[Rei93] J. H. Reif, editor. *A Synthesis of Parallel Algorithms*. Morgan-Kaufmann, San Mateo, CA, 1993.

[RR89] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[Sie89] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 20–25, 1989.

[Val75] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Vis83] U. Vishkin. On choice of a model of parallel computation. Technical Report 61, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012, 1983.

[GMR96b] P.B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. In *Proc. of EURO-PAR'96, Springer LNCS*, August 1996. To appear. Preliminary version appears in *QRQW: Accounting for Concurrency in PRAMs and Asynchronous PRAMs*, AT&T Bell Laboratories Technical Report, March 1993.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[Hag89] T. Hagerup. Hybridsort revisited and parallelized. *Information Processing Letters*, 32:35–39, 1989.

[Hag91] T. Hagerup. Fast parallel generation of random permutations. In *Proc. of 18th Int. Colloquium on Automata Languages and Programming, Springer LNCS 510*, pages 405–416, 1991.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 869–941. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[KRS90] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[Lei85] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[Lei92] F. T. Leighton. Methods for message routing in parallel machines. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 77–96, May 1992. Invited paper.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[MA80] H. Meijer and S.G. Akl. The design and analysis of a new hybrid sorting algorithm. *Information Processing Letters*, 10:213–218, 1980.

[Mas91] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA 94086. *MasPar System Overview, document 9300-0100, revision A3*, March 1991.

[Mat92] Y. Matias. *Highly Parallel Randomized Algorithmics*. PhD thesis, Tel Aviv University, Israel, 1992.

[MR89] G. L. Miller and J. H. Reif. Parallel tree contraction, Part 1: Fundamentals. In S. Micali, editor, *Randomness and Computation*, volume 5, pages 47–72. JAI Press, 1989.

[MS91] P.D. MacKenzie and Q.F. Stout. Ultra-fast expected time parallel algorithms. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 414–423, 1991.

[MV91a] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

[MV91b] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[MV95] Y. Matias and U. Vishkin. A note on reducing parallel model simulations to integer sorting. In *Proc. 9th IEEE Int. Parallel Processing Symp.*, pages 208–212, 1995.

[Rei85] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, May 1985.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, 1985.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, 1974.

[Chl89] B.S. Chlebus. Parallel iterated bucket sort. *Information Processing Letters*, 31:181–183, 1989.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. of 17th Int. Colloquium on Automata Languages and Programming, Springer LNCS 443*, pages 6–19, 1990.

[FBR93] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. In *Proc. 1993 IEEE Compcon Spring*, pages 285–294, February 1993.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[GGK$^+$83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers*, C-32(2):175–189, 1983.

[Gil91] J. Gil. Fast load balancing on a PRAM. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Computing*, pages 10–17, December 1991.

[Gil94] J. Gil. Renaming and dispersing: Techniques for fast load balancing. *Journal of Parallel and Distributed Computing*, 23(2):149–157, November 1994.

[GM91] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, 1991.

[GM94a] J. Gil and Y. Matias. Simple fast parallel hashing. In *Proc. of 21st Int. Colloquium on Automata Languages and Programming, Springer LNCS 820*, July 1994.

[GM94b] J. Gil and Y. Matias. Simple fast parallel hashing by oblivious execution. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1994. *SIAM Journal on Computing*, 1996, accepted for publication.

[GMR93] P. B. Gibbons, Y. Matias, and V. Ramachandran. QRQW: Accounting for concurrency in PRAMs and Asynchronous PRAMs. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, March 1993.

[GMR94a] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236–247, June 1994.

[GMR94b] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638–648, January 1994.

[GMR96a] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 1996. To appear. Preliminary version appears in *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638-648, January 1994.

8 Conclusions

In this paper we have presented highly parallel work-optimal algorithms for several fundamental problems for the QRQW PRAM. These include linear work, logarithmic time algorithms for multiple compaction, generating a random permutation, and hashing; a sublogarithmic time, linear work algorithm for load balancing when the maximum initial load is small; and a sublogarithmic time linear work algorithm for generating a random cyclic permutation. We have also presented several simple algorithms for the sorting problem that improve on algorithms known for exclusive memory access PRAM models. Complementing these algorithmic results, we have shown an $\Omega(\lg L)$ time lower bound on the QRQW PRAM for the load balancing problem with maximum load L. All of the algorithms we have presented in this paper are randomized algorithms with high probability performance guarantees, and our lower bound applies to randomized as well as deterministic algorithms.

We have also provided experimental results from an implementation, on the MasPar MP-1, of our QRQW PRAM algorithm for generating a random permutation as well as the best EREW PRAM algorithm for this problem; our experimental results show that the QRQW PRAM algorithm does, indeed, run faster than the EREW PRAM algorithm.

The QRQW PRAM models the mechanism used by a number of currently available commercial shared memory machines to handle memory contention. As has been illustrated in the algorithms presented in this paper, novel techniques may be needed in the design of efficient algorithms in the QRQW models. We expect that further research will help obtain a clearer understanding of the capabilities of this model and its applicability to the design of efficient and cost effective parallel algorithms that can be implemented on currently available parallel machines.

Among the important open problems remaining are to obtain tight upper and lower bounds for the running times of (additional) fundamental problems on the QRQW PRAM, and to obtain a work-optimal, polylog time simulation of the CRCW PRAM on a QRQW PRAM (or prove that such a simulation does not exist).

Acknowledgements

The authors thank the anonymous referees for their helpful comments.

References

[ACC$^+$90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 International Conf. on Supercomputing*, pages 1–6, June 1990.

[AH92] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 463–472, January 1992.

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \lg n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Summer Computer Conference*, pages 307–314, 1968.

[BGMZ95] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 84–94, July 1995.

7. Compact the items in B into an array of size n.

By Observation 2.6, the maximum contention in step 1 is $O(\lg n/\lg\lg n)$ w.h.p. For step 2, we can apply Theorem 7.2 (or use any other algorithm that sorts n keys in $O(\lg n)$ time and at most $O(n\lg^2 n)$ work on a CRQW PRAM). Steps 3, 4, 5, and 7 can be done in $O(\lg n)$ time and linear work using prefix sums computations. For step 6, we replace the procedure used in [RR89] with our algorithm for "relaxed" heavy multiple compaction (Lemma 4.2). Thus w.h.p., the total time is $O(\lg n)$ and the total work is $O(n)$. Processor allocation is straightforward, yielding the desired result. ■

We observe that, with the exception of step 5 above, the entire algorithm can be adapted to run on the QRQW PRAM within the same resource bounds. In step 5, each item needs to learn the estimate of the set size for its key, and the pointer to its allocated subarray; we use the concurrent-read capability to stay within the desired resource bounds.

Theorem 7.4 matches the bounds obtained for the CRCW PRAM in [RR89]. (There is also a more involved, optimal CRCW PRAM algorithm that runs in $O(\lg n/\lg\lg n)$ time and linear work w.h.p.; see, e.g. [Mat92].)

We conclude this section with the following application of integer sorting to emulating the powerful FETCH&ADD PRAM on the CRQW PRAM.

Emulating Fetch&Add PRAM on CRQW PRAM

The FETCH&ADD PRAM model [GGK$^+$83, Vis83] is stronger than the CRCW PRAM; for instance, the parity and the prefix sums problems with input size n can be solved in constant time on a FETCH&ADD using n processors, while requiring $\Omega(\lg n/\lg\lg n)$ time on a CRCW PRAM when using n^c processors, for any constant $c > 0$. The following lemma gives a reduction from the problem of emulating one step of a FETCH&ADD PRAM on an EREW PRAM, to the integer sorting problem.

Lemma 7.5 ([MV95]) *Emulating one step of a* FETCH&ADD PRAM *with n processors and memory of arbitrary size m on an* EREW PRAM *can be reduced to $[1, n]$-integer sorting in $O(j\lg n)$ time and $O(n)$ work w.h.p., using $O(n\lg^{(j)} n)$ space, for any $j = 1, \ldots, \lg^* n$. In particular, it can be reduced:*

(ii) *to $[1, n]$-integer sorting, in $O(\lg n\lg^* n)$ time and $O(n)$ operations with high probability, using $O(n)$ space; and*

(iii) *to $[1, n]$-integer sorting, in $O(\lg n)$ time and $O(n)$ operations with high probability, using $O(n\lg^{(j)} n)$ space, for any constant $j > 0$.*

By using the CRQW integer sorting algorithm of Theorem 7.4 we obtain:

Theorem 7.6 *One step of an n-processor* FETCH&ADD PRAM *can be emulated on an $n/\lg n$-processor* CRQW PRAM *in $O(j\lg n)$ time w.h.p., and $O(n\lg^{(j)} n)$ space, for any $j = 1, \ldots, \lg^* n$. In particular, the emulation takes linear work and $O(\lg n\lg^* n)$ time w.h.p., using $O(n)$ space; and furthermore, for any constant j the emulation takes linear work and $O(\lg n)$ time w.h.p., using $O(n\lg^{(j)} n)$ space.*

The total work is, w.h.p.,

$$\sum_{1\leq i\leq \tau} O(n\lg n_i) = O(n\lg n).$$

The time for bitonic sort on groups of size at most $2^{\sqrt{\lg n}}$ is $O(\lg n)$, while the total work performed is $O(n\lg n)$ over all groups. Broadcasting whether any failure has occurred is done only after the bitonic sort, and takes $O(\lg n)$ time and linear work.

