Communication-Efficient Parallel Sorting

(Preliminary Version)

MICHAEL T. GOODRICH*
Johns Hopkins Univ.
goodrich@cs.jhu.edu

Abstract

We study the problem of sorting n numbers on a p-processor bulk-synchronous parallel (BSP) computer, which is a parallel multicomputer that allows for general processor-to-processor communication rounds provided each processor sends and receives at most h items in any round. We provide parallel sorting methods that use internal computation time that is $O(\frac{n \log n}{p})$ and a number of communication rounds that is $O(\frac{\log n}{\log(h+1)})$ for $h = \Theta(n/p)$. The internal computation bound is optimal for any comparison-based sorting algorithm. Moreover, the number of communication rounds is bounded by a constant for the (practical) situations when $p \leq n^{1-1/c}$ for a constant $c \geq 1$. In fact, we show that our bound on the number of communication rounds is asymptotically optimal for the full range of values for p, for we show that just computing the "or" of n bits distributed evenly to the first $O(n/h)$ of an arbitrary number of processors in a BSP computer requires $\Omega(\log n/\log(h+1))$ communication rounds.

1 Introduction

Most of the research on parallel algorithm design in the 1970's and 1980's was focused on fine-grain massively-parallel models of computation (e.g., see [4, 7, 22, 24, 28, 37]), where the ratio of memory to processors is fairly small (typically $O(1)$), and this focus was independent of whether the model of computation was a parallel random-access machine (PRAM) or a network model, such as a mesh-of-processors. But, as more and more parallel computer systems are being built, researchers are realizing that processor-to-processor communication is a prime bottleneck in parallel computing [2, 6, 12, 26, 30, 31, 34, 41, 40]. The real potential of parallel computation, therefore, will most likely only be realized for coarse-to-medium-grain parallel systems, where the ratio of memory to processors is non-constant, for such systems allow an algorithm designer to balance communication latency with internal computation time. Indeed, this realization has given rise to several new computation models for parallel algorithm design, which all use what Valiant [40] calls "bulk synchronous" processing. In such a model an input of size n is distributed evenly across a p-processor parallel computer. In a single computation *round* (which Valiant calls a *superstep*) each processor may send and receive h messages and then perform an internal computation on its internal memory cells using the messages it has just received. To avoid any conflicts that might be caused by asynchronies in the network (whose topology is left undefined) the messages sent out in a round t by some processor cannot depend upon any messages that processor receives in round t (but, of course, they may depend upon messages received in round $t-1$). We refer to this model of computation as the Bulk-Synchronous Parallel (BSP) model.

*This research supported by the NSF under Grants IRI-9116843 and CCR-9300079, and by ARO under grant DAAH04-96-1-0013.

STOC'96, Philadelphia PA,USA
© 1996 ACM 0-89791-785-5/96/05..$3.50

1.1 The BSP Model

As with the PRAM family of computer models[1], the BSP model is distinguished by the broadcast and combining abilities of the network connecting the various processors. In the weakest version, which is the only version Valiant [40] considers, the network may not duplicate nor combine messages, but instead may only realize h-relations between the processors. We call this the *EREW BSP* model, noting that it is essentially the same as a model Valiant elsewhere [41] calls the XPRAM and one that Gibbons [19] calls the EREW phase-PRAM. It is also the communication structure assumed by the LogP model [12, 25], which is the same as the BSP model except that the LogP model does not explicitly require bulk-synchronous processing.

But it is also natural to allow for a slightly more powerful bulk-synchronous model, which we call the

[1] Indeed, a PRAM with as many processors and memory cells is a BSP model with $h = 1$, as is a *module parallel computer* (MPC) [31], which is also known as is a *distributed-memory machine* (DMM) [23], for any memory size.

weak-CREW BSP model. In this model we assume processors are numbered 1, 2, ..., p, and that messages can be duplicated by the network so long as the destinations for any message are a contiguous set of processors $\{i, i+1, \ldots, j\}$. This is essentially the same as a model Dehne *et al.* [15, 16] refer to as the coarse-grain multi-computer. In designing an algorithm for this model one must take care to ensure that, even with message duplication, the number of messages received by a processor in a single round is at most h. Nevertheless, as we demonstrate in this paper, this limited broadcast capability can sometimes be employed to yield weak-CREW BSP algorithms that are conceptually simpler than their EREW BSP counterparts.

Finally, one can imagine more powerful instances of the BSP model, such as a *CREW BSP* model, which would allow for arbitrary broadcasts, or even a *CRCW BSP* model, which would also allow for messages to the same location to be combined (using some arbitration rule). (See also [19, 32].)

The running time of a BSP algorithm is characterized by two parameters: T_I, the internal computation time, and T_C, the number of communication rounds. A prime goal in designing a BSP algorithm is to minimize both of these parameters. Alternatively, by introducing additional characterizing parameters of the BSP model, we can combine T_I and T_C into a single running time parameter, which we call the *combined running time.* Specifically, if we let L denote the latency of the network—that is, the worst-case time needed to send one processor-to-processor message—and we let g denote the time "gap" between consecutive messages received by a processor in a communication round, then we can characterize the total running time of a BSP computation as $O(T_I + (L+g \quad {}_C)$. Incidentally, this is also the running time of implementing a BSP computation in the analogous[2] LogP model [12, 25].

The goal of this paper is to further the study of bulk-synchronous parallel algorithms by addressing the fundamental problem of sorting n elements distributed evenly across a p-processor BSP computer.

