Programming Research Group

PRIMITIVE OPERATIONS ON THE BSP MODEL

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

PRG-TR-23-96

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

Primitive Operations on the BSP Model*

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis

Computing Laboratory,
Oxford University,
Wolfson Building,
Oxford, OX1 3QD,
UK.

October 1996

Abstract

The design of a complex algorithm relies heavily on a set of primitive operations and the instruments required to compile these operations into an algorithm. In this work, we examine some of these basic primitive operations and present algorithms that are suitable for the Bulk-Synchronous Parallel model. In particular, we consider algorithms for the following primitive operations: broadcasting, parallel-prefix, merging, generalized and integer sorting.

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 software to be written once and for all that can be migrated efficiently among a variety of parallel machines.

Keywords. BSP algorithms - broadcasting - parallel-prefix - merging - sorting - integer sorting

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

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 some of these basic primitive operations and present algorithms for these operations that are suitable for the Bulk-Synchronous Parallel model. In particular, we consider algorithms for the following primitive operations:

(1) n-word broadcasting.

(2) n-word parallel-prefix.

(3) Merging.

(4) Generalized sorting.

(5) Integer sorting.

1.1 The Bulk-Synchronous Parallel Model

The introduction of the von-Neumann model for sequential computing in the mid 1940s provided the hardware industry with a stable model whose universal acceptance encouraged technological innovations and long-term investments that led to the explosive growth of sequential computing since then. The software industry followed and subsequently led this growth without fearing that program development would become obsolete with the next round of hardware innovations.

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.

We argue that 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 been for sequential computing. The introduction of the Bulk-Synchronous Parallel model 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.

The *Bulk-Synchronous Parallel (BSP)* model of computation has been proposed by L.G. Valiant [22, 23], as a unified framework for the design, analysis, and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a precise way. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems (a global clock is utilized to achieve synchronization on every instruction), and (ii) fully asynchronous systems (synchronization is realized pairwise whenever communication is required). Processing units in the BSP model advance jointly through the program, with the required remote communication occurring between supersteps. The BSP computer as described in [9, 15, 16, 17, 22, 23, 24] consists of the following three components.

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

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

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

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

(1) p - the number of processing units.

(2) g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. More formally, g is related to the time required to realize h-relations in continuous message usage in gh time units [16].

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

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each processing unit is given a task to perform using data already available there locally before the start of the superstep. The task may include (i) computation steps on values held locally at the start of the superstep, (ii) message transmissions, and (iii) message receipts. A superstep can complete at any time after the first L time units. It should be noted that, although the model stresses global barrier mode synchronization, pairs of processing units may synchronize pairwise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules [9].

The time complexity of a superstep $\mathcal{S}$ in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x + gh\}$ time units, 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 [9], and in order to simplify the description of the performance of our algorithms, we assume that each superstep is either pure computation or pure communication. This way, the two kinds of costs are easily distinguished.

The minimum superstep size is bounded below by the frequency with which barrier synchronization can be realized. As it was noted in [9], any particular BSP machine may impose a lower bound on L on a twofold basis.

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

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

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

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 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 2 and e respectively. A dominant term in the BSP merging and sorting algorithms to be claimed later is contributed by sequential merging performed independently at each processor. We charge $T^*_{mrg}(m, k)$ time units for sequentially merging two non-decreasing sequences X and Y, of sizes m and k respectively. We also charge T^*_{srt} time units for sorting n elements sequentially. For other operations we charge $c_\otimes$ time units for operations over the associative operator $\otimes$ in parallel-prefix computations and unit cost for each comparison performed.

2 Broadcasting and Parallel-Prefix

In this section we describe and analyze BSP algorithms for two fundamental operations, namely broadcasting and parallel-prefix. These primitive operations are auxiliary routines for the merging and sorting algorithms described in later sections.

2.1 Broadcasting

The *broadcast* operation is a fundamental primitive in many parallel applications. In this section we derive an efficient BSP algorithm for a generalization of the single-item broadcast operation.