It follows that the entire algorithm runs in $O(\lg^2 n/\lg\lg n)$ time and $O(n\lg n)$ work w.h.p., using $O(n)$ space. ■

In [GMR96b], we consider the QRQW ASYNCHRONOUS PRAM model, a more asynchronous QRQW model in which individual processors may proceed at their own pace without waiting for the contention encountered by other processors. We show how to adapt the above QRQW PRAM sorting algorithm to obtain a fairly simple randomized sorting algorithm on the QRQW ASYNCHRONOUS PRAM that runs in $O(\lg n)$ time with $O(n\lg n)$ work w.h.p.

7.3 Integer sorting

The final class of sorting problems we consider is that of sorting an arbitrary collection of n integers in the range $[1..n\lg^c n]$, for a constant c. For this problem, we obtain an $O(\lg n)$ time, linear work randomized algorithm for the CRQW PRAM. In contrast, no algorithm with $O(\lg n)$ time and simultaneously $o(n\lg n)$ work is known for the CREW PRAM.

Theorem 7.4 *Sorting n integers in the range $[1..n\lg^c n]$, for any constant c, can be done in $O(\lg n)$ time and linear work w.h.p. on a* CRQW PRAM.

Proof. The integer sorting algorithm follows the steps of the Rajasekaran and Reif algorithm for the CRCW PRAM [RR89]. The main phase of the algorithm sorts the input keys based on their $\lg(n/\lg^3 n)$ least significant bits. Then Fact 4.3 can be applied to stably-sort the resulting sequence based on the $\lg(\lg^{c+3} n)$ most significant bits of the input keys, to obtain the final sorted sequence. In what follows, we list the steps of the main phase of the Rajasekaran-Reif algorithm, and then discuss how to implement the steps on a CRQW PRAM within the bounds stated in the theorem.

Let $D = n/\lg^3 n$ and for each input item, let its $\lg D$ least significant bits be its label.

1. Select in parallel $n/\lg^2 n$ input items drawn uniformly at random.
2. Sort these sample items according to their labels.
3. For each label $j \in [1..D]$, compute the number, N_j, of items in the sample with label j. Let $count_j = d(\lg^2 n)\max(N_j, \lg n)$, for a constant d. Rajasekaran and Reif show that for a suitable d, $count_j$ is an upper bound on the number of input items with label j and $\sum_{j=1}^{D} count_j \leq 2dn$, w.h.p.
4. Let B be an array of size $8dn$. Partition array B into subarrays such that the jth subarray is of size $4count_j$. Let $pointer_j$ be the starting point in B of the jth subarray.
5. Each item with label j reads $count_j$ and $pointer_j$.
6. Apply a multiple compaction algorithm to place each item into a private cell in the subarray for its label.

tree ensures that, if n searches are performed in parallel such that not too many searches result in the same leaf of the (non-fat) tree, then each step of the search will encounter low contention.

The process of fattening a search tree can be done in $O(\lg n)$ time and $O(n \lg n)$ work using binary broadcasting.

In the case of our sorting algorithm, at the ith level of recursion we make n_i copies of the median splitter, $n_i/2$ copies of the 1/4 and 3/4 splitters, and so forth, down to $n_i^{1/2+\epsilon}$ copies of the $n_i^{1/2-\epsilon}$ splitters in the leaves of the tree.[8] Since there are $\Theta(n_i^{1/2+\epsilon})$ items per splitter bucket w.h.p., it can be shown that at each step in the binary search, an item selecting a random copy of the splitter encounters constant expected contention. Thus by Observation 2.6, the maximum contention over all items at each step in the search is $O(\lg n/\lg\lg n)$ w.h.p. Thus each item can determine its bucket in $O(\lg n_i \lg n/\lg\lg n)$ time and $O(\lg n_i)$ work w.h.p.

At the ith level of recursion, there are n/n_i fat-trees, each of which uses $O(n_i \lg n_i)$ space. To reduce the space per fat-tree to $O(n_i)$, we initially make only some of the copies, and then reuse the space as needed. Specifically, we make n_i copies of the median splitter stored in an array A_0, $n_i/4$ copies of the 1/4 and 3/4 splitters stored in an array A_1, and in general $n_i/4^j$ copies of each splitter at the jth level of the fat-tree, for a total of $n_i/2^j$ copies of splitters stored in an array A_j. This is $O(n_i)$ copies in all. The processors begin by probing A_0, encountering constant expected contention. Then for each array A_j, $j > 0$, the contents of A_j are duplicated and stored in array A_{j-1}, in constant time and $O(n_i)$ work. The processors again probe A_0, which contains $n_i/2$ copies of the 1/4 and 3/4 splitters, followed by the duplication of all splitter copies, and so forth, alternating probe steps and duplication steps, until finally probing the $n_i^{1/2+\epsilon}$ copies of the $n_i^{1/2-\epsilon}$ splitters placed in A_0 in the previous duplication step. In this way, the maximum contention over all items at each step in the search is $O(\lg n/\lg\lg n)$ w.h.p. as before, while the space for all the fat-trees is $O(n)$.

This leads to the following theorem.

Theorem 7.3 *Algorithm $\mathcal{A}$ for sorting n arbitrary keys can be implemented on a* QRQW PRAM *in $O(\lg^2 n/\lg\lg n)$ time and $O(n \lg n)$ work w.h.p., using $O(n)$ space.*

Proof. The analysis proceeds as in Theorem 7.2. Since $n_i \leq (1 + 1/\lg n)^i \cdot n^{(1/2+\epsilon)^i}$ for all i, there exists a $\tau = \Theta(\lg\lg n)$ such that $n_\tau < 2^{\sqrt{\lg n}}$. Moreover, since $n_i \geq \lg^c n$, we have that, w.h.p., n_{i+1} will be an upper bound on the number of items with the same label, the subarrays designated for each label are of sufficient size, and the heavy multiple compaction will succeed — therefore the algorithm will complete without restarting.

We now analyze the QRQW PRAM complexity of Algorithm $\mathcal{A}$. Consider all $O(n/n_i)$ subproblems at the ith level of recursion. By Observation 2.6 and since $n_i > 2^{\sqrt{\lg n}}$, the maximum contention in step 1 is $O(\sqrt{\lg n})$ w.h.p. The work is $O(n/\sqrt{n_i})$. Step 2 can be done in $O(\lg n_i)$ time and $O(n)$ work by first making $\sqrt{n_i}$ copies of each item in the sample. For step 3, we build a binary search fat-tree of depth $\lg(n_i^{1/2-\epsilon})$, and then label each item using a random search into the fat-tree, as described above. This takes $O(\lg n_i \cdot \lg n/\lg\lg n)$ time w.h.p. and $O(n \lg n_i)$ work. Step 4 can be done in $O(\lg^* n_i \lg n/\lg\lg n)$ time and $O(n)$ work w.h.p. Thus the total time spent on all recursive calls is, w.h.p.,

$$\sum_{1 \leq i \leq \tau} O(\lg n_i \lg n/\lg\lg n) = O(\lg^2 n/\lg\lg n).$$

[8] A similar idea was used implicitly in [RV87] in the context of sorting on the cube-connected cycles network. In [RV87], multiple copies of the splitters are placed at nodes in the network. These are used to direct the routing of each item to a subnetwork designated for the splitter bucket in which its key belongs.

Proof. We first show that $n_i < n^{1/\lg\lg n}$ after $\tau = \Theta(\lg\lg\lg n)$ recursive calls to Algorithm $\mathcal{A}$. We claim that for all i:

$$n_i \leq (1+1/\lg n)^i \cdot n^{(\frac{1}{2}+\epsilon)^i}.$$

The proof of this claim is by induction on i. The case $i=0$ is straightforward. Assume that the claim holds for an arbitrary $i \geq 0$. We have that $n_{i+1} = (1+1/\lg n)\cdot n_i^{\frac{1}{2}+\epsilon}$, which by the inductive hypothesis is at most $(1+1/\lg n)((1+1/\lg n)^i \cdot n^{(\frac{1}{2}+\epsilon)^i})^{\frac{1}{2}+\epsilon}$. Since $\epsilon < 1/2$, we have that $n_{i+1} < (1+1/\lg n)^{i+1}\cdot n^{(\frac{1}{2}+\epsilon)^{i+1}}$, and the claim is proved. It follows that there exists a $\tau = \Theta(\lg\lg\lg n)$ such that $n_\tau < n^{1/\lg\lg n}$. Also, for all $i \leq \tau$, we have that $(1+1/\lg n)^i < (1+1/\tau)^\tau < e$.

Algorithm $\mathcal{A}$ applies the technique of *oversampling* as used in [RV87] to obtain a sample B' with better performance guarantees. Specifically, let X_i be the size of the largest group created for a given subproblem (of size at most n_i) at the ith level of recursion. Then from Lemma 7.1 in [RV87], we have

$$Pr\{X_i > (1+n_i^{-\epsilon/6})n_i^{\frac{1}{2}+\epsilon}\} = (n_i^{\epsilon/2}2^{n_i^{\epsilon/2}})^{-\omega(1)} \qquad (1)$$

Since $n_i \geq \lg^c n$ and $c > 6/\epsilon$,

$$\begin{aligned} Pr\{X_i > n_{i+1}\} &= Pr\{X_i > (1+1/\lg n)n_i^{\frac{1}{2}+\epsilon}\} < Pr\{X_i > (1+\lg^{-c\epsilon/6} n)n_i^{\frac{1}{2}+\epsilon}\} \\ &\leq Pr\{X_i > (1+n_i^{-\epsilon/6})n_i^{\frac{1}{2}+\epsilon}\} = o(n^{-c\epsilon/2}) \qquad \text{(by 1).} \end{aligned}$$

Thus w.h.p., n_{i+1} will be an upper bound on the number of items with the same label, the subarrays designated for each label are of sufficient size, and the heavy multiple compaction will succeed — therefore the algorithm will complete without restarting.

