Merging on the BSP Model

Alexandros V. Gerbessiotis
CIS Department
New Jersey Institute of Technology
Newark, NJ 07102
USA.

Constantinos J. Siniolakis
The American College of Greece
6 Gravias St.
Aghia Paraskevi
Athens, 15342
Greece.

June 26, 2000

Abstract

The design of complex parallel algorithms relies heavily on a set of primitive operations. In this work, we examine the problem of merging two sorted sequences in an architecture independent setting. We derive parallel algorithms that can be used on a variety of parallel machines and whose performance can be reliably predicted if these algorithms are analyzed under the Bulk-Synchronous Parallel (BSP) model. While our algorithms are fairly simple themselves, description of their performance in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables parallel software to be written once and for all that can be migrated efficiently among a variety of parallel platforms. The optimal merging algorithm presented in this work achieves asymptotically optimal parallel efficiency compared to any optimal sequential merging algorithm.

Keywords. BSP algorithms - merging - architecture independent - parallel algorithms

1 Introduction

The design of a complex algorithm relies on a set of primitive operations and the instruments required to compile these operations into an algorithm. In this work, we examine the problem of merging sorted sequences in an architecture independent setting. We derive parallel algorithms that can be used on a variety of parallel machines and whose performance can be reliably predicted if these algorithms are analyzed under the Bulk-Synchronous Parallel (BSP) model. While our algorithms are fairly simple themselves, description of their performance in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables parallel software to be written once and for all that can be migrated efficiently among a variety of parallel platforms. The main merging algorithm presented in this work achieves asymptotically optimal parallel efficiency compared to the best sequential merging algorithm for a wide range of the BSP parameters p, L and g, and input problem size.

1.1 The Bulk-Synchronous Parallel Model

In the early days of computing it was assumed that sequential machines would be superseded by parallel ones, however, this has yet to happen despite the rapid advances in communication and processor technologies. Attempts in the mid 1980s and early 1990s to popularize parallel computers and garner enough acceptance for parallel computing failed. The main reason for this failure of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von-Neumann model has served sequential computing. The introduction of the Bulk-Synchronous Parallel model comes to address these limitations and to 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.

The *Bulk-Synchronous Parallel (BSP)* model of computation has been proposed by L.G. Valiant [20, 21], as a unified framework for the design, analysis, and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which

allows the performance of parallel and distributed systems to be analyzed and measured reliably. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems (a global clock is utilized to achieve synchronization on every instruction), and (ii) fully asynchronous systems (synchronization is realized pairwise whenever communication is required). Processing units in the BSP model advance jointly through the program, with the required remote communication occurring between supersteps. The BSP computer as described in [20] consists of the following three mechanisms.

(1) A *collection of p processor/memory components* numbered $0, \ldots, p-1$. Each such component consists of a sequential processor with a block of local memory.

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

(3) A *mechanism* for efficiently synchronizing, all, or a subset, of the components.

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

(1) p - the number of processor/memory components.

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

(3) g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. Let an h-relation be a routing instance where each processor sends or receives at most h messages. The time to realize an h-relation is then, gh for large enough h (or L, otherwise).

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each component is given a task to perform using locally available data. The task may include computations on values held locally at the start of the superstep, message transmissions, and message receipts. A superstep can complete at any time after the first L time steps. Although the model stresses global, barrier-style, synchronization, pairs of components may synchronize pairwise by sending messages to and from an agreed memory location. Such message exchanges, however, should respect the superstep rules [20].

The time complexity of a superstep $\mathcal{S}$ in a BSP algorithm is determined according to the original cost model of [20]. 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 $\mathcal{S}$, and h is the maximum number of messages transmitted or received by any processor. Note that, if h_t is the maximum number of messages transmitted by any processor during $\mathcal{S}$, and h_r is the maximum number of messages received by any processor during $\mathcal{S}$, then $h = \max\{h_t, h_r\}$. Along the lines of [11], and in order to simplify the description of the performance of our algorithms, we assume that each superstep is either pure computation or pure communication. This way, the two kinds of costs are distinguishable.

The minimum superstep size is bounded below by the frequency with which barrier synchronization can be realized. As it was noted in [11], any particular BSP machine may impose a lower bound on L on a twofold basis: (a) there is a minimum time requirement for doing synchronization, (b) the theoretical bound on g may be realizable only for large enough L, so that the communication network can achieve the stated capacity. The maximum superstep size, however, is a property of the application, since by definition a superstep can accommodate as much work as can be done without a remote data exchange. The actual values of L and g for any BSP machine are likely to depend on p. In particular, the parameters L and g are considered to be non-decreasing functions of p.

The purpose of efficient BSP algorithm design (and of parallel algorithm design in general) is to minimize the number of supersteps (synchronization operations), minimize communication and maximize processor utilization. Description of the performance of the algorithms in terms of the BSP parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables parallel software to be written once and for all that can be migrated efficiently among a variety of parallel machines.

