Operation	$r = 0,\ m = \Theta(1)$	$r \geq 0,\ m = \Theta(1)$	$r \geq 0,\ m = \Omega(1)$
$t_{BPC}(a,d)$	$\Theta(2^d)$	$\Theta(2^{d-r})$	$\Theta(2^{d-r})$
$t_{CYC}(a,d)$	$\Theta(2^d)$	$\Theta(2^{d-r})$	$\Theta(2^{d-r})$
$t_{RR}(a,d)$	$\Theta(2^d)$	$\Theta(2^{d-r})$	$\Theta(2^{d-r})$
$t_{SS}(a,d)$	$\Theta(2^d)$	$\Theta(2^{d-r})$	$\Theta(2^{d-r})$
$T_{OP}(a,b,d)$	$\Theta(2^{d-b})$	$\Theta(2^{d-b-r})$	$\Theta(2^{d-b-r})$
$S(a,b,d)$	$\Theta((d/b) \cdot 2^{d-b})$	$\Theta((d-r) \cdot 2^{d-b-r}/b)$	$\Theta((d-r) \cdot 2^{d-b-r}/(b+m))$

Table 5: I/O complexities for the *P–DISK* model.

storage behaves like a RAM), and (ii) a zero cost assumption. The motivation for the zero cost assumption is that, for computational problems involving large amount data (e.g., much more than can fit in the internal storage), it is often found that the fastest algorithm is the one which performs fewer I/Os. By making the zero cost assumption, we focus our attention entirely on minimizing I/O complexity.

The disk model has a parallel version *P–DISK*. (*P–DISK* is to the disk model as $\mathcal{G}_{par}$ is to $\mathcal{G}_{seq}$.) Our goal in this section is to determine the complexity of sorting in the *P–DISK* model. Following previous authors, we will focus on the I/O complexity measure. We begin by focusing our attention on the important special case where $r = 0$ and $m = \Theta(1)$.

Because we wish to calculate I/O complexity instead of running time, we need to make a slight change to our general scheme of analysis. In particular, Equation (2) should be changed to:

$$T_{OP}(a, b, d) = O(\max_{b \le d' \le d}(t_{PRIM}(a - b, d - b) + t_{PRIM}(a - b, d' - b) \cdot 2^{d-d'})).$$

(Note that we have dropped the term corresponding to computation over the fixed-connection network, as it does not contribute to the I/O complexity.)

We now define the mapping of cubes to secondary storage that will be employed to emulate $\mathcal{G}_{seq}(a)$. Recall that a machine in $\mathcal{G}_{seq}(a)$ has $a + 1$ associated cubes, one of each dimension d, $0 \le d \le a$. For this purpose, we assume that the blocks of secondary storage are numbered from 0, and map the dimension-d cube to the contiguous block of memory locations $[2^d, 2^{d+1})$, $0 \le d \le a$. It is now trivial to determine the complexities of the four primitive operations. The remaining steps in the calculation of $S(a, b, d)$ are purely mechanical. Our results for $r = 0$ and $m = \Theta(1)$ are summarized in the first column of Table 5.

It is not difficult to generalize our approach to handle arbitrary r and m. First let us consider the case of arbitrary r and $m = \Theta(1)$. This case can be easily reduced to the $r = 0$, $m = \Theta(1)$ case by simply performing all of the analysis in terms of numbers of blocks of values (as opposed to numbers of values), and then correcting appropriately. Our results for arbitrary r and $m = \Theta(1)$ are summarized in the second column of Table 5. Our bounds on the running times of the four primitive operations assume that $r \le d$, and our remaining bounds assume that $r \le d - b$.

Now let us consider the case of arbitrary r and $m = \Omega(1)$. This case is similar to the preceding one, except that for m sufficiently large we can improve the sorting I/O bound by cutting off the recurrences of Equation (3) through Equation (6) at $d = O(b + m)$ instead of $d = O(b)$. This modification leads to the bounds stated in the third column of Table 5. The matching lower bound for sorting is established in [3].

Remark: The above modificiations used to handle arbitrary r and m could have been avoided by incorporating the parameters r and m into our generic model of multi-level storage. In order to simplify our earlier presentation, however, we chose not to clutter the generic model with these additional parameters.

Operation	$f(\ell)=1$	$f(\ell)=1/(\ell+1)$	$f(\ell)=\rho^{-c\ell}$, $c>0$
$t_{BPC}(a,d)$	$\Theta(2^d)$	$\Theta(d\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{CYC}(a,d)$	$\Theta(2^d)$	$\Theta(d\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{RR}(a,d)$	$\Theta(2^d)$	$\Theta(d\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{SS}(a,d)$	$\Theta(2^d)$	$\Theta(d\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$T_{OP}(a,b,d)$	$\Theta(b\cdot 2^{d-b})$	$\Theta(d\cdot 2^{d-b})$	$\Theta(2^{(c/2+1)(d-b)}+b\cdot 2^{d-b})$
$S(a,b,d)$	$\Theta(d\cdot 2^{d-b})$	$O((\lg d-\lg b)\cdot d\cdot 2^{d-b})$	$\Theta(2^{(c/2+1)(d-b)}+d\cdot 2^{d-b})$
$S^*(n,p)$	$\Theta((n\lg n)/p)$	$O((n\lg n)(\lg\lg n-\lg\lg p)/p)$	$\Theta((n/p)^{c/2+1}+(n\lg n)/p)$

Table 3: Running times for *P–UMH* and *P–RUMH*. The two O-bounds are known to be tight for *P–RUMH*.

Operation	$f(\ell)=1$	$f(\ell)=1/(\ell+1)$	$f(\ell)=\rho^{-c\cdot\ell}$, $c>0$
$t_{BPC}(a,d)$	$\Theta(d\cdot 2^d)$	$\Theta(d^2\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{CYC}(a,d)$	$\Theta(d\cdot 2^d)$	$\Theta(d^2\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{RR}(a,d)$	$\Theta(2^d)$	$\Theta(d\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$t_{SS}(a,d)$	$\Theta(d\cdot 2^d)$	$\Theta(d^2\cdot 2^d)$	$\Theta(2^{(c/2+1)\cdot d})$
$T_{OP}(a,b,d)$	$\Theta(d\cdot 2^{d-b})$	$\Theta(((d-b)^2+b)\cdot 2^{d-b})$	$\Theta(2^{(c/2+1)(d-b)}+b\cdot 2^{d-b})$
$S(a,b,d)$	$\Theta((\lg d-\lg b)\cdot d\cdot 2^{d-b})$	$\Theta(((d-b)^2+d)\cdot 2^{d-b})$	$\Theta(2^{(c/2+1)(d-b)}+d\cdot 2^{d-b})$
$S^*(n,p)$	$\Theta((n\lg n)(\lg\lg n-\lg\lg p)/p)$	$\Theta((n/p)(\lg^2(n/p)+\lg n))$	$\Theta((n/p)^{c/2+1}+(n\lg n)/p)$

Table 4: Running times for *P–SUMH*.

B.2 Results for the *P–SUMH* Model

The complexity of the four primitive operations is not too difficult to determine for all instances of the *P–SUMH* model that we consider. Our results for $f(\ell)=1$, $f(\ell)=1/(\ell+1)$, and $f(\ell)=\rho^{-c\cdot\ell}$, $c>0$, are summarized in Table 4.

The sorting lower bounds are all non-trivial and are established in [19].

C Parallel Disk Model

The sequential disk model is a two-level model: A processor has a a "slow" external storage partitioned into blocks of size 2^r, $r\geq 0$, and a "fast" internal storage that can hold up to 2^m blocks, $m=\Omega(1)$. Data can only be read from (resp., written to) the external storage one block at a time. We call each such read or write of a block an *I/O operation*, or simply an *I/O*. There is a certain fixed cost associated with each I/O. The minimum number of I/Os needed to solve a given problem is referred to as the I/O complexity of the problem.