We now analyze the CRQW PRAM complexity of Algorithm $\mathcal{A}$. Consider all $O(n/n_i)$ subproblems at the ith level of recursion. Step 1 takes $O(1)$ time and $O(n/\sqrt{n_i})$ work. Step 2 takes $O(\lg n_i)$ time and $O(n)$ work. Step 3 takes $O(\lg n_i)$ time and $O(n \lg n_i)$ work. By Lemma 4.2 and the analysis in the previous paragraph, step 4 can be done in $O(\lg^* n_i \lg n/\lg\lg n)$ time and $O(n)$ work w.h.p. Thus the total time spent on all recursive calls is, w.h.p., $\sum_{1\leq i\leq\tau} O(\lg n_i + \lg^* n_i \lg n/\lg\lg n)$. Since $\lg n_i = O((1/2+\epsilon)^i \lg n)$ and $\lg^* n_i < \lg^* n$, the total time is, w.h.p.,

$$O((\tau \lg^* n/\lg\lg n)\lg n) + \sum_{1\leq i\leq\tau} O((1/2+\epsilon)^i \lg n) = O(\lg n).$$

The total work is, w.h.p.,

$$\sum_{1\leq i\leq\tau} O(n\lg n_i) = O(n\lg n).$$

The time for mergesort on groups of size at most $n^{1/\lg\lg n}$ is $O(\lg n)$, while the total work performed is $O(n\lg n)$ over all groups. Broadcasting whether any failure has occurred is done only after the mergesort, and takes $O(\lg n)$ time and linear work.

It follows that the entire algorithm runs in $O(\lg n)$ time and $O(n\lg n)$ work w.h.p. Moreover, all steps can be done in $O(n)$ space. ∎

To implement Algorithm $\mathcal{A}$ on a QRQW PRAM, we must replace all the high-contention read steps with techniques that use only low-contention steps. The main obstacle is step 3, in which each item needs to learn its position relative to the sorted sample. A straightforward binary search on B' would encounter $\Theta(n)$ contention. Instead, for the QRQW, we employ the following novel data structure:

Binary search fat-tree. In a *binary search fat-tree*, there are n copies of the root node, $n/2$ copies of the two children of the root node, and in general, $n/2^j$ copies of each of the 2^j distinct nodes at level j down from the root of the tree. The added fatness over a traditional binary search

- For the CRQW PRAM, we improve the space bound (to $O(n)$ space) over the CREW PRAM while maintaining the $O(\lg n)$ time, $O(n \lg n)$ work bounds.

These algorithms are arguably as simple as the ones cited earlier.

To obtain these improved results, we modify the $\sqrt{n}$-sample sort algorithm given above. In the last phase of our algorithm, we use a work-inefficient, but simple deterministic sorting algorithm. For our QRQW result, we use bitonic sorting [Bat68]; this runs in $O(\lg^2 n)$ time and $O(n \lg^2 n)$ work on an EREW PRAM. For our CRQW result, we use a parallelization of mergesort that applies Valiant's $O(\lg \lg n)$ time merging algorithm [Val75, BH85] at each round; this runs in $O(\lg n \lg \lg n)$ time with n processors on a CREW PRAM. (The work can be improved to $O(n \lg n)$, see e.g. [JáJ92].) Algorithm $\mathcal{A}$ below describes the generic modified algorithm.

Algorithm $\mathcal{A}$.

Let ϵ be any constant such that $0 < \epsilon < 1/2$. Let $n = n_0$ be the number of input items, and for $i \geq 1$, let

$$n_i = (1 + 1/\lg n) \cdot n_{i-1}^{\frac{1}{2}+\epsilon}.$$

W.h.p., n_i is an upper bound on the number of items in each subproblem at the ith recursive call to $\mathcal{A}$.

For subproblems at the ith level of recursion:

1. Let S be the set of at most n_i items in this subproblem. Select in parallel $\sqrt{n_i}$ items drawn uniformly at random from S.
2. Sort these sample items by comparing all pairs of items, using summation computations to compute the ranks of each item, and then storing the items in an array B in sorted order. Move every (n_i^{ϵ})th item in B to an array B'.
3. For each item $v \in S$, determine the largest item, w, in B' that is smaller than v, using a binary search on B'. Label v with the index of w in B'.
4. Place all items with the same label into a subarray of size $\Theta(n_i^{1/2+\epsilon})$ designated for the label, using heavy multiple compaction. W.h.p., the number of items with the same label is at most n_{i+1} and thus the heavy multiple compaction succeeds in placing all items in each such group into its designated subarray.
5. Recursively sort the items within each group, for all groups in parallel. When n_{i+1} is at most $n^{1/\lg \lg n}$, finish sorting the group using the CREW PRAM mergesort algorithm. Alternatively, for our QRQW PRAM result, when n_{i+1} is at most $2^{(\lg n)^{1/2}}$, finish sorting the group using the EREW PRAM bitonic sort algorithm. These cut-off points suffice for n sufficiently large; for general n, the cut-off points are $\max\left\{n^{1/\lg \lg n}, \lg^c n\right\}$ and $\max\left\{2^{(\lg n)^{1/2}}, \lg^c n\right\}$, respectively, for $c > 6/\epsilon$ a suitable constant.

We use "relaxed" heavy multiple compaction, which reports failure if a set size exceeds its upper bound count (recall the discussion at the end of Section 4.1). If failure is reported for any subproblem, we restart the algorithm from the beginning.

Algorithm $\mathcal{A}$ is readily implemented on a CRQW PRAM, as follows.

Theorem 7.2 *Algorithm $\mathcal{A}$ for sorting n arbitrary keys can be implemented on a* CRQW PRAM *in $O(\lg n)$ time and $O(n \lg n)$ work w.h.p., using $O(n)$ space.*

To obtain a sorted output it remains to sort within each subinterval. Each subinterval contains $O(\lg n)$ items w.h.p., and we assign one processor to the items in each subinterval. Each subinterval can be sequentially sorted in $O(\lg n)$ expected time by further dividing the subintervals into $\lg n$ buckets (sub-subintervals), having each processor assign its items to the appropriate bucket, and then having each processor use heapsort to sort within the buckets [MA80]. A more precise analysis [Hag89] shows that each processor fails to complete its sorting in $O(\lg n)$ time with probability less than $1/\lg n$ (the failure probability is in fact much smaller). We can achieve $O(\lg n)$ time w.h.p., as follows. Each processor applies the sequential sorting algorithm for $O(\lg n)$ steps. We expect $O(n/\lg n)$ processors to fail to complete their sorting, and by Fact 2.5, this occurs w.h.p. Use a parallel prefix sums algorithm to compact the unsuccessful subintervals and then assign $O(\lg n)$ processors to each such subinterval; each processor gets a constant number of unsorted items. In $O(\lg n)$ time, each processor compares its items against the other items in its assigned subinterval, computes their ranks within the subinterval, and places the items in the appropriate positions in the output array. Finally, the output array is compacted to size n using a parallel prefix sums algorithm.

At this point, w.h.p., the n numbers drawn from $U(0,1)$ are successfully sorted, and the stated time and work bounds are achieved w.h.p. However, for some inputs, e.g. when the number of items in a subinterval exceeds $4c\lg n$, we will have failed to sort the items. To obtain a Las Vegas algorithm, in such cases, we sort the input using a single processor; this does not affect the time and work bounds for the algorithm. ■

Theorem 7.1 matches the bounds obtained for the CRCW PRAM in [Chl89, Hag89]. (There is also a more involved $O(\lg n/\lg\lg n)$ time CRCW PRAM algorithm as implied by applying first the $O(\lg\lg n)$ padded-sorting algorithm of [MS91], followed by the $O(\lg n/\lg\lg n)$ prefix sums algorithm of [CV89].)

7.2 General Sorting

In this section we consider the problem of general sorting, i.e. sorting an arbitrary collection of n keys from some totally-ordered set. On the EREW PRAM, there are two known $O(\lg n)$ time, $O(n\lg n)$ work algorithms for general sorting [AKS83, Col88]; these deterministic algorithms match the asymptotic lower bounds for general sorting on the EREW and CREW PRAM models. Unfortunately, these two algorithms are not as simple and practical as one would like. Simple parallel $O(n\lg n)$ work algorithms for sorting include a simple straightforward parallelization of mergesort that runs in $O(\lg^2 n)$ time on a CREW PRAM and an $O(\lg^2 n)$ time randomized quicksort algorithm on an EREW PRAM (see, e.g. [JáJ92]).

Another relatively simple parallel sorting algorithm is a randomized $\sqrt{n}$-sample sort algorithm for the CREW PRAM that runs in $O(\lg n)$ time, $O(n\lg n)$ work, and $O(n^{1+\epsilon})$ space [Rei85].[7] This algorithm consists of the following high-level steps: (1) randomly sample $\sqrt{n}$ keys, (2) sort the sample by comparing all pairs of keys, (3) each item determines by binary search its position among the sorted sample and labels itself accordingly, (4) sort the items based on their labels using integer sorting, and (5) recursively sort within groups with the same label. When the size of a group is at most $\lg n$, finish sorting the group by comparing all pairs of items.

We build on this $\sqrt{n}$-sample sort algorithm and obtain the following two results:

- For the QRQW PRAM, we obtain an $O(\lg^2 n/\lg\lg n)$ time, $O(n\lg n)$ work, $O(n)$ space randomized sorting algorithm, thus improving the time bound by a factor of $\lg\lg n$ over the EREW PRAM quicksort algorithm.

