Load Balancing, Selection and Sorting on the Hypercube

C. Greg Plaxton*
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

This paper presents novel load balancing, selection and sorting algorithms for the hypercube with 1-port communication. The main result is an algorithm for sorting n values on p processors, SmoothSort, that runs asymptotically faster (in the worst case) than any previously known algorithm over a wide range of the ratio n/p. The load balancing and selection algorithms upon which SmoothSort is based are expected to be of independent interest. Although the analysis of our algorithms is limited to obtaining asymptotic bounds, the constant factors being ignored are quite small.

1 Introduction

This paper presents novel load balancing, selection and sorting algorithms for the hypercube. The following model of computation is assumed. Each processor has an infinite local memory configured in $O(\log p)$-bit words and can perform the usual set of ALU operations in constant time on word-sized operands. Processors communicate with one another by sending *packets* over the bidirectional channels provided by the hypercube network. A packet consists of a constant number of words. We derive asymptotic bounds for the worst case time complexity of our algorithms, and these are stated in terms of *time steps*. In a single time step, each processor is allowed to send and/or receive *at most one* packet, and perform a bounded local computation. Our algorithms operate in a SIMD fashion, but we assume that the instruction format does not restrict all packets to cross the same dimension in a given time step. We refer to this model as the *weak* hypercube model, to distinguish it from other more powerful models that allow every edge to transmit a packet in each time step.

Consider the problem of sorting n keys initially distributed uniformly over a hypercube with p processors, $n \geq p$. The well-known sequential lower bound for sorting implies an $\Omega((n \log n)/p)$ bound on the running time of any parallel sorting algorithm on the weak model. For the case $n = p$, the best known sorting algorithm for the weak model is Batcher's bitonic sort, which runs in $O(\log^2 p)$ time [2]. For $n \neq p$, a number of other algorithms have been proposed. The running time and range of applicability of each of these algorithms is summarized in Table 1. Note that BitonicSort refers to the straightforward split-and-merge generalization of bitonic sort, due independently to Baudet and Stevenson [3] and Johnsson [9]. Also, it should be emphasized that we have restricted attention to deterministic, worst-case complexity algorithms running on the weak model. For examples of results based on other assumptions, we refer the reader to [15], [16] and [18].

One may verify that BitonicSort provides optimal speedup over sequential sorting only if $p = O(2^{\sqrt{\log n}})$. Two recent algorithms, which we refer to as CubeSort (Cypher and Sanz, [5]) and ColumnSort (Aggarwal and Huang, [1]), have improved this result significantly. Both of these algorithms are optimal if n exceeds p by a polynomial factor, that is, if $n = p^{1+\epsilon}$ for any constant $\epsilon > 0$. ColumnSort is based on Leighton's technique for sorting n values by performing a constant number of smaller sorts [12]. Note that Table 1 does not indicate the running time of ColumnSort when ϵ is allowed to vary. This algorithm is not competitive for $\epsilon = o(1)$ since the hidden constant in the running time is proportional to $(1 + 1/\epsilon)^\alpha$ with $\alpha = 2/(\log 3 - 1) \approx 3.419$, as opposed to $\alpha = 1$ for CubeSort. Of course, it may be possible to obtain an algorithm based on Leighton's column sorting technique that achieves a smaller value of α.

*This work was supported in part by a grant from the AT&T Foundation, NSF grant DCR-8351757 and ONR grant N00014-88-K-0166. The author is primarily supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada.

The main result of this paper is a new sorting algorithm, SmoothSort, that runs asymptotically faster (in the worst case) than any previously known algorithm over a wide range of the ratio n/p. We also present a simple variant of this algorithm, which we call QuickSort, that is almost as fast as SmoothSort. The following example illustrates the nature of our results. For $n = p\log^2 p$, the sequential lower bound implies a lower bound of $\Omega(\log^3 p)$, the running time of BitonicSort is $O(\log^4 p)$, the running time of CubeSort is $O(\log^4 p/\log\log p)$, the running time of QuickSort is $O(\log^{7/2} p)$ and the running time of SmoothSort is $O(\log^{7/2} p(\log\log p)^{-1/2})$. ColumnSort is not competitive in this range, and has a running time of about $O(\log^{6.419} p)$.

The remainder of this paper is organized as follows. Sections 2 and 3 obtains upper and lower bounds for two load balancing operations. Section 4 provides a selection algorithm that is independent of the load balancing results. Section 5 defines and analyzes the QuickSort and SmoothSort algorithms, making use of the load balancing and selection operations defined earlier.

Note: The statement of a theorem or lemma for which the proof has been omitted is immediately followed by a $\Box$.

2 Load Balancing

It is important to maintain a balanced load on any parallel computer since an imbalanced load generally leads to low processor utilization. The particular load balancing problem that we consider, Balance, is defined as follows. Let n tokens be distributed over the p processors of the hypercube, with no more than m tokens assigned to any single processor, $\lceil n/p\rceil \le m \le n$. We assume that $n = O(p^c)$ for some constant c in order that calculations involving token counts can be performed with a constant number of CPU operations. The problem is to redistribute the tokens so that each processor has either $\lfloor n/p\rfloor$ or $\lceil n/p\rceil$ tokens, that is, so that the load is distributed as evenly as possible. Peleg and Upfal have exhibited tight bounds for this operation on a certain class of expander networks [14]. In many applications, it is not necessary to balance the population of tokens exactly. If the difference between the maximum number of tokens at any processor and the minimum number of tokens at any processor is ξ, we will say that the tokens have been *balanced with error* ξ. The corresponding operation will be referred to as Balance *with error* ξ.

2.1 Lower Bound

In order to obtain a lower bound for Balance on the weak model, we concentrate the tokens in a small number of processors and bound the time required to eliminate the excess tokens residing at this set of processors. In the following, recall that $d = \log p$ denotes the dimension of the given hypercube.