Lemma 1 *There exists a p-processor BSP algorithm for broadcasting an n-word vector, that for any integer t in $[2, \ldots, p]$, requires time at most*[1]

$$M_{brd}^{n}(p) = 2\max\left\{L, g(p-1)\left\lceil\frac{n}{p}\right\rceil\right\} \cdot [n \geq p] + (2h-1)\max\{L, gt\lceil n/h\rceil\} \cdot [n < p],$$

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 $T_{brd}^{n}(p) = M_{brd}^{n}(p)$.

Proof Consider an n-word vector $X = (x_1, \ldots, x_n)$. Without loss of generality we assume that X resides in the local memory of processor p_0. The procedure to manage the broadcast operation follows. We distinguish two cases.

Case 1: If $n \geq p$, then processor p_0 sends $\lceil n/p\rceil$-word vector $(x_{(j-1)\lceil n/p\rceil+1}, \ldots, x_{j\lceil n/p\rceil})$ to processor p_{j-1}, for all j in $[1, \ldots, p]$. Subsequently, each processor p_{j-1}, sends the vector it received in the previous step to all other processors. The cost of this operation is at most $2\max\{L, g(p-1)\lceil n/p\rceil\}$.

Case 2: If $n < p$, then processor p_0 employs a pipelined t-ary tree, consisting of p nodes, for some appropriate t. The depth of the tree is $h = \lceil\log_t((t-1)p+1)\rceil - 1$. In step s, processor p_{j-1}, for all j in $[1, \ldots, t^{s-1}+1]$, sends t copies of the $(s - \lceil(j-1)/t\rceil)$-th component of size $\lceil n/h\rceil$ of X, which is vector $(x_{(s-\lceil(j-1)/t\rceil)\lceil n/h\rceil+1}, \ldots, x_{(s-\lceil(j-1)/t\rceil+1)\lceil n/h\rceil})$, to processors $\langle p_{(j-1)t+1}, \ldots, p_{jt}\rangle$. This process completes within $h + \lceil n/\lceil n/h\rceil\rceil - 1$ steps.

The lemma follows. ■

A similar result has been shown in [11] and is given in the following lemma.

Lemma 2 *There exists a p-processor BSP algorithm for broadcasting an n-word vector, that for any integer t in $[2, \ldots, p]$, requires time at most*

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

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 $T_{brd}^{n}(p) = M_{brd}^{n}(p)$.

2.2 Parallel-Prefix

The *parallel-prefix* operation [13, 14] is a powerful primitive operation with many parallel applications [4].

Definition 1 *Let $\langle X_1, X_2, \ldots, X_p\rangle$ be a sequence of p vectors each of size n, such that $X_i = (x_{1,i}, \ldots x_{n,i})$, i in $[1, \ldots, p]$. 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 $X_i \otimes X_{i+1} = (x_{1,i} \otimes x_{1,i+1}, \ldots, x_{n,i} \otimes x_{n,i+1})$, i in $[1, \ldots, p-1]$.*

An optimal linear-time sequential algorithm exists for the prefix problem. In the remainder of this section we present a non-trivial BSP algorithm.

Lemma 3 *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 t in $[2, \ldots, p]$, requires time at most*

[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.

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

and

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

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* $T^n_{ppf}(p, \oplus) = C^n_{ppf}(p, \oplus) + M^n_{ppf}(p, \oplus)$.

Proof The lemma follows by way of arguments similar to that of lemma 2, the difference being that the t-ary tree utilized in the analysis of case 2 consists of p leaf nodes and is of depth $h = \lceil \log_t p \rceil$. ∎

The following result has also been shown in [11].

Lemma 4 *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* t *in* $[2, \ldots, p]$*, 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* $T^n_{ppf}(p, \oplus) = C^n_{ppf}(p, \oplus) + M^n_{ppf}(p, \oplus)$.