1.2 Previous work on parallel sorting

Let us, then, briefly review a small sample of the work previously done for parallel sorting. Batcher [5] in 1968 gave what is considered to be the first parallel sorting scheme, showing that in a fine-grained parallel sorting network one can sort in $O(\log^2 n)$ time using $O(n)$ processors. Since this early work there has been much effort directed at fine-grain parallel sorting algorithms (e.g., see Akl [4], Bitton *et al.* [7], JáJá [22], Karp and Ramachandran [24], and Reif [37]). Nevertheless, it was not until 1983 that it was shown, by Ajtai, Komlós, and Szemerédi [3], that n elements can be sorted in $O(\log n)$ time with an $O(n \log n)$-sized network (see also Paterson [35]). In 1985 Leighton [27] extended this result to show that one can produce an $O(n)$-node bounded-degree network capable of sorting in $O(\log n)$ steps, based upon an algorithm he called "columnsort." In 1988 Cole [10] gave simple methods for optimal sorting in the CREW and EREW PRAM models in $O(\log n)$ time using $O(n)$ processors, based upon an elegant "cascade mergesort" paradigm using arrays, and this result was recently extended to the Parallel Pointer Machine by Goodrich and Kosaraju [20]. Thus, one can sort optimally in these fine-grained models.

These previous methods are not optimal, however, when implemented in bulk-synchronous models. Nevertheless, Leighton's columnsort method [27] can be used to design a bulk-synchronous parallel sorting algorithm that uses a constant number of communication rounds, provided $p^3 \leq n$. Indeed, there are a host of published algorithms for achieving such a result when the ratio of input size to number of processors is as large as this. For example, a randomized strategy, called sample sort, achieves this result with high probability [8, 17, 18, 21, 29, 38], as do deterministic strategies based upon regular sampling [33, 39]. These methods based upon sampling do not seem to scale nicely for smaller n/p ratios, however. If columnsort is implemented in a recursive fashion, then it yields an EREW BSP algorithm that uses $T_C = O([\log n / \log(n/p)]^\delta)$ communication rounds and internal computation time that is $O(T_C(n/p) \log(n/p))$, where $\delta = 2/(\log 3 - h)\mathbb{I})$, which is approximately 3.419. Using an algorithm they call "cubesort," Cypher and Sanz [13] show how to improve the T_C term in these bounds to be $O((25)^{(\log^* n - \log^*(n/p))}[\log n / \log(n/p)]^2)$, and Plaxton [36] shows how cubesort can be modified to achieve $T_C = O([\log n / \log(n/p)]^2)$. Indeed, Plaxton[3] can modify the "sharesort" method of Cypher and Plaxton [14] to achieve $T_C = O((\log n / \log(n/p)) \log^2(\log n / \log(n/p)))$. Finally, Chvátal [9] describes an approach of Ajtai, Komlós, Paterson, and Szemerédi for adapting the sorting network of Ajtai, Komlós, and Szemerédi [3] to achieve a depth of $O(\log n / \log(n/p))$ where the basic unit in the network is a "black box" that can sort $\lceil n/p \rceil$ elements. An effective method for constructing such a network is not included in Chvátal's report, however, for the method he describes is a non-uniform procedure based upon the probabilistic

[2]There is also an o parameter in the LogP model, but it would be redundant with L and g in this bound.

[3]Personal communication.

method. In addition, the constant factor in the running time appears to be fairly large. Incidentally, these latter methods [9, 14, 13, 27, 36] are actually defined for more-restrictive BSP models where the data elements cannot be duplicated and each internal computation must be a sorting of the internal-memory elements.

The only previous sorting algorithms we are aware of that were designed with the BSP model in mind are recent methods of Adler, Byers, and Karp [1] and Gerbessiotis and Valiant [18]. The method of Adler *et al.* runs in a combined time that is $O(\frac{ng\log n}{p} + pg + gL)$, provided $p \leq n^{1-\delta}$ for some constant $0 < \delta < 1$. They do not define their algorithm for larger values of p, but they do give a slightly better implementation of their method in the LogP model so as to achieve a running time of $O(\frac{ng\log n}{p}+pg+L)$ for p similarly bounded. Gerbessiotis and Valiant give several randomized methods, the best[4] of which runs with a combined time of $O(\frac{n\log n}{p} + gp^{\epsilon} + gn/p + L)$, with high probability, for any constant $0 < \epsilon < 1$, provided $p \leq n^{1-\delta}$, where δ is a small constant depending upon ϵ.

1.3 Our results

Given a set S of n items distributed evenly across p processors in a weak-CREW BSP computer we show how S can be sorted in $O(\log n/\log(h+1))$ communication rounds and $O((n\log n)/p)$ internal computation time, for $h = \Theta(n/p)$. The method is fairly simple and the constant factors in the running time are fairly small. Moreover, we also show how to extend our result to the EREW BSP model while achieving the same asymptotic bounds on the number of communication rounds and internal computation time. Our bounds on internal computation time are optimal for any comparison-based parallel algorithm. In addition, we achieve a deterministic combined running time that is $O(\frac{n\log n}{p} + (L + gn/p)(\log n/\log(n/p)))$, which is valid for all values of p and improves the best bounds of Adler *et al.* [1] and Gerbessiotis and Valiant [18] even when $p \leq n^{1-\delta}$ for some constant $0 < \delta < 1$, in which case our method sorts in a constant number of communication rounds. In fact, if $p^3 \leq n$, then our method essentially amounts to a sample sort (with regular sampling). If $p = \Theta(n)$, then our method amounts to a pipelined parallel mergesort, achieving the same asymptotic performance as the fine-grained algorithms of Cole [10] and Goodrich and Kosaraju [20]. Thus, our method provides a sorting method that is fully-scalable over all values of p while achieving an optimal internal computation time over this entire range.