Two different assumptions are commonly made with regard to the complexity of accessing locations of the internal storage: (i) a unit cost assumption (in this model the internal

B Uniform Memory Hierarchies

The *uniform memory hierarchy* (*UMH*) model was defined by Alpern, Carter, and Feig [4]. The model is parameterized by positive integers α and ρ, and a monotone, non-increasing *bandwidth function* f. The memory is partitioned into *levels*, and each level is in turn partitioned into *blocks*. The number of locations in each block at level ℓ, $\ell \geq 0$, is ρ^{ℓ}. (We will assume that $\rho > 1$ so that the size of blocks at successive levels increases geometrically.) The number of blocks at level ℓ is $\alpha \cdot \rho^{\ell}$. Thus, the total number of storage locations at level ℓ is $\alpha \cdot \rho^{2\ell}$. In the *UMH* model, data stored at level ℓ can only be directly moved to either level $\ell+1$ or level $\ell-1$. (We can think of the processor as residing at level 0. Of course, information is never passed down from level 0 or up from the highest level.)

Data is passed up from level ℓ to level $\ell+1$ in level-ℓ blocks. When a level-ℓ block arrives at level $\ell+1$, it is placed into a *subblock* of some block at level $\ell+1$. (Each block at level $\ell+1$ is partitioned into ρ subblocks of size ρ^{ℓ}.) The cost of transferring a level-ℓ block of data from level ℓ to level $\ell+1$ is $\rho^{\ell}/f(\ell)$.

Data is passed down from level $\ell+1$ to level ℓ in level-$(\ell+1)$ subblocks. When a level-$(\ell+1)$ subblock arrives at level ℓ, it is placed into a block at level ℓ. The cost of transferring a level-ℓ block of data from level $\ell+1$ to level ℓ is $\rho^{\ell}/f(\ell)$.

Distinct block transfers between adjacent levels cannot overlap in time. On the other hand, the *UMH* model does allow simultaneous transfers between distinct pairs of levels. We will also consider two variants of the *UMH* model that have been proposed by Nodine and Vitter [17]: *sequential UMH* (*SUMH*), and *random-access UMH* (*RUMH*). The *SUMH* model is the same as *UMH* except that it disallows simultaneous transfers of any kind. The *RUMH* is a much less restrictive variant of the original *UMH* model; we refer the reader to [17] for the definition of this model.

The *UMH*, *RUMH*, and *SUMH* models have parallel versions *P–UMH*, *P–RUMH*, *P–SUMH*, respectively. (For example, *P–UMH* is to *UMH* as $\mathcal{G}_{par}$ is to $\mathcal{G}_{seq}$.)

We now define the mapping of cubes to memory that will be employed (for all variants of *UMH*) to emulate $\mathcal{G}_{seq}(a)$. Recall that a machine in $\mathcal{G}_{seq}(a)$ has $a+1$ associated cubes, one of each dimension d, $0 \leq d \leq a$. We map the dimension-d cube to the smallest level that will hold it, i.e., to level ℓ where ℓ is the least integer such that $\alpha\rho^{2\ell} \geq 2^{d}$. (Within the level, we map the cube to the smallest possible number of blocks.) For each of the models that we consider, it remains to determine the complexities of the four primitive operations. The remaining steps in the calculation of $S^{*}(n,p)$ are purely mechanical, and have been omitted.

B.1 Results for the *P–UMH* and *P–RUMH* Models

The complexity of the four primitive operations is not too difficult to determine for all instances of the *P–UMH* and *P–RUMH* model that we consider. The results for $f(\ell) = 1$, $f(\ell) = 1/(\ell+1)$, and $f(\ell) = \rho^{-c\cdot\ell}$, $c > 0$, are summarized in Table 3.

The sorting lower bounds for $f(\ell) = 1/(\ell+1)$ (*P–RUMH* only) and $f(\ell) = \rho^{-c\ell}$, $c > 0$, are non-trivial and are established in [19].

Operation	$P\text{–}HMM_{\lg x}$	$P\text{–}HMM_{x^\alpha}$, $\alpha > 0$
$t_{BPC}(a,d)$	$\Theta(d \cdot 2^d)$	$\Theta(2^{(\alpha+1)\cdot d})$
$t_{CYC}(a,d)$	$\Theta(d \cdot 2^d)$	$\Theta(2^{(\alpha+1)\cdot d})$
$t_{RR}(a,d)$	$\Theta(d \cdot 2^d)$	$\Theta(2^{(\alpha+1)\cdot d})$
$t_{SS}(a,d)$	$\Theta(d \cdot 2^d)$	$\Theta(2^{(\alpha+1)\cdot d})$
$T_{OP}(a,b,d)$	$\Theta(d \cdot 2^{d-b})$	$\Theta(2^{(\alpha+1)(d-b)} + b \cdot 2^{d-b})$
$S(a,b,d)$	$\Theta((\lg d - \lg b) \cdot d \cdot 2^{d-b})$	$\Theta(2^{(\alpha+1)(d-b)} + d \cdot 2^{d-b})$
$S^*(n,p)$	$\Theta((n \lg n)(\lg\lg n - \lg\lg p)/p)$	$\Theta((n/p)^{\alpha+1} + (n \lg n)/p)$

Table 1: Running times for $P\text{–}HMM$.

Operation	$P\text{–}BT_{x^\alpha}$, $0 < \alpha < 1$	$P\text{–}BT_x$	$P\text{–}BT_{x^\alpha}$, $\alpha > 1$
$t_{BPC}(a,d)$	$\Theta((\lg d) \cdot 2^d)$	$\Theta(d^2 \cdot 2^d)$	$\Theta(2^{\alpha \cdot d})$
$t_{CYC}(a,d)$	$\Theta(2^d)$	$\Theta(2^d)$	$\Theta(2^{\alpha \cdot d})$
$t_{RR}(a,d)$	$\Theta((\lg d) \cdot 2^d)$	$\Theta(d^2 \cdot 2^d)$	$\Theta(2^{\alpha \cdot d})$
$t_{SS}(a,d)$	$\Theta((\lg d) \cdot 2^d)$	$\Theta(d \cdot 2^d)$	$\Theta(2^{\alpha \cdot d})$
$T_{OP}(a,b,d)$	$\Theta(b \cdot 2^{d-b})$	$\Theta(((d-b)^2 + b) \cdot 2^{d-b})$	$\Theta(2^{\alpha \cdot (d-b)} + b \cdot 2^{d-b})$
$S(a,b,d)$	$\Theta(d \cdot 2^{d-b})$	$\Theta(((d-b)^2 + d) \cdot 2^{d-b})$	$\Theta(2^{\alpha \cdot (d-b)} + d \cdot 2^{d-b})$
$S^*(n,p)$	$\Theta((n \lg n)/p)$	$\Theta((n/p)(\lg^2(n/p) + \lg n))$	$\Theta((n/p)^\alpha + (n \lg n)/p)$

Table 2: Running times for $P\text{–}BT$.

[19] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:41–57, 1993.

[20] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 159–169, May 1990. To appear in *Algorithmica*.

A The Hierarchical Memory Model

The *hierarchical memory model* (*HMM*) was defined by Aggarwal, Alpern, Chandra and Snir [1]. This model is equivalent to the usual sequential RAM model except that the cost of reading/writing location x is $f(x)$ instead of 1, where f is some nondecreasing function. (The RAM model is the same as *HMM* with $f(x) = 1$.) We will consider the following two important special cases of *HMM*: $HMM_{\lg x}$ and HMM_{x^α}, $\alpha > 0$.

The *hierarchical memory model with block transfer* (*BT*) was defined by Aggarwal, Chandra, and Snir [2]. In this model, access to location x costs $f(x)$ (as in *HMM*), but contiguous blocks of locations can be copied more cheaply. In particular, the contiguous block of $\ell + 1$ locations $[x - \ell, x]$ can be copied to any disjoint block $[y - \ell, y]$ at a cost of $f(\max\{x, y\}) + 1$. We will consider the following three important special cases of *BT*: BT_{x^α}, $0 < \alpha < 1$, BT_x, and BT_{x^α}, $\alpha > 1$.

The *HMM* and *BT* models have parallel versions *P–HMM* and *P–BT*, respectively. (For example, *P–HMM* is to *HMM* as $\mathcal{G}_{par}$ is to $\mathcal{G}_{seq}$.)

We now define the mapping of cubes to memory that will be employed (for all variants of *HMM* and *BT*) to emulate $\mathcal{G}_{seq}(a)$. Recall that a machine in $\mathcal{G}_{seq}(a)$ has $a + 1$ associated cubes, one of each dimension d, $0 \le d \le a$. We map the dimension-d cube to the contiguous block of memory locations $[2^d, 2^{d+1})$, $0 \le d \le a$. For each of the models that we consider, it remains to determine the complexities of the four primitive operations. The remaining steps in the calculation of $S^*(n, p)$ are purely mechanical, and have been omitted.