Definition 2.1 Let $\Gamma^r(i)$ denote the set of $\binom{d}{r}$ processors at Hamming distance r from processor i, $0 \le r \le d$.

Definition 2.2 Let $B(i,r)$ denote the *complete Hamming ball* of radius r centered at processor i. More formally, this is the set of processors given by $B(i,r) = \cup_{0\le l\le r}\Gamma^l(i)$.

Note that $|B(i,r)| = \sum_{0\le l\le r}\binom{d}{l}$. We also need to consider "incomplete" Hamming balls, that is, Hamming balls with only a partially filled outer layer.

Definition 2.3 Given a positive integer x, let r and y be the unique pair of nonnegative integers such that $x = y + \sum_{0\le l\le r-1}\binom{d}{l}$, $1 \le y \le \binom{d}{r}$. We say that a set of processors B is a *Hamming ball of size* x *and radius* r if there is some processor i and some set of processors S such that $B = B(i, r-1)\cup S$, $S \subseteq \Gamma^r(i)$ and $|S| = y$. Let $\mathcal{B}(x)$ denote the set of all Hamming balls of size x.

Definition 2.4 Let the graph G with vertex set $V(G)$ and edge set $E(G)$ be given. For every $U \subseteq V(G)$, we define the *fringe of* U *with respect to* G, $\mathcal{F}(G,U)$, as the set

$$\{u \in U \mid (u,v) \in E(G) \text{ for some } v \in V(G)\setminus U\}.$$

Finally, we define the function

$$f(G,x) = \min_{\substack{U\subseteq V(G)\\|U|=x}} |\mathcal{F}(G,U)|,$$

where G is a graph and x is an integer, $0 \le x \le |V(G)|$, and let H_d denote the undirected graph corresponding to a hypercube of dimension d.

Lemma 2.1 The Balance operation requires $\Omega((n - \lceil n/m\rceil\lceil n/p\rceil)/f(H_d,\lceil n/m\rceil))$ time.

Proof: We can concentrate the n tokens into a set S of $\lceil n/m\rceil$ processors. Under a balanced load, S contains at most $\lceil n/m\rceil\lceil n/p\rceil$ tokens. Hence, at least $n - \lceil n/m\rceil\lceil n/p\rceil$ tokens must leave the set S. Only processors located at the fringe of S, those in $\mathcal{F}(H_d,S)$, can send tokens out of S, and these can only transmit one token per time step. Therefore,

Algorithm	*Running Time*	*Range*
`BitonicSort` [2][3][9]	$O((n/p)\log^2 p)$	$n = \Omega(p)$
`MergeSort` [13]	$O(\log^2 p/\log(p/n))$	$n = O(p)$
`ColumnSort` [1]	$O((n \log n)/p)$	$n = \Omega(p^{1+\epsilon})$, $\epsilon > 0$
`CubeSort` [5]	$O((n/p)\log^2 p/\log(n/p))$	essentially $n = \Omega(p)$

Table 1: Previous sorting algorithms for the weak model.

the running time of `Balance` must be at least $(n - \lceil n/m \rceil \lceil n/p \rceil)/(f(H_d, \lceil n/m \rceil))$. □

Given Lemma 2.1, we can now obtain the desired lower bound by computing $f(H_d, \lceil n/m \rceil)$. Intuitively, one might expect that the value of $f(H_d, x)$ is determined by a Hamming ball configuration. The correctness of this intuition is borne out by the following theorem due to Harper [7]. Note that Frankl and Füredi have given a simpler proof of the same result [6].

Theorem 1 There exists a ball $B \in \mathcal{B}(x)$ such that $f(H_d, x) = |\mathcal{F}(H_d, B)|$, $0 \leq x \leq p$. □

We need the following estimate of the "volume-to-surface" ratio of a Hamming ball.

Lemma 2.2 For positive integers d and r, $0 \leq r \leq d/2$, let $S = \binom{d}{r}$ and let $V = \sum_{0 \leq k \leq r} \binom{d}{k}$. Then $V/S = \Theta\left(\sqrt{\frac{d}{z+1}}\right)$, where z is the real value defined by the equation $r = d/2 - \sqrt{dz}$.

Proof Sketch: Split the analysis into three cases: $0 \leq z \leq 1$, $1 \leq z \leq d/16$, and $d/16 \leq z \leq d/4$. We sketch one approach for the most difficult case: $1 \leq z \leq d/16$. For the lower bound, examine the $(d/z)^{1/2}$ largest terms of V and show that each of these is a constant fraction of S. For the upper bound, show that V is dominated by the infinite geometric progression with initial value S and ratio $r/(d-r+1)$. □

Theorem 2 Let d, r, S, V be as defined in Lemma 2.2. If $V = \Theta(2^{d(1-1/k)})$, then $V/S = \Theta(k^{1/2})$, $1 \leq k \leq d$.

Proof Sketch: Let $r = d/2 - \epsilon d$ and use the logarithmic form of Stirling's approximation to estimate $\log S$, expanding the logarithms with a power series in ϵ. This leads to $S = 2^{d(1-\Theta(\epsilon^2))}$, implying that $\epsilon = \Theta(k^{-1/2})$ whenever $V = \Theta(2^{d(1-1/k)})$. Considering the cases $\epsilon \leq d^{-1/2}$ and $\epsilon \geq d^{-1/2}$ separately, we can use Lemma 2.2 to establish the desired bound. □

The following theorem is an easy consequence of the preceding discussion.