Indeed, we show that our bounds on the number of communication rounds needed to sort n elements on a BSP computer are also worst-case optimal for this entire range of values of p. We establish this by showing that simply computing the "or" of n bits distributed evenly across $\Theta(n/h)$ processors requires $\Omega(\log n/\log(h+1))$ number of communication rounds, where each processor can send and receive h messages in a CREW BSP computer. This lower bound holds even if the number of additional processors and the number of additional memory cells per processor are unbounded. Since this lower bound is independent of the total number of processors and amount of memory in the multicomputer, it joins lower bounds of Mansour *et al.* [30] and Adler *et al.* [1] in giving further evidence that the prime bottleneck in parallel computing is communication, and not the number of processors nor the memory size.

[4] They also give a method with a combined running time of $O([(n/p)\log^{a+1} p + L\log^2 p + g\log^{a+2} p + g(n/p)\log p]/\log\log p)$, with high probability, provided $p < n/\log^{a+1} p$.

2 A weak-CREW BSP Algorithm

Let S be a set of n items distributed evenly in a p-processor weak-CREW BSP computer. We sort the elements of S using a d-way parallel mergesort, pipelined in a way analogous to the binary parallel mergesort procedures of Cole [10] and Goodrich and Kosaraju [20].

Specifically, we choose $d = \max\{\lceil\sqrt{n/p}\rceil, 2\}$, and let T be a d-way rooted, complete, balanced tree such that each leaf is associated with a subset $S_i \subseteq S$ of size at most $\lceil n/p\rceil$. For each node v in T define $U(v)$ to be the sorted list of elements stored at descendents of v in T, where we define v to be a descendent of itself if it is a leaf. Note that if $\{w_1, w_2, \ldots, w_d\}$ denote the children of a node v in T, then $U(v) = U(w_1) \cup U(w_2) \cup \cdots \cup U(w_d)$. Our goal, then, is to construct $U(root(T))$. We may assume, without loss of generality, that the elements are distinct, for otherwise we can break ties using the original positions of the elements of S.

We perform this construction in a bottom-up pipelined way. In particular, we perform a series of *stages*, where in a Stage t we construct a list $U_t(v) \subseteq U(v)$ for each node v that we identify as being *active*. A node is *full* in Stage t if $U_t(v) = U(v)$, and a node is *active* if $U_t(v) \neq \emptyset$ and v was not full in Stage $t-3$. Likewise, we say that a list A stored at a node v in T is *full* if $A = U(v)$. Initially, each leaf of T is full and active, whereas each internal node is initially inactive.

We say that a list B is a *k-sample* of a list A if B consists of every k-th element of A. For each active node v in T we define a sample $L_t(v)$ defined as follows:

- If v is not full, then $L_t(v)$ is a d^2-sample of $U_t(v)$.
- If v first became full in Stage t, then we define $L_t(v)$ to be a d^2-sample of $U_t(v) = U(v)$; we define $L_{t+1}(v)$ to be a d-sample of $U_t(v)$, and we define $L_{t+2}(v) = U(v)$ (i.e., $L_{t+2}(v)$ is full).

We then define

$$U_t(v) = L_{t-1}(w_1) \cup L_{t-1}(w_2) \cup \cdots \cup L_{t-1}(w_d),$$

where, again, $\{w_1, w_2, \ldots, w_d\}$ denote the children of node v in T. Note that by our definition of $L_t(v)$, if a node v becomes full in Stage t, then v's parent becomes full in Stage $t+3$. Thus, assuming we can implement each stage with a constant number of communication rounds using the p processors, then we will be able to sort the elements of S, by constructing $U(root(T))$, in just $O(\log_d n) = O\left(\frac{\log n}{\log(h+1)}\right)$ communication rounds, for $h = \Theta(n/p)$. Before we give the details for implementing each stage in our algorithm, however, we establish the following bounds (whose proofs are included in the full version):

Lemma 2.1: *If at most k elements of $U_t(v)$ are in an interval $[a, b]$, then at most $dk + 2d^2$ elements of $U_{t+1}(v)$ are in $[a, b]$.*

Intuitively, this lemma says that $U_{t+1}(v)$ will not be wildly different from $U_t(v)$. Similarly, we have the following corollary that relates $L_{t+1}(v)$ and $L_t(v)$:

Corollary 2.2: *If at most k elements of $L_t(v)$ are in an interval $[a, b]$, then at most $d(k+1)+2$ elements of $L_{t+1}(v)$ are in $[a, b]$.*

Having given this important lemma and its corollary, let us now turn to the details of implementing each stage in our pipelined procedure using just a constant number of communication rounds.

2.1 Implementing each stage using a constant number of communication rounds

We say that a list A is *ranked* [10, 20] into a list B if, for each element $a \in A$, we know the rank of a's predecessor in B (based upon the ordering of elements in $A \cup B$). If A is ranked in B and B is ranked in A, then A and B are *cross-ranked*. The generic situation at the end of any Stage t is that we have the following conditions satisfied at each node v in T.

Induction Invariants:

1. $L_t(v)$ is ranked into $L_{t-1}(v)$.
2. If v is not full, then $L_{t-1}(w_i)$ is ranked in $U_t(v)$, for each child w_i of v in T.
3. $L_t(v)$ is ranked into $U_t(v)$.

We maintain copies of the lists $L_{t-1}(v)$, $L_t(v)$, $U_{t-1}(v)$, and $U_t(v)$ for each active node v in T, and we do not maintain any other lists during the computation. As we shall show, this will allow us to implement the entire computation efficiently using just p processors. In order to implement each stage in our computation using just $O(1)$ communication rounds we also maintain the following important load-balancing invariant at each node v in T.

