.0009	0.0097	0.0021	0.0010
$4K$	0.0191	0.0062	0.0019	0.0391	0.0072	0.0026
$16K$	0.0938	0.0258	0.0071	0.1561	0.0261	0.0084

Table 7: Execution time for InsertQ* on a Cray T3D.

6 Conclusion

In this work we have presented efficient and practical PPQ and PD algorithms on the BSP model of computation along with an experimental study of the PPQ operations. We have shown that our algorithms are communication efficient and achieve optimality to within small multiplicative constant factors for a range of parallel machines, including the SGI Power Challenge and the Cray T3D.

A Tail Inequalities

This section is devoted to the proof of a technical combinatorial lemma. We start by presenting some tight probabilistic bounds.

Theorem A.1 *Let $Y_1, Y_2, \ldots, Y_n$ be independent random variables such that, for $1 \leq i \leq n$, $0 \leq Y_i \leq z$. Then, for $Y = \sum_{i=1}^{n} Y_i$, $\mu = E[Y]$, and any $u > 0$,*

$$Pr[Y \geq (1+u)\mu] \leq \left(\left(\frac{1}{1+u}\right)^{(1+u)\mu/n} \left(\frac{z - \mu/n}{z-(1+u)\mu/n}\right)^{z-(1+u)\mu/n} \right)^{n/z} \leq \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{\mu/z}.$$

Theorem A.2 *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = P_i$, where $0 < P_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = E[X] = \sum_{i=1}^{n} P_i$, and any $u > 0$,*

Deletion (Heap Size: 64K)						
	Ordinary Heap			*Leftist Heap*		
n	$p=1$	$p=8$	$p=32$	$p=1$	$p=8$	$p=32$
$2K$	0.0715	0.0229	0.0326	0.0957	0.0225	0.0309
$4K$	0.1368	0.0372	0.0342	0.1865	0.0369	0.0345
$8K$	0.2611	0.0697	0.0431	0.3578	0.0642	0.0429

Deletion (Heap Size: 256K)						
	Ordinary Heap			*Leftist Heap*		
n	$p=1$	$p=8$	$p=32$	$p=1$	$p=8$	$p=32$
$2K$	0.0810	0.0260	0.0332	0.1108	0.0253	0.0333
$8K$	0.3065	0.0594	0.0483	0.4282	0.0644	0.0469
$16K$	1.160	0.1847	0.0842	1.633	0.2145	0.0787

Table 8: Execution time for DeleteMinQ* on a Cray T3D.

$$Pr[X \geq (1+u)\mu] \leq \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{\mu}.$$

The aforementioned tail inequalities are attributed to [23] and [6] respectively. A simpler bound ([1]) is obtained for the special case $0 < u < 1$ and is summarized bellow [34].

Theorem A.3 *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = P_i$, where $0 < P_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n P_i$, and any $0 < \varepsilon < 1$,*

$$Pr[X \geq (1+\varepsilon)\mu] \leq e^{-(1/3)\varepsilon^2 \mu},$$

and

$$Pr[X \leq (1-\varepsilon)\mu] \leq e^{-(1/2)\varepsilon^2 \mu}.$$

Theorem A.4 (Azuma's Inequality) *Let $Z_0, Z_1, \ldots$ be a martingale sequence such that for each i,*

$$|Z_i - Z_{i-1}| \leq \vartheta_i,$$

where ϑ_i may depend on i. Then, for all $n \geq 0$ and any $\lambda > 0$,

$$Pr[|Z_n - Z_0| \geq \lambda] \leq 2e^{-\lambda^2/(2\sum_{i=1}^n \vartheta_i^2)}.$$

The aforementioned tail inequality (referred to as Azuma's inequality) is due to [2, 23]. The application of Azuma's inequality has its origins in [31], and various forms and applications are surveyed in [34].