1.2 Specifying the Performance of BSP Algorithms

We specify the performance of a BSP algorithm $\mathcal{A}^*$ as follows. First, we define a sequential algorithm $\mathcal{A}$ with which we are comparing it and make explicit how we charge for the basic operations of both $\mathcal{A}^*$ and $\mathcal{A}$. Since the mechanisms of the BSP model are similar to those used by sequential models and we are interested in performance ratios of these two we can define

operations in a higher level than machine level instructions. Throughout this work, lg and log denote logarithms to the base two and e respectively. A dominant term in the BSP merging algorithms to be claimed later is contributed by sequential merging performed independently at each processor. We charge $T^*_{mrg}(m,k)$ time steps for sequentially merging two non-decreasing sequences X and Y, of sizes m and k respectively. For other operations we charge $c_\otimes$ time steps for operations over the associative operator $\otimes$ in parallel-prefix computations and unit cost for every comparison performed.

2 Fundamental Operations

The *parallel-prefix* operation [15, 16], and its variant, *segmented parallel-prefix*, is a powerful primitive operation with many parallel applications [3]. BSP algorithms for parallel-prefix are described in [10]. They will become auxiliary routines for the merging algorithm described in a later section.

Definition 1 *Let $\langle X_1, X_2, \ldots, X_p\rangle$ be a sequence of p vectors each of size n, such that $X_i = (x_{1,i}, \ldots x_{n,i})$, $1 \le i \le p$. The prefix of sequence $\langle X_1, X_2, \ldots, X_p\rangle$, over an associative operator $\otimes$, is defined to be the sequence $\langle X_1, X_1 \otimes X_2, \ldots, X_1 \otimes X_2 \otimes \ldots \otimes X_p\rangle$, where $X_i \otimes X_{i+1} = (x_{1,i} \otimes x_{1,i+1}, \ldots, x_{n,i} \otimes x_{n,i+1})$, $1 \le i \le p-1$.*

An optimal linear-time sequential algorithm exists for the prefix problem. A non-trivial BSP algorithm is presented in [10] and its performance is described by the following lemma.

Lemma 1 *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* [1]

$$C^n_{ppf}(p,\otimes) = \max\left\{L, c_\otimes(p-1)\left\lceil\frac{n}{p}\right\rceil\right\}\cdot[n \ge p] + (4h-2)\max\left\{L, c_\otimes(t-1)\left\lceil n/h\right\rceil\right\}\cdot[n < p]$$

and

$$M^n_{ppf}(p,\otimes) = 2\max\left\{L, gp\left\lceil\frac{n}{p}\right\rceil\right\}\cdot[n \ge p] + (4h-2)\max\left\{L, 2gt\left\lceil n/h\right\rceil\right\}\cdot[n < p],$$

[1] 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 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 of the algorithm is* $T^n_{ppf}(p, \oplus) = C^n_{ppf}(p, \oplus) + M^n_{ppf}(p, \oplus)$.

If n is smaller than p the parallel-prefix algorithm works in a pipelined fashion on a binary tree with p leaves. For larger n, each vector of size n is split into p pieces, piece i is transmitted to processor i and that processor computes the prefices on piece i of all vectors sequentially, and in the following superstep the results are returned to the sending processors. The proof of this lemma appears in [10]. A similar result has also been shown in [13].

For $c_\otimes = O(1)$, we write $T^n_{ppf}(p)$ (respectively $C^n_{ppf}(p)$, $M^n_{ppf}(p)$) for $T^n_{ppf}(p, \otimes)$ (respectively $C^n_{ppf}(p, \otimes)$, $M^n_{ppf}(p, \otimes)$). We also write $T_{ppf}(p, \otimes)$ (respectively $C_{ppf}(p, \otimes)$, $M_{ppf}(p, \otimes)$) for $T^1_{ppf}(p, \otimes)$ (respectively $C^1_{ppf}(p, \otimes)$, $M^1_{ppf}(p, \otimes)$).

The *segmented parallel-prefix* operation [16] consists of a sequence of prefix computations over the same associative operator $\otimes$ but on k disjoint sets of total size p. As shown in [16], one can solve the segmented prefix problem as a standard prefix problem, by first concatenating the input sets into a single set, and inserting a *barrier* at the start of each set of data. One next defines an associative operator $\bar{\otimes}$ that respects the barriers, but otherwise performs identically to the original operator $\otimes$. This augmented parallel-prefix operation is then realized as the standard parallel-prefix operation. We therefore obtain that the time required to realize n disjoint segmented parallel-prefix operations, each on k disjoint sets of total size p, over an augmented associative operator $\bar{\otimes}$ is at most, $T^n_{seg_ppf}(p, \bar{\otimes}) = 2T^n_{ppf}(p, \otimes)$.

3 Merging

Let $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_k \rangle$ be two non-decreasing sequences whose keys belong to a universe U equipped with a total order $<$ that can be evaluated in unit time by a sequential processor for any given pair of arguments. Without loss of generality, we assume that all the keys appearing in sequence X are distinct, i.e., if the keys are not distinct we can make them so by appending to each one their index/memory location storing it. We consider, in this section, the problem of *merging* these two sequences into a non-decreasing sequence - say, $Z = \langle z_1, \ldots, z_{m+k} \rangle$. Linear-time sequential algorithms to solve the merging problem are

discussed in [14]. In the following sections we present simple sub-optimal and rather more complicated optimal BSP merging algorithms.

We first introduce a few definitions. The *rank* of a key x in sequence X, denoted by $\text{rank}(x : X)$, is the number of keys of X that are less than or equal to x. Sequence $\bar{X}$ is a c-sample of sequence X, denoted by $\bar{X} = \text{sample}_c(X)$, if $\bar{X}$ consists of every c-th key of X, i.e., $\bar{X} = \text{sample}_c(X) = \langle x_c, \ldots, x_{ic}, \ldots, x_{\lfloor m/c \rfloor c} \rangle$, for all $1 \leq i \leq \lfloor m/c \rfloor$.