Theorem 3 `Balance` requires $\Omega(k^{1/2}m)$ time under the weak model if $m = \Theta(p^{1/k}(n/p))$ and $m \geq 2n/p$. □

2.2 Upper Bound

Now consider the task of obtaining an efficient implementation of the `Balance` operation on the weak hypercube model. Let a subcube of dimension 1 be given, with a tokens at processor 0 and b tokens at processor 1. Further assume that each processor knows the number of tokens that it is holding, that is, the value a (b) is stored in the local memory of processor 0 (1). In this case, it is easy to see that the `Balance` operation can be performed over the given subcube in $|a-b|/2 + O(1)$ time. This motivates the following definition.

Definition 2.5 Let a set of n tokens be arbitrarily distributed over the processors of a p processor hypercube, $p \geq 2$. Let the `Balance` operation be applied to each of the $p/2$ subcubes of dimension 1 induced by the set of $p/2$ edges across dimension i. We say that the hypercube has been *balanced across dimension i*.

Clearly the amount of time required to balance across dimension i is $\xi/2 + O(1)$, where ξ is the maximum discrepancy between the number of tokens at a given processor and its neighbor in dimension i.

Lemma 2.3 Let the low and high subcubes with respect to dimension i of a given hypercube be balanced with error ξ. Then after balancing across dimension i, the entire hypercube is balanced with error $\xi + 1$. □

By repeated application of Lemma 2.3 we find that successively balancing across every dimension of the hypercube yields an implementation of `Balance` with error $\log p$. The running time of this algorithm is $O(m \log p)$, since no processor will ever contain more than m tokens and hence each balancing step requires at most $m/2 + O(1)$ time. Unfortunately, it is possible to distribute the tokens so that this performance is achieved to within a constant factor. In other words, the worst-case running time of such a balancing algorithm is $\Theta(m \log p)$. The following algorithm improves on this time bound by making a more careful decomposition of the hypercube. In order to present the algorithm, we require one additional definition.

Definition 2.6 Let the *discrepancy across dimension* i, denoted δ_i, represent the absolute value of the difference between the total number of tokens in the high and low subcubes with respect to dimension i.

The efficiency of the following recursive procedure for `Balance` with error $\log p$ relies upon the fact (shown below) that there is always some dimension with a small associated discrepancy.

Algorithm `Balance`

1. Each processor counts how many tokens it has and stores the result. This takes $O(m)$ time.
2. Let l denote the dimension of the subcube being balanced ($l = d$ initially). If $l = 0$, return.
3. Compute δ_i, $0 \leq i < l$. This involves performing l independent sums over the entire subcube. These sum operations can be pipelined to run in a total of $O(l)$ time [8].
4. Determine the dimension i^* with least associated discrepancy δ_{i^*}. This takes $O(l)$ time.
5. Recursively balance the high and low subcubes with respect to dimension i^*, using steps 2 to 6.
6. Balance across dimension i^*, adjusting the token counts appropriately. The running time of this step is analyzed below.

In order to establish the correctness of the above algorithm we need to show that the output hypercube is balanced with error $\log p$. This follows easily by induction using Lemma 2.3.

To analyze the time complexity, only the cost of step 6 remains to be determined. When this step is executed, the $(l-1)$-dimensional high and low subcubes with respect to dimension i^* are balanced with error $l-1$. Hence, there are integers a and b such that each processor in the high subcube contains a number of tokens that is in the range $[a, a+l-1]$, and each processor in the low subcube contains a number of tokens in the range $[b, b+l-1]$. Letting $\delta = \delta_{i^*}$, we have

$$2^{l-1}(b - (a+l-1)) \leq \delta,$$

and thus $(b+l-1) - a \leq \delta 2^{1-l} + 2l - 2$. We can bound $(a+l-1)-b$ by the same quantity. Therefore, the cost of the balancing step is $O(\delta 2^{-l} + l)$. We now proceed to bound the minimum discrepancy δ. In the following sequence of lemmas, let Δ denote the sum of the discrepancies in the given hypercube of dimension d, $\sum_{0 \leq i < d} \delta_i$.

Lemma 2.4 The value of Δ is maximized by packing all of the tokens into a Hamming ball B of size $\lceil n/m \rceil$.

Proof: Given an arbitrary distribution of the tokens, we show that the corresponding value of Δ is at most that achieved by a particular Hamming ball configuration. First, transform the processor ids of the given hypercube by toggling each bit that corresponds to a dimension for which there are more tokens in the low subcube than in the high subcube. Note that the transformed hypercube yields precisely the same value of Δ as the given hypercube. It has the additional property that for every dimension, the high subcube contains at least as many tokens as the low subcube. Let $w(i)$ denote the number of 1's in the d-bit processor id i, and let $n(i)$ denote the number of tokens at processor i. Now observe that Δ may be expressed as

$$\begin{aligned} \Delta &= \sum_{0 \leq i < p} w(i)n(i) - \sum_{0 \leq i < p} (d - w(i))n(i) \\ &= 2 \sum_{0 \leq i < p} w(i)n(i) - nd, \end{aligned}$$

where $p = 2^d$. Thus, maximizing Δ is equivalent to maximizing $\sum_{0 \leq i < p} w(i)n(i)$. This new sum is clearly maximized by distributing tokens according to the following algorithm: while there are tokens left to distribute, put m tokens (or the number of tokens remaining, if less than m) into an empty processor with largest $w(i)$ in the set of empty processors. The result follows since this algorithm fills a Hamming ball centered at processor $2^d - 1$. ☐

Lemma 2.5 Let an instance of `Balance` be given for which the tokens are packed into a Hamming ball of size $\lceil n/m \rceil$. Then $\Delta = O(md^{1/2}p)$. Furthermore, if $m \geq 2n/p$ and $\lceil n/m \rceil = p^{1-1/k}$, then $\Delta = \Theta(mdk^{-1/2}p^{1-1/k})$.