A.1 Results for the *P–HMM* Model

The complexity of the four primitive operations is easy to determine for all variants of the *P–HMM* model. Our results for $P\text{–}HMM_{\lg x}$ and $P\text{–}HMM_{x^\alpha}$, $\alpha > 0$, are summarized in Table 1.

A.2 Results for the *P–BT* Model

The complexity of the four primitive operations is non-trivial to determine for some variants of the *P–BT* model. Fortunately, though, the *P–BT* complexity of many operations is known [2], and these existing bounds readily imply tight bounds for each of the four primitive operations. Our results for $P\text{–}BT_{x^\alpha}$, $0 < \alpha < 1$, $P\text{–}BT_x$, and $P\text{–}BT_{x^\alpha}$, $\alpha > 1$, are summarized in Table 2. (Note that we have not considered $P\text{–}BT_{\lg x}$ because we already obtain a sorting bound of $O((n \lg n)/p)$ for $P\text{–}BT_x$. We cannot hope to improve this bound with any comparison-based sort.)

[4] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 600–608, October 1990.

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

[6] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26–47, 1992.

[7] T. H. Cormen. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 130–139, July 1993.

[8] T. H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17:41–57, 1993.

[9] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, May 1990. To appear in *JCSS*.

[10] R. E. Cypher and C. G. Plaxton. Techniques for shared key sorting. Technical report, IBM Almaden Research Center, March 1990.

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

[12] R. W. Floyd. Permuting information in idealized two-level storage. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, New York, 1972.

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

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

[15] D. Nassimi and S. Sahni. A self routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers*, C–30:332–340, 1981.

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

[17] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[18] M. H. Nodine and J. S. Vitter. Deterministic distibution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, July 1993.

basic operations. Instead, a sparse sorting routine should be implemented directly. Similar comments apply to the Sharesort subroutine FindSplitters.

The reader may have questioned the significance of the particular exponents appearing in the Sharesort recurrences. These were chosen for consistency with the original presentation of Sharesort, but other possibilities exist. In general, we can hope to make use of "nicer" exponents (in terms of the constant factors induced) on computational models having a more powerful row routing capability.

To make a fair comparison between the practical performance of Sharesort and that of Balance Sort, it would first be necessary to fix a particular model and then compare optimized versions of both algorithms. This has yet to be done. One point of contrast may be worth mentioning, however, for the case where the processors happen to be interconnected by a hypercubic network. While both algorithms make use of sorting operations over the interconnection network, Sharesort applies such operations only to "actual" keys (i.e., keys of the original sorting problem). All of the other operations performed over the interconnection network by Sharesort are standard, logarithmic-time operations (e.g., prefix, monotone route). In contrast, Balance Sort makes use of sorting over the interconnection network for both PRAM emulation as well as sorting of actual keys.

It is possible that both Sharesort and Balance Sort are impractical in their strict deterministic form: The most practical algorithms for sorting in parallel models of multi-level storage may well be randomized.

9 Concluding Remarks

Sharesort may be useful in other parallel sorting applications where some form regularity is enforced (e.g., in an environment where BPC permutations are less expensive than arbitrary permutations).

Finally, the set of primitives underlying Sharesort also underlies a host of other algorithms for hypercubic networks. Do many or all of these algorithms for hypercubic networks map to optimal algorithms for parallel models of multi-level storage? This is an intriguing question for further investigation.

References

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, May 1987.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, October 1987.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31:1116–1127, 1988.

3. Map the cubes of $\mathcal{G}_{seq}(a)$ to the specific storage locations in M. The mapping should be chosen in order to minimize $T_{OP}(a,b,d)$.

4. Determine the complexity of the four primitive operations: $t_{BPC}(a,d)$, $t_{CYC}(a,d)$, $t_{RR}(a,d)$, and $t_{SS}(a,d)$. This step is quite model-specific but will generally be straightforward. Note that the four primitive operations are sequential. Their complexities are determined with respect to machine M.

5. Determine $T_{OP}(a,b,d)$. Substitute the functions computed in the previous step into Equations (1) and (2).

6. Determine $M(a,b,d)$. Solve the recurrence of Equations (3) and (4).

7. Determine $S(a,b,d)$. Solve the recurrence of Equations (5) and (6).

8. Determine $S^*(n,p)$. In order to present the complexity of our Sharesort implementation in more usual terms, we compute

$$S^*(n,p) \overset{\text{def}}{=} S(\lg n, \lg(p/c_{sh}), \lg n)$$

   Here n denotes the number of keys being sorted and p denotes the number of processors. This time bound applies to the model $\mathcal{M}_{par}$.

It is worth commenting on our requirement that p be a multiple of the constant c_{sh}. All of the models that we consider have the property that a single multi-level storage hierarchy can emulate any constant number of multi-level storage hierarchies with constant slowdown (and a constant factor blow-up in storage capacity). As a consequence, there exists a machine in $\mathcal{M}_{par}$ with $\Theta(2^a)$ storage and only $\epsilon \cdot 2^b$ processors, for any $\epsilon > 0$, that sorts in time $O(S^*(n,p))$.

Appendices A through C apply the preceding strategy to several specific models of multilevel storage that have been proposed in the literature. The results are summarized in table format. Although we do not prove any non-trivial lower bounds in this paper, the reader will notice that almost all of our table entries (all but one, in fact) are given in the form of Θ-bounds rather than O-bounds. The source of each non-trivial lower bound will be cited in the relevant section. Those lower bounds which are not discussed are trivial to prove.

8 Practical Considerations

It should be emphasized that the simulation-based approach we have taken towards analyzing the big-Oh complexity of Sharesort-based algorithms is geared towards streamlining the asymptotic analysis, and is almost certainly inappropriate (due to accumulated constant factors) for actual implementation. In particular, note that the primitives of our generic model must be rather weak in order to be efficiently implementable on *all* of the particular models that we intend to address. When implementing Sharesort on any specific model, a variety of significant constant-factor optimizations should be possible. For example, each call to sparse enumeration sort is currently handled by simulating the corresponding sequence of

2^b processors of the relevant dimension-d par-cube.) Running time: $O(T_{OP}(a,b,d) + 2^{5d/9} M(a,b,4d/9))$.

3. A set of $2^{4d/9}$ merges of dimension $5d/9$. Implementation: A subcube scan primitive interleaved with the sequence of $2^{4d/9}$ recursive high-order merging problems. (The subcube scan primitive has argument $5d/9$ and is applied to each of the 2^b processors of the relevant dimension-d par-cube.) Running time: $O(T_{OP}(a,b,d) + 2^{4d/9} M(a,b,5d/9))$.

(The initial sorting problem is such that it resides within a dimension-a par-cube with 2^b processors. As is easily proven by induction, each recursive sorting or high-order merging problem of dimension d arises within some dimension-d par-cube with 2^b processors. Furthermore, each sorted list of length $2^{4d/5}$ passed to a high-order merging problem resides in a dimension-$(4d/5)$ par-cube with 2^b processors.)

Thus, we have

$$\begin{aligned} & d > c^* \cdot b \\ \Longrightarrow\quad & M(a,b,d) \le M(a,b,4d/9) \cdot 2^{5d/9} + M(a,b,5d/9) \cdot 2^{4d/9} + O(T_{OP}(a,b,d)). \end{aligned} \tag{4}$$

Given the function $T_{OP}(a,b,d)$ associated with a specific model of multi-level storage, we can apply the recurrence of Equations (3) and (4) to calculate $M(a,b,d)$. This turns out to be a straightforward exercise for each of the models that we consider in Section 7.

Let $S(a,b,d)$ denote the complexity of performing a sort of dimension d on $\mathcal{G}_{par}(a,b)$. Cubesort gives

$$d \le c^* \cdot b \implies S(a,b,d) = O(T_{OP}(a,b,d)), \tag{5}$$

and Sharesort gives

$$d > c^* \cdot b \implies S(a,b,d) \le S(a,b,4d/5) + M(a,b,d) + O(T_{OP}(a,b,d)). \tag{6}$$

Given the functions $T_{OP}(a,b,d)$ and $M(a,b,d)$ associated with a specific model of multi-level storage, we can apply the recurrence of Equations (5) and (6). This turns out to give $S(a,b,d) = O(M(a,b,d))$ for each of the models that we consider in Section 7.