A.1 Two Combinatorial Lemmas

Lemma A.1 *Let W be the number of bins containing exactly one ball when m balls are thrown independently and uniformly into k bins. Then,*

$$\mu_1 = E[W] = m\left(1 - \frac{1}{k}\right)^{m-1} \sim m\, e^{-(m-1)/k},$$

and for $\lambda > 0$,

$$\begin{aligned} Pr[|W - \mu_1| \geq \lambda] &\leq 2\exp\left(-\frac{\lambda^2(k-1/2)}{\left(3 + \frac{m-1}{k-1}\right)^2 (k^2 - \mu_0^2)^2}\right) \\ &= 2\exp\left(-\frac{\lambda^2(k-1/2)}{\left(3 + \frac{m-1}{k-1}\right)^2 (k^2 - (\frac{k-1}{m}\mu_1)^2)}\right). \end{aligned}$$

Proof The probability that a bin gets exactly i balls when m balls are placed randomly into k bins is

$$\binom{m}{i}\left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{m-i}.$$

Therefore, the expected number of bins with exactly i balls is bounded above by

$$k\binom{m}{i}\left(\frac{1}{k}\right)^i \left(1 - \frac{1}{k}\right)^{m-i}.$$

Substituting in the above bound 1 for i we obtain the desired bound for μ_1. In the remainder of this proof we concentrate on proving the tail bound.

Let $G_0 \subseteq G_1 \subseteq \ldots \subseteq G_m$ be a filter, where G_t is generated by the random choice of bins for the first t balls, i.e., the set of events corresponding to the state of the bins at time t. Let W be the random variable denoting the number of bins with exactly one ball at time m, and let $W_t = \mathrm{E}[W \mid G_t]$ denote the conditional expectation of W at time t. Then, the random variables $W_0, W_1, \ldots, W_m$ form a martingale with $W_0 = \mathrm{E}[W]$ and $W_m = W$.

Let $w(X, Y, t)$ denote the expectation of W given that X bins are empty at time t and Y bins contain exactly one ball. The probability that any of these empty bins receives exactly one ball during the last $m - t$ time units is given by $\binom{m-t}{1}(1/k)(1-1/k)^{m-t-1}$. Similarly, the probability that any of these bins containing exactly one ball does not receive a ball during the last $m-t$ time units is given by $(1-1/k)^{m-t}$. By linearity of expectations, we obtain that the number of these bins that contain exactly one ball at the end of the m placements is given by

$$w(X, Y, t) = X\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + Y\left(1-\frac{1}{k}\right)^{m-t}.$$

Let the random variable X_t denote the number of empty bins at time t, and, similarly, the random variable Y_t denote the number of bins containing exactly one ball at time t. Then,

$$W_{t-1} = w(X_{t-1}, Y_{t-1}, t-1) = X_{t-1}\binom{m-t+1}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t} + Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t+1}.$$

At time $t-1$ the values of X_{t-1}, Y_{t-1} and W_{t-1} are determined. Then, at time t, there are three possibilities.

(1) With probability X_{t-1}/k, the t-th ball goes into a currently empty bin. Then, $X_t = X_{t-1} - 1$, $Y_t = Y_{t-1} + 1$, and

$$W_t = w(X_{t-1}-1, Y_{t-1}+1, t) = (X_{t-1}-1)\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + (Y_{t-1}+1)\left(1-\frac{1}{k}\right)^{m-t}.$$

(2) With probability Y_{t-1}/k, the t-th ball goes into a bin currently containing exactly one ball. Then, $X_t = X_{t-1}$, $Y_t = Y_{t-1} - 1$, and

$$W_t = w(X_{t-1}, Y_{t-1}-1, t) = X_{t-1}\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + (Y_{t-1}-1)\left(1-\frac{1}{k}\right)^{m-t}.$$

(3) With probability $1 - (X_{t-1} + Y_{t-1})/k$, the t-th ball goes into a bin currently containing more than one balls. Then, $X_t = X_{t-1}$, $Y_t = Y_{t-1}$, and

$$W_t = w(X_{t-1}, Y_{t-1}, t) = X_{t-1}\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t}.$$

We next focus on the difference random variable $\Delta_t = W_t - W_{t-1}$. Corresponding to W_t, the distribution of Δ_t (given the state at time $t-1$) can be characterized as follows.