Proof: Assume the tokens are packed into a Hamming ball B of radius r. The total discrepancy Δ is bounded by m times the number of edges between B and $H_d \setminus B$. The number of such edges is maximized when $r = \lfloor d/2 \rfloor$, so $\Delta = O(md\binom{d}{\lfloor d/2 \rfloor}) = O(md^{1/2}p)$. For the sharper bound, Theorem 2 tells us that the cardinality of the fringe of B is $f(H_d, p^{1-1/k}) = \Theta(k^{-1/2}p^{1-1/k})$. The number of edges between B and $H_d \setminus B$ is $d - r = \Theta(d)$ (r is at most $\lfloor d/2 \rfloor$ since $m \geq 2n/p$) times the size of the fringe of B. Hence, $\Delta = \Theta(mdk^{-1/2}p^{1-1/k})$, as claimed. ☐

In this section, all that we need is the $O(md^{1/2}p)$ bound of Lemma 2.5. The more detailed bound involving k will be used in the next section when we analyze a load balancing algorithm involving multiple token types. The following theorem is an immediate consequence of the preceding two lemmas.

Theorem 4 For any instance of `Balance`, the average discrepancy across a dimension, Δ/d, is $O(md^{-1/2}p)$. □

Recall that the cost of the balancing step in algorithm `Balance` was shown above to be $O(\delta 2^{-l} + l)$, where $\delta = \delta_{i^*}$ is the minimum discrepancy. Now the minimum discrepancy is certainly no larger than the average discrepancy, so δ must be $O(ml^{-1/2}2^l)$ by Theorem 4. Hence, the cost of the balancing step is $O(ml^{-1/2} + l)$, and the total running time of algorithm `Balance` is $O(\sum_{1 \le l \le d} ml^{-1/2} + l) = O(m \log^{1/2} p + \log^2 p)$.

Of course, our algorithm performs balancing with error $\log p$. This should be good enough for most applications but if a minimum error (0 if $p|n$, 1 otherwise) balancing is desired, some post-processing is needed. Note that we can view the post-processing task as an instance of `Balance` with $m = \log p$. An algorithm due to Leighton gives an $O(m \log p)$ implementation of `Balance` with minimum error [11]. The algorithm runs in m phases, and each phase takes care of one token from every processor for which the supply of tokens has not yet been exhausted. In a phase, the designated tokens are routed to a contiguous block (with respect to processor id modulo p) of processors. Each token is routed in exactly one phase. The first block begins at processor 0 (say), and each subsequent block begins at the processor following the end of the previous block. In this manner, the set of tokens will be distributed as evenly as possible.

A single phase of Leighton's algorithm is implemented by performing a prefix sum over the designated tokens followed by a concentration route as defined by Nassimi and Sahni [13]. The prefix sum gives the offset of each token within the contiguous block of destination processors. The size of the block is broadcast so that all processors can compute the start of the next block. All of these operations can be performed in $O(\log p)$ time, so the over-all running time of Leighton's algorithm is $O(m \log p)$. We can now state the main result of this section.

Theorem 5 The `Balance` operation, with minimum error, runs in $O(m \log^{1/2} p + \log^2 p)$ time on the weak model. □

It should be mentioned that Leighton's algorithm is best suited for a particular pipelined model of the hypercube. For a definition of this model, see the work of Varman and Doshi [16]. The prefix sum and broadcast operations can be pipelined on the weak model but the concentration routes cannot (provably). The pipelined model of the hypercube allows concentration routes to be pipelined as well, so that `Balance` can be implemented to run in $O(m + \log p)$ time. Note that the lower bound of Theorem 3 is not violated since it applies to a different computational model.

3 Multiple Types

In this section, we consider a slightly more complicated load balancing problem than `Balance`. Let n tokens be evenly distributed over p processors, that is, there are $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ tokens located at each processor. Each token has an associated *type*. There are $g \ge 2$ different types of tokens, and nothing is known about the distribution or proportion of the tokens of any particular type. We call the set of tokens of a particular type a *group*, and we assume that the g group types are given by the integers $\{0, \ldots, g-1\}$. The problem is to execute g `Balance` (with error ξ) operations, one over each group of tokens. We call this operation `MultiBalance` (with error ξ). Our motivation for considering `MultiBalance` is that it turns out to be useful for sorting, as will be seen in Section 5.

3.1 Upper Bound

We now present an implementation of `MultiBalance` that runs in $O((n/p)(\log g \log p)^{1/2} + g \log^2 p)$ time on the weak hypercube model. The following definitions, which build on Definitions 2.5 and 2.6, are required.

Definition 3.1 Given an instance of `MultiBalance`, we say that the n tokens have been *multi-balanced across dimension* i if and only if each group j has been balanced across dimension i, $0 \le j < g$.

Definition 3.2 Let δ_i^j denote the discrepancy across dimension i with respect to group j, $0 \le j < g$. Define the *multi-discrepancy across dimension* i, denoted δ_i^M, as the sum $\sum_{0 \le j < g} \delta_i^j$.

Algorithm `MultiBalance`

1. Each processor partitions its set of tokens into g subsets, one subset corresponding to each of the g token types. Each of the subsets is counted and the results are stored. This takes $O(n/p)$ time.

2. Let l denote the dimension of the subcube being multi-balanced ($l = d$ initially). If $l = 0$, return.

3. Compute δ_i^M, $0 \le i < l$. This involves performing l independent sums over the entire subcube for each of the g groups. Each set of l sum operations can be pipelined to run in $O(l)$ time [8], so the total running time is $O(gl)$.

4. Determine the dimension i^* with least associated multi-discrepancy $\delta_{i^*}^M$. This takes $O(l)$ time.

5. Recursively multi-balance the high and low subcubes with respect to dimension i^*, using steps 2 to 6.

6. Multi-balance across dimension i^*, adjusting the token counts appropriately. The running time of this step is analyzed below.

The correctness of the preceding implementation of `MultiBalance` with error $\log p$ follows by induction using Lemma 2.3. If a minimum error multi-balancing is desired, we can perform $O(\log^2 p)$ post-processing per group as described in Section 2. The total cost of post-processing is thus $O(g \log^2 p)$ time.