7 Sorting in Specific Models

Given the framework provided by Section 6, it is no easy to analyze the complexity of Sharesort on a variety of specific models of multi-level storage. For any specific model $\mathcal{M}$, we follow the same procedure to analyze the complexity of Sharesort:

1. Define $\mathcal{M}_{seq}$, the sequential version of model $\mathcal{M}$. (The parallel version $\mathcal{M}_{par}$ is obtained from $\mathcal{M}_{seq}$ in the same way as $\mathcal{G}_{par}$ was obtained from $\mathcal{G}_{seq}$.)

2. Choose the machine M in $\mathcal{M}_{seq}$ corresponding to $\mathcal{G}_{seq}(a)$. The desired machine M should have a total storage capacity of $\Theta(2^a)$.

Cubesort immediately gives a method for sorting the 2^a keys of par-cube C in a constant number of phases, where in each phase the following steps are performed: (i) a parallel application of the subcube scan primitive is applied to par-cube C, interleaving sorting operations over the fixed-connection network, and (ii) a BPC route operation is applied to par-cube C.

Letting $T_{CUBE}(a,b)$ denote the running time of Cubesort, we conclude that

$$T_{CUBE}(a,b) = O(T_{OP}(a,b,a))$$

whenever $a = O(b)$. This bound will turn out to be optimal in all of the models of multi-level storage that we consider in Section 7. On the other hand, when $a = \omega(b)$, Cubesort does not lead to optimal sorting bounds. (Nor does column sort.)

6.2 Sharesort

We now devise a sorting algorithm for our generic parallel model of multi-level storage that will prove to be asymptotically optimal for all $a \geq b$ on a variety of particular models of multi-level storage.

Our sorting algorithm will be a bottom-up merging algorithm. As a result, for given parameters a and b, we will perform recursive sorts corresponding to parameters d and b where d takes on many different values in the range $b \leq d \leq a$. For each recursive sort with $d \leq c^* \cdot b$, where c^* is some sufficiently large positive constant to be determined, we will simply apply Cubesort as described in Section 6.1. On the other hand, each recursive sort with $d > c^* \cdot b$ will be solved by applying the first of two main recurrences of the Sharesort algorithm to reduce the given sorting problem to a number of smaller sorting and high-order merging problems. Let us define a *sort of dimension* d as the problem of sorting 2^d keys, and a *high-order merge of dimension* d as the problem of merging $2^{d/5}$ sorted lists of length $2^{4d/5}$. Sharesort reduces a sort of dimension d (i.e., a sort of 2^d keys) to $2^{d/5}$ sorts of dimension $4d/5$, followed by a high-order merge of dimension d.

It remains to describe how to perform a high-order merge of dimension d, $b \leq d \leq a$. Let $M(a,b,d)$ denote the complexity of performing a high-order merge of dimension d on $\mathcal{G}_{par}(a,b)$. If $d \leq c^* \cdot b$, the high-order merge will be performed using Cubesort, and so

$$d \leq c^* \cdot b \implies M(a,b,d) = O(T_{OP}(a,b,d)). \tag{3}$$

If $d > c^* \cdot b$, then we apply the second main recurrence of Sharesort, which reduces a high-order merging problem of dimension d to:

1. A constant number of basic operations of dimension d, including a dimension-d row routing operation with $c_{rr} = 29/45$. (Note that c'_{rr} can be brought arbitrarily close to c_{rr} by choosing the constant c^* sufficiently large. Hence, for an appropriate choice of c^*, the constraint $c'_{rr} > 0$ will be satisfied.) Implementation: See Section 5. Running time: $O(T_{OP}(a,b,d))$.

2. A set of $2^{5d/9}$ high-order merges of dimension $4d/9$. Implementation: A subcube scan primitive interleaved with the sequence of $2^{5d/9}$ recursive high-order merging problems. (The subcube scan primitive has argument $4d/9$ and is applied to each of the

Note that $c'_{rr} > 0$ if and only if $c_{rr} \cdot d - b > 0$, and so the row route operation is ill-defined unless this inequality holds. In our applications, we will verify that $c_{rr} \cdot d - b > 0$ whenever a row route operation is performed. Running time (when the operation is well-defined): $O(t_{RR}(a-b, d-b))$.

We now generalize the preceding implementations to handle the case where a basic operation is applied to all subpar-cubes of a given par-cube. As in Section 3, let us take the prefix operation as a canonical example. Assume that dimension-d par-cube C is contained in some machine of $\mathcal{G}_{par}(a, b)$, $b \le d \le a$, and we would like to apply prefix to every dimension-d' subpar-cube of C, where $b \le d' \le d$. This could be implemented by interleaving: (i) a parallel application of the subcube scan primitive, with argument d', to par-cube C, and (ii) $2^{d-d'}$ prefix operations, one applied to each of the dimension-d' subpar-cubes of C. Running time: $O(t_{SS}(a-b, d-b) + t_{SS}(a-b, d'-b) \cdot 2^{d-d'} + b \cdot 2^{d-b})$.

The five remaining basic operations can be generalized in a similar fashion. Letting $T_{OP}(a, b, d)$ represent the maximum running time of any basic operation applied to the dimension-d' subpar-cubes of some dimension-d par-cube of a machine in $\mathcal{G}_{par}(a, b)$, $b \le d' \le d \le a$, we find that

$$T_{OP}(a, b, d) = O\left(\max_{b \le d' \le d}\left(t_{PRIM}(a-b, d-b) + t_{PRIM}(a-b, d'-b) \cdot 2^{d-d'} + b \cdot 2^{d-b}\right)\right). \qquad (2)$$

Note that the additive term $b \cdot 2^{d-b}$ term represents the cost of all parallel computations over the fixed-connection network, since the primitive operations do not induce any communication between the processors.

6 Sorting in the Generic Model

In this section we derive bounds on the running time of sorting algorithms in the generic parallel model of multi-level storage $\mathcal{G}$ defined in Section 5. Our time bounds will be expressed in terms of $T_{OP}(a, b, d)$ (see Equation (2)).

The input to a sorting problem is a set of 2^a keys. We will investigate the time required to sort these keys on a machine M in $\mathcal{G}_{par}(a, b)$. Note that the total storage capacity of M is only a constant factor larger than the size of the data set being sorted. The input data is assumed to be provided to the sorting algorithm in a single dimension-a par-cube C. The algorithm must produce the sorted output in the same par-cube C.

6.1 Cubesort

If $a = O(b)$ (i.e., if the number of keys is bounded by some polynomial in the number of processors) then Cubesort [11] provides a simple method for obtaining an efficient sorting algorithm. (Leighton's column sort [13] could also be used to obtain the bounds that follow. We prefer to make use of Cubesort only because the permutations performed by Cubesort can be implemented as BPC routes, one of our basic operations. The permutations involved in column sort are equally "easy", but are not all directly implementable as BPC routes.)

intra-processor index bits), (ii) a parallel application of the subcube scan primitive with parameter 0, interleaving appropriate $O(b)$-time computations over the fixed-connection network (to permute/complement inter-processor index bits), (iii) a parallel application of the cyclic shift primitive (with arguments set to prepare for the "transpose" of step (iv)), (iv) a parallel application of the subcube scan primitive with parameter 0, interleaving appropriate $O(b)$-time computations over the fixed connection network (in combination with the previous step, to exchange intra-processor index bits with inter-processor index bits as necessary), (v) a parallel application of the BPC route primitive (to permute intra-processor bits once again if necessary). Running time: $O(t_{BPC}(a-b, d-b) + t_{CYC}(a-b, d-b) + t_{SS}(a-b, d-b) + b \cdot 2^{d-b})$.

3. **Merge.** A merge operation takes two dimension-d par-cubes C' and C'' as input, $b \leq d \leq a$, and produces a dimension-$(d+1)$ output par-cube C. The merge can be implemented by interleaving: (i) three parallel applications of the subcube scan primitive, each with argument 0, applied (one each) to par-cubes C, C', and C'', and (ii) appropriate $O(b)$-time computations over the fixed-connection network. (The idea is to implement a block-sequential merge of the data in cubes C' and C'', where the block size is 2^b and the output is written a block at a time to output cube C. A new block is read from C' or C'' only when there is insufficient data in the fixed-connection network to form the next output block.) Running time: $O(t_{SS}(a-b, d-b) + b \cdot 2^{d-b})$.