Load-balancing Invariant:

- If a list A is not full, then A is partitioned into contiguous subarrays of size d each, with each subarray stored on a different processor.
- If a list A is full, then A is partitioned into contiguous subarrays of size d^2 each, with each subarray stored on a different processor.

We assume that the names of the nodes of v in T and the four lists stored at each node v are defined so that given an index, i, into one of these lists, A, one can determine the processor holding $A[i]$ as a local computation (not needing a communication step)[5]. Given that the induction and load-balancing invariants are satisfied for each node v in T, we can construct $U_{t+1}(v)$ at each active node, with the above invariants satisfied for it, as follows.

Computation for Stage $t+1$:

1. For each element a in $L_t(w_i)$, let $b(a)$ and $c(a)$ respectively be the predecessor and successor of a in $L_{t-1}(w_i)$. We can determine $b(a)$ and $c(a)$ in $O(1)$ communication rounds, for each such a, since $L_t(w_i)$ is ranked in $L_t(w_i)$ by Induction Invariant 1. In fact, if $L_t(w_i) = U(w_i)$, then this is essentially a local computation. Moreover, by our load-balancing invariant and Corollary 2.2, even in the general case, each processor (storing a portion of some $L_{t-1}(w_i)$) will receive (and then send) at most $d(d+1)+2 = \Theta(h)$ messages to implement this step.

[5]We maintain this assumption inductively, as we show in the full version.

2. Determine the location (rank) of $b(a)$ and $c(a)$ in $U_t(v)$. This can also be easily implemented with a $O(1)$ communication rounds, as in the previous step.

3. Broadcast a (and its rank in $L_t(w_i)$) to all processors holding elements of $U_t(v)$ between $b(a)$ and $c(a)$. By our load-balancing invariant and Lemma 2.1 we can guarantee that each processor will receive at most $3d^2 + d = \Theta(h)$ messages to implement this step (each processor receives at least one element from each child of v plus as many elements as fall in its interval of $U_t(v)$); hence, it can be done in $O(1)$ communication rounds.

4. Each processor assigned to a contiguous portion $[e, f)$ of $U_t(v)$ receives elements sent in the previous round and merges them via a simple d-way mergesort procedure to form a sublist of $U_{t+1}(v)$ of size $O(d^2) = O(h)$. It is important to observe that the processor for $[e, f)$ receives at least one element from each child of v so as to include all all the elements that may intersect the interval $[e, f)$, even if none actually fall inside $[e, f)$. This allows us to accurately compute the rank of each element in $U_{t+1}(v)$ locally; hence, it gives us $U_t(v)$ cross-ranked with $U_{t+1}(v)$. Moreover, this step can be accomplished in $O(1)$ communication rounds and $O(d^2 \log d) = O((n/p)\log(n/p))$ internal computation time.

5. For each element a in $U_{t+1}(v)$ send a message to the processor holding $a \in L_t(w_i)$ informing that copy of a of its rank in $U_{t+1}(v)$. This step can easily be accomplished in $O(1)$ communication rounds, and gives us Induction Invariant 2.

6. Determine the sample $L_{t+1}(v)$ and rank it into $U_{t+1}(v)$, giving us Induction Invariant 3. Also, use the cross-ranking of $U_{t+1}(v)$ and $U_t(v)$ to rank $L_{t+1}(v)$ into $L_t(v)$, giving us Induction Invariant 1. This step can easily be accomplished in $O(1)$ communication rounds.

7. Finally, partition the four lists stored at each node v so as to satisfy the load-balancing invariant. Assuming the total size of all the non-full lists in T is $O(n/d)$, then this can easily be performed in $O(1)$ communication rounds using $p = \Theta(n/d^2)$ processors.

Therefore, given the above assumption regarding the total size of all the lists, in a constant number of communication rounds and an internal computation time that is $O((n/p)\log(n/p))$ we can build the set $U_{t+1}(v)$ and establish the induction and load-balancing invariants so as to repeat this procedure in Stage $t + 2$.

Let us therefore analyze the total size of all the lists stored at nodes in T. Clearly, the size of all the full lists in T is $O(n)$. Moreover, each such list contributes at most $1/d$ of its elements to the next higher level in T, and from then on up T each lists on a level l contribute at most $1/d^2$ of its elements to lists on the next higher level in T. Thus, the total size of all non-full $U_{t-1}(v)$ or $U_t(v)$ lists forms a geometric series that sums to be $O(n/d)$, which is what we require. In addition, any sample $L_t(v)$ or $L_{t-1}(v)$ that is not full can contain at most $1/d$ of the elements of $U(v)$; hence, the total space needed for all these lists is also $O(n/d)$. This establishes the following:

Theorem 2.3: *Given a set S of n items stored $O(n/p)$ per processor on a p-processor weak-CREW BSP computer, one can sort S in $O(\log n/\log(h+1))$ communication steps and $O(n \log n/p)$ internal computation time, where $h = \Theta(n/p)$.*

In achieving this result we exploited the broadcast capability of the weak-CREW BSP model (in Step 3). In the next section we show how to match the asymptotic performance of Theorem 2.3 without using such a capability.

3 An EREW BSP Algorithm

Suppose we are now given a set S of n items, which are distributed evenly across the p processors of an EREW BSP computer. Our goal is to sort S in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ internal computation time without using any broadcasts, for $h = \Theta(n/p)$. We achieve this result using a cascading method similar to one used by Cole [10].