[7] The algorithm in [Rei85] uses $\Theta(n)$ memory locations of size $O(\sqrt{n}\lg n)$ bits. Under the standard assumption for the PRAM, adopted as well in this paper, that each memory location is of size $O(\lg n)$ bits, the algorithm in [Rei85] uses $\Theta(n^{1.5})$ space. This has been improved to $O(n^{1+\epsilon})$ space, for any constant $\epsilon > 0$ (see, e.g. [JáJ92]).

number of active buckets is, w.h.p., smaller than the number of memory blocks. In such cases, the contention to a memory block in step 2(a) is a binomial random variable with an expected value less than 1. It follows by Observation 2.6 that w.h.p., the maximum contention to a memory block is $O(\lg n/\lg\lg n)$. By Fact 6.3, all buckets contain $O(\lg n/\lg\lg n)$ keys w.h.p. Thus, in a constant number of steps of $O(\lg n/\lg\lg n)$ contention w.h.p., keys of each active bucket can learn if their bucket is allocated with a memory block, read the random linear function selected by their bucket, and test for injectiveness.

The work for an iteration of step 2 is bounded by the number of keys in active buckets; Gil and Matias [GM94a] show that w.h.p. this number decreases faster than a geometric series. Thus step 2 of the algorithm can be described in a QRQW work-time presentation as a geometric decaying algorithm with $O(n)$ work, consisting of $O(\lg\lg n)$ steps, each with contention $O(\lg n/\lg\lg n)$ w.h.p.

This implies an $O(\lg n)$ time $O(n)$ work algorithm that, by using Theorem 2.3 and Theorem 2.4, can be implemented on a QRQW PRAM in $O(\lg n)$ time, using $n/\lg n$ processors.

We now analyze the lookup queries algorithm. By Lemma 6.4, $h(x)$ can be computed for each query key in parallel in $O(\lg n/\lg\lg n)$ time and linear work w.h.p. By Fact 6.3, at most $O(\lg n/\lg\lg n)$ query keys map to any single bucket w.h.p. Thus the contention encountered for a query key to read its block address, its secondary hash function, and its hash table location is $O(\lg n/\lg\lg n)$ w.h.p.

This completes the proof of Theorem 6.1. ■

7 Sorting

In this section, we present results for three classes of sorting algorithms. First, we consider sorting keys drawn uniformly at random, and present an $O(\lg n)$ time, linear work w.h.p. algorithm. Second, we consider sorting general keys, and present two simple, work-optimal, comparison-based sorting algorithms, one running in $O(\lg^2 n/\lg\lg n)$ time w.h.p. and the other running in $O(\lg n)$ time w.h.p. Third, we consider sorting small integer keys, and present an $O(\lg n)$ time, linear work w.h.p. algorithm. We apply this result to obtain an $O(\lg n)$ time, linear work w.h.p. algorithm for emulating the powerful FETCH&ADD PRAM. The first two results are for the QRQW PRAM model; the latter three are for the stronger CRQW PRAM model.

7.1 Distributive Sorting

The *sorting from* $U(0,1)$ problem is to sort n numbers chosen uniformly at random from the range $(0,1)$. As indicated in Table 1, the best known linear work EREW PRAM algorithm for this problem runs in $O(n^\epsilon)$ time, for fixed $\epsilon > 0$. EREW PRAM algorithms that run in polylog time are work inefficient by at least a $\sqrt{\lg n \lg\lg n}$ factor. We obtain the following:

Theorem 7.1 *Sorting from* $U(0,1)$ *can be done in* $O(\lg n)$ *time and linear work w.h.p. on a* QRQW PRAM.

Proof. First partition the real interval $(0,1)$ into $n/\lg n$ subintervals. It follows from Fact 2.5 that the number of input items in each subinterval is with high probability at most $c\lg n$ for some constant c. We allocate to each subinterval an array of size $4c\lg n$ and employ our multiple compaction algorithm (Theorem 4.1) to place each input item in a private cell in the subarray allocated to its subinterval.

The Gil and Matias algorithm sketched above requires a careful selection of its constants and parameters, so that $O(\lg\lg n)$ iterations provably suffice. Likewise, our adaptation of their algorithm requires a careful selection of its constants and parameters to leverage their analysis and obtain the desired result, as follows. In selecting the hash function that defines the buckets, it suffices to take $\mathcal{R}' = \mathcal{R}^{d_1,d_2}(k,n)$ with $d_1 = 7$, $d_2 = 11$, and $k = n^{3/7}$. Let $\lambda = 18/13$, and let $t^* = 2\lg\lg n/\lg\lambda$ be the number of iterations. Let $t' = \lceil t/2 \rceil$. For $t = 1, 2, \ldots, t^*$, let x_t, the block size at iteration t, and m_t, the number of blocks at iteration t, be:

$$\begin{aligned} x_t &= 2^{a\lambda^{t'}+b_1t'+c_1} \\ m_t &= n2^{-a\lambda^{t'}-b_2t'+c_2}, \end{aligned}$$

where $a = 8/13$, $b_1 = 1/5$, $b_2 = 9/20$, $c_1 = 73/25$, and $c_2 = 89/20$ (these are the same constants used in the Gil and Matias algorithm). Then the QRQW hashing algorithm is:

Constructing a hash table.

1. Select a random hash function h from $\mathcal{R}'$, duplicate the parameters of h, and partition the input set into n buckets according to h.
2. For $t := 1$ to t^* do
   - (a) *Allocation:* Allocate m_t memory blocks, each of size x_t. Let each bucket select a block at random, and try to claim it by writing the bucket number in a designated memory cell.
   - (b) *Hashing:* Each bucket that successfully claimed an allocated block in the previous step tries to injectively map its keys into the block using a random linear hash function from $\mathcal{H}^1_{x_t}$. If it succeeds, record the description of the hash function and the address of the memory block for that bucket. Buckets that fail carry on to the next iteration. For the last iteration, $t = t^*$, repeat this hashing substep a total of 8 times.
3. If there are any buckets that have yet to succeed, return to step 1 and restart the algorithm from the beginning.

Lookup queries for n distinct keys are performed as follows:

Lookup queries.

1. For each query key x, $h(x)$ is computed to locate the memory block for this bucket and the secondary hash function h_i, $i = h(x)$, used within this block.
2. The key x is in the hash table if and only if location $h_i(x)$ of this memory block contains the key x.

Proof of Theorem 6.1. We first analyze the hash table construction algorithm, then the lookup queries algorithm.

By Lemma 6.4, step 1 of the hash table construction algorithm takes $O(\lg n)$ time and linear work w.h.p. As for step 2, Gil and Matias [GM94a] show that, for their algorithm, the number of active buckets decreases more rapidly than the number of memory blocks, and hence w.h.p., all buckets have become inactive after $O(\lg\lg n)$ iterations. A straightforward adaptation of their analysis to our algorithm (which uses hash functions from $\mathcal{R}'$), shows that w.h.p., all buckets have become inactive after t^* iterations. Thus w.h.p., the algorithm will not be restarted. Step 3 can be performed in $O(\lg n)$ time and linear work, using an OR computation.

To complete the analysis for the QRQW PRAM, we determine the contention encountered in step 2. For each active bucket we have a processor standing by that acts in step 2(a) in claiming a memory block, and in step 2(b) in selecting a random function from $\mathcal{H}^1_{x_t}$. As argued above, the

6.2 The hashing algorithm

Our algorithm is based on an $O(\lg\lg n)$ time CRCW hashing algorithm of Gil and Matias [GM94a, GM94b] (see also [GM91]). Their algorithm uses a technique of oblivious execution that circumvents the need to learn the bucket sizes b_i^h, in order to allocate appropriately-sized memory blocks and construct the second level functions h_i. We first sketch the high-contention CRCW algorithm and then derive our low-contention QRQW algorithm.

1. Partition the input set into n buckets by a random hash function from $\mathcal{H}_n^d$, where d is an appropriate constant.
2. For $t := 1$ to $O(\lg\lg n)$ do
   (a) *Allocation:* Allocate m_t memory blocks, each of size x_t, where m_t and x_t are carefully selected parameters (x_t behaves as 2^{λ^t} for some constant λ and $m_t \approx n/2^t x_t$). Let each bucket select a block at random, and try to claim it by writing the bucket number in a designated memory cell.
   (b) *Hashing:* Each bucket that successfully claimed an allocated block in the previous step tries to injectively map its keys into the block using a random linear hash function from $\mathcal{H}^1_{x_t}$. If it succeeds, it records the description of the hash function and the address of the memory block for that bucket. Buckets that fail carry on to the next iteration.

The algorithm above is a high-contention one, since the bucket sizes when using a hash function from $\mathcal{H}_n^d$ may be polynomially large, while the memory block sizes x_t are small (e.g. x_0 is a constant). To obtain an efficient low-contention algorithm, we first replace the polynomial class $\mathcal{H}_n^d$ in step 1 with the class $\mathcal{R}$ defined above, taking $k = n^{1-\delta}$, $0 < \delta < 1/2$; functions from this class have relatively small bucket sizes (Fact 6.3). The disadvantage of using functions from $\mathcal{R}$ is that each function $h \in \mathcal{R}$ is represented by $n^{1-\delta} + \Theta(1)$ numbers that need to be selected at random in an initialization step, and then used to evaluate in parallel $h(x)$ for $x \in S$ as well as any subsequent query set. A straightforward implementation of this evaluation results in polynomial contention. We devise a low-contention scheme for the evaluation, yielding the following result:

Lemma 6.4 *A function h can be selected at random from $\mathcal{R}$ and preprocessed for efficient evaluation in $O(\lg n)$ time and linear work w.h.p. Subsequently, for any set $S \subset U$ of size n, $h(x)$ can be evaluated in parallel for all $x \in S$ on a* QRQW PRAM *in $O(\lg n/\lg\lg n)$ time and linear work w.h.p.*

Proof. Recall that $h = \langle f, g, a_1, a_2, \ldots, a_{n^{1-\delta}}\rangle$, for some constant δ, where each a_j is selected at random from $\{0, \ldots, n-1\}$. These $n^{1-\delta} + \Theta(1)$ parameters are selected by as many processors and then duplicated in $O(\lg n)$ time and linear work, using a simple binary broadcasting algorithm: the functions f and g are duplicated n times and each of the a_j is duplicated $4n/n^{1-\delta} = 4n^\delta$ times. The total representation requires linear space.