3.1 Batcher's Bitonic-Sort Algorithm on the BSP model

Let $V = \langle v_1, \ldots, v_n \rangle$ be a sequence of keys drawn from a linearly ordered set. The sequence V is bitonic if, for some positive integer $i \leq n$, we have $v_1 \geq \ldots \geq v_i \leq v_{i+1} \leq \ldots \leq v_n$. In [1] it is shown that there exists a sorting network of depth $\lceil \lg n \rceil$ with $n\lceil \lg n \rceil/2$ comparators that correctly sorts V. The following result generalizes the construction of [1] by providing an efficient BSP implementation of Batcher's bitonic-sort algorithm. The algorithm derives from [1] (immediate for $t = 2$ case) and is also described in [10].

Lemma 2 *There exists a p-processor BSP algorithm for sorting a bitonic sequence of length n, that for all integers n, p and t, such that $n \geq p$ and $2 \leq t \leq p$, requires time at most*

$$C_{btn}(n, p; t) = \lceil \log_t p + 1 \rceil \max \left\{ L, T^*_{mrg} \left(2^{\lceil \lg t \rceil - 1} \left\lceil \frac{n}{p} \right\rceil, 2^{\lceil \lg t \rceil - 1} \left\lceil \frac{n}{p} \right\rceil \right) \right\}$$

and

$$M_{btn}(n, p; t) = \lceil \log_t p + 1 \rceil \max \left\{ L, g 2^{\lceil \lg t \rceil - 1} \left\lceil \frac{n}{p} \right\rceil \right\},$$

for communication and computation respectively, where T^*_{mrg} refers to the time required by sequential merging. The overall time complexity is denoted by $T_{btn}(n, p; t) = C_{btn}(n, p; t) + M_{btn}(n, p; t)$.

Proof: The input is assumed to be evenly and orderly distributed among the p processors, i.e., key x_i in X, for all $1 \leq i \leq m$, resides in the local memory of processor $\lfloor i/q \rfloor$ at position $i \bmod q$ and key y_j in Y, for all $1 \leq j \leq k$, resides in the local memory of processor $\lfloor (m + k - j + 1)/q \rfloor$ at position $(m + k - j + 1) \bmod q$, where $q = \lceil (m + k)/p \rceil$. This distribution of keys to processors results in a sorted sequence for each processor except possibly for processor $\lfloor m/q \rfloor$ that might result in a bitonic sequence.

Let $W = \langle\langle w_{1,1}, \ldots, w_{1,l}\rangle, \ldots, \langle w_{i,1}, \ldots, w_{i,l}\rangle, \ldots, \langle w_{p,1}, \ldots, w_{p,l}\rangle\rangle$ be a sequence of keys drawn from a linearly ordered set, such that $w_{i,1} \leq \ldots \leq w_{i,l}$, for all $1 \leq i \leq p$, and sequence $\bar{W} = \langle w_{1,l}, \ldots, w_{p,l}\rangle$ is bitonic. We next outline how to modify Batcher's bitonic-sort algorithm to obtain a p-processor BSP algorithm to sort W. We replace each input to the bitonic-sort network by a non-decreasing sequence $w_{i,1} \leq \ldots \leq w_{i,l}$ of length l. We also replace each comparator by a sequential merging algorithm that receives as input two non-decreasing sequences each of length l, merges them, and outputs separately the top and bottom halves of the merged sequences. As shown in [2, 14] the concatenation of the merged sequences at the output of the bitonic-sort network yields the desired sorted sequence. The performance of the aforementioned algorithm can be improved as in [13], by combining $\lceil \lg t \rceil$ consecutive levels of comparators of the bitonic-sort network into a single level, for some integer $2 \leq t \leq p$. The so constructed network has a total of at most $\lceil \lceil \lg p \rceil / \lceil \lg t \rceil \rceil$ levels.

To simulate this sorting network on a BSP machine, we replace comparators at each level of the network by a pairs of processors. Each processor, therefore, communicates a total of at most $2^{\lceil \lg t \rceil - 1} l$ messages per level. Since the total number of levels is at most $\lceil \log_t p + 1 \rceil$ we obtain that the time complexity of the BSP algorithm is bounded above by $\lceil \log_t p + 1 \rceil \max\{L, T^*_{mrg}(2^{\lceil \lg t \rceil - 1} l, 2^{\lceil \lg t \rceil - 1} l)\}$ and $\lceil \log_t p + 1 \rceil \max\{L, g 2^{\lceil \lg t \rceil - 1} l\}$ for communication and computation respectively. By substituting in the above analysis for $l = \lceil n/p \rceil$ we obtain the desired result. ■

Two sorted sequences of lengths m and k respectively can be merged by Batcher's algorithm if the concatenation of the fomer sequence and the reverse of the latter one is input to the algorithm for sorting.