Let T be a complete rooted d-way tree with each of its leaves associated with a sublist $S_i \subset S$ of size at most $\lceil n/p \rceil$, where $d = \max\{\lceil (n/p)^{1/7} \rceil, 2\}$ (the reason for this choice will become apparent in the analysis). Our method proceeds in a series of stages, as in the weak-CREW BSP algorithm, with us constructing the set $U_t(v)$ in each stage, as before:

$$U_t(v) = \bigcup_{i=1}^{d} L_{t-1}(w_i),$$

where each $L_t(v)$ list is defined to be a sample of $U_t(v)$ as in our weak-CREW algorithm.

In order to perform this construction so as to avoid broadcasts, however, we will accomplish this by constructing a larger, augmented list, $A_t(v)$, such that $U_t(v) \subseteq A_t(v)$. We also define a list $D_t(v)$ to be a d^2-sample of $A_t(v)$. For each active node v, with parent u and children $w_1, w_2, \ldots, w_d$, we then define

$$A_t(v) = D_{t-1}(u) \cup \bigcup_{i=1}^{d} L_{t-1}(w_i),$$

i.e., $A_t(v) = D_{t-1}(u) \cup U_t(v)$. Intuitively, the D_t lists communicate information "down" the tree T in a way that allows us to avoid broadcasts. Indeed, once a copy of an element begins to traverse down the tree, then it will never again traverse up (since the D lists are only sent to children).

Still, even though we are assuming, without loss of generality, that the elements of S are distinct, this definition may create duplicate entries of an element in the same list, with some traversing down and at most one traversing up. We resolve any ambiguities this may create by breaking comparison ties based upon an upward-traversing element always being greater than any downward-traversing element, and any comparison between downward-traversing elements being resolved based upon the level in T where the elements first began traversing down (where level numbers increase as one traverses down T).

The goal of each Stage t in the computation, then, is to construct $A_t(v)$ and $U_t(v)$, together with their respective samples $D_t(v)$ and $L_t(v)$. In order to prove that each stage of our algorithm can indeed be performed in a constant number of communication rounds on an EREW BSP computer we must establish the following bounds (whose proofs are included in the full version):

Lemma 3.1: *If at most k elements of $A_t(v)$ are in an interval $[a, b]$, then at most $(d+1)k + 2(d+1)^2$ elements of $A_{t+1}(v)$ are in $[a, b]$.*

This immediately implies the following:

Corollary 3.2: *If at most k elements of $D_t(v)$ are in an interval $[a, b]$, then at most $(d+1)(k+1)+3$ elements of $D_{t+1}(v)$ are in $[a, b]$.*

In addition, we can also show the following:

Lemma 3.3: *For any two consecutive elements b and c in $A_t(v)$ let b' and c' respectively be the predecessor of b and the successor of c in $A_t(u)$, where u is the parent of v in T. There are at most $(d+1)(d^2+1) + 2(d+1)^2 + 2$ elements of $A_t(u)$ in the interval $[b', c']$.*

Finally, we have the following:

Lemma 3.4: *For any two consecutive elements b and c in $D_{t-1}(u)$ there are at most $(d+1)^2(d^4+5)$ elements of $A_t(v)$ in the interval $[b, c]$, where u is the parent of v in T.*

As will become apparent in our algorithm description, these bounds are all crucial for establishing that our algorithm runs in the EREW BSP model using a constant number of communication rounds per stage. In order to perform the computation for Stage $t+1$ using a constant number of communication rounds we assume that we maintain the following induction invariants for each active node v in T:

Induction Invariants:

1. $A_t(v)$ is ranked into $U_t(v)$.
2. $A_t(v)$ and $D_{t-1}(u)$ are cross-ranked, where u is the parent of v.
3. $A_{t-1}(v)$ is ranked into $A_t(v)$.
4. $D_t(v)$ is ranked in $D_{t-1}(v)$.

We also maintain a load-balancing invariant, similar to the one we used in our weak-CREW BSP algorithm, except that we now define a list A stored at a node v to be *full* if $A \supseteq U(v)$.

Load-balancing Invariant:

- If a list A is not full, then A is partitioned into contiguous subarrays of size d^6 each, with each subarray stored on a different processor.
- If a list A is full, then A is partitioned into contiguous subarrays of size d^7 each, with each subarray stored on a different processor.

Given that the induction and load-balancing invariants hold after the completion of Stage t, our method for performing Stage $t+1$ is as follows.

Computation for Stage $t+1$:

For each child w_i of v we perform the following computation.

1. For each element a in $A_t(w_i)$ use the ranking of $A_t(w_i)$ in $U_t(w_i)$ to determine if a is also in $L_t(w_i)$ (together with its rank in $L_t(w_i)$ if so). No communication is necessary for this step, given Induction Invariant 1.
2. For each such element a in $L_t(w_i)$ use the ranking of $A_t(w_i)$ in $D_{t-1}(v)$ to determine the

ranks of the predecessor, $b(a)$, of a and successor, $c(a)$, of a in $D_{t-1}(v)$. No communication is necessary for this step either, given Induction Invariant 2.

3. For each a in $L_t(w_i)$, use the ranks of the processor(s) for $b(a)$ and $c(a)$ in $D_{t-1}(v)$ to determine the respective ranks of $b(a)$ $c(a)$ in $A_{t-1}(v)$. No communication is necessary for this step.

4. For each a in $L_t(w_i)$, request that the processor(s) for $b(a)$ and $c(a)$ in $A_{t-1}(v)$ send(s) the processor for a the name of predecessor, $b'(a)$, of $b(a)$ and the name of successor, $c'(a)$, of $b(a)$ in $A_t(v)$, using Invariant 3. By Lemma 3.1 and our load-balancing invariant, each processor will receive and send at most $(d+1)d^6 + 2(d+1)^2 = \Theta(h)$ messages to implement this step.