Recall that for a key $x \in S$, we compute $h(x) := (g(x) + a_{f(x)}) \bmod n$. Thus, for each key we need to read the values of f, g, and $a_{f(x)}$. Reading f and g is easy: the i'th key reads the i'th copies of these two functions. The main difficulty is in reading $a_{f(x)}$ as contention cannot be entirely avoided. For each key $x \in S$, a processor allocated to the key evaluates $f(x)$ and then chooses at random one of the copies of $a_{f(x)}$ and reads it. By Fact 6.2,

$$\textbf{Prob}\left(b_i^f \le 2n^\delta \text{ for } 0 \le i < n^{1-\delta}\right) \ge 1 - O(n^{1-\delta-\delta d_1/2}) \ .$$

Therefore, w.h.p. the contention distribution obtained in the read step of $a_{f(x)}$ is upper bounded by a distribution obtained by $n^{1-\delta}$ instances of throwing $2n^\delta$ balls into $4n^\delta$ urns at random. In particular, it follows from Fact 2.5 that the maximum contention is $O(\lg n/\lg\lg n)$ w.h.p. ∎

Theorem 6.1 *A hash table for S can be constructed in $O(\lg n)$ time and linear work w.h.p. on a* QRQW PRAM. *Subsequently, lookup queries for n given distinct keys can be completed in $O(\lg n / \lg\lg n)$ time and linear work w.h.p. on a* QRQW PRAM.

The set S of keys to be stored in the hash table as well as the set of keys appearing in lookup queries can be arbitrary subsets of U. We assume that the choice of sets is independent of the random bits used by the algorithm. Our result is for distinct keys. As shown in Table 1, the best known linear work EREW PRAM algorithm for this problem runs in $O(n^\epsilon)$ time.

6.1 Basics

Consider the universe $U = \{0, 1, \ldots, q-1\}$ where q is some prime. A *hash function* h, $U \overset{h}{\mapsto} [0, \ldots, s-1]$, maps the universe U into a smaller universe of size s. Given a set $S \subset U$ of size n, the hash function h splits S into buckets $B_i^h := \{x \in S \mid h(x) = i\}$ of sizes $b_i^h = |B_i^h|$, $0 \le i < s$. The function h is *c-perfect for S* if $b_i^h \le c$ for all $0 \le i < s$; h is *perfect* for S if it is 1-perfect for it.

Let d be a constant. The class of d-degree *polynomial hash functions* is defined as follows:

$$\mathcal{H}_s^d := \left\{ h \;\middle|\; h(x) := \left(\sum_{i=0}^{d} a_i x^i \bmod q \right) \bmod s,\ a_i \in U \right\}.$$

Fact 6.2 ([KRS90]) *Let h be selected at random from $\mathcal{H}^d_{n^{1-\delta}}$. Then, for each i, $i = 1, \ldots, n^{1-\delta}$*

$$\mathbf{Prob}\left(b_i^h > 2n^\delta\right) = O(n^{-\delta d/2}) \ .$$

The class $\mathcal{H}_s^1$ is denoted the class of *linear hash functions*.

Siegel [Sie89] and then Dietzfelbinger and Meyer auf der Heide [DM90] showed how polynomial hash functions can be combined to create a new class of hash functions. The class $\mathcal{R} = \mathcal{R}^{d_1,d_2}(k,n)$ of hash functions, defined in [DM90], is the set of all $(k+2)$-tuples $h = \langle f, g, a_1, a_2, \ldots, a_k \rangle$, where $f \in \mathcal{H}_k^{d_1}$, for some constant d_1, $g \in \mathcal{H}_n^{d_2}$, for some constant d_2, and $a_1, a_2, \ldots, a_k \in \{0, \ldots, n-1\}$. The action of $h \in \mathcal{R}$ on $x \in U$ is defined as $h(x) := (g(x) + a_{f(x)}) \bmod n$.

With high probability, a random hash function from $\mathcal{R}$ has a distribution of bucket sizes that is very close to that of a truly random function. In particular:

Fact 6.3 ([DM90]) *Let $0 < \delta < 1/2$ and let $k = n^{1-\delta}$. For h randomly chosen from $\mathcal{R}$, h is $O(\lg n / \lg\lg n)$-perfect with high probability.*

The two-level hashing scheme. Fredman, Komlos, and Szemeredi [FKS84] introduced a simple and elegant two-level scheme for constructing a perfect hash function: a first-level hash function h partitions the input set S into n buckets B_i^h, $0 \le i < n$; this function is constructed in a first phase and is assumed to imply a certain distribution on the bucket sizes b_i^h. For each bucket B_i^h, a private memory block of appropriate size is allocated and a second-level function h_i maps the elements of B_i^h injectively into its block; these functions are constructed in a second phase. Fredman, Komlos, and Szemeredi showed that both the first level and the second level can be constructed in linear expected time, by using linear hash functions only, and by allocating to each bucket B_i^h a memory block of quadratic size $O((b_i^h)^2)$.

in Table 2 on the choice resulting in the best performance, namely, an initial subarray of size $4n-1$. Note that the inactive 15K processors are used solely for the extra memory they provide; only the active 1K processors execute useful steps in the program.

The second column of timings in Table 2 shows the results of these experiments. As can be seen from this table, the QRQW algorithm is over three times faster than the EREW algorithm, and the dart-throwing with SCANs algorithm is in between.

Asymptotic analysis of the implemented algorithms. We provide an asymptotic analysis of the implemented algorithms to determine if the relative order of the analyzed bounds corresponds to the relative order of the measured performance on the MP-1. We consider two possible models on which to base our analysis: the SIMD-QRQW PRAM described at the end of Section 2.1, and the SCAN-SIMD-QRQW PRAM, defined to be a SIMD-QRQW PRAM augmented with a unit time SCAN operation. As mentioned above, features of the MP-1 are more closely reflected in the SIMD-QRQW PRAM model. Considering both the SIMD-QRQW PRAM and SCAN-SIMD-QRQW PRAM models allows us to explore whether the builtin SCAN operations on the MP-1 should be considered unit time operations when modeling the MP-1.

We analyze the three implemented algorithms in turn.

The sorting-based algorithm uses bitonic sorting (the sorting method employed by the MP-1 system sort routines), and hence takes $O(\lg^2 n)$ time w.h.p. on the n-processor SIMD-QRQW PRAM or SCAN-SIMD-QRQW PRAM (same bounds as for the EREW PRAM).

The first dart-throwing algorithm takes $O(\lg n \lg\lg n)$ time w.h.p. on the n-processor SIMD-QRQW PRAM, and is readily shown to take $O(\lg n)$ time w.h.p. on the n-processor SCAN-SIMD-QRQW PRAM. (A more careful analysis for the SCAN-SIMD-QRQW PRAM yields a time bound that is slightly sublogarithmic.)

The random permutation algorithm given in Theorem 5.1 takes $O(\lg n)$ time w.h.p. on the n-processor SIMD-QRQW PRAM. On the n-processor SCAN-SIMD-QRQW PRAM, the time is again slightly sublogarithmic.

We conclude that for the particular implementations studied above, the relative order according to the SIMD-QRQW PRAM matches the observed performance, and to a lesser extent, the same can be said for the SCAN-SIMD-QRQW PRAM. The SIMD-QRQW PRAM has the advantage over the SCAN-SIMD-QRQW PRAM in predicting the faster of the two dart-throwing algorithms.

Related experimental results. Recall that the random permutation algorithm described in Theorem 5.1 permitted each processor to have multiple reads/writes in progress at a time, and that this pipelining feature was exploited to obtain a work-optimal algorithm on the QRQW PRAM. On the MasPar, however, each processor can have at most one read/write in progress at a time, so we were not able to exploit this aspect of the algorithm (and in fact the resulting implemented algorithm is not work-optimal). Recently, the random permutation algorithm described in Theorem 5.1 was implemented on an 8-processor CRAY J90, a parallel vector machine that permits this pipelining feature. This algorithm was compared with the fastest known sorting-based random permutation algorithm on the CRAY J90, and was shown to be considerably faster over a range of problem sizes (e.g. a factor of 2.5 faster in generating a random permutation for $n = 16,384$) [BGMZ95].

6 Parallel hashing

Given a finite universe U and a set $S \subset U$ of size n, the *hashing* problem is to construct a linear-size data structure (a "hash table") that can support lookup operations, i.e. queries of the type "is $x \in S$," for any $x \in U$. We show:

A number of builtin library routines are provided with the MPL language, including primitives for routing on the multistage network or the X-Net, for various SCAN operations, and for random number generation. The timings were done using the timing functions provided with MPL, and did not include the cost of generating an initial random seed for each processor at the start of the experiments.