In order to complete the analysis of the running time of algorithm `MultiBalance`, we need to consider the cost of step 6. Let Δ_j denote the sum of the discrepancies with respect to group j in the given hypercube of dimension d, $\sum_{0 \le i < d} \delta_i^j$. Let Δ^M denote the sum of the multi-discrepancies $\sum_{0 \le i < d} \delta_i^M = \sum_{0 \le j < g} \Delta_j$.

Lemma 3.1 For any instance of `MultiBalance`, $\Delta^M = O(dk^{-1/2}n)$, where k satisfies $g = p^{1/k}$.

Proof: Let the number of tokens in group j be denoted x_j, $0 \le j < g$, and consider the contribution of Δ_j to Δ^M. Since $\sum_{0 \le j < g} x_j = n$, there is at most one x_j that exceeds $n/2$. For such an x_j, we have $\Delta_j = O((n/p) d^{1/2} p) = O(d^{1/2} n)$ by Theorem 4. Now $g \ge 2$, so $k \le \log p$, and this contribution is $O(dk^{-1/2}n)$.

Hence, we can assume that $x_j \le n/2$, $0 \le j < g$. Let k_j satisfy $x_j/n = p^{-1/k_j}$, $0 \le j < g$. The tokens of group j can be packed into p^{1-1/k_j} processors, and by Lemmas 2.4 and 2.5, $\Delta_j = O((n/p) d k_j^{-1/2} p^{1-1/k_j}) = O(d x_j k_j^{-1/2}) = O(d^{1/2} x_j \log^{1/2}(n/x_j))$. Consider the function $f(x) = x \log^{1/2}(n/x)$, where we allow x to take on real values over the range $[1, n/2]$. One may easily verify that $f''(x) < 0$ in this range. In other words, $f(x)$ is a concave function. Therefore, the sum $\sum_{0 \le j < g} f(x_j)$, subject to the constraint $\sum_{0 \le j < g} x_j = n$, is maximized when all of the x_j's are equal, that is, when $x_j = n/g$. Forcing the x_j's to be integers can only decrease this sum, so $\Delta^M = O(d^{1/2} g f(n/g)) = O(d^{1/2} g (n/g) \log^{1/2} g) = O(dk^{-1/2}n)$, as required. □

Lemma 3.2 For any instance of `MultiBalance`, the average multi-discrepancy Δ^M/d across a dimension is $O(n(\log g/\log p)^{1/2})$.

Proof: Immediate from Lemma 3.1, since $g = p^{1/k}$ and $k = \log p/\log g$. □

Theorem 6 `MultiBalance`, with minimum error, runs in $O((n/p)(\log g \log p)^{1/2} + g \log^2 p)$ time on the weak model.

Proof: By a simple extension of the argument given in Section 2, the time required to perform the multi-balancing step is $O(\delta_{i^*}^M 2^{-l} + gl)$. Now $\delta_{i^*}^M$ is certainly no more than Δ^M/d, and the number of tokens in a subcube of dimension l at depth $d-l$ of the recursion is $O(n2^{l-d} + g2^l(d-l))$, where the latter term bounds the cumulative effect of odd parity in the balancing operations. Thus, using Lemma 3.2 we find that the total time expended in step 6 is $O\left(\sum_{1 \le l \le d} 2^{-l}(n2^{l-d} + g2^l(d-l))(\log g/l)^{1/2} + gl\right)$, which reduces to $O((n/p)(\log g \log p)^{1/2} + g \log^2 p)$. This dominates the time required by all other steps of the algorithm, including the post-processing. □

3.2 Lower Bound

It is easy to see that $\frac{1}{2}(\Delta^M - dgp)/p$ is a lower bound on the running time of `MultiBalance`, since Δ^M can only decrease by $2p$ each time step and it is certainly less than dgp after the `MultiBalance` operation has been performed. Hence, exhibiting a particular input instance with a high value of Δ^M gives a good lower bound for the worst-case running time of `MultiBalance`. Consider the following input configuration, assuming for convenience that $g = 2^l$, $1 \le l < d$. Divide the d-bit id of each processor into l fields of approximately d/l bits apiece. Let each field of id bits determine a single "super-bit" of an l-bit "condensed" id as follows. If the the majority of the bits in a field are 0, then the corresponding super-bit is a 0, otherwise it is a 1. The condensed ids induce a partition of the set of processors into 2^l equal-sized subsets. For the desired configuration, fill each of the processors having condensed id i with n/p tokens from the ith group, $0 \le i < 2^l$. It can be shown that each group contributes $\Theta((n/g)(\log g \log p)^{1/2})$ to Δ^M, which leads to the following result.

Theorem 7 The `MultiBalance` operation requires $\Omega((n/p)(\log g \log p)^{1/2} - g \log p)$ time on the weak model. □

4 Selection

The `Select` operation is defined as follows. Given n $O(\log p)$-bit values, determine the jth largest and broadcast it to all processors, $0 \leq j < n$. We assume that the n values are initially distributed in an approximately uniform fashion, that is, $O(n/p)$ values are stored at each processor. We assume that $n = O(p^c)$ for some constant c, so that array indices and value counts can be manipulated with a constant number of CPU operations. Finally, we assume that the n values are *distinct*. We make the latter assumption without loss of generality, since ties can always be broken consistently by making use of the following convention.

1. If two values originating at different processors are equal, the one originating from the processor with the higher id is deemed to be larger.

2. If two values originating at the same processor are equal, the one initially stored at the higher memory location is deemed to be larger.

This tie-breaking procedure appends $\lceil \log n \rceil = O(\log p)$ bits to each value, and thus produces at most a constant factor overhead.