(1) With probability X_{t-1}/k, the value of Δ_t is

$$\begin{aligned}
\delta_0 &= W_t - W_{t-1} \\
&= (X_{t-1}-1)\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + (Y_{t-1}+1)\left(1-\frac{1}{k}\right)^{m-t} - \\
&\quad X_{t-1}\binom{m-t+1}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t} - Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t+1} \\
&= \left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t}\left(X_{t-1}\frac{m-t-k+1}{k-1} + Y_{t-1} - \frac{(m-t)k}{k-1} + k\right).
\end{aligned}$$

Thus, since $1 \le t \le m$ and $0 \le X_{t-1}, Y_{t-1} \le k$, it follows that the absolute value of δ_0 is bounded as follows:

$$
\begin{aligned}
|\delta_0| &= |W_t - W_{t-1}| \\
&\leq \left(3 + \frac{m-1}{k-1}\right)\left(1 - \frac{1}{k}\right)^{m-t}.
\end{aligned}
$$

(2) With probability Y_{t-1}/k, the value of Δ_t is

$$
\begin{aligned}
\delta_1 &= W_t - W_{t-1} \\
&= X_{t-1}\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + (Y_{t-1}-1)\left(1-\frac{1}{k}\right)^{m-t} - \\
&\quad X_{t-1}\binom{m-t+1}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t} - Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t+1} \\
&= \left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t}\left(X_{t-1}\frac{m-t-k+1}{k-1} + Y_{t-1} - k\right).
\end{aligned}
$$

Thus, since $1 \leq t \leq m$ and by definition $0 \leq X_{t-1}, Y_{t-1} \leq k$, it follows that the absolute value of δ_1 is bounded as follows:

$$
\begin{aligned}
|\delta_1| &= |W_t - W_{t-1}| \\
&\leq \left(3 + \frac{m-1}{k-1}\right)\left(1 - \frac{1}{k}\right)^{m-t}.
\end{aligned}
$$

(3) With probability $1 - (X_{t-1} + Y_{t-1})/k$, the value of Δ_t is

$$
\begin{aligned}
\delta_2 &= W_t - W_{t-1} \\
&= X_{t-1}\binom{m-t}{1}\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t-1} + Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t} - \\
&\quad X_{t-1}(m-t+1)\left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t} - Y_{t-1}\left(1-\frac{1}{k}\right)^{m-t+1} \\
&= \left(\frac{1}{k}\right)\left(1-\frac{1}{k}\right)^{m-t}\left(X_{t-1}\frac{m-t-k+1}{k-1} + Y_{t-1}\right).
\end{aligned}
$$

Thus, since $1 \leq t \leq m$ and by definition $0 \leq X_{t-1}, Y_{t-1} \leq k$, it follows that the absolute value of δ_2 is bounded as follows:

$$
\begin{aligned}
|\delta_2| &= |W_t - W_{t-1}| \\
&\leq \left(2 + \frac{m-1}{k-1}\right)\left(1 - \frac{1}{k}\right)^{m-t}.
\end{aligned}
$$

It therefore follows that the value of the difference Δ_t is bounded by

$$
-\left(3 + \frac{m-1}{k-1}\right)\left(1-\frac{1}{k}\right)^{m-t} \leq \Delta_t \leq \left(3 + \frac{m-1}{k-1}\right)\left(1-\frac{1}{k}\right)^{m-t}.
$$

For $1 \leq t \leq m$, we set $r = (m-1)/k$, $\vartheta_t = (3 + (m-1)/(k-1))(1-1/k)^{m-t}$, and we have that $|\Delta_t| = |W_t - W_{t-1}| \leq \vartheta_t$. By a straightforward calculation,

$$
\begin{aligned}
\sum_{t=1}^{m} \vartheta_t^2 &= \left(3 + \frac{m-1}{k-1}\right)^2 \sum_{t=0}^{m-1} \left(1 - \frac{1}{k}\right)^{2t} \\
&= \left(3 + \frac{m-1}{k-1}\right)^2 \frac{1-(1-(1/k))^{2m}}{1-(1-(1/k))^2} \\
&= \left(3 + \frac{m-1}{k-1}\right)^2 \frac{k^2 - (\mu_1(k/m)(1-(1/k)))^2}{2k-1} \\
&= \left(3 + \frac{m-1}{k-1}\right)^2 \frac{k^2 - \mu_0^2}{2k-1} \\
&= \left(3 + \frac{m-1}{k-1}\right)^2 \frac{k^2 - (\frac{k-1}{m}\mu_1)^2}{2k-1}.
\end{aligned}
$$

The desired result is obtained by invoking theorem A.4 and substituting the above expression for $\sum_{t=1}^{m} \vartheta_t^2$. ■

Similarly, we apply Azuma's inequality to bound the number of empty bins when m balls are placed randomly into k bins.