In our first set of experiments, we compare the following three randomized Las Vegas algorithms, for $16,384$ processors ($n = p = 16,384$):

- **A sorting-based algorithm:** Each processor selects a random number between 1 and $2^{31}-1$. These numbers are sorted, and $\pi(i) =$ the rank of i's number in the sorted order. In the unlikely event that two processors select the same number, we repeat the algorithm.

  We use a builtin library routine for the sorting and ranking (`rank32`) and for detecting if the algorithm needs to be repeated (`globalor`). This is arguably the simplest and most popular EREW PRAM algorithm for random permutation.

- **A dart-throwing algorithm using SCAN:** At each iteration, until all items have been placed: Each unplaced item selects a random cell from an array A of size $n-1$; an item succeeds in claiming a cell if no other item selects the same cell this iteration. (This is detected using the "write,read,write,read" procedure outlined at the beginning of Section 5.1.) Compact the successful items in A and transfer them to locations $\pi(K+1), \pi(K+2), \ldots, \pi(K+k)$, where K is the number of items that succeeded in previous iterations and k is the number of items that succeeded in this iteration. Array π will contain the random permutation.

  We use a builtin SCAN-type routine for the compaction (`enumerate`) and for detecting when all items have been placed (`globalor`).

- **A dart-throwing algorithm for the QRQW:** We implement the algorithm described in Theorem 5.1, using n processors (and no reallocation), and taking the initial subarray size to be $2n-1$.

  We use a builtin library routine for detecting when all items have been placed (`globalor`) and for the compaction at the end (`scanAdd16`).

The MP-1 provides for single-step data parallel operation on plural variables, i.e. parallel operation on p data items, one per processor. In the initial iterations of the dart-throwing algorithm for the QRQW, p processors throw darts into a subarray of size m, for some m greater than p; however parallel operation on p data items out of a larger set m of possible data items is not efficiently supported by the MP-1. We employ m/p plural variables to represent the subarray of size m. We emulate each dart throwing step by m/p substeps cycling through these plural variables, such that each processor throws its dart only during the substep for the plural variable containing its randomly selected cell. This overhead increases with m; on the other hand, decreasing m results in a lower success probability for each item and hence extra iterations may be needed before all items succeed in claiming a cell. With this trade-off in mind, we have explored a range of possible array sizes for each of the dart-throwing algorithms, and selected the one that resulted in the best performance.

The first column of timings in Table 2 shows the results of these experiments. Both dart-throwing algorithms outperform the EREW algorithm, with the QRQW algorithm the fastest.

In our second set of experiments, we explore the performance of the three algorithms on an optimistic configuration of the MP-1. In particular, we employ only 1024 processors of the MP-1, one per cluster, so that each processor has its own input port and output port to the multistage network. Moreover, we use plural variables that are the full size of the machine, permitting one-step parallel operation on $p = 1024$ data items out of a larger set $m \leq 16,384$ of possible data items (overcoming the bottleneck described above). This improves the relative performance of the QRQW algorithm. For this configuration, we again explored a range of possible initial array sizes, and report

The implementation of the algorithm described above on a QRQW PRAM is similar to the implementation of the heavy multiple compaction algorithm. That is, it can be described in an $O(\lg n)$-spawning model and be implemented using Corollary 3.8, or it can be implemented directly as in the proof of Lemma 4.2. The theorem follows. ∎

5.2 Preliminary experimental results

We have performed several illustrative experiments comparing random permutation algorithms; these experiments were performed on a 16,384 processor MasPar MP-1 [Mas91]. The goal was to see whether a good QRQW algorithm would outperform the popular EREW algorithm. We have implemented the random permutation algorithm given in Theorem 5.1, as well as a variant of this algorithm that uses more extensively the builtin library routine provided by the MP-1 for performing SCAN operations, and compared their performance to the popular sorting-based EREW random permutation algorithm.[6]

We perform two sets of experiments. In the first set, we use all 16,384 processors to generate random permutations of $\{1, \ldots, 16384\}$, i.e. we study the case where $n = p = 16,384$. Then in the second set, we use only 1024 processors of the full machine to generate random permutations of $\{1, \ldots, 1024\}$, i.e. we study the case where $n = p = 1024$. The results are shown in Table 2. In both cases, the QRQW algorithm described in Theorem 5.1 is the fastest. In the rest of this section, we present the details of our experiments. We begin with a brief description of the MasPar MP-1.

Random Permutation on the MasPar MP-1		
algorithm	**16K proc.**	**1K proc.**
sorting-based (EREW)	11.25 ms	10.01 ms
dart-throwing with SCANs	8.02 ms	6.05 ms
dart-throwing for QRQW	7.57 ms	2.88 ms

Table 2: Each running time represents the average of generating 1000 random permutations of $\{1, \ldots, p\}$, where p is the number of processors. The experiments with 1K processors were run on the same machine as the experiments with 16K processors, but using only one processor per router cluster. See the text for more details.

In the MasPar MP-1, the 16,384 processors are connected by a mesh-like point-to-point network called the *X-Net*, as well as by a multistage network used for global routing. Processors are partitioned into clusters, such that the 16 processors in a cluster share a single output port and a single input port to the multistage network. Each processor has 16K bytes of local memory; processors can read or write to locations in each other's local memories using either network. The MP-1 is a SIMD machine.

In SIMD machines, the processors execute in lock-step; thus if any processor is delayed due to contention at a location, all processors are delayed. On the MasPar, processors wait after each read/write for the read/write with the maximum contention. This feature is captured by the SIMD-QRQW PRAM model.

Our implementations were done using version 2.0 of the system software provided for the MP-1. The programs were written in the MPL language, an extension of C that permits data parallel operations. MPL provides "plural" versions of many C data types, for defining variables suitable for data parallel operation. A `plural int` for example is a data type with an integer on each processor; adding two `plural int` variables results in a `plural int` variable that is the component-wise sum.

[6] Random cyclic permutation algorithms (such as those given in Theorem 5.2 and Theorem 5.3) were not considered in our comparison.

Note that detecting whether we are done and notifying all the processors requires $\Omega(\lg n)$ time, by Theorem 3.1, so this cannot be done. We can ensure, however, that the algorithm always produces a valid random cyclic permutation, by adding the following steps to handle the unlikely scenario where there are unplaced items or items whose successors have not been determined. Let x be a memory location apart from the array A. Any processor assigned an item that remains unplaced or without a known successor writes its ID to x; the resulting value in x designates the processor that will complete the work sequentially. The designated processor checks each item to see if it is unplaced, and if so, attempts to place the item into a random cell of A until it succeeds in finding an unclaimed cell. Finally, after all the items have been placed, the processor steps through A to determine the successors for all items, and fills in the output array. Thus we have a Las Vegas algorithm, but since we do not inform all the processors when the algorithm completes, some processors may not know when it is safe to use the output.

To complete the proof of the theorem, we show that the time and work for the algorithm matches the bounds stated in the theorem. Step 1 is $O(nf)$ work and, by Observation 2.6, $O(f)$ contention w.h.p. Step 2 is $O(n)$ work and $O(1)$ contention. Step 3 is $O(f)$ substeps of $O(n)$ work and $O(1)$ contention each. Step 4 is $O(n)$ work and $O(1)$ contention. The sequential cleanup phase described in the previous paragraph occurs with polynomially small probability, and can be ignored in the analysis. ∎

5.1.3 An efficient random cyclic permutation algorithm

We next show how to solve the random cyclic permutation problem in sublogarithmic time and linear work. The algorithm is based on an $O(\lg^* n)$ time CRCW PRAM algorithms for linear compaction and random permutation [MV91a].

Theorem 5.3 *The random cyclic permutation problem can be solved by a* QRQW PRAM *algorithm in $O(\lg n \lg^* n / \lg\lg n)$ time and linear work w.h.p.*

Proof. We adapt the heavy multiple compaction algorithm from Section 4.1 as follows. First, we consider the special case where there is but a single label. Second, we permit an item to claim a cell only if it is the only item attempting to claim the cell, to ensure that the items are placed at random into the array. Third, after completing all the rounds of the log-star paradigm, we determine the successor for each item, using the approach described in Theorem 5.2, as follows. Consider a binary tree imposed on A and walk up the tree $2\lg\lg n$ levels: At each node, v, maintain a linked list of the items in the subtree rooted at v by linking the rightmost item in v's left subtree with the leftmost item in v's right subtree. Then for each node, v, at level $2\lg\lg n$, link v's rightmost item to the leftmost item of the next node to v's right at this level (with wrap-around). This finds successors for all items whose successors are within a distance of $\lg^2 n$ cells. We complete the algorithm by having each item, i, with successor j, write j to the ith output cell. A Las Vegas algorithm can be obtained by following the procedure given in Theorem 5.2.

The analysis of the heavy multiple compaction algorithm using the $q_{i'}$-spawning model given in Section 4.1 can be readily adapted to show that the time for each of the $O(\lg^* n)$ rounds is $O(\lg n / \lg\lg n)$ w.h.p., that the overall work is $O(n)$ w.h.p., and that w.h.p., all items are placed prior to finding the successors. Walking up the tree takes $O(\lg\lg n)$ time and $O(n)$ work (the work is linear here since the tree has only $O(n)$ nodes). To analyze the probability that all successors will be found in walking up the tree, consider an arbitrary subarray of A of size $\lg^2 n$. Each dart hits a cell in the subarray with probability $p = \lg^2 n / cn$, where cn is the size of A, c a constant. The probability that no item is in the subarray is less than $(1-p)^n < 1/e^{\lg^2 n / c}$. It follows that w.h.p., all subarrays of A of size $\lg^2 n$ have at least one item. In particular, for any given item, the subarray starting just to its right in A (with wrap-around) will contain its successor w.h.p.

compression step) then the time is sublogarithmic, and can be somewhat improved if the algorithm from [MV91a] is used. Such an algorithm is actually described in the proof of Theorem 5.3.