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 same notational convention applies to all pipelined operations.

2.2.1 Segmented Parallel-Prefix

The *segmented parallel-prefix* operation [14] 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 [14], to solve the segmented prefix problem as the standard prefix problem, we first concatenate the input sets into a single set, inserting a *barrier* at the start of each set of data. We next define 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,

$$T^n_{seg\_ppf}(p, \bar{\otimes}) = 2T^n_{ppf}(p, \bar{\otimes}).$$

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 elements 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. Without loss of generality, we assume that all the elements appearing in sequence X are distinct, i.e., if the keys are not distinct we can make them so by appending to them their index. We consider, in this section, the problem of *merging* these two sequences into a non-decreasing sequence - say, $Z = \langle z_1, \ldots, z_{m+k} \rangle$.

Simple linear-time sequential algorithms to solve the merging problem are discussed in [5, 12]. 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 an element x in sequence X, denoted by $\mathrm{rank}(x : X)$, is the number of elements of X that are less than or equal to x. *Ranking* a sequence $X = \langle x_1, \ldots, x_m \rangle$ in a sequence Y amounts to computing the integer array $\mathrm{rank}(X : Y) = \langle r_1, \ldots, r_m \rangle$, where $r_i = \mathrm{rank}(x_i : Y)$, for all i in $[1, \ldots, m]$. Sequence $\bar{X}$ is a c-sample of sequence X, denoted by $\mathrm{sample}_c(X)$, if $\bar{X}$ consists of every c-th element of X, i.e., $\bar{X} = \mathrm{sample}_c(X) = \langle x_c, \ldots, x_{ic}, \ldots, x_{\lfloor m/c \rfloor c} \rangle$, for all i in $[1, \ldots, \lfloor m/c \rfloor]$.

3.1 A BSP Adaptation of Batcher's Bitonic-Sort Algorithm

In this section we present a BSP adaptation of Batcher's bitonic-sort algorithm [2]. Let $V = \langle v_1, \ldots, v_n \rangle$ be a sequence of elements 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 [2] 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 [2] by providing an efficient BSP implementation of Batcher's bitonic-sort algorithm.

Lemma 5 *There exists a p-processor BSP algorithm for merging two non-decreasing sequences X and Y, of sizes m and k respectively, that for all integers $n = m + k$, p and t, such that $n \geq p$ and t in $[2, \ldots, p]$, requires time at most*

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

and

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

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

Proof The input is assumed to be evenly and orderly distributed among the p processors, i.e., element x_i in X, for all i in $[1, \ldots, m]$, resides in the local memory of processor $p_{\lfloor i/q \rfloor}$ at position $i \bmod q$ and element y_j in Y, for all j in $[1, \ldots, k]$, resides in the local memory of processor $p_{\lfloor (m+k-j+1)/q \rfloor}$ at position $(m+k-j+1) \bmod q$, where $q = \lceil (m+k)/p \rceil$. This distribution of elements to processors results in a sorted sequence for each processor except possibly for processor $p_{\lfloor m/q \rfloor}$ 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 elements drawn from a linearly ordered set, such that $w_{i,1} \leq \ldots \leq w_{i,l}$, for all i in $[1, \ldots, p]$, 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 size l. We also replace each comparator by a sequential merging algorithm that receives as input two non-decreasing sequences each of size l, merges them, and outputs separately the top and bottom halves of the merged sequences. As shown in [3, 12] 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 [11], by combining $\lceil \lg t \rceil$ consecutive levels of comparators of the bitonic-sort network into a single level, for some integer t in $[2, \ldots, p]$. 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. ■

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 by concurrently merging groups of pairs of subsequences. The basic algorithmic idea derives from a merging algorithm of [20] as adapted in [10].