Lemma A.2 ([35]) *Let Z be the number of empty bins when m balls are thrown independently and uniformly into k bins. Then,*

$$\mu_0 = E[Z] = k\left(1 - \frac{1}{k}\right)^m \sim e^{-m/k}k,$$

and for $\lambda > 0$,

$$Pr[|Z - \mu_0| \geq \lambda] \leq 2e^{-\lambda^2\left((k-1/2)/\left(k^2-\mu_0^2\right)\right)}.$$

$$Pr[|Z - \mu_0| \geq \lambda]\, 2\exp\left(-\frac{\lambda^2(k-1/2)}{k^2-\mu_0^2}\right)$$

References

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

[2] K. Azuma. Weighted sums of certain dependent random variables. *Tohoku Mathematical Journal*, 19:357-367, 1967.

[3] A. Baumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of Euro-Par'96 Parallel Processing*, LNCS 1124, Springer-Verlag, August 1996.

[4] A. Baumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 233-242, Padova, Italy, June 1996.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proceedings of the 9-th ACM STOC*, pp. 106-112, 1977.

[6] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annuals of Mathematical Statistics*, 23:493-509, 1952.

[7] H. T. Cormen, C. E. Leiserson, and L. R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[8] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of ICALP'90*, pp. 6-19, LNCS 443, Springer Verlag, July 1990.

[9] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proccedings of the 22-nd ACM STOC*, pp. 117-127, 1990.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Computing*, 23(4):738-761, 1994.

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

[12] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, May 1996.

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

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

[15] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Sunnyvale, USA, August 9-11, 1996.

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

[17] A. V. Gerbessiotis and C. J. Siniolakis. Efficient Deterministic Sorting on the BSP Model. Technical Report PRG-TR-19-96, Oxford University Computing Laboratory, October 1996.

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

[19] A. V. Gerbessiotis and C. J. Siniolakis. A Randomized Sorting Algorithm on the BSP model. In Proceedings of IPPS'97, April 1997.

[20] M. T. Goodrich. Communication-Efficient Parallel Sorting. *Proceedings 26th ACM STOC*, 1996.

[21] M. W. Goudreau, J. M. D. Hill, K. Lang, W. F. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library (preliminary version). Technical Report, Oxford University Computing Laboratory, April 1996 (see `www.bsp-worldwide.org` for details).

[22] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *IPL*, 33:305-308, 1990.

[23] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13-30, 1963.

[24] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[25] A. R. Karlin and E. Upfal. Parallel hashing – An efficient implementation of shared memory. In *Proceedings of the ACM Symposium on theory of computing*, pp. 160-168, 1986.

[26] R. M. Karp, M. Saks, and A. Wigderson. On a search problem related to branch-and-bound procedures. In *Proceedings of the 27-th IEEE FOCS*, pp. 19-28, 1986.

[27] R. M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Symposium on theory of computing*, pp. 290-300, 1988.

[28] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765-789, 1993.

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

[30] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95-132, 1990.

[31] B. Maurey. Construction de suites symètriques. *Compt. Rend. Acad. Sci. Paris*, 288:679-681, 1979.

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

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

[34] C. J. H. McDiarmid. On the method of bounded differences. *Surveys in Combinatorics: Invited Papers at the 12-th British Combinatorial Conference*, (J. Siemons, ed.), pp. 148-188, 1989.

[35] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1996.

[36] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceeding of ICALP'83: Automate, Languages and Programming*, pp. 597-609, LNCS 154, Springer Verlag, 1983.

[37] M. C. Pinotti and G. Pucci. Parallel priority queues. *IPL*, 40:33-40, 1990.

[38] A. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proceedings of the 4-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 127-137, 1992.

[39] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Proceedings of the 6-th IEEE Symposium on Parallel and Distributed Processing*, pp. 97-103, 1994.

[40] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Proceedings of the 2-nd International Workshop, Irregular'95*, pp. 379-393, LNCS 980, Springer Verlag, 1995.

[41] C. J. Siniolakis. On the Complexity of BSP Sorting. Technical Report PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996.

[42] R. E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, 1983.

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

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

[45] L. G. Valiant. A combining mechanism for parallel computers. In *Parallel Architectures and Their Efficient Use*, pp. 1-10, LNCS 678, Springer-Verlag, 1993.