An application of the presented algorithm is in parallel sorting [4, 5, 6, 8, 9]. In parallel sorting using regular sampling [17], a sample can be sorted either sequentially (as was also done in [11] in the context of a randomized sorting algorithm) or in parallel. In experimental work reported in [4, 5] a BSP implementation of Batcher's sorter is used for sample sorting. Given the simplicity of the algorithm and its low overhead in non-essential operations, its practical performance exceeds those of any sequential approaches (eg sequential sorting or merging of p sorted sample sequences) even for small values of p, and especially in cases where the ratio n/p is low. Such experimental results can be found in [5] where a comparison to the

the work of [19] that uses sequential sample sorting is presented, and more recently, in [4].

3.2 An Optimal BSP Merging Algorithm

In this section, we provide an optimal BSP solution to the problem of merging two sequences X and Y that is based on *overpartitioning* X and Y into many pairs of subsequences such that we can construct sequence $Z = merge(X, Y)$ by concurrently merging groups of pairs of subsequences. The basic algorithmic idea derives from a merging algorithm of [18] as adapted in [12]. The merging algorithm presented in thise section was first described in a less detailed form in [10]. The most important operation in the merging algorithm is a binary search operation of keys of one of the input sequences into subsets of the input keys of the other sequence. The requirement of subset processor synchronization assumed in [10] is replaced in this work by using segmented parallel prefix.

Lemma 3 *Let ω_n be any function of n, such that $\omega_n > 1$ for sufficiently large n. Then, there exists a p-processor BSP algorithm for merging two non-decreasing sequences X and Y, of lengths m and k respectively and combined length $n = m + k$, that for all integers m, k, t and p such that $n/p = \omega_n \lg n$ and $2 \leq t \leq p$, requires time at most*

$$\begin{aligned} C_{mrg}(m,k,p;t) &= \max\{L, T^*_{mrg}(n_1,n_2)\} + C_{btn}(p\lceil\sqrt{\omega_n}\rceil, p; t) + 6\max\{L, O(\sqrt{\omega_n})\} + \\ &\quad 2\max\{L, O(1)\} + \max\{L, O(\frac{n}{p\sqrt{\omega_n}})\} + \max\{L, \sqrt{\omega_n}O(\lg\frac{n}{p\sqrt{\omega_n}})\} + 2C^{\lceil\frac{n}{p\sqrt{\omega_n}}\rceil}_{ppf}(p) \end{aligned}$$

and

$$\begin{aligned} M_{mrg}(m,k,p;t) &= 3\max\{L, g(n_1+n_2)\} + M_{btn}(p\sqrt{\omega_n}, p; t) + 2\max\{L, O(g)\} \\ &\quad \max\{L, O(g\sqrt{\omega_n})\} + 2M^{\lceil\frac{n}{p\sqrt{\omega_n}}\rceil}_{ppf}(p) + \max\{L, gO(\frac{n}{p\sqrt{\omega_n}})\} \end{aligned}$$

for communication and computation respectively, where $n_1 + n_2 = n/p + O(n/(p\sqrt{\omega_n}))$ and T^*_{mrg} refers to the time required by sequential merging. The overall time complexity of the algorithm is denoted by $T_{mrg}(m,k,p;t) = C_{mrg}(m,k,p;t) + M_{mrg}(m,k,p;t)$.

Proof: The input is initially evenly and orderly distributed among the p processors, i.e., key x_i in X, for all i such that $1 \leq i \leq m$ resides in the local memory of processor $\lfloor i/q \rfloor$ at

position $i \bmod q$ and key y_j in Y, for all j such that $1 \leq j \leq k$, resides in the local memory of processor $\lfloor (m+j)/q \rfloor$ at position $(m+j) \bmod q$, where $q = \lceil (m+k)/p \rceil$. Moreover, it is assumed that the keys are distinct. This can be achieved by attaching to each key the address of the memory location it is stored in. For data types whose bit or byte length is comparable to the length of the address describing them, such a transformation leads – in most cases – to a doubling of the overall number of comparisons performed in the worst case. For more complex data types such as strings of characters the extra cost may be negligible.

The procedure to manage the merging operation is outlined next.

Procedure MERGE (X, Y)

begin

Stage 1 : Subsequences $\bar{X}$ and $\bar{Y}$ are constructed such that $\bar{X}$ is a $\lceil n/(p\sqrt{\omega_n}) \rceil$-sample of X and, $\bar{Y}$ is a $\lceil n/(p\sqrt{\omega_n}) \rceil$-sample of Y (ie according to the definition of sample, every $\lceil n/(p\sqrt{\omega_n}) \rceil$-th key of X or Y is picked). Let $\bar{X} = \langle \bar{x}_1, \bar{x}_2, \ldots \rangle$ and $\bar{Y} = \langle \bar{y}_1, \bar{y}_2, \ldots \rangle$. For every key of $\bar{X}$ and $\bar{Y}$ information about whether it belongs to sequence X or Y along with its rank ($\text{rank}(\bar{x}_i : X)$ for any $\bar{x}_i \in \bar{X}$, $\text{rank}(\bar{y}_j : Y)$ for any $\bar{y}_j \in \bar{Y}$) in these sequences is also maintained . Sets $\bar{X}$ and $\bar{Y}$ partition X and Y respectively into sets $X_0, X_1, \ldots$ and $Y_0, Y_1, \ldots$ each of cardinality at most $\lceil n/(p\sqrt{\omega_n}) \rceil$.

[**Analysis of Stage 1:** Each processor selects at most $\lceil \sqrt{\omega_n} \rceil$ keys from its local memory to form $\bar{X}$ or $\bar{Y}$, and thus, the time of this stage is bounded above by $\max\{L, O(\sqrt{\omega_n})\}$.]