5. Send a (together with its rank in $L_t(w_i)$) to the processor(s) assigned to elements of $A_t(v)$ between $b'(a)$ and $c'(a)$ to be merged with all other elements of $A_{t+1}(v)$ that fall in this range. As with the previous step, by Lemma 3.1 and our load-balancing invariant, each processor will receive at most $(d+1)d^6 + 2(d+1)^2 = \Theta(h)$ messages to implement this step. More importantly, by Lemma 3.3, each processor will send an element a to at most $\lceil((d+1)(d^2+1)+2(d+1)^2+2)/d^6\rceil+1 = O(1)$ other processors. Thus, no broadcasting is needed in order to implement this step.

At the parent u of v we assume a similar (but simpler) computation is being performed. Finally, at node v we perform the following computation:

1. For each interval $[e, f)$ of elements of $A_t(v)$ assigned to a single processor, merge all the elements coming from the parent u and children $w_1, w_2, \ldots, w_d$ to form $A_{t+1}(v)$. Such a processor will receive at least one element from each node adjacent to v, plus as many elements of $A_{t+1}(v)$ as fall in $[e, f)$, for a total of at most $d + 1 + (d+1)d^6 + 2(d+1)^2 = \Theta(h)$. This mergesort computation amounts to a $(d+1)$-way mergesort and can easily be implemented in $O(d^7 \log(d+1)) = O((n/p)\log(n/p))$ internal steps.

2. Likewise, for each interval $[e, f)$ of elements of $A_t(v)$ assigned to a single processor, merge all the elements coming just from v's children $w_1, w_2, \ldots, w_d$ to form $U_{t+1}(v)$ (and $A_{t+1}(v)$ ranked in $U_{t+1}(v)$, which gives us Induction Invariant 1).

3. Use the rank information derived from the previous two steps to rank $A_{t+1}(v)$ in $D_t(u)$, giving us half of Induction Invariant 2. Also, rank $A_t(v)$ in $A_{t+1}(v)$ giving us Invariant 3 and by an additional calculation a ranking of $D_{t+1}(v)$ in $D_t(v)$, which is Invariant 4. Finally, send a message to each element a in $D_t(u)$ informing it of its rank in $A_{t+1}(v)$ so as to complete the other half of Invariant 2. To implement this step requires that each processor send at most h messages and each processor receive at most $d^6(d) = O(h)$ messages.

4. Finally, repartition the lists at each node v so as to satisfy the load-balancing invariant. Assuming that the total size of all non-full lists is $O(n/d)$ and the size of all full lists is $O(n)$, then this step can easily be implemented in $O(1)$ communication rounds.

Let us, therefore, analyze the space requirements of this algorithm. The total size of all the $U(v)$ lists on the full level clearly is $O(n)$. Each such list causes at most $\lceil |U(v)|/d \rceil$ elements to be sent to v's parent, u. Now the inclusion of these elements in u causes at most $(d+1)\lceil |U(v)|/d^3 \rceil$ elements to be sent to nodes at distance 1 from u (including v itself). But once an element starts traversing down the tree T it never is sent up again. We can repeat this argument to establish that the existence of $U(v)$ causes at most $(d+1)^2\lceil |U(v)|/d^5 \rceil$ elements to be sent to nodes at distance 2 from u, and so on. Thus, the number of all of these elements that originate from u sum to be a geometric series that is $O(n/d)$. Therefore, the total size of all the non-full lists is $O(n/d)$. Likewise, the total size of all the lists (and hence the lists on the full level) is $O(n)$. This gives us the following theorem:

Theorem 3.5: *Given a set S of n items stored $O(n/p)$ per processor on a p-processor EREW BSP computer, one can sort S in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ internal computation time, for $h = \Theta(n/p)$.*

This immediately implies the following:

Corollary 3.6: *Given a set S of n items stored $O(n/p)$ per processor, one can sort S on an EREW BSP computer with a combined running time that is $O(\frac{n \log n}{p} + (L + gn/p)(\log n/\log(n/p)))$.*

This bound also applies to the LogP model.

4 A Lower Bound for BSP Computations

In this section we show that our upper bounds on the number of communication rounds needed to sort n numbers on a p-processor BSP computer are optimal. Specifically, we show that $\Omega(\log n/\log(h+1))$ communication steps are needed to compute the "or" of n bits using an arbitrary number of processors in a CREW BSP computer, where h is the number of message that can be sent and received by a single processor in a single communication round.

Let us begin by formalizing the framework for proving our lower bound. Assume we have a set S of n Boolean values $x_1, x_2, \ldots, x_n$ initially placed in memory locations $m_1, m_2, \ldots, m_n$ with memory cells $m_{(i-1)h+1}, \ldots, m_{ih}$ stored in the local memory of processor p_i, for $i \in \{1, 2, \ldots, \lceil n/h \rceil\}$. This, of course, implies that we have at least $\lceil n/h \rceil$ processors, but for the sake of the lower bound we allow for an arbitrary number of processors. Moreover, we place no upper bound on the amount of additional memory cells that each processor may store internally. The goal of the computation is that after some T steps the "or" of the values in S should be stored in memory location m_1.

Our lower bound proof will be an adaptation of a lower bound proof of Cook, Dwork, and Reischuk [11] for computing the "or" of n bits on a CREW PRAM. The main difficulties in adapting this proof come from the way the fact that each processor in a BSP computer can send h messages in each communication round, rather than just a single value, complicates arguments that bound the amount of information processors can communicate by *not* sending messages.