Our selection algorithm obtains efficient performance by eliminating from consideration a constant fraction of the values *at every processor* in each iteration. Interestingly, this is accomplished without redistributing the values, in contrast to the best known PRAM algorithm due to Vishkin [17].

Algorithm `Select`

1. At every processor, mark each of the $O(n/p)$ local values "live". This takes $O(n/p)$ time.

2. Let S_i denote the set of live items located at processor i, and let m_i denote the median of S_i. Let $S = \cup_{0 \leq i < p} S_i$ and let $M = \{m_0, \ldots, m_{p-1}\}$. Let $s = \max_{0 \leq i < p} |S_i|$. All of the medians can be obtained in $O(s)$ time using a linear time sequential method. Having found the medians, partition the set S_i into its upper and lower half. Continue this partitioning process to depth $\min\{\log(n/p), \log\log p\}$, that is, until S_i has either been fully sorted or has been split into $\log p$ subsets, each with approximately $s/\log p$ values. Build a binary tree of partition elements to facilitate searching. The total cost of this preprocessing is $O(s \min\{\log(n/p), \log\log p\})$. Given an arbitrary value, we can now determine its rank in S_i in $O(\min\{\log(n/p), \log\log p\} + s/\log p)$ time by locating the correct subset and then looking at every element in that subset. This sequential trade-off between preprocessing time and search time is well understood, see [4] and [10].

3. Find that $m \in M$ with rank in S closest to j (if there is a tie, break it arbitrarily) and broadcast it to all processors. We compute m in time $O(s + \log^2 p)$ as follows. First, let $m'_i = m_i$ at each processor i. Now sort the set $M' = \{m'_0, \ldots, m'_{p-1}\}$ so that $m'_i = m_k$, where m_k has rank i in M. This can be done using bitonic sort in $O(\log^2 p)$ time. Next perform a binary search over the set M' to determine m. This requires $O(\log p)$ rank computations over S, each of which may be performed as follows.

   (a) Let m' be that element of M' whose rank in S is currently required by the binary search. Broadcast m', along with the id of the processor that it is stored in, to all processors. This takes $O(\log p)$ time.

   (b) At each processor i, compute the rank r_i of m' in S_i. As observed above, the preprocessing performed in step 2 allows this to be done in $O(s/\log p + \min\{\log(n/p), \log\log p\})$ time.

   (c) Sum the r_i values to obtain the rank of m' in S. This takes $O(\log p)$ time.

4. Let j' be the rank of m in S. If $j' = j$, return m and halt.

5. At each processor i, kill off those elements of S_i that cannot possibly have rank j in S. Let r_i be the rank of m in S_i as computed in step 3b. Assume that $j' < j$; the case $j' > j$ is similar. We can certainly eliminate all of the elements in S_i with ranks in S_i less than or equal to r_i. If $m \geq m_i$, note that this has eliminated at least half of the elements of S_i. If $m < m_i$, we can also eliminate those elements of S_i with ranks greater than or equal to that of m_i. The justification for this is as follows: all of these elements have ranks in S that are at least as large as the rank of m_i in S, and the rank of m_i in S must be larger than j to avoid contradicting our choice of m. Once again, and hence in all cases, the number of live elements in S_i is reduced by at least a factor of 2. Given the preprocessing performed in step 2, this step can be performed in $O(s/\log p + \min\{\log(n/p), \log\log p\})$ time.

6. Set j to $j - \Delta$, where Δ is the total number of values eliminated in step 5 because their rank in S had to be less than j. Note that Δ can

be computed and broadcast to all processors in $O(\log p)$ time. Go to step 2.

From the above analysis, each iteration of steps 2 through 6 runs in time $O(s \min\{\log(n/p), \log\log p\} + \log^2 p)$. Since each $|S_i|$ initially has size $O(n/p)$ and is cut at least in half every iteration, after $O(\log(n/p))$ iterations every S_i will contain at most one element, and the next m computed in step 3 will have rank j in S. Hence, the algorithm terminates within $O(\log(n/p))$ iterations. Since s decreases geometrically from an initial value of n/p, the total running time is $O((n/p)\min\{\log(n/p), \log\log p\} + \log^2 p \log(n/p)) = O((n/p)\log\log p + \log^2 p \log(n/p))$.

Note that the $(n/p)\log\log p$ term in the running time is entirely due to the cost of local preprocessing, as opposed to communication. The results of this section are summarized by the following theorem.

Theorem 8 The `Select` operation runs on the weak model in $O((n/p)\log\log p + \log^2 p\log(n/p))$ time. If the values are given in locally sorted form, `Select` runs in $O(\log^2 p \log(n/p))$ time.

The communication component of algorithm `Select` can be expressed in terms of the primitive operations: sort p values, broadcast and sum. Thus, it adapts easily to a variety of networks. However, simulating Vishkin's algorithm yields a faster selection algorithm on any network family (eg. certain expander graphs, see [14]) that can implement the `Balance` operation of Section 2 in $o(m \log\log p)$ time.

5 Sorting

The `Sort` operation is defined as follows. Given n $O(\log p)$-bit values, distributed $O(n/p + 1)$ per processor, rearrange the n values so that they are distributed uniformly (that is, balanced with minimum error) and every value in processor i is less than or equal to every value in processor j whenever $0 \le i < j < p$. In addition, the set of values within any particular processor should be sorted. As for the `Select` operation, we assume that $n = O(p^c)$ for some constant c, and that the n values are distinct.

By making use of the `Balance` and `Select` operations developed earlier, we obtain the following straightforward sorting algorithm. We restrict attention to the case $n = \Omega(p)$.

Algorithm `QuickSort`

1. If the dimension of the hypercube being sorted is 0, locally sort the $O(n/p)$ values located at each processor, and return. If performed, this operation takes $O((n/p)\log(n/p))$ time.