Stage 2 : Subsequences $\bar{X}$ and $\bar{Y}$ are merged into subsequence $\bar{Z} = \langle \bar{z}_1, \ldots, \bar{z}_l \rangle$, where $l \leq p\lceil \sqrt{\omega_n} \rceil$. For every key $\bar{z}_i \in \bar{Z}$ information about whether it belongs to sequence X or Y along with its rank in that sequence is maintained.

[**Analysis of Stage 2:** The merging operation is realized by employing the BSP bitonic-sort algorithm of Lemma 2. As $\bar{X}$ and $\bar{Y}$ are ordered and stored identically in the processors, one of the two sequences needs to be reversed so that their concatenation would form a bitonic sequence, an operation that requires $\max\{L, O(g\sqrt{\omega_n})\}$ communication time. Bitonic-sort requires total time at most $T_{btn}(|\bar{X}| + |\bar{Y}|, p; t)$, where $|\bar{X}| + |\bar{Y}| = p\lceil \sqrt{\omega_n} \rceil$.

For a key $\bar{z}_i \in \bar{Z}$ its ranks in $\bar{X}$ and $\bar{Y}$ are immediately available.

For rank($\bar{z}_i : \bar{X}$), if $\bar{z}_i \in \bar{X}$, then rank($\bar{z}_i : \bar{X}$) became available in stage 1. Otherwise, $\bar{z}_i \in \bar{Y}$, and rank($\bar{z}_i : \bar{X}$) $= i -$ rank($\bar{z}_i : \bar{Y}$).

Rank rank($\bar{z}_i : \bar{Y}$) is computed analogously. Therefore, the total time for maintaining these ranks is an extra $\max\{L, O(\sqrt{\omega_n})\}$.]

Stage 3 : For every $\bar{z}_i \in \bar{Z}$, ranks rank($\bar{z}_i : X$) and rank($\bar{z}_i : Y$) are computed.

[**Analysis of Stage 3:** A reference to a key in the remainder refers to a key of set $\bar{Z}$. We next show how to compute for key $\bar{z}_i \in \bar{X}$, rank($\bar{z}_i \in X$) and rank($\bar{z}_i \in Y$). The case for $\bar{z}_i \in \bar{Y}$ is handled similarly.

If $\bar{z}_i \in \bar{X}$, then rank($\bar{z}_i : X$) $=$ rank($\bar{x}_{\text{rank}(\bar{z}_i:\bar{X})} : X$) can be computed in constant time using the rank information obtained in stages 1 and 2. In order to find rank($\bar{z}_i : Y$) the following procedure consisting of steps (a) and (b) is executed.

(a) By stage 2, rank($\bar{z}_i : \bar{Y}$) $= a_i$ is available in $O(1)$ time per key $\bar{z}_i$.

(b) A binary search of $\bar{z}_i$ into Y_{a_i} is performed and the result is added to the rank in Y of the first key of Y_{a_i}, which is immediately available, as Y_{a_i} is formed from Y.

We show below how step (b) can be performed in parallel for all $\bar{z}_i$ such that $\bar{z}_i \in \bar{X}$, by providing more details about steps (a) and (b) and consequently breaking step (b) into substeps (b1), (b2), (b31), (b32), (b33) and (b34).

(a) For every key $\bar{z}_i$ stored in some processor, rank($\bar{z}_i : \bar{Y}$) $= a_i$ becomes available at the end of stage 2. The sequence rank($\bar{z}_i : \bar{Y}$), $i = 1, \ldots$, is a non-decreasing sequence. Let a_{l_j} be the rank in $\bar{Y}$ of the leftmost key $\bar{z}_{l_j}$ stored in processor j; $\bar{z}_1$, the leftmost key of $\bar{Z}$, is stored in processor 0, the leftmost processor, and a canonical left-to-right ordering is then defined. Similarly, let a_{r_j} be the rank of the rightmost key $\bar{z}_{r_j}$ stored in processor j.

Step (a) thus requires parallel computation time $\max\{L, O(\sqrt{\omega_n})\}$.

(b1) This step prepares a segmented parallel prefix operation that will be performed in step (b3). The length of the sequence to which the operation will be applied is larger than the number of available processors. The first part of the prefix will involve a sequential scan/prefix on the part of the sequence that is local to a particular processor, followed by a segmented prefix on p values (one per processor). At the completion of the parallel part, the values thus obtained are combined with the values obtained in the sequential scan by

performing an additional scan/prefix operation within a processor. The associative operator is a copy operator that will copy sets $X_0, X_1, \ldots$ and $Y_0, Y_1, \ldots$ as needed to the processors in preparation for a binary search operation into these sets so that the final ranks be determined. Each segment of the prefix operation consists of keys with the same a_i. For every $\bar{z}_i$, a_i is compared to a_{i-1}. The result of the comparison of a_i to a_{i-1} determines whether a new segment starts or not at i. If a segment starts at i, then a variable w_i is set $w_i = 1$, otherwise $w_i = 0$ (for $\bar{z}_1$, it is $w_1 = 1$). Independently of the value of w_i, variable u_i is set to $u_i = a_{i-1}$ to copy the rank of the key to the left of $\bar{z}_i$.