Each processor p_i is assumed initially to be in a starting state, q_1^i, taken from a possibly-unbounded set of states. At the beginning of a round t processor p_i is assumed to be in some state q_t^i. A round begins with each processor sending up to h messages, some of which may be (arbitrary) partial broadcasts, and simultaneously receiving up to h messages from other processors. Without loss of generality, each message may be assumed to be the contents of one of the memory cells associated with the sending processor, since we place no constraints on the amount of information that may be stored in a memory cell nor on the number of memory cells that a processor may contain. A processor then enters a new state q_{t+1}^i that depends upon its previous state q_t^i and the values of the messages it has received. A round completes with a processor possibly writing new values to some of its internal memory cells based upon its new state q_{t+1}^i.

Before analyzing the most general situation, let us first prove a lower bound for the *oblivious* case, where the determination of whether a processor p_i will send a message to processor p_j in round t depends only upon the value of p_i and t, and not on the input. Of course, the contents of such a message could depend upon the input. For input string $I = (x_1, x_2, \ldots, x_n)$ of Boolean values, let $I(k)$ denote the input string $(x_1, x_2, \ldots, \bar{x}_k, \ldots, x_n)$, where $\bar{x}_k$ denotes the complement of Boolean value x_k. I is a *critical input* for function $f(I)$ if $f(I) \neq f(I(k))$ for all $k \in \{1, 2, \ldots, n\}$. (Note that $I = (0, 0, \ldots, 0)$ is a critical input for the "or" function.) Say that input index k *affects* [11] processor p_i in round t with input I if the state of p_i on input I after round t differs from the state of processor p_i on input $I(k)$ after round t. Likewise, say that input index k *affects* memory cell m_i in round t with input I if the contents of m_i on input I after round t differs from the contents of m_i on input $I(k)$ after round t.

Theorem 4.1: *If $f : \{0,1\}^n \to \{0,1\}$ has a critical input, then any oblivious CREW BSP computer that computes f requires $\Omega(\log n/\log(h+1))$ communication rounds.*

Proof: Let $K(p_i, t, I)$ (respectively, $L(m_i, t, I)$) be the set of input indices that affect processor p_i (resp., memory cell m_i) in round t with input I. Further, let K_t and L_t satisfy the following recurrence equations:

$$K_0 = 0, \quad (1)$$

$$L_0 = 1, \quad (2)$$

$$K_{t+1} = K_t + hL_t, \quad (3)$$

$$L_{t+1} = K_{t+1} + L_t. \quad (4)$$

Note that it suffices to prove that $|K(p_i, t, I)| \leq K_t$ and $|L(p_i, t, I)| \leq L_t$, for K_t and L_t are both at most $[2(h+1)]^t$, and if I is a critical input for f, then every one of the input indices must affect memory cell m_1. That is, if $m_1 = f(I)$, then $|L(m_1, T, I)| = n$, which implies that T is $\Omega(\log n/\log(h+1))$. In the full version we show how to establish the above bounds on $|K(p_i, t, I)|$ and $|L(p_i, t, I)|$ by induction on t. ■

The main difficulty in generalizing this result to non-oblivious computations is that in the non-oblivious case a processor p_i can receive information from a processor p_j by p_j *not* sending a message to p_i. Still, as we show in the next theorem, this ability cannot alter the asymptotic performance of a CREW BSP computer by more than a constant factor for computing the value of a function with a critical input.

Theorem 4.2: *If $f : \{0,1\}^n \to \{0,1\}$ has a critical input, then any CREW BSP computer that computes f requires $\Omega(\log n / \log(h+1))$ communication rounds.*

Proof: Let $K(p_i, t, I)$ and $L(m_i, t, I))$ be as in the proof of Theorem 4.1. But now let K_t and L_t be defined by the following recurrence relations:

$$
\begin{aligned}
K_0 &= 0, & (5)\\
L_0 &= 1, & (6)\\
K_{t+1} &= (2h+1)K_t + hL_t, & (7)\\
L_{t+1} &= K_{t+1} + L_t. & (8)
\end{aligned}
$$

As in the previous proof, it suffices to show that $|K(p_i, t, I)| \le K_t$ and $|L(m_i, t, I)| \le L_t$, for K_t and L_t are both at most $[3(h+1)]^t$.

We establish these bounds on $|K(p_i, t, I)|$ and $|L(p_i, t, I)|$ by induction on t. First, note that $K(p_i, t, I)$ is empty in round $t = 0$, and $L(m_i, 0, I) = \{i\}$ if $i \in \{1, 2, \ldots, n\}$ and otherwise $L(m_i, 0, I)$ is empty. At the beginning of round t a processor p_i receives the contents of at most h memory locations, and it also receives information by noting that some processors did not send p_i a message. Still, after it incorporates this information into its new state q^i_{t+1} it optionally writes to its local memory, as in the previous proof. Thus, if we can establish Equation (7), then Equation (8) immediately follows.

Say that input index k *possibly-causes* a processor p_j to send a message to processor p_i in round t with I if p_j sends a message to processor p_i in round t on input $I(k)$. Using this notion we bound $K(p_i, t+1, I)$ as a subset of

$$K(p_i, t, I) \cup \bigcup_{j \in \mathcal{I}} L(m_j, t, I) \cup Y(p_i, t, I),$$

for some index set $\mathcal{I}$ with $|\mathcal{I}| \le h$, where $Y(p_i, t, I)$ denotes the set of all indices k that possibly-cause some processor p_j to send a message to p_i with I. Thus, we must bound $r = |Y(p_i, t, I)|$. So, let $Y = Y(p_i, t, I) = \{k_1, k_2, \ldots, k_r\}$ be the set of indices k_j that possibly-cause a processor $p(k_j)$ to send a message to p_i with I. Note that if $r \le hK_t$, then we have established Equation (7), so for the remainder of this proof let us assume that $r > hK_t$ (we will show that if this is the case, then $r \le 2hK_t$). Say that a subset $Y' \subseteq Y$ is *processor-disjoint* if, for any k_j and $k_{j'}$ in Y', $p(k_j) \ne p(k_{j'})$.