2. Let S denote the set of n values. Call `Select` to find the value with rank $\lceil n/2 \rceil$ in S. Let this value be m. Using algorithm `Select` this takes time $O((n/p)\log\log p + \log^2 p \log(n/p))$.

3. Route all values that are strictly less than m to the low subcube. Route all values that are greater than or equal to m to the high subcube. To do this each processor splits the sorted list that it currently holds into two sorted sublists and sends the appropriate sublist to its neighbor in the highest dimension. This takes $O(n/p)$ time.

4. At this point, each processor contains between 0 and $2n/p$ values. Call `Balance` to smooth out the load, that is, to redistribute the values so that each processor holds a list of length n/p. This takes $O((n/p)\log^{1/2} p + \log^2 p)$ time.

5. Sort the low and high subcubes recursively.

The correctness of the preceding `QuickSort` algorithm should be obvious. The overall time complexity of `QuickSort` is readily seen to be

$$O((n/p)\log^{3/2} p + \log^3 p \log(n/p)).$$

We now describe the `SmoothSort` sorting algorithm, which is also designed to run on the weak model. It makes use of the `MultiBalance` operation described in Section 3.

Algorithm `SmoothSort`

1. Locally sort the $O(n/p)$ values located at each processor. This takes $O((n/p)\log(n/p))$ time.

2. If the dimension of the hypercube being sorted is 0, return.

3. Determine the 2^l values with ranks $\lfloor \frac{in}{2^l} \rfloor$, $0 \le i < 2^l$, and broadcast them to all processors. We call these *splitter values*. An appropriate choice for the parameter l will be specified later. For now we assume only that $2^l \le n/p$. These selections can each be performed in $O(\log^2 p \log(n/p))$ time as described in Section 4, since the values have been sorted locally. Each broadcast takes $O(\log p)$ time. Thus, the total time required for this step is $O(2^l \log^2 p \log(n/p))$. Each processor now contains a sorted list of n/p values and a sorted list of 2^l splitter values.

4. The 2^l splitter values naturally partition the n values into 2^l groups. The ith group consists of those values with ranks between $\lfloor \frac{in}{2^l} \rfloor$ and

$\left\lfloor \frac{(i+1)n}{2^l} \right\rfloor - 1$ inclusive, $0 \leq i < 2^l$. At each processor, label each of the n/p local values with the appropriate l-bit group number. Since the list of values and the list of splitter values are sorted, this takes $O(2^l + n/p) = O(n/p)$ time.

5. Call `MultiBalance` to smooth out each of the 2^l groups of tokens. We have $g = 2^l$ so this takes $O((n/p)(l \log p)^{1/2} + 2^l \log^2 p)$ time.

6. Loop over the high order l dimensions, routing each group to the appropriate subcube. This takes $O(ln/p)$ time.

7. Call `Balance` to smooth out the load in each of the 2^l subcubes. The error in these subcubes is at most 2^l, so this takes $O(2^l \log^{1/2} p + \log^2 p)$ time. **Remark:** If n/p is a power of 2 then there is no error, and this step can be omitted.

8. Sort the 2^l subcubes recursively.

It remains to determine the value of l that minimizes the total running time of `SmoothSort` subject to the constraints $l \geq 1$, $2^l \leq n/p$ and $l \leq \log p$. From the analysis accompanying the description of the algorithm, we find that the cost of the top level of the recursion (that is, excluding the recursive calls) is dominated by an expression of the form

$$O\left(2^l \log^2 p \log(n/p) + (n/p)(l \log p)^{1/2}\right),$$

for all valid choices of l. The running time of `SmoothSort` is minimized (to within a constant factor) by increasing l to the point where the cost of performing the selections balances the cost of the `MultiBalance` operation. This leads to setting $l = \max(1, \lfloor \log(n/(pq)) \rfloor)$, where $q = \log^{3/2} p \log\log p$. Substituting this choice of l into the above expression, we find that the cost of a given level of the recursion is a function of p and n/p. The value of n/p does not vary with the depth of the recursion, while p is halved at each level. If $n \leq pq$, then l is forced to 1 and the running time of `SmoothSort` is the same as that of `QuickSort`. If $n \geq pq$, we have $2^l = \Theta(n/(pq))$ and the depth of the recursion is $\Theta((\log p)/l)$. Furthermore, the cost of any level is at most that of the top level, so the total running time of `SmoothSort` is

$$O\left((n/p) \log p \sqrt{\frac{\log p}{\log(n/(pq))}} + \log^3 p \log(n/p)\right),$$

Note that the deviation from optimality (that is, from the time required by the sequential lower bound) for $n \geq pq$ is given by the square root factor. Previously, the best known algorithm for this range was `CubeSort`, which deviates from the lower bound by a factor of $\log p / \log(n/p)$.

Table 2 summarizes the running times of the best known sorting algorithms for the weak model of the hypercube, over specific ranges of n/p. The entries are arranged in ascending order of n/p. `MergeSort` is listed first because it is the best known sorting method (in the sense of asymptotic complexity) when $n \ll p$. The last column indicates that `MergeSort` remains the best known algorithm up to $n = \Theta(p)$, at which point `BitonicSort` has the same complexity. As the ratio n/p is increased beyond a constant, we soon attain the region in which `CubeSort` overtakes `BitonicSort`. Computing this point requires some care, however. The formula given in the table for `CubeSort` only holds if $n/p = \Omega(\log^{(k)} p)$ for some constant k, where $\log^{(k)} p$ denotes the kth iterated logarithm. If k is not held constant, there is an additional c^k factor in the running time, for some constant $c > 1$. This puts the transition region at $n/p = \Theta(\log^{(k)} p)$ with k satisfying $c^k = \log(n/p) = \log^{(k+1)} p$. This occurs when k is roughly $\log^* p - \Theta(\log^* \log^* p)$!