Step (b1) requires computation time $\max\{L, O(\sqrt{\omega_n})\}$ and communication time equal to $\max\{L, O(g)\}$ as the leftmost key $\bar{z}_{l_j}$ of processor j requires a remote memory access.

(b2) Each processor that holds a $w_i = 1$ fetches set Y_{a_i} and stores it to a local variable W_i by allocating the necessary memory space. Otherwise, $W_i = \emptyset$ (empty set/null pointer).

As there are at most $\lceil\sqrt{\omega_n}\rceil$ such w_i variables per processor, an $n/p+O(n/(p\sqrt{\omega_n}))$-relation needs to be routed for communication time $\max\{L, gn/p + O(gn/(p\sqrt{\omega_n}))\}$.

(b3) If at some processor j, $w_{l_j} = 0$, processor j also allocates memory for W_{l_j} even though it may not fetch (yet) $Y_{a_{l_j}}$ (memory copy takes place in (b33)).

A segmented parallel prefix operation is then performed on the sequence W_i; the beginning of a new segment is signified by a $w_i = 1$. The associative operator is a copy operator. A local (sequential) copy only updates pointers i.e. it is a constant time operator; a non-local (cross processor) copy copies memory by performing a communication step.

After the completion of the prefix operation a single copy of Y_{a_i} will exist in any processor that will operate on it and all W_r of that processor with $a_r = a_i$ will point to that single copy. As the segments correspond to sets of keys with the same a_i rank, keys of the same segment will perform a binary search operation on the same set Y_{a_i}. Step (b3) is described in more detail and analyzed by breaking it up similarly into substeps below.

(b31) If $w_{l_j} = 0$, memory is allocated for W_{l_j} that will hold a copy of $Y_{a_{l_j}}$ in processor j, even if $w_{l_j} \neq 1$. Overall time complexity of this step is $\max\{L, O(n/(p\sqrt{\omega_n}))\}$.

(b32) Since the prefix operation is performed on a number of $O(p\sqrt{\omega_n})$ sets W_i, and the number of processors is p, the first step of segmented parallel prefix involves a sequential left-to-right scan/prefix of the W_i's and related pointer updating.

This step requires computation time $\max\{L, O(\sqrt{\omega_n})\}$, as all necessary space has been preallocated in steps (b2) and (b31) and only pointer updating takes place in this step.

At the completion of the scan operation, each processor computes a single partial result in W_{r_j} and sets a variable $V_j = W_{r_j}$ and also initializes $U_j = a_{r_{j-1}}$. The latter step requires $\max\{L, O(1)\}$ time for computation. Note that $a_{r_{j-1}}$ is locally available as u_{l_j}.

(b33) A segmented parallel prefix on the p values V_j is performed. The segments are determined by U_j (a segment starts at j if $U_j \neq a_{r_j}$). The associative operator is a (memory) copy operator. The time complexity of this step is $T_{ppf}^{\lceil n/(p\sqrt{\omega_n})\rceil}(p)$.

At the completion of the segmented parallel prefix, some processors may not have all the segments required for the following step that performs the binary search operation. These are segments of step (b31), where $a_{r_{j-1}} = a_{l_j} \neq a_{r_j}$. In such a case, a copy of $Y_{a_{r_{j-1}}}$ pointed by $W_{a_{r_{j-1}}}$ is stored in processor $j-1$ but processor j does not yet store a copy of $Y_{a_{l_j}} = Y_{a_{r_{j-1}}}$.

Each processor may thus copy a single segment. Detection of this deficit and the required remote fetching of the segment adds $\max\{L, gO(n/(p\sqrt{\omega_n}))\}$ to the communication time required.

The segmented prefix is completed by a sequential scan that combines the results of the local scan of step (b32) to the results of the parallel prefix of this step in time $\max\{L, O(\sqrt{\omega_n})\}$.

(b34) In any processor j, for each one of its required keys $\bar{z}_i$, a binary search on locally available set Y_{a_i} is performed and $\text{rank}(\bar{z}_i : Y)$ becomes available as described in step (b).

The computation time of this step is at most $\max\{L, \sqrt{\omega_n}O(\lg(n/(p\sqrt{\omega_n})))\}$.

]

Stage 4 : Subsequence $\hat{Z} = \langle \hat{z}_1, \hat{z}_2, \ldots \rangle$ is constructed such that $\hat{Z}$ is a $\lceil\sqrt{\omega_n}\rceil$-sample of $\bar{Z} - \{\bar{z}_{|\bar{Z}|}\}$ (ie of $\bar{Z}$ without its last key). For every $\hat{z}_i$, ranks $\text{rank}(\hat{z}_i : X)$ and $\text{rank}(\hat{z}_i : Y)$ become available.

[**Analysis of Stage 4:** Since $|\bar{Z}| \leq p\lceil\sqrt{\omega_n}\rceil$ we have that $\bar{Z} - \{\bar{z}_{|\bar{Z}|}\}$ is of size at most $p\lceil\sqrt{\omega_n}\rceil - 1$ and we thus have that $|\hat{Z}| < p$ and consequently, each processor selects at most one key from its local memory. The time complexity of this step is bounded above by $\max\{L, O(1)\}$. As every key $\hat{z}_i$ of $\hat{Z}$ is also a key of $\bar{Z}$, the rank $\text{rank}(\hat{z}_i : \bar{Z})$ is immediately available. As for every key of $\bar{Z}$ and therefore for key $\hat{z}_i$ of that set $\text{rank}(\hat{z}_i : X)$ is available from stage 3, the

rank computations $\text{rank}(\hat{z}_i : X)$ and $\text{rank}(\hat{z}_i : Y)$ thus require $\max\{L, O(1)\} + \max\{L, O(g)\}$ time.]