Lemma 6 *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 sizes m and k respectively, that for all integers $n = m + k$ and p, such that $n/p = \omega_n \lg n$, requires time at most*

$$\begin{aligned} C_{mrg}(m,k,p) &= \max\{L, T^*_{mrg}(n_1,n_2)\} + C_{btn}(\frac{p\lceil\sqrt{\omega_n}\rceil}{2}, \frac{p\lceil\sqrt{\omega_n}\rceil}{2}) + 3\max\{L, O(\sqrt{\omega_n})\} + \\ & \quad 2\max\{L, O(1)\} + \max\left\{L, O\left(\lg\frac{n}{p\sqrt{\omega_n}}\right)\right\} + 2C^2_{ppf}(p) \end{aligned}$$

and

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

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 $T_{mrg}(m,k,p) = C_{mrg}(m,k,p) + M_{mrg}(m,k,p)$.

Proof The input is assumed to be evenly and orderly distributed among the p processors, i.e., element x_i in X, for all i in $[1,\ldots,m]$, resides in the local memory of processor $p_{\lfloor i/q\rfloor}$ at position $i \bmod q$ and element y_j in Y, for all j in $[1,\ldots,k]$, resides in the local memory of processor $p_{\lfloor (m+j)/q\rfloor}$ at position $(m+j) \bmod q$, where $q = \lceil (m+k)/p\rceil$. Moreover, it is assumed that the keys are distinct since they can always be made so by, for example, appending to them the index of the memory location in which they are stored. 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, similarly, $\bar{Y}$ is a $\lceil n/(p\sqrt{\omega_n})\rceil$-sample of Y. For each element in $\bar{X}$ and $\bar{Y}$ information about whether it belongs to sequence X or Y along with its rank in that sequence is also maintained.

[**Analysis of Stage 1:** Each processor selects at most $\lceil\sqrt{\omega_n}\rceil$ elements from its local memory, and therefore, the time complexity 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 $\bar{Z}$. For each element 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 5, for a total time of at most $T_{btn}(|\bar{X}|, |\bar{Y}|, p)$, where $|\bar{X}| + |\bar{Y}| = p\lceil\sqrt{\omega_n}\rceil$.]

Stage 3 :

Subsequence $\hat{Z} = \langle \hat{z}_1, \ldots, \hat{z}_{|\hat{Z}|}\rangle$ is constructed such that $\hat{Z}$ is a $\lceil\sqrt{\omega_n}\rceil$-sample of $\bar{Z} - \{\bar{z}_{|\bar{Z}|}\}$.

[**Analysis of Stage 3:** Since $|\bar{Z}| \le p\lceil\sqrt{\omega_n}\rceil$ we have that $|\hat{Z}| < p$, and therefore, each processor selects at most one element from its local memory. The time complexity of this stage is bounded above by $\max\{L, O(1)\}$.]

Stage 4 :

Let $\hat{X} = \bar{X} \cap \hat{Z}$ and $\hat{Y} = \bar{Y} \cap \hat{Z}$. Integer arrays $R_{\hat{X}:X} = \text{rank}(\hat{X} : X)$, $R_{\hat{X}:Y} = \text{rank}(\hat{X} : Y)$, $R_{\hat{Y}:X} = \text{rank}(\hat{Y} : X)$ and $R_{\hat{Y}:Y} = \text{rank}(\hat{Y} : Y)$ are constructed. As a consequence, integer arrays $R_{\hat{Z}:X} = \text{rank}(\hat{Z} : X)$ and $R_{\hat{Z}:Y} = \text{rank}(\hat{Z} : Y)$ are obtained.

[**Analysis of Stage 4:** Integer arrays $R_{\hat{X}:X}$ and $R_{\hat{Y}:Y}$ can be generated in $\max\{L, O(1)\}$ time since $\hat{X} \subset \bar{X} \subset X$ and $\hat{Y} \subset \bar{Y} \subset Y$.