Claim: If $Y' = \{k_{j_1}, k_{j_2}, \ldots, k_{j_{h+1}}\}$ is a processor-disjoint subset of Y of size $h+1$, then there is an index k_{j_i} in Y' such that k_{j_i} affects processor $p(k_{j_l})$ in round t with $I(k_{j_l})$, where $k_{j_i} \ne k_{j_l}$.

Proof (of claim): If this is not the case, then on input $I(k_{j_1})(k_{j_2}) \cdots (k_{j_{h+1}})$ there would be $h+1$ different messages sent to processor p_i, which would violate the correctness of the BSP computation. ■ (of claim)

As done by Cook, Dwork, and Reischuk [11], we employ a combinatorial graph argument to derive a bound on $|Y|$. Consider a bipartite graph G whose two node sets are $\{k_1, k_2, \ldots, k_r\}$ and $\{p(k_1), p(k_2), \ldots, p(k_r)\}$. Let there be an edge between k_j and $p(k_l)$ if k_j affects $p(k_l)$ in round t with $I(k_l)$. Let e denote the number of edges in G. The degree of any node $p(k_j)$ is $|K(p(k_j), t, I(k_j))|$, which, by our induction hypothesis, is bounded by K_t. Thus, $e \le rK_t$. We can also derive a lower bound on e, using our claim above. Let G' be a subgraph of G defined by a processor-disjoint subset of Y of size $h+1$ together with the processor nodes associated with this subset. Then, by our claim, G' must contain at least one edge of G. Thus, letting g denote the number of processor-disjoint subsets (like G') of size $h+1$, together with the associated processor nodes, we can write $g \le e$. Since placing a node in such a G' eliminates at most K_t other candidates,

$$
\begin{aligned}
g &\ge \frac{r(r-K_t)(r-2K_t)\cdots(r-hK_t)}{(h+1)!}\\
&\ge \frac{r(r-hK_t)^h}{(h+1)!}.
\end{aligned}
$$

Therefore, $r(r-hK_t)^h \le (h+1)!rK_t$, which implies that $r \le hK_t + [(h+1)!K_t]^{1/h}$. By Sterling's approximation, then, $r \le 2hK_t$, which establishes Equation (7) and completes the proof of the theorem. ■

Acknowledgements

We would like to thank Bob Cypher, Faith Fich, S. Rao Kosaraju, Greg Plaxton, and Les Valiant for several helpful conversations or e-mail exchanges regarding topics related to this paper.

References

[1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. . *7th SPAA*, 129–136, 1995.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *T.C.S.*, 71:3–28, 1990.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

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

[5] K. E. Batcher. Sorting networks and their applications. *Proc. 1968 Spring Joint Computer Conf.*, 307–314, Reston, VA, 1968.

[6] G. Bilardi and F. P. Preparata. Lower bounds to processor-time tradeoffs under bounded-speed message propagation. *4th WADS, LNCS 955*, 1–12. Springer-Verlag, 1995.

[7] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *ACM Comp. Surveys*, 16(3):287–318, 1984.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. *3rd SPAA*, 3–16, 1991.

[9] V. Chvátal. Lecture notes on the new AKS sorting network. Report DCS-TR-294, Rutgers Univ., 1992.

[10] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[11] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *4th ACM Symp. on Princ. and Pract. of Par. Prog.*, 1–12, 1993.

[13] R. Cypher and J. L. C. Sanz. Cubesort: A parallel algorithm for sorting N data items with S-sorters. *J. of Algorithms*, 13:211–234, 1992.

[14] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *J.C.S.S.*, 47:501–548, 1993.

[15] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar. A randomized parallel 3D convex hull algorithm for course grained multicomputers. *7th SPAA*, 27–33, 1995.

[16] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. *9th SCG*, 298–307, 1993.

[17] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, 1970.

[18] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. of Par. and Dist. Comp.*, 22:251–267, 1994.

[19] P. B. Gibbons. A more practical PRAM model. *(1st) SPAA*, 158–168, 1989.

[20] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, to appear.

[21] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementation of randomized sorting on large parallel machines. *4th SPAA*, 158–167, 1992.

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

[23] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on distributed machines. *24th STOC*, 318–326, 1992.

[24] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, 869–941. Elsevier/MIT Press, 1990.

[25] R. M. Karp, A. Sahay, E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. *5th SPAA*, 142–153, 1993.

[26] C. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *T.C.S.*, 71:95–132, 1990.

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

[28] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[29] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. *6th SPAA* 46–56, 1994.

[30] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. *26th STOC*, 372–381, 1994.

[31] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 9(1):29–59, 1984.

[32] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy. Parallel algorithm design on the WPRAM model. Report 94.24, School of Comp. Sci., Univ. of Leeds, 1994.

[33] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *J. ACM*, 29(3):642–667, 1982.

[34] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *20th STOC*, 510–513, 1988.

[35] M. Paterson. Improved sorting networks with $o(\log n)$ depth. *Algorithmica*, 5(1):75–92, 1990.

[36] C. G. Plaxton. *Efficient Computation on Sparse Interconnection Networks.* PhD thesis, Dept. of Comp. Sci., Stanford Univ., 1989.

[37] J. H. Reif. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, 1993.

[38] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *J. ACM*, 34(1):60–76, 1987.

[39] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. and Dist. Comp.*, 14:362–372, 1992.

[40] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, 1990.

[41] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, 943–972. Elsevier/MIT Press, 1990.