Stage 5 : Processor i reads subsequence

$$X'_i = \begin{cases} \langle x_1, \ldots, x_{\text{rank}(\hat{z}_{i+1}:X)} \rangle & \text{if } i = 0, \\ \langle x_{\text{rank}(\hat{z}_i:X)+1}, \ldots, x_{\text{rank}(\hat{z}_{i+1}:X)} \rangle & \text{if } 1 \leq i < |\hat{Z}|, \\ \langle x_{\text{rank}(\hat{z}_i:X)+1}, \ldots, x_m \rangle & \text{if } i = |\hat{Z}|, \end{cases}$$

and

$$Y'_i = \begin{cases} \langle y_1, \ldots, y_{\text{rank}(\hat{z}_{i+1}:Y)} \rangle & \text{if } i = 0, \\ \langle y_{\text{rank}(\hat{z}_i:Y)+1}, \ldots, y_{\text{rank}(\hat{z}_{i+1}:Y)} \rangle & \text{if } 1 \leq i < |\hat{Z}|, \\ \langle y_{\text{rank}(\hat{z}_i:Y)+1}, \ldots, y_k \rangle & \text{if } i = |\hat{Z}|, \end{cases}$$

and subsequently merges them into subsequence Z_i by employing some optimal deterministic sequential merging algorithm [14]. The desired sequence $merge(X, Y)$ is the concatenation of sequences Z_i, for all i, in increasing order of i.

[**Analysis of Stage 5:** By way of stages 1-3, the cardinality of $X'_i \cup Y'_i$, for all i stored in some processor j is bounded above by $\lceil\sqrt{\omega_n}\rceil\lceil n/(p\sqrt{\omega_n})\rceil = n/p + O(n/(p\sqrt{\omega_n}))$. Thus, this stage takes time at most $\max\{L, T^*_{mrg}(n_1, n_2)\}$ and $\max\{L, g(n_1 + n_2)\}$ for computation and communication respectively, where $n_1 + n_2 = n/p + O(n/(p\sqrt{\omega_n}))$.]

end

A proof of correctness follows from the discussion of the algorithm. As far as the running time is concerned, in the statement of Lemma 3 the first term of computation time is contributed by stage 5, the second term by stage 2, the third term by stages 1, 2, and stage 3(a), 3(b1), 3(b32), 3(b33). Note that the rank of an element of $\bar{Z}$ in both X and Y can be computed, within the same time bound in stage 3. The fourth term is contributed by stage 4 and 3(b32), the fourth term by stage 3(b31), and the sixth term by stage 3(b34). For the last term two prefix operations are performed in stage 3(b33) one for computing ranks in X and one for computing the ranks in Y.

Similarly the first term for communication time is contributed by stage 5 and stage 3(b2) (one term for fetching $Y_0, Y_1, \ldots$ and one for fetching $X_0, X_1, \ldots$ segments), the second term by stage 2, the third term by stages 3(b1) and 4, the fourth by stage 2, the fifth by stage

3(b33) (for the two prefix operations also accounted in computation time) and the last term by stage 3(b33). ■

In spite of the apparent complexity of rank computations most of them are not difficult to compute. The non-trivial step of the algorithm is the segmented prefix which is a segmented copy operation. In practice, the sequential cost of this copy operation will be minimal as locality in real processor architectures renders sequential scan operations efficient; the parallel part of the operation is also an ordinary segmented prefix operation. If the keys are non-trivial data types (eg strings of characters, arbitrary-length integers) the cost of this segmented copy becomes negligible compared to the cost of comparisons. We present below an example that shows how the various ranks of keys are computed.

Example. Let us assume that $X = \langle 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31\rangle$, and $Y = \langle 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32\rangle$. Let $p = 4$ and $\omega_n = 4$. As $n = 32 = |X| + |Y|$, then $\bar{X} = \langle 7^{X,1}_{1,4},\ 15^{X,2}_{2,8},\ 23^{X,3}_{3,12},\ 31^{X,4}_{4,16}\rangle$, $\bar{Y} = \langle 8^{Y,1}_{1,4},\ 16^{Y,2}_{2,8},\ 24^{Y,3}_{3,12},\ 32^{Y,4}_{4,16}\rangle$. $\bar{Y}$ splits Y into sets $Y_0 = \langle 2, 4, 6, 8\rangle$, $Y_1 = \langle 10, 12, 14, 16\rangle$ and so on.

For a key $32^{Y,4}_{4,16}$ the two superscripts $Y, 4$ indicate the set, in this case Y, to which key 32 belongs, and the rank of key 32 in $\bar{Y}$. The former piece of information is collected at the end of stage 1 and the latter at the end of stage 2. The first subscript describes the index in the current sequence (ie rank in $\bar{Y}$) and the second the rank in Y.

After stage 2, $\bar{Z} = \langle 7^{X,1}_{1,4}\ , 8^{Y,1}_{2,4}\ , 15^{X,2}_{3,8}\ , 16^{Y,2}_{4,8}\ , 23^{X,3}_{5,12}\ , 24^{Y,3}_{6,12}\ , 31^{X,4}_{7,16}\ , 32^{Y,4}_{8,16}\ , \rangle$, where the first subscript gives the rank of a key in $\bar{Z}$ and the second, the rank in $\bar{X}$ or $\bar{Y}$.