5.1.2 A fast random cyclic permutation algorithm

For random cyclic permutation, we observe that the contention during the dart throwing can be reduced by using a larger array; this was the technique used in the linear compaction algorithm given in [GMR96a]. However, this reduction in contention due to throwing into a larger array must be balanced against the additional time spent by an item finding its successor in the larger array. Consider an array of size $O(n2^f)$, for $\lg\lg n \leq f \leq \lg n$, into which n random darts are thrown. By Observation 2.6, the maximum contention will be $O(\lg n/f)$ w.h.p.; the maximum gap between darts can be shown to be $O(2^f)$ w.h.p. Successors can be found in time logarithmic in the maximum gap. Hence we have an $O(\lg n/f + f)$ time requirement for this approach, which is minimized when $f = \sqrt{\lg n}$. The algorithm given below is based on this approach. Since the contention at each round of dart throwing is $O(\sqrt{\lg n})$, even after many of the items have been placed, we aim for only a constant number of rounds.

Theorem 5.2 *The random cyclic permutation problem can be solved by an n-processor* QRQW PRAM *algorithm in $O(\sqrt{\lg n})$ time w.h.p.*

Proof. Let A be an auxiliary array of size $m = nf2^{c \cdot f}$, where $f = \sqrt{\lg n}$, for a constant $c \geq 1$ determined by the analysis.

1. Each item attempts to claim f random cells in A; an attempt succeeds if there is no other claim on that cell.
2. W.h.p., each item will have at least one claimed cell. Each item marks all but its first such claimed cell as unclaimed.
3. Each item finds its successor in A (with wrap-around), as follows. Consider a binary tree imposed on A. Each item begins at its leaf and walks up the tree level by level for at most $2cf$ levels, until it encounters an item to its left and to its right in A. In particular, at each node, v, we maintain a linked list of the items in the subtree rooted at v by linking the rightmost item in v's left subtree with the leftmost item in v's right subtree. Then, for each item that is the rightmost item in its subtree at level $2cf$ (and hence has failed to find its successor), link the item to the leftmost item (if any) in the subtree immediately to its right at this level. Note that this finds successors for all items whose successors are within a distance of 2^{2cf} cells.
4. For each item, i, with successor j, write j to the ith output cell.

The probability of an item failing to be placed in step 1 is less than

$$(nf/m)^f = (1/2^{cf})^f = 1/2^{c\lg n} = 1/n^c.$$

To analyze the probability that all successors will be found in step 3, consider an arbitrary subarray of A of size 2^{2cf}. Each dart hits a cell in the subarray with probability $p = 2^{2cf}/m$. The probability that no item is in the subarray is less than

$$(1-p)^n < (1/e)^{pn} = 1/e^{2^{2cf}/f2^{cf}} = 1/e^{2^{c\sqrt{\lg n}}/\sqrt{\lg n}} \ .$$

It follows that w.h.p., all subarrays of A of size 2^{2cf} have at least one item. In particular, for any given item, the subarray starting just to its right in A will contain its successor w.h.p. Thus w.h.p., the above algorithm outputs a random cyclic permutation.

failures; this ensures that the policy for arbitrating between multiple writers to a cell does not bias the random permutation. At the end of this first step, the relative order of the items in the array gives an implicit random permutation. In the second step, the items are compressed into an array $[1..n]$, in order to compute the permutation explicitly.

A simple compression can be obtained by compacting the items using a prefix sums algorithm [MR89, RR89]. An alternative compression technique that circumvents the need for compaction was presented in [MV91a]: each item in the linear size array finds its neighboring item, and points to it; using the pointers all items can be placed in an array $[1..n]$ in constant time, resulting with a random cyclic permutation. (A general random permutation is obtained in [MV91a] by breaking the global cycle into smaller cycles in an appropriate manner, using a prefix-minima computation.)

The difference between the two compression techniques is illustrated by the following example. Let $n = 5$, and consider the items placed at random into an array of size 10, as follows:

4		5	2				1	3	

In the first technique, the items are compacted in order, yielding the permutation on the right in Figure 1. In the second technique, the items specify the cycle representation, yielding the permutation on the left in Figure 1.

In each of the QRQW PRAM algorithms in this section, we need to detect whether a processor attempting to claim a cell x succeeds, i.e. whether the attempt is the only claim on cell x. This is accomplished for all attempts over all cells in a constant number of steps as follows. Each processor first writes its index into its selected cell, then reads the cell. Any processor that does not read its own index has detected multiple claims on that cell and hence has failed to claim the cell; it writes again to the cell. Finally, each processor that did read its own index reads again the cell; if the cell no longer contains its index, it has failed to claim the cell, otherwise it has succeeded.

5.1.1 A random permutation algorithm

Theorem 5.1 *The random permutation problem can be solved by a* QRQW PRAM *algorithm in $O(\lg n)$ time and linear work w.h.p.*

Proof. We use an algorithm adapted from a randomized CRCW algorithm of Gil [Gil91] for the *renaming* problem, in which the processors in an anonymous set of at most n processors are given distinct names from $[1..O(n)]$. For each of $c \lg\lg n$ rounds, for a constant $c \geq 1$, each unplaced item selects a random cell from a subarray of an array A (a new subarray is used for each round); if no other item selects the same cell, the item has been successfully placed. The size of the subarray used in the first round is $d \cdot n$, for some constant $d > 1$, and the size decreases by a factor of two at each round. If, after $c \lg\lg n$ rounds, not all items have been placed, restart from the beginning. After all items have been placed, the array A is compacted to size n.

Gil [Gil91] shows that, w.h.p., the algorithm completes without restarting. Moreover, w.h.p., the number of active items decreases more rapidly than the subarray size. In such cases, the contention to a memory cell at each round is a binomial random variable with an expected value less than 1. It follows by Observation 2.6 that w.h.p., the maximum contention is $O(\lg n / \lg\lg n)$ at each round, and hence the total time is $O(\lg n)$ w.h.p. The total work is $O(n)$ w.h.p. Processor allocation can be done directly or by applying Theorem 2.4. ∎

We note that there are other CRCW algorithms that may also give similar complexity bounds. Also, if the output may consist of an implicit (or "padded") random permutation (i.e. without the order as in the input; (iii) the p processors compute the prefix sums of the numbers $N[i,j]$ (in row major order) into the two-dimensional array $S_{\lg n,p}$; (iv) each processor j traverses its group j, and computes the global rank r of each element in its group; if x is an element in a subgroup i that is ranked $r_i(x)$ in its subgroup's list, then the global rank of x is $r(x) = S[i, j-1] + r_i(x)$; and (v) each processor copies all the items in its group into the output array in sorted order by their global rank. All steps can be easily implemented in $O(\lg n)$ time. ■

This gives us the following lemma.

Lemma 4.4 *The multiple compaction problem in which the count of each set is at most $\alpha \cdot \lg^2 n$ for the constant α in Lemma 4.2 (i.e. the light multiple compaction problem) can be solved on a* QRQW PRAM *in $O(\lg n)$ time and linear work w.h.p.*

5 Random permutation

The *random permutation* problem is to generate a permutation of $\{1, \ldots, n\}$ such that all permutations are equally likely. The *random cyclic permutation* problem is to generate a cyclic permutation (one that consists of a single cycle) of $\{1, \ldots, n\}$ such that all such permutations are equally likely. Examples of cyclic and noncyclic permutations are given in Figure 1. As indicated in Table 1, the best known linear work random permutation algorithm for the EREW PRAM run in $O(n^\epsilon)$ time, for fixed $\epsilon > 0$. This is also the best bound known for the random cyclic permutation problem. Polylog time EREW algorithms known for both problems are work inefficient by at least a $\sqrt{\lg n \lg\lg n}$ factor.

i	1	2	3	4	5
$\pi(i)$	3	1	4	5	2

(45213)

i	1	2	3	4	5
$\phi(i)$	4	5	2	1	3

(41)(253)

Figure 1: Permutations. On the left, a cyclic permutation, π, and a corresponding cycle representation. On the right, a noncyclic permutation, ϕ, and a corresponding cycle representation.

In this section, we present three QRQW PRAM algorithms that significantly improve upon the best EREW algorithms. The first, an adapted CRCW algorithm, solves the random permutation problem in $O(\lg n)$ time, linear work w.h.p. The second, a newly designed algorithm, solves the random cyclic permutation problem in $O(\sqrt{\lg n})$ time w.h.p., using n processors. The third, an adapted CRCW algorithm, solves the random cyclic permutation problem in $O(\lg n \lg^* n / \lg\lg n)$ time, linear work w.h.p. This section concludes with some results obtained from running random permutation algorithms on the MasPar MP-1 [Mas91].

5.1 Algorithms

Dart throwing is a popular technique for random permutation on the CRCW PRAM [MR89, RR89, MV91a, Hag91, Mat92]. The random permutation algorithms in the cited references all essentially consist of two basic steps. First, the items $1, \ldots, n$ are placed at random into a linear size array, by a process in which each item attempts to claim a random cell in the array until it succeeds (in later rounds, multiple processors may work on behalf of each item). If multiple items attempt to claim the same cell in the same step (by writing to the cell), all such attempts are considered to be

4.2 The light multiple compaction algorithm

In this section we present an $O(\lg n)$ time, linear work QRQW PRAM algorithm for the multiple compaction problem when the count of every set is at most $\alpha \lg^2 n$, i.e. for the light multiple compaction problem. The main steps in the algorithm are as follows.