4. **Monotone route.** A monotone route operation takes two dimension-d par-cubes C and C' as input, $b \leq d \leq a$, and applies a particular permutation to the values of par-cube C. The permutation can be implemented by interleaving: (i) two parallel applications of the subcube scan primitive, each with argument 0, applied (one each) to par-cubes C and C', and (ii) appropriate $O(b)$-time computations over the fixed-connection network. (The idea is to implement a block-sequential monotone route of the data in cube C, where the block size is 2^b and the output is written a block at a time to output cube C. A new block is read from C or C' only when there is insufficient data in the fixed-connection network to form the next output block.) Running time: $O(t_{SS}(a-b, d-b) + b \cdot 2^{d-b})$.

5. **Prefix.** A prefix operation acts on a single dimension-d par-cube C, $b \leq d \leq a$. The prefix operation can be implemented by interleaving: (i) a parallel application of the subcube scan primitive, with argument 0, to par-cube C, and (ii) appropriate $O(b)$-time computations over the fixed-connection network. (The idea is to implement a block-sequential prefix operation over the data in cube C, where the block size is 2^b and we alternately read/write blocks from/to C.) Running time: $O(t_{SS}(a-b, d-b) + b \cdot 2^{d-b})$.

6. **Row route.** A row route operation takes two dimension-d par-cubes C and C' as input, $b \leq d \leq a$, and applies a particular permutation to the values of par-cube C. The permutation can be implemented by a single parallel application of the row route primitive to par-cubes C and C', with argument

$$c'_{rr} \overset{\text{def}}{=} \frac{c_{rr} \cdot d - b}{d - b}.$$

Although the preceding paragraph gives a complete description of the model $\mathcal{G}$, it will be useful to introduce some additional terminology. First, let us arbitrarily partition the p processors into c_{sh} *par-groups (parallel groups)* of size 2^b. Each par-group has a total of 2^b associated cubes of dimension d, $0 \le d \le a - b$. We call these cubes seq-cubes (sequential cubes). Within each par-group, we form a single par-cube (parallel cube) of dimension d, $b \le d \le a$, by combining together the 2^b seq-cubes of dimension $d - b$. The values of each par-cube are distributed over the corresponding set of seq-cubes by applying a 2^b-way unshuffle of the index set of the par-cube. More formally, suppose that a given dimension-d par-cube C corresponds to the set of 2^b seq-cubes $C^{(j)}$, $0 \le j < 2^b$. Further assume that C has associated values x_i, $0 \le i < 2^d$, and that $C^{(j)}$ has associated values $x_k^{(j)}$, $0 \le k < 2^{d-b}$. Then x_i is assigned to seq-cube $C^{(j)}$ with $j = (i \bmod 2^b)$, and corresponds to the value $x_k^{(j)}$ with $k = \lfloor i/2^b \rfloor$.

We have chosen to introduce par-cubes because all of our algorithms are most easily understood in terms of their action on par-cubes (as opposed to seq-cubes). In our algorithms, whenever a primitive operation is applied at a particular processor P, the same primitive operation is simultaneously applied at every other processor in the par-group of P. We call this a *parallel application* of the primitive to the par-group (or par-cube). Furthermore, although each par-cube is physically distributed over 2^b processors, we will shortly demonstrate that all of the basic operations defined in Section 3 can be efficiently applied to par-cubes. In fact, each basic operation will require only a constant number of parallel applications of the four primitive operations.

Recall that each of the basic operations of family $\mathcal{A}$ takes $O(1)$ distinct cubes of data as input, and produces a single output cube (in some cases the output cube is also one of the input cubes). We now describe how each of these basic operations can be implemented on our generic model of multi-level storage. In particular, we will be interested in the case where: (i) the input/output cubes to a particular operation correspond to par-cubes in the generic model, and (ii) distinct input/output cubes correspond to par-cubes of distinct par-groups in the generic model. We now address each of the basic operations of family $\mathcal{A}$ in turn. Following the same approach as in Section 3, we will first treat the case where a basic operation is applied over an entire par-cube, and then generalize to basic operations that are applied to all subpar-cubes of some par-cube.

1. **Assignment.** An assignment operation involves k (k a constant) dimension-d input par-cubes $C^{(j)}$, $0 \le j < k$, and a dimension-d output par-cube C, $b \le d \le a$. Let ℓ denote the number of distinct input/output par-cubes (the output par-cube could be the same as one of the input par-cubes). The assignment operation can be implemented by interleaving: (i) $\ell = O(1)$ parallel applications of the subcube scan primitive, each with argument 0, applied to the set of input/output par-cubes, and (ii) $O(1)$-time local computations over the processors of the fixed-connection network. Running time: $O(t_{SS}(a - b, d - b) + 2^{d-b})$.

2. **BPC route.** A BPC route operation takes a single dimension-d par-cube C, $b \le d \le a$, and permutes the values of C. The permutation is not difficult to implement via a sequence of primitive operations applied to C. One possible sequence may be sketched as follows: (i) a parallel application of the BPC route primitive (to permute/complement

in $\mathcal{G}_{seq}(a)$

2. **Cyclic shift.** When applied to the cube C of dimension d, $0 \le d \le a$, this primitive takes an integer argument k such that $0 \le k < 2^d$. Assume that cube C has associated integer values x_i, and let $y_j = x_i$ where $j = (i+k) \bmod 2^d$, $0 \le i < 2^d$. Then the effect of applying a cyclic shift primitive to cube C is to replace each x_i by y_i, $0 \le i < 2^d$. Let $t_{CYC}(a,d)$ denote the running time of this primitive.

3. **Row route.** This primitive is defined in the same way as the corresponding "basic operation" in Section 3, except that the constant c_{rr} in the definition of r is replaced by a real value c'_{rr}, $c_{rr} \le c'_{rr} < 1$, that is passed as an argument. (Also, we do not extend the definition to allow for simultaneous application across all of the subcubes of a given cube.) In our applications, c'_{rr} will always be bounded away from 1. Let $t_{RR}(a,d)$ denote the running time of this primitive.

4. **Subcube scan.** When applied to a cube C of dimension d, $0 \le d \le a$, this primitive takes an integer argument d', $0 \le d' \le d$. The subcube scan operation proceeds in $2^{d-d'}$ "steps". Step i, $0 \le i < 2^{d-d'}$, consists of the following three "sub-steps":

   (a) The ith dimension-d' subcube of C (i.e., the subcube corresponding to the range of indices $[i \cdot 2^{d'}, (i+1) \cdot 2^{d'})$) is copied to the cube of dimension d'.

   (b) An arbitrary sequence of primitive operations (possibly including one or more subcube scan primitives) is applied to the cube of dimension d'. (In our applications, this sequence will be the same for all i.) If $d' = 0$, then an arbitrary computation may be applied at the processor level. Note: The running time of this step is not included in $t_{SS}(a,d')$, defined below, as it will always be accounted for elsewhere in our analysis.

   (c) The values stored in the cube of dimension d' are copied back into the ith dimension-d' subcube of C.

   Let $t_{SS}(a,d)$ denote the running time of this primitive, for a worst-case choice of d'.

Let us define $t_{PRIM}(a,d)$ as the maximum worst-case running time of any of the four primitive operations applied to the cube of size d of some machine in $\mathcal{G}_{seq}(a)$, that is,

$$t_{PRIM}(a,d) \overset{\text{def}}{=} \max\{t_{BPC}(a,d), t_{CYC}(a,d), t_{RR}(a,d), t_{SS}(a,d)\}. \tag{1}$$

We now define our generic parallel model of multi-level storage $\mathcal{G}_{par}$, which takes two nonnegative integer parameters. A machine in $\mathcal{G}_{par}(a,b)$, $0 \le b \le a$, may be constructed as follows: For some sufficiently large positive constant c_{sh} (related to the maximum number of words of data used by the hypercube version of the Sharesort algorithm at any one processor, which is $O(1)$), take $p = c_{sh} \cdot 2^b$ machines in $\mathcal{G}_{seq}(a-b)$, and interconnect the p processors via any bounded-degree network that can perform both prefix and sorting operations (where the input consists of p values/keys stored one per processor) in $O(\lg p) = O(b)$ time. Such networks are known to exist [13].

storage model $\mathcal{M}$, we will apply basic operation X *sequentially* to each of the dimension-d' cubes. Operation Y will also be applied sequentially to each of the dimension-d'' cubes. However, in many cases we will not want to apply X everywhere before beginning to apply Y. Instead, it may be more efficient to interleave the applications of X and Y as in the following $2^{d-d'}$-phase algorithm: In phase i, apply X to the ith dimension-d' subcube of C (call this subcube C_i), and then apply Y sequentially to each of the $2^{d'-d''}$ dimension-d'' subcubes of C_i, $0 \leq i < 2^{d-d'}$.