The construction of integer arrays $R_{\hat{X}:Y}$ and $R_{\hat{Y}:X}$ is rather more complicated. In the remainder of the analysis of this stage we outline how to generate $R_{\hat{X}:Y}$. Integer array $R_{\hat{Y}:X}$ is obtained analogously. To simplify the exposition, it is assumed that sequences $\hat{X}$ and $\hat{Y}$ have been padded with elements $-\infty$ and $+\infty$. This assumption can be removed without affecting the asymptotic performance of our algorithm.

Let $\hat{x}$ be an element in $\hat{X}$, and let $\bar{y}_l(\hat{x})$ and $\bar{y}_r(\hat{x})$ be the two consecutive elements in $\bar{Y}$, such that $\bar{y}_l(\hat{x}) \leq \hat{x} \leq \bar{y}_r(\hat{x})$. In order to determine $\text{rank}(\hat{x} : Y)$ it suffices to perform a binary search in subsequence $\langle y_{\text{rank}(\bar{y}_l(\hat{x}):Y)}, \ldots, y_{\text{rank}(\bar{y}_r(\hat{x}):Y)} \rangle$.

A naive implementation of this step would require a concurrent read BSP model to achieve optimality. Along the lines of [10], concurrency is eliminated by providing each processor with a private copy of the sequence in which it carries out the search. To this end, for each element $\bar{y} \in \bar{Y}$ the two consecutive elements $\hat{x}_l(\bar{y})$ and $\hat{x}_r(\bar{y})$ in $\hat{X}$ are computed, such that $\hat{x}_l(\bar{y}) \leq \bar{y} \leq \hat{x}_r(\bar{y})$. The computation of these two elements, for each element $\bar{y}$ in $\bar{Y}$, is realized by two distinct parallel-prefix operations on subsequence $\bar{Z}$. The first parallel-prefix operation is from left to right of sequence $\bar{Z}$ and computes for each element $\bar{y}$ in $\bar{Y}$ element $\hat{x}_l(\bar{y})$. Analogously, the second parallel-prefix operation is from right to left of sequence $\bar{Z}$ and computes for each element $\bar{y}$ in $\bar{Y}$ element $\hat{x}_r(\bar{y})$. The time complexity of this step is at most $2T_{ppf}(p) + 2\max\{L, O(\sqrt{\omega_n})\}$.

For each pair $(\bar{y}_i, \bar{y}_j)$, $i < j$, of consecutive elements in $\bar{Y}$, subsequence $\langle y_{\text{rank}(\bar{y}_i:Y)}, \ldots, y_{\text{rank}(\bar{y}_j:Y)} \rangle$ is broadcast to the consecutive block of processors maintaining elements $\hat{x}_r(\bar{y}_i)$ through $\hat{x}_l(\bar{y}_j)$ of $\bar{y}_j$. By way of stages 1-3, no processor broadcasts more than $\lceil n/p \rceil$ elements, and no processor receives more than $\lceil n/(p\lceil\sqrt{\omega_n}\rceil)\rceil$ elements. Therefore, since different broadcast operations can be realized on disjoint sets of processors, the time complexity of this stage is bounded above by $T_{brd}^{\lceil n/p \rceil}(p)$.

Integer array $R_{\hat{X}:Y}$ is generated by having each processor perform a binary search on the subsequence of size at most $\lceil n/(p\sqrt{\omega_n})\rceil$ it received in the previous step. This step takes time $\max\{L, O(\lg(n/(p\omega_n)))\}$. Integer array $R_{\hat{Y}:X}$ is computed similarly. We note that integer arrays $R_{\hat{X}:Y}$ and $R_{\hat{Y}:X}$ can be generated simultaneously. Therefore, the overall time complexity of this stage is bounded above by $2T_{ppf}^2(p) + T_{brd}^{\lceil n/p \rceil}(p) + 2\max\{L, O(\sqrt{\omega_n})\} + \max\{L, O(\lg(n/(p\sqrt{\omega_n})))\}$.]