Then, $\text{rank}(\bar{z}_i : \bar{X}) = \langle 1, 2-1 = 1, 2, 4-2 = 2, 3, 6-3 = 3, 4, 8-4 = 4\rangle = \langle 1, 1, 2, 2, 3, 4, 4\rangle$, and $\text{rank}(\bar{z}_i : \bar{Y}) = \langle 1 - 1 = 0, 1, 3 - 2 = 1, 2, 5 - 3 = 2, 3, 7 - 4 = 3, 4\rangle = \langle 0, 1, 1, 2, 2, 3, 3, 4\rangle$ are then computed from the information available to the keys of $\bar{Z}$.

We next show how to compute $\text{rank}(16 : Y)$. As the information for 16 is $16^{Y,2}_{4,8}$, key 16 belongs to Y and its rank in Y is 8, therefore $\text{rank}(16 : Y) = 8$. In order to find $\text{rank}(16 : X)$, we note that $\text{rank}(16 : \bar{Z}) = 4$ (by reading off the first subscript value from the information attached to key 16), and that the fourth entry of $\text{rank}(\bar{z}_i : \bar{X})$ is $\text{rank}(16 : \bar{X}) = a_{\text{rank}(16:\bar{Z})} = a_4 = 2$. A binary search of 16 in X_2, which is $X_2 = \langle 17, 19, 21, 23\rangle$, returns -1, since 16 is less than the leftmosts element 17. We use the convention that a binary search operation on

a sorted sequence of k keys returns an index in the range $\langle -1, 0, \ldots, k-1 \rangle$ that shows the potential insertion position of the searched key.

The return value is then added to the rank of the leftmost element of X_2 in X, that is, to $\text{rank}(17 : X) = 9$, resulting in $\text{rank}(16 : X) = -1 + 9 = 8$. ■

By substituting appropriate values for the parameters in Lemma 3 a multiplicity of bounds on the runtime of algorithm MERGE can be obtained. The following is one such bound.

Corollary 1 *There exists a p-processor BSP algorithm for merging two non-decreasing sequences of lengths m and k, that for all integers $n = m+k$ and p, such that $p = O(n/\lg n)$ and $L = O(n/(p \log_t p)$, requires time $O(n/p)$ and $O(gn/p)$ for computation and communication respectively, where $t = \min\{O(L/g), p\}$.*

Proof: Substitute in Lemma 3 for T_{btn} the corresponding expression in Lemma 2 and $\min\{O(L/g), p\}$ for t. For T_{ppf} apply Lemma 1 with $t = \min\{O(L/g), p\}$. ■

We note that stage 4 could have been executed before stage 3. In such a case only ranks of keys of set $\hat{Z}$ instead of $\bar{Z}$ ought to have been computed.

4 Conclusions

Sub-optimal and optimal merging algorithms on the BSP model were presented in this work. Our algorithms were described in an architecture independent way, suitable for coarse-grained parallel computer models. A comparison of our algorithm to say a PRAM algorithm for the same problem (such as the one in [12]) indicates the issues involved with algorithm design on realistic parallel computer models such as the BSP. Processor size is in our case independent of problem size, a cost is assigned to synchronization operations and to communication and the resulting algorithm exhibits efficiency on coarse-grained and fine-grain architectures as well. If the algorithm is analyzed under the BSP cost model, then its performance on various parallel platforms can be estimated provided that the BSP parameters of such platforms can be made available (say, by measuring them).

5 Acknowledgements

Part of this work was initiated while both authors were with the Oxford University Computing Laboratory, The University of Oxford, Oxford, UK. The work of the first author there was supported in part by EPSRC under grant GR/K16999 and that of the second author was supported in part by a Bodossaki Foundation Graduate Scholarship. The work of the first author was subsequently supported in part by NJIT SBR grant 421350.

References

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

[2] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C-27(1):84-87, January 1978.

[3] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers,* C-38(11):1526-1538, 1989.

[4] A. V. Gerbessiotis and C. J. Siniolakis. A. V. Gerbessiotis, C. J. Siniolakis. BSP Sorting: An experimental Study. Submitted for publication. `http://www.cis.njit.edu/~alexg/pubs/exsort.ps`

[5] A. V. Gerbessiotis and C. J. Siniolakis. An Experimental Study of BSP Sorting Algorithms. In Proceedings of 6th EuroMicro Workshop on Parallel and Distributed Processing, Madrid, Spain, January, IEEE Computer Societty Press, 1998.

[6] 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*, June 1996.

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

[8] 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.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Efficient deterministic sorting on the BSP model. Parallel Processing Letters, Vol 9 No 1 (1999), pp 69-79, World Scientific Publishing Company.

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

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

[12] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters,* 33:181-185, 1989.

[13] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters,* 58:303-310, 1996.

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

[15] R. E. Ladner and M. J. Fisher. Parallel-prefix computation. *Journal of the ACM,* 27(4):831-838, 1980.

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

[17] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362-372, 1992.

[18] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms,* 2:88-102, 1981.

[19] K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *5-th EuroMicro Workshop on Parallel and Distributed Processing*, 1997.

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

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