While this interleaving idea is quite simple, it cannot be applied effectively without further investigating the structure of the Sharesort algorithm. Fortunately, Sharesort is based on a system of recurrences that capture everything we will need to know about the sequence of subcube dimensions corresponding to the sequence of basic operations applied by Sharesort. These recurrences are reviewed in Section 6. The "interleaving" scheme that we will use to implement Sharesort on various models of multi-level storage is also given in Section 6.

5 A Generic Model of Multi-Level Storage

In Section 7 we will determine the asymptotic complexity of certain operations on a variety of models of multi-level storage. Fortunately, we will be able to expedite this process by following the same basic strategy for each model. The purpose of the present section is to define a "generic" model $\mathcal{G}$ of multi-level storage which captures certain common features of the specific models that we will study later on. Section 6 will describe our basic strategy in terms of the generic model.

We first define the sequential version of our generic model of multi-level storage $\mathcal{G}$. We will then extend this definition to the parallel case. Our sequential model of multi-level storage $\mathcal{G}_{seq}$ is parameterized by a single nonnegative integer. A machine in $\mathcal{G}_{seq}(a)$ is a single processor with $O(1)$ local memory and an associated multi-level storage consisting of $a+1$ disjoint cubes. More specifically, the multi-level storage contains one cube of each dimension d, $0 \leq d \leq a$. Input and output are provided/computed in specific subsets of the multi-level storage. The processor has direct access to the value in the dimension-0 cube. (We make no assumption regarding whether or not the processor can directly access the values stored in any other cube. Thus, in algorithms that we develop for the generic model, such values will first be transferred to the dimension-0 cube before they are read by the processor.)

The computation consists of a sequence of steps, where in each step one of four *primitive operations* is applied to a cube. After a primitive operation has been applied to the cube of some dimension d, $0 \leq d \leq a$, all values belonging only to cubes of dimension strictly less than d should be assumed to have been set arbitrarily. (In other words, any data that may have been stored in such a location should be assumed lost.) The more useful effect of each of the four primitive operations is defined below.

1. **BPC route.** This primitive is defined in the same way as the corresponding "basic operation" in Section 3. (But we do not extend the definition to allow for simultaneous application across all of the subcubes of a given cube.) Let $t_{BPC}(a, d)$ denote the running time of this primitive when applied to the cube of dimension d of a machine

A number of sorting algorithms that have been proposed in the literature belong to the family $\mathcal{A}$. Hence these algorithms can be implemented by making use of the subroutines cited above. For each of the following three sorting algorithms, this approach leads to efficient hypercube implementations.

1. **Bitonic sort.** This sorting algorithm, due to Batcher [5], can be expressed in terms of the assignment, BPC route, and merge operations applied to a dimension-d cube containing 2^d keys. The corresponding hypercube algorithm runs in $O(d^2)$ time. Upon termination, the key with rank i (ties can be broken in a stable fashion) is stored in processor i, $0 \leq i < 2^d$. (Note: It is possible to implement bitonic sort without any explicit list reversal operations, but this would not improve the running time by more than a constant factor.)

2. **Sparse enumeration sort.** This algorithm, due to Nassimi and Sahni [16], can be expressed in terms of the assignment, BPC route, merge, monotone route, and prefix operations applied to a constant number of dimension-d cubes, one of which initially contains the set of input keys. When the number of input keys is $2^{\gamma \cdot d}$ for some constant $\gamma < 1$, the corresponding hypercube algorithm runs in $O(d)$ time. Upon termination, the key with rank i (ties can be broken in a stable fashion) is stored in processor i, $0 \leq i < 2^{\gamma \cdot d}$.

3. **Sharesort.** This algorithm, due to Cypher and Plaxton [9], can be expressed in terms of the assignment, BPC route, merge, monotone route, prefix, and row route operations applied to a constant number of dimension-d cubes, one of which initially contains the set of 2^d input keys. (Sharesort also makes use of bitonic sort and sparse enumeration sort, but as we have just seen these algorithms can themselves be expressed in terms of the basic operations.) Depending on the amount of preprocessing allowed (see the discussion of the row route bounds above), the running time of Sharesort is $O(d \lg d)$ (exponential preprocessing), $O(d \lg d \lg^* d)$ (polynomial preprocessing), or $O(d \lg^2 d)$ (no preprocessing). The output condition is the same as for bitonic sort.

The main goal of this paper is to demonstrate that Sharesort can also be implemented efficiently on various models of multi-level storage. For a particular model of multi-level storage $\mathcal{M}$, one might attempt to achieve this goal by: (i) determining the complexity of implementing each of the basic operations on $\mathcal{M}$, and (ii) calculating the corresponding running time for Sharesort by adding up the times of the various basic operations. Although this approach will certainly yield a correct sorting algorithm, it will also generally lead to a running time that is very far from optimal. In order to obtain optimal bounds, we will make use of the observation that the scheduling of the basic operations can be *interleaved*, as in the simple example of the following paragraph.

Suppose that basic operations X and then Y are to be applied to the dimension-d' and dimension-d'' subcubes of some dimension-d cube C, with $0 \leq d'' \leq d' \leq d$. In the hypercube implementation outlined above, basic operation X would be applied to all dimension-d' subcubes of C in parallel, and then basic operation Y would be applied to all dimension-d'' subcubes of C in parallel. In an implementation corresponding to a typical multi-level

$\oplus$ to each of the $2^{d-d'}$ dimension-d' subcubes of C. Each of the other five basic operations can be generalized in a similar fashion; in each case, an additional integer d' is specified, $0 \leq d' \leq d$, and the operation as defined above is performed over all subcubes of dimension d'. The resulting set of six generalized basic operations, as applied to a dimension-d cube, will be referred to as the *basic operations of dimension d*.

We can now define the family of algorithms $\mathcal{A}$. The input and working storage of an algorithm in $\mathcal{A}$ is given by a constant number of dimension-d cubes. The algorithm transforms the cube values by applying some sequence of basic operations of dimension d. (For each such operation, a subcube dimension d', $0 \leq d' \leq d$, is specified as discussed in the preceding paragraph.)

4 Hypercube Algorithms

This section begins with a discussion of the general strategy for implementing algorithms of family $\mathcal{A}$ on the hypercube. (The same strategy can be applied to obtain efficient algorithms for bounded-degree variants of the hypercube such as the butterfly, cube-connected cycles, and shuffle-exchange.) We then review three specific sorting algorithms that have been proposed in the past, all of which belong to family $\mathcal{A}$.

Consider an algorithm of family $\mathcal{A}$ involving $k = O(1)$ dimension-d cubes with associated values $x_i^{(j)}$, $0 \leq i < 2^d$, $0 \leq j < k$. It is natural to map the cube values to a hypercube of dimension d by storing the values $\{x_i^{(0)}, \ldots, x_i^{(k-1)}\}$ at processor i, $0 \leq i < 2^d$. The following running times are known to be achievable for implementing each of the basic operations. Note: In each case, the running time is given in terms of d', $0 \leq d' \leq d$, the dimension of the subcubes to which the operation is being applied. Unless otherwise stated, the running times are for on-line algorithms with no preprocessing.

1. **Assignment**. An assignment operation requires only $O(1)$ time, as it can be implemented with a constant number of local operations at each processor (no inter-processor communication is required).

2. **BPC route.** This operation can be performed in $O(d')$ time [15].

3. **Merge.** This operation can be performed in $O(d')$ time by reversing one of the two lists (using a BPC route) and then applying Batcher's bitonic merge algorithm [5].

4. **Monotone route.** This operation can be performed in $O(d')$ time [16].

5. **Prefix.** This operation can be performed in $O(d')$ time (see [14], for example).