Stage 5 :
Each processor p_i, for all i in $[0, \ldots, |\hat{Z}|]$, reads subsequences

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

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 merging algorithm [5, 12]. The desired sequence is the concatenation of sequences Z_i, for all i in $[0, \ldots, |\hat{Z}|]$, in increasing order of i. Note that by way of stage 3 it is that $|\hat{Z}| < p$.

[**Analysis of Stage 5:** Since integer arrays $R_{\hat{Z}:X}$ and $R_{\hat{Z}:Y}$ are available, the determination of $\text{rank}(\hat{z}_i : X)$ and $\text{rank}(\hat{z}_i : Y)$, for all i in $[1, \ldots, |\hat{Z}|$, can be performed, by realizing a constant number of local operations and a constant number of message exchanges, in time $\max\{L, O(1)\}$ and $\max\{L, O(g)\}$ for computation and communication respectively.

By way of stages 1-3, the size of $X_i \cup Y_i$, for all i in $[0, \ldots, |\hat{Z}|]$, 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

The lemma follows. ■

By substituting appropriate values for the parameters in lemma 6 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 sizes 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 6 for T_{btn} the corresponding expression in lemma 5 and $\min\{O(L/g), p\}$ for t. For T_{brd} and T_{ppf} apply lemmas 2 and 4 respectively, and let $t = \min\{O(L/g), p\}$ in these lemmas. ■

4 Sorting

Let $X = \langle x_1, \ldots, x_m \rangle$ be a sequence whose elements 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. Without loss of generality, we assume that all the elements appearing in sequence X are distinct, i.e., if the keys are not distinct we can make them so by appending to them their index. We consider, in this section, the problem of *sorting* a sequence X into a non-decreasing sequence - say $Z = \langle x_{\pi(1)}, \ldots, x_{\pi(n)} \rangle$.

Simple $O(n \lg n)$ time sequential sorting algorithms are well known [5, 12]. In this section, we provide simple BSP sorting algorithms. A more detailed account of the sorting problem on the BSP model can be found in [1, 9, 6, 8, 21].

4.1 Sorting with $n/\lg n$ Processors

In this section, we provide a simple BSP solution to the problem of sorting a sequence X that is based on the technique of merge-sort [5, 12] and derives from algorithm MERGE of section 3.2.

Lemma 7 *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 sorting a sequences X of size n, that for all integers n and p, such that $n/p = \omega_n \lg n$, requires time at most*

$$C_{srt}(n,p) = \max\left\{L, T^*_{srt}\left(\left\lceil\frac{n}{p}\right\rceil\right)\right\} + \sum_{i=1}^{\lceil \lg p\rceil}\left(\left\lfloor\frac{p}{2^i}\right\rfloor C_{mrg}\left(2^{i-1}\left\lceil\frac{n}{p}\right\rceil, 2^{i-1}\left\lceil\frac{n}{p}\right\rceil, 2^i\right)\right)$$

and

$$M_{srt}(n,p) = \sum_{i=1}^{\lceil \lg p\rceil}\left(\left\lfloor\frac{p}{2^i}\right\rfloor M_{mrg}\left(2^{i-1}\left\lceil\frac{n}{p}\right\rceil, 2^{i-1}\left\lceil\frac{n}{p}\right\rceil, 2^i\right)\right),$$

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

Proof The sorting algorithm is a BSP adaptation of sequential merge-sort [5, 12]. Initially, each processor, in parallel with all other processors, sorts the elements in its possession by some optimal deterministic sorting algorithm [5, 12]. When this step is executed, each processor holds at most $\lceil n/p\rceil$ elements, and therefore, this step takes time $\max\{L, T^*_{srt}(\lceil n/p\rceil)\}$. Subsequently, processors merge the resulting p sequences via $\lceil \lg p\rceil$ merges of pairs of sequences of total size n, and the lemma follows. ■

By substituting appropriate values for the parameters in lemma 7 a multiplicity of bounds on the runtime of BSP sorting can be obtained. The following is one such bound.