The next entry in the table indicates that the running time of `SmoothSort` improves upon that of `CubeSort` when n exceeds $\Theta(p \log p (\log\log p)^2)$. The reason that `CubeSort` performs faster close to $n = p$ is that `SmoothSort` has the additive $O(\log^3 p \log(n/p))$ term in its complexity. This additive term becomes insignificant at $n = \Theta(pq)$, where q is as defined above. For larger values of n, the running time of `SmoothSort` is given by the last entry in the table.

Finally, it should be mentioned that Table 2 does not present the entire picture. Since `CubeSort` is a sorting algorithm that performs a large sort by doing many smaller sorts, it turns out that a hybrid algorithm which uses `CubeSort` at the top-level and `SmoothSort` to perform the smaller sorts can be used to obtain improvements over at least a portion of the range $p \leq n \leq pq$. For instance, at $n = p \log p$ the hybrid algorithm has a running time of $O(\log^{8/3+\epsilon} p)$, where ϵ is an arbitrarily small positive constant, as opposed to the $O(\log^3 p / \log\log p)$ running time of `CubeSort`. A full analysis of the hybrid algorithm remains to be performed.

6 Concluding Remarks

Many algorithms for the weak model of the hypercube can be adapted to run within the same time bound on the perfect shuffle. This is true for `BitonicSort` and `CubeSort`, for example. Unfortunately, it is not

Algorithm	*Running Time*	*Transition Region*
`MergeSort`	$O(\log^2 p/\log(p/n))$	$n = \Theta(p)$
`BitonicSort`	$O((n/p)\log^2 p)$	see text
`CubeSort`	$O((n/p)\log^2 p/\log(n/p))$	$n = \Theta(p\log p(\log\log p)^2)$
`SmoothSort`	$\Theta(\log^3 p\log\log p)$	$n = \Theta(pq)$
`SmoothSort`	$O((n/p)\log^{3/2} p/\log^{1/2}(n/(pq))$	

Table 2: Running times of sorting algorithms for the weak model.

true for the `Balance` and `MultiBalance` algorithms given in this paper. The reason is that these algorithms exploit the ability of the hypercube to communicate along its dimensions in an arbitrary order. The perfect shuffle versions of these algorithms must balance along each of the dimensions in either ascending or descending order. Doing this, we find that `SmoothSort` is easy to implement on the perfect shuffle, though its asymptotic performance degrades to that of `CubeSort`. On the other hand, it appears that `SmoothSort` would remain faster than `CubeSort` by a significant constant factor in terms of worst case complexity.

In contrast with both `BitonicSort` and `CubeSort`, `SmoothSort` is not an oblivious algorithm and its average case behavior on the hypercube can be shown to be optimal for $n \geq p\log^{2+\epsilon} p$, $\epsilon = \Omega(1)$. By simultaneously guaranteeing good worst-case performance, `SmoothSort` avoids the potential pitfalls of a simpler scheme such as `HyperQuickSort` [18]. For solving the related problem of permutation routing, `SmoothSort` is even more practical because the cost of performing selections goes away. On the perfect shuffle, `SmoothSort` performs permutation routing in $O((n/p)\log^2 p/\log(n/p))$ time. The constant hidden by the O-notation is small and, unlike `CubeSort`, this bound holds for all $n \geq p$. With regard to permutation routing on the hypercube, an interesting open question is the following: is there an algorithm for the `Balance` operation that runs in $O(mf(p) + \log p)$ time for some non-constant function $f(p)$? Such a result would imply an $O(\log^2 p/\log f(p)) = o(\log^2 p)$ permutation routing algorithm for the case $n = p$, that is, faster than `BitonicSort`.

Finally, it is clear that the time bounds for our algorithms are intimately related to the expansion properties of the hypercube. It would be interesting to investigate this connection in a more general setting.

7 Acknowledgments

I would like to thank Sherry Listgarten, Tom Mathies, Ernst Mayr, Jorge Sanz and Alex Wang for their valuable comments on this work. In particular, Sherry provided the elegant proof of Lemma 2.4.

References

[1] A. Aggarwal and M.-D. A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks. In J. H. Reif, editor, *Lecture Notes in Computer Science: VLSI Algorithms and Architectures (AWOC 88),* vol. 319, pages 339–350. Springer-Verlag, 1988.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

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

[4] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[5] R. E. Cypher and J. L. C. Sanz. Cubesort: An optimal sorting algorithm for feasible parallel computers. In J. H. Reif, editor, *Lecture Notes in Computer Science: VLSI Algorithms and Architectures (AWOC 88),* vol. 319, pages 456–464. Springer-Verlag, 1988.

[6] P. Frankl and Z. Füredi. A short proof for a theorem of Harper about Hamming spheres. *Discrete Mathematics*, 34:311–313, 1981.

[7] L. Harper. Optimal numberings and isoperimetric problems on graphs. *J. Combinatorial Theory*, 1:385–393, 1966.

[8] C.-T. Ho and S. L. Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, pages 640–648, 1986.

[9] S. L. Johnsson. Combining parallel and sequential sorting on a Boolean n-cube. In *Proceedings of the 1984 IEEE International Conference on Parallel Processing*, pages 444–448, 1984.

[10] R. M. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17:883–902, 1988.

[11] F. T. Leighton. Personal communication.

[12] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34:344–354, 1985.

[13] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM*, 29:642–667, 1982.

[14] D. Peleg and E. Upfal. The token distribution problem. *SIAM J. Comput.*, 18:229–243, 1989.

[15] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *JACM*, 34:60–76, 1987.

[16] P. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. Technical Report TR–8802, Rice University, Department of Electrical and Computer Engineering, February 1988.

[17] U. Vishkin. An optimal parallel algorithm for selection. Technical Report 106, Courant Institute of Mathematical Sciences, Department of Computer Science, December 1983.

[18] B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 292–299, 1986.