6. **Row route.** The asymptotic complexity of this operation is known to be the same (to within a constant factor) as that of the shared key sorting operation of Cypher and Plaxton [9]. (This claim follows from the optimal $O(d')$ complexity of sparse enumeration sort, discussed below.) If exponential preprocessing time (to compute certain tables used by the algorithm) is allowed, an optimal $O(d')$ running time can be achieved [10]. With polynomial preprocessing, $O(d' \lg^* d')$ is achievable [10]. With no preprocessing, $O(d' \lg d')$ is currently the best bound known [9].

denote some dimension-$(d+1)$ cube with associated integer values x_i, $0 \leq i < 2^{d+1}$. Then the effect of merging "input" cubes C' and C'' to "output" cube C is to assign the x_i's to the sorted sequence corresponding to the union of the y_i's and z_i's.

4. **Monotone route.** Let distinct dimension-d cubes C, C', and C'' be given, with associated integer values x_i, y_i, and z_i, $0 \leq i < 2^d$, respectively. Further assume that $-1 \leq z_i < 2^d$, $0 \leq i < 2^d$, and that the nonnegative z_i's form a monotonically increasing sequence. Then the effect of applying a monotone route operation to "destination" cube C, "source" cube C', and "address" cube C'' is to assign $x_{z(i)}$ to y_i for each i such that $0 \leq i < 2^d$ and $z(i)$ is nonnegative.

5. **Prefix.** Let C denote a dimension-d cube of data containing the integer values x_i, $0 \leq i < 2^d$. Let $y_i = \bigoplus_{0 \leq j < i} x_j$, $0 \leq i < 2^d$, for some binary associative operator $\oplus$. Then the effect of applying a prefix operation to cube C with operator $\oplus$ is to replace x_i with y_i, $0 \leq i < 2^d$.

6. **Row route.** Let distinct dimension-d cubes C and C' be given, with associated integer values x_i and y_i, $0 \leq i < 2^d$, respectively. Let $r = \lfloor c_{rr} \cdot d \rfloor$ where $c_{rr} > 1$ is a positive constant to be specified later (see Section 6). Let $f(i) = \lfloor i/2^{d-r} \rfloor$, π be a permutation over $[0, 2^r)$, and $\Pi(i) = \pi(f(i)) \cdot 2^{d-r} + (i \bmod 2^{d-r})$, $0 \leq i < 2^d$. Note that Π is a permutation over $[0, 2^d)$. (Viewing C as a $2^r \times 2^{d-r}$ matrix stored in row-major order, Π corresponds to a permutation of the rows of C.) Assume that $y_i = \Pi(i)$ and let $z_i = x_{\Pi(i)}$, $0 \leq i < 2^d$. Then the effect of applying a row route operation to cube C with respect to the "row permutation" specified by cube C' is to replace x_i with z_i, $0 \leq i < 2^d$.

The reader may wonder why we have chosen to include three special-purpose permutation routing operations in the preceding list (BPC route, monotone route, and row route) rather than a general permutation routing operation. The reason is that on many models of computation (such as the hypercube, and certain of the models of multi-level storage that we will consider), the worst-case complexity of these special-purpose routing operations is asymptotically lower than that of general permutation routing. In such cases we may be able to obtain a more efficient sorting algorithm by restricting ourselves to special-purpose routing operations.

We now generalize each of our six basic operations in the same fashion by introducing the notion of a subcube. Our goal is to allow parallel application of any of the basic operations over disjoint subcubes of some cube.

Definition 3.2 For any d' such that $0 \leq d' \leq d$, we define a unique partitioning of a given dimension-d cube of data into $2^{d-d'}$ dimension-d' *subcubes* of data as follows: Subcube j consists of those values with indices in the interval $[j \cdot 2^{d'}, (j+1) \cdot 2^{d'})$.

If re-indexed from 0, note that any subcube corresponds to a cube. Consider then the following generalization of the prefix operation defined above: Given a dimension-d cube C, a binary associative operator $\oplus$, and an integer d', $0 \leq d' \leq d$, apply prefix with operator lem of sorting efficiently on hypercubic networks (e.g., the main routine is overly-sequential in nature), though it remains to be seen whether Balance Sort could be modified to yield an efficient hypercubic sorting algorithm (while preserving the same basic overall structure). Thus, Sharesort has a broader proven range of applicability.

3 The Family of Algorithms $\mathcal{A}$

This section introduces an abstract family of algorithms which we will refer to as family $\mathcal{A}$. Algorithms in this family are obtained by composing certain *basic operations* over a constant number of *cubes* of data, as defined below. In subsequent sections we will analyze the running time of algorithms in this family under various models of computation.

Definition 3.1 For any $d \geq 0$, a *dimension-d cube* of data is a set of 2^d integer values indexed from 0 to $2^d - 1$.

Note that a dimension-d cube of data may be viewed as simply a linear array of length 2^d. We now define the *basic operations* that may be applied to cubes of data within algorithms of family $\mathcal{A}$. With the exception of the "row route" operation, all of these operations have appeared previously (and extensively) in the literature. Row routing is not really a new operation either, as it is a minor variant of the shared key sorting operation used within the Sharesort algorithm of Cypher and Plaxton [9].

The set of six basic operations stated below may seem to have been somewhat arbitrarily chosen. As we will soon see, however, these basic operations satisfy two important properties: (i) each can be efficiently implemented both on hypercubic networks as well as on various models of multi-level storage, and (ii) together they can be used to define an efficient sorting algorithm.

1. **Assignment.** Let $E(x, x^{(0)}, \ldots, x^{(k-1)})$ denote an arbitrary constant-size, integer-valued arithmetic expression in $k+1$ integer variables. Corresponding to each variable $x^{(j)}$, assume that we are given a dimension-d cube $C^{(j)}$ with associated integer values $x_i^{(j)}$, $0 \leq i < 2^d$. Further assume that cube C has associated integer values x_i, $0 \leq i < 2^d$. Then the result of assigning expression E to cube C is to replace x_i by $E(i, x_i^{(0)}, \ldots, x_i^{(k-1)})$, $0 \leq i < 2^d$.

2. **BPC route.** Let C be a cube of dimension d with associated integer values x_i, $0 \leq i \leq 2^d$. Let π be a permutation over $[0, d)$, and $\langle b_0, \ldots, b_{d-1} \rangle$ be a sequence of d bits. For any integer i in $[0, 2^d)$, we define the function $\Pi(i)$ as follows: If i has binary representation $i_{d-1} \cdots i_0$ then $\Pi(i) = j$ where j has binary representation $j_{d-1} \cdots j_0$ with $j_k = (i_{\pi(k)} + b_k) \bmod 2$, $0 \leq k < d$. Note that Π is a permutation over $[0, 2^d)$. Let $y_{\Pi(i)} = x_i$, $0 \leq i < 2^d$. Given π and $\langle b_0, \ldots, b_{d-1} \rangle$, the effect of applying the corresponding BPC route operation to cube C is to replace x_i with y_i, $0 \leq i < 2^d$.

3. **Merge.** Let distinct dimension-d cubes C' and C'' be given, with associated integer values y_i and z_i, $0 \leq i < 2^d$, respectively. Assume that the y_i's and z_i's are sorted in ascending order. (In other words, $y_i \leq y_{i+1}$ and $z_i \leq z_{i+1}$, $0 \leq i < 2^d - 1$.) Let C

in the current setting (which is more sequential in nature) than in the hypercube setting. For example, the shared key sorting subroutine takes up approximately half of the original Sharesort description, but corresponds to a simple primitive in the present context.

It is interesting to compare the high-level structure of Sharesort with that of Balance Sort. Balance Sort is a deterministic, comparison-based distribution sort: The keys are sorted in a top-down fashion by partitioning them into buckets (as given by a set of approximately evenly-spaced splitter keys) and then sorting each of the buckets recursively. On the other hand, Sharesort is a deterministic, comparison-based (high-order) merge sort: The keys are sorted in a bottom-up fashion by arbitrarily partitioning them into n^{γ} lists of length $n^{(1-\gamma)}$ (for some constant γ, $0 < \gamma < 1$), sorting each of these lists recursively, and then merging the sorted lists via a single high-order merge operation. This cursory examination of the structure of the two algorithms in question suggests that they are entirely different. In fact, such an analysis is somewhat misleading, since the critical high-order merge operation of Sharesort, which dominates the running time of Sharesort in all computational models of interest, is actually a "distribution merge": The keys are merged in a top-down fashion by computing a set of (precisely) evenly-spaced splitter keys, breaking the sorted lists into smaller sorted lists of similar keys (i.e., keys lying between the same pair of adjacent splitters), partitioning these smaller sorted lists into buckets (as given by the splitters), and then merging the lists within each bucket recursively.