Corollary 2 *There exists a p-processor BSP algorithm for sorting a sequence of size n, that for all integers n and p, such that $p = O(n/\lg n)$ and $L = O(n/(p \log_t p))$, requires time $O(n \lg n/p)$ and $O(gn \lg n/p)$ for computation and communication respectively, where $t = \min\{O(L/g), p\}$.*

Proof The proof is similar to that of lemma 6. ■

4.2 Sorting with n^2 Processors

In this section we present a BSP adaptation of a sorting algorithm by Nassimi and Sahni [18]. The algorithm is realized by employing exclusively broadcasting, parallel-prefix, regular routing, and local operations.

Lemma 8 *There exists a p-processor BSP algorithm for sorting a sequence X of size n, that for all integers n and p, such that $p = n^2$, requires time at most*

$$C_{quad\_srt}(n,p) = \max\{L, O(1)\} + C_{ppf}(\sqrt{p})$$

$$M_{quad\_srt}(n,p) = \max\{L, O(g)\} + 2M_{brd}(\sqrt{p}) + M_{ppf}(\sqrt{p}),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{quad\_srt}(n,p) = C_{quad\_srt}(n,p) + M_{quad\_srt}(n,p)$.

Proof Consider input sequence $X = \langle x_1, \ldots, x_n \rangle$ of size n and n^2 processors. We number processors in a mesh-like fashion, so that we can refer to processor at row i and column j, as $p_{i,j}$, i, j in $[1, \ldots, n]$. Input elements are assumed to be distributed among the processors in the first row, i.e., element x_j resides in the local memory of processor $p_{1,j}$. By executing n disjoint broadcast operations, processor $p_{1,j}$, for all j in $[1, \ldots, n]$, broadcasts element x_j to all processors of column j. Similarly, processor $p_{i,i}$, for all i in $[1, \ldots, n]$, broadcasts element x_i to all processors of row i. In a single superstep, processor $p_{i,j}$, for all i, j in $[1, \ldots, n]$, compares x_i and x_j, and subsequently, by executing n disjoint parallel-prefix sum operations determines the rank $\mathrm{rank}(x_i : X)$, for all i in $[1, \ldots, n]$. The sorting operation is completed by routing each element to its final destination. The overall time complexity of the algorithm can be easily seen to be at most $\max\{L, O(1)\} + \max\{L, O(g)\} + 2T_{brd}(\sqrt{p}) + T_{ppf}(\sqrt{p})$, and the lemma follows. ■

We therefore obtain the following result.

Corollary 3 *There exists a p-processor BSP algorithm for sorting a sequence of size n, that for all integers n and p, such that $p = n^2$, requires time $O(L \log_t p)$ and $O(L \log_t p)$ for computation and communication respectively, where $t = \min\{O(L/g), p\}$.*

Proof Substitute in lemma 8 for T_{brd} and T_{ppf} the corresponding expressions in lemmas 2 and 4, and let $t = \min\{O(L/g), p\}$ in these lemmas. ■

4.3 Sorting with $n^{1+\epsilon}$ Processors

In this section we present a randomized BSP generalization of the sorting algorithm of section 4.2. The algorithm is realized by employing exclusively broadcasting, parallel-prefix, regular routing, and local operations.

Lemma 9 *There exists a randomized p-processor BSP algorithm for sorting a sequence X of size n, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 1$, that for all integers n and p, such that $p = n^{1+\epsilon}$, for any constant $0 < \epsilon < 1$, requires time at most*

$$C_{poly\_srt}(n,p) = O\left(\frac{1}{\epsilon}\left(\max\{L, O(1)\} + C_{ppf}(p)\right)\right)$$

$$M_{poly\_srt}(n,p) = O\left(\frac{1}{\epsilon}\left(\max\{L, O(g)\} + M_{ppf}(p)\right)\right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{poly\_srt}(n,p) = C_{poly\_srt}(n,p) + M_{poly\_srt}(n,p)$.

Proof Refer to [7]. ■

We thus derive the following corollary.

Corollary 4 *There exists a* p*-processor BSP algorithm for sorting a sequence of size* n*, that for all integers* n *and* p*, such that* $p = n^{1+\epsilon}$*, for any constant* $0 < \epsilon < 1$*, requires time* $O((1/\epsilon)L\log_t p)$ *and* $O((1/\epsilon)L\log_t p)$ *for computation and communication respectively, where* $t = \min\{O(L/g), p\}$.

Proof Substitute in lemma 9 for T_{brd} and T_{ppf} the corresponding expressions in lemmas 2 and 4, and let $t = \min\{O(L/g), p\}$ in these lemmas. ■

5 Integer Sorting

In this section we focus our attention on a BSP algorithm for stable sorting integers in an arbitrary range. This is an important special case of sorting with a significant number of applications [12]. Simple sequential algorithms to solve the integer problem are well known [5, 12]. In the following section, we provide a BSP solution to the integer sorting problem that is based on *radix-sort* techniques [12].

5.1 Parallel Radix-Sort

In this section we present a radix-sort based integer sorting algorithm that extends an idea of [19]. The algorithm is realized by employing exclusively parallel-prefix, regular routing, and local operations.

Lemma 10 *There exists a* p*-processor BSP algorithm for stable sorting a sequence* X *consisting of* n *integers in the range* $[0,\ldots,R-1]$*, that for all integers* n*,* p *and* t*, such that* $n \geq p$ *and* t *in* $[2,\ldots,p]$*, requires time at most*

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

and

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

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* $T_{rdx\_srt}(n,p) = C_{rdx\_srt}(n,p) + M_{rdx\_srt}(n,p)$.

Proof Let $m = \max\{\lceil n/p\rceil, t\lceil\log_t p\rceil\}$. We show how to stably sort n keys in the range $[0,\ldots,m-1]$, for any positive integers n and p such that $n \geq p$. We can then apply radix-sort techniques [12] to prove that we can sort n keys in the range $[0,\ldots,R-1]$, within $r = \lceil\lg R/\lg m\rceil$ iterations of the basic procedure. The input is assumed to be evenly and orderly distributed among the p processors, i.e., element x_i in X, for all i in $[1,\ldots,n]$ resides in the local memory of processor $p_{\lfloor i/q\rfloor}$ at position $i \bmod q$, where $q = \lceil n/p\rceil$.

Program RADIX_SORT (X)

begin
Stage 1 :
Every processor, in parallel with all the other processors, counts the number of its elements that fall into the m intervals induced by $[0, \ldots, m-1]$. Let $S_{k,i}$ denote the number of keys with value k in processor i, k in $[0, \ldots, m-1]$ and i in $[0, \ldots, p-1]$.
[**Analysis of Stage 1:** When this step is executed, every processor holds at most $\lceil n/p \rceil$ keys. The cost of this stage is therefore bounded above by $\max\{L, O(m)\}$.]

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

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

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

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

By applying radix-sort techniques [12] we obtain that n keys in the range $[0, \ldots, R-1]$, can be sorted within $r = \lceil \lg R / \lg \max\{\lceil n/p \rceil, t\lceil \log_t p \rceil\} \rceil$ iterations of procedure RADIX_SORT, and the lemma follows. ■

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

We thus derive the following corollary.

Corollary 5 *There exists a p-processor BSP algorithm for sorting a sequence of n integers in the range $[0, \ldots, n-1]$, that for all integers n and p, such that $p = n^{1-\epsilon}$ and $L = O(n/(p\log_t p))$, requires time $O((1/\epsilon)n/p)$ and $O(g(1/\epsilon)n/p)$ for computation and communication respectively, where $t = \min\{O(L/g), p\}$.*

Proof Substitute in lemma 10 for T_{ppf} the corresponding expression in lemma 4 and $\min\{O(L/g), p\}$ for t. ■

References

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

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