Thus, Sharesort is more properly viewed as a "mixed-mode" sorting method, with both "bottom-up" and "top-down" characteristics. Indeed, the power of Sharesort stems directly from this curious combination of computational paradigms: The bottom-up sorting eases the complexity of the top-down high-order merge while not increasing the overall complexity of the algorithm by more than a constant factor. Because Balance Sort takes an entirely top-down approach, the implementation details turn out to be more complicated, for example: (i) a complicated bipartite matching procedure (obtained via advanced de-randomization techniques) is needed to balance the load across the processors; (ii) a CRCW PRAM interconnection is required to obtain optimal complexity for a certain range of parameter settings within the parallel disk model; (iii) the main balancing routine needs to maintain several matrices containing certain distribution-related counts.

Unlike Sharesort, the Balance Sort algorithm does not seem to be expressible in terms of a small number of fundamental primitive operations. (This comment applies to both interprocessor operations as well as operations acting on the multi-level storage.) As a result, we find that the implementation details of Balance Sort tend to vary more from one model to another (e.g., different techniques are used to bound the internal processing time in the parallel disk model than in the other models). While we have not attempted to formulate the following claim as a theorem, we believe that Balance Sort is unlikely to yield better asymptotic performance than Sharesort on any "reasonable" model of multi-level storage. This is because the running time of each algorithm is ultimately determined by solving a similar recurrence, where the additive (i.e., overhead) term is determined by the complexity of certain basic operations, and the basic operations associated with Sharesort appear to be simpler.

Finally, it is clear that the approach of Balance Sort is not directly applicable to the probof important reasons why our results are of interest: (i) the Sharesort-based algorithms are conceptually simple, in that they are based upon a handful of well-known basic operations applied to "cubes" of data according to a small system of recurrences; (ii) the Sharesort-based approach seems to apply to a strictly wider class of computational models than does Balance Sort; (iii) the compact mathematical formulation of Sharesort enables us to more easily give a complete analysis of its complexity on a given model; (iv) the Sharesort-based results reveal an unexpected connection between efficient sorting algorithms for hypercubic networks and efficient sorting algorithms for parallel models of multi-level storage.

A number of technical challenges had to be overcome in order to establish our main result. To simplify the task of developing Sharesort-based algorithms for each of the many parallel models of multi-level storage that exist, we will proceed by: (i) defining a "generic" parallel model of multi-level storage, (ii) providing an implemention of Sharesort within the generic model, and (iii) showing how to efficiently simulate the generic model on each of the specific models of interest. The generic model must be carefully specified in order to facilitate this approach. Even so, some of the generic model implementations of basic Sharesort operations are non-trivial (e.g., BPC route). Still other operations (e.g., monotone route) are intriguing because the efficient generic model implementation, though straightforward, is seemingly unrelated to the efficient hypercube implementation. (Such "coincidences" suggest that the optimality of Sharesort on parallel models of multi-level storage may be somewhat fortuitous.)

The remainder of this paper is organized as follows. Section 3 defines an abstract family of algorithms $\mathcal{A}$ in terms of a number of "basic operations" applied to "cubes" of data. Section 4 considers hypercube implementations of algorithms in $\mathcal{A}$, and discusses the running time of several sorting algorithms in $\mathcal{A}$. Section 5 defines a "generic" sequential model of multi-level storage, $\mathcal{G}_{seq}$, in terms of four "primitive operations". Section 5 then extends $\mathcal{G}_{seq}$ to a generic parallel model, $\mathcal{G}_{par}$, and shows how to implement the basic operations of $\mathcal{A}$ in $\mathcal{G}_{par}$. Section 6 describes how to efficiently implement the Sharesort algorithm in $\mathcal{G}_{par}$. Section 7 presents a simple scheme for determining the complexity of Sharesort on a given parallel model of multi-level storage. (Appendices A through C apply this scheme to the various models that have been proposed in the literature: *P–HMM*, *P–BT*, *P–UMH*, *P–RUMH*, *P–SUMH*, and *P–DISK*.) Section 8 discusses practical considerations. Section 9 provides concluding remarks.

2 Comparison with Balance Sort

The purpose of this section is to contrast the asymptotic performance of Sharesort and Balance Sort on various parallel models of multi-level storage. Constant-factor issues will not be addressed in this section. (Section 8 discusses a number of practical considerations that one should take into account in an actual Sharesort implementation.)

Readers familiar with the intricacies of the Sharesort implementation on hypercubic networks [9] may be skeptical about the claim made in Section 1 that our Sharesort-based algorithms for parallel models of multi-level storage are "conceptually simple". Accordingly, it must be emphasized that the implementation details of Sharesort are far easier

1 Introduction

Under the classic sequential RAM model, it is assumed that any memory location can be accessed in unit time. In practice, however, we find that the memory of machines is partitioned into multiple levels of storage with significantly different access times (e.g., registers, cache, memory, disk, tape). In an effort to properly capture this phenomenon, a variety of sequential models of multi-level storage (often referred to as memory hierarchy models) have been proposed [1, 2, 3, 4, 12]. For example, one simple model assumes that accessing memory location i costs $\lg i$ units of time [1]. (More elaborate models tend to allow special block operations, or to define discontinuous access functions [2, 4].) For each particular model of multi-level storage, it is natural to analyze the complexity of routing (permuting) and sorting data. The complexity of these fundamental operations is intimately related to our notion of the "power" of the model.

An obvious question to ask is whether existing research on sequential models of multi-level storage can be extended to the parallel domain in an interesting way. Although many parallel programs involve only a small amount of data at each processor (and thus a negligible memory hierarchy at each processor), such applications are probably closer to the exception than the rule. The coming generation of tera-computers can be expected to consist of thousands of processors, each with its own multi-gigabyte storage [6]. Thus, an extension of sequential multi-level storage models to the parallel domain would seem to be well-motivated. In fact, Vitter and Shriver [20], Nodine and Vitter [17], and Vitter and Nodine [19] have proposed just such a series of extensions, and have examined the complexity of sorting in various parallel models of multi-level storage. In each of these papers, a new parallel model $\mathcal{M}_{par}$ is defined in terms of an existing sequential model $\mathcal{M}_{seq}$ by interconnecting a number of model $\mathcal{M}_{seq}$ sequential processors via some standard parallel model (e.g., PRAM, fixed-connection network). This seems to be a natural and appropriate approach. The so-called "parallel disk model" of Vitter and Shriver [20] has also been studied by Cormen [7, 8], who provides an extremely tight complexity analysis for certain classes of routing operations.

The sorting results presented in [17, 19, 20] are quite non-trivial; each of these papers provides tight sorting bounds for the specific family of computational models that it addresses. However, until recently, no single algorithm (or single paradigm) was known that could be used to obtain optimal time bounds for sorting on *all* models in these families. The question of existence of such an algorithm was largely resolved by the recent Balance Sort algorithm of Nodine and Vitter [18]. Balance Sort is a deterministic sorting scheme that leads to optimal or best-known complexity bounds for virtually all parallel models of multi-level storage yet proposed.

The main result of our paper is that the deterministic Sharesort algorithm of Cypher and Plaxton [9], originally designed as a one-item-per-processor sorting algorithm for hypercubic networks, is readily adaptable to all known parallel models of multi-level storage, where in all cases it matches the asymptotic performance of Balance Sort. (In fact, in limited cases Sharesort provides technical improvements over Balance Sort by requiring a less powerful model of computation in terms of the processor interconnection. We are not aware of any case in which Balance Sort requires a less powerful model. The relative merits of Balance Sort and Sharesort are discussed more extensively in Sections 2 and 8.) There are a number

Optimal Parallel Sorting in Multi-Level Storage

Alok Aggarwal
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
aggarwa@watson.ibm.com

C. Greg Plaxton[*]
Department of Computer Science
University of Texas at Austin
Austin, TX 78712
plaxton@cs.utexas.edu

Abstract

We adapt the Sharesort algorithm of Cypher and Plaxton to run on various parallel models of multi-level storage, and analyze its resulting performance. Sharesort was originally defined in the context of sorting n records on an n-processor hypercubic network. In that context, it is not known whether Sharesort is asymptotically optimal. Nonetheless, we find that Sharesort achieves optimal time bounds for parallel sorting in multi-level storage, under a variety of models that have been defined in the literature.

[*]Supported by NSF Research Initiation Award CCR–9111591, and the Texas Advanced Research Program under Grant No. 003658–480.