(i) Elect a leader for every set Φ_j as follows: Write each item into a random location in its output subarray. Then use a simple prefix sums computation on the output array to identify the item written in the first non-empty location in each subarray. Designate this item as the leader for its set.

(ii) Have the leader of every set Φ_j write the value of n_j in location j of an array $C[1..n]$. For every empty set Φ_j write the value 1 (empty sets are assumed to have one dummy member).

(iii) Let each subarray of size $\alpha \lg^2 n$ in C define a superset containing the sets represented in this subarray. Note that each superset is of size between $\alpha \lg^2 n$ and $(\alpha \lg^2 n)^2$.

(iv) Process the data for the supersets defined in step (iii) to serve as an input for the heavy multiple compaction problem as follows: Compute prefix sums in array C to determine the starting position of the subarray for each superset in the (new) output array for the supersets. The leader for each set in the superset writes the label of the superset, its count, and its pointer in the starting position of the output subarray for its set. The processors then apply a simple broadcast computation to broadcast this information to all locations within each subarray in an optimal logarithmic time EREW PRAM computation. Each item then reads a random location in its output subarray to determine the label of its superset, its count, and its pointer.

(v) Apply the heavy multiple compaction algorithm of Lemma 4.2 to place each superset item in the appropriate subarray.

(vi) Within each superset, sort the items with the keys being the input labels modulo $\alpha \lg^2 n$. This places items with the same input label consecutively within the subarray.

(vii) Rank each item within the consecutive subarray for its input label, using a prefix sums computation. Then move each item, say with rank i, to the ith position in the original output subarray for its input label, using its input pointer.

The maximum contention in steps (i) and (iv) is $O(\lg n / \lg\lg n)$ w.h.p., by Observation 2.6. Thus each of the steps (i)–(v) and (vii) above is easily seen to run on a QRQW PRAM in $O(\lg n)$ time and linear work w.h.p. For step (vi), we apply the following result.

Fact 4.3 (see, e.g. [Rei93]) *The* EREW PRAM *can stably-sort n integers in the range $[1..\lg^c n]$, for any integer constant c, in $O(\lg n)$ time and linear work.*

Proof. The following steps stably-sort integer keys in the range $[1..\lg n]$; the desired result is obtained by repeating these steps c times on increasingly significant bits of the input integers.

We use $p = n/\lg n$ processors. The input items are partitioned into p groups of size $\lg n$, by their location in the input array. Each group consists of $\lg n$ subgroups (some of them perhaps empty), according to the key values. We use a two-dimensional array $N_{\lg n, p}$; $N[i,j]$ will represent the number of keys with value i in group j. Thus, each row i in N will represent the sizes of subgroups of keys with value i, whereas each column j in N will represent the subgroups of group j. The algorithm consists of the following steps: (i) each processor j, $j = 1, \ldots, p$, traverses its group j, counts the number of items in each subgroup i, and records them into $N[i,j]$, $i = 1, \ldots, \lg n$; (ii) each processor j traverses its group and puts the items of each subgroup in a separate list, ordered in the same relative the number of items that fail is less than $\lg m$ w.h.p. in m. This establishes the claim on the number of active items remaining at the end of each round.

Thus at the beginning of round i', the number of active items in each set Φ_j is at most $\max(n_j/(2^{i'-1}q_{i'}), \lg m)$ w.h.p. in m, i.e. at most $n_j/\alpha \lg m$ (recall that $n_j \geq \alpha \lg^2 m$). Since $q_{i'} = \alpha \lg m$ processors are allocated to each item in round i', all active items succeed in this round w.h.p. in m. A Las Vegas algorithm can be obtained by repeating this last round on the remaining active items until all such items have been placed.

Now we describe an implementation of this algorithm on the QRQW PRAM. The algorithm can be easily implemented on the L-spawning model of Section 3.3, taking $L = q_{i'} = \alpha \lg m$. Moreover, the L-spawning algorithm is predicted. The number of parallel steps is $t' = O(\lg^* n)$. The expected contention at each deactivation step is less than 1, so by Observation 2.6, the maximum contention at each deactivation step is $O(\lg m / \lg\lg m)$ w.h.p. in m, and the time of the algorithm is therefore $t = O(\lg^* n \lg m / \lg\lg m)$. The work of the algorithm is $O(\sum_{i=1}^{i'} n/2^i)$ which is $O(n)$. By Corollary 3.8, the algorithm described above can be implemented on the QRQW PRAM in $O(n)$ work and $O(\lg^* n \lg m / \lg\lg m + \lg^* n \sqrt{\lg m} \lg\lg\lg m)$ time, i.e. $O(\lg^* n \lg m / \lg\lg m)$ time, w.h.p. in m.

We next describe a more direct implementation of the L-spawning algorithm above, which does not require the use of the linear compaction algorithm (as in Corollary 3.8). Consider a partition of the input elements in array A into *groups* of size $\lg^2 m$. Since the expected number of active items in each group is $\Omega(\lg m)$ in each round, by Chernoff bounds (Fact 2.5), the number of active items within each group is, w.h.p. in m, within a constant factor of the expected value. Therefore, the allocation step can be implemented within each group. Specifically, within each group a linear-work $O(\lg(\lg^2 m))$-time prefix sum algorithm is used to

(i) identify successful copies and select one of them to deactivate their item;

(ii) count the number of active items in the group;

(iii) duplicate each active item into q_i copies; and

(iv) partition the set of copies into equal-sized chunks, one chunk per processor.

Thus, the deactivation step of round i can be implemented in $O(\lg\lg m)$ time and $O(n/2^i)$ work w.h.p. in m. This leads to the following lemma.

Lemma 4.2 *The multiple compaction problem in which the count of each set is at least $\alpha \cdot \lg^2 m$ for a suitable constant $\alpha > 0$ can be solved by a* QRQW PRAM *algorithm in $O(\lg^* n \lg m / \lg\lg m)$ time and $O(n)$ work w.h.p. in m. The heavy multiple compaction problem (the case $n = m$) can be solved in $O(\lg n \lg^* n / \lg\lg n)$ time and linear work w.h.p. in n.*

In Section 7, we will use a relaxed version of the heavy multiple compaction problem in which the input assumption that all counts n_j are upper bounds on the sizes of their respective sets Φ_j is true *w.h.p. only*. When some set Φ_j has more than n_j items, the algorithm is permitted to report failure. The algorithm given above can be readily adapted to handle this relaxed version, within the same time and work bounds, as follows: After round i', use the output subarray to count the number of items in each set Φ_j; if there exists a set Φ_j with more than n_j items, report failure. This can be done in $O(\lg n)$ time and linear work using prefix sum computations. Repeat round i' and this test until either all items are placed or failure is reported.

describe our algorithm for heavy multiple compaction, and prove that it runs in $O(\lg n)$ time and linear work w.h.p. on a QRQW PRAM (Lemma 4.2). Then in Section 4.2 we describe our algorithm for light multiple compaction, and prove that it also runs in $O(\lg n)$ time and linear work w.h.p. on a QRQW PRAM (Lemma 4.4). To solve the overall multiple compaction problem, it suffices to perform one application each of the heavy and light multiple compaction algorithms. Thus the theorem follows from Lemma 4.2 and Lemma 4.4. ■

4.1 The heavy multiple compaction algorithm

We follow the general strategy used in the multiple compaction algorithm given in [GMV91] for the CRCW PRAM, and the log-star paradigm of [MV91a, Mat92]. To highlight and distinguish the dependence of our algorithm on the input size and the confidence bounds, we consider an input of size n and show a Las Vegas algorithm that, for any m, obtains its time bounds (which are a function of n and m) with high probability in m (i.e. with probability $1 - 1/m^\delta$ for any constant $\delta > 0$).

The log-star paradigm as adapted to our algorithm consists of $O(\lg^* n)$ basic rounds. An item is initially *active* and becomes *inactive* when it is moved into a private cell in the subarray for its set. The number of active items in set Φ_j at the beginning of round $i > 1$ is at most $n_j/(2^{i-1}q_i)$, where $\{q_i\}$ is a sequence defined by

$$q_{i+1} = \min\{2^{q_i}, \alpha \cdot \lg m\},$$

with q_1 a sufficiently large constant. Round 1 is repeated a constant number of times to establish the base case of this invariant. The number of rounds is defined as $i' = \min\{i : q_i = \alpha \lg m\}$.

Round i consists of two steps:

(i) *Allocation*, where each active item in Φ_j is allocated with a set of q_i processors (a "team"); and

(ii) *Deactivation*, where a processor handling an active item of a set Φ_j tries to get hold of a private cell in the subarray assigned to Φ_j, by selecting a cell in the subarray at random and writing its index into that cell. An active item is deactivated if any one of the processors assigned to it is able to obtain a private cell for the item.

In each round, the number of processors trying to write to the subarray for Φ_j (of size $4n_j$) is at most n_j. A processor fails in a write attempt if there is already a value written in that location from a previous step. To simplify the analysis, we will also consider a write attempt to be a failure if another processor tries to write into the location in the same step; this only increases the probability of failure. Then, the failure probability of each processor is at most $1/2$; moreover, these probabilities are "pseudo-independent" in the sense that the bound on the failure probability of an item is valid no matter what happens with other items. If any of the processors for an active item succeeds in claiming a cell, then the item becomes inactive by selecting one of its successful processors. Since q_i processors are allocated to each item, the probability that an entire team for an item fails is at most 2^{-q_i}.

We claim that the number of active items in each set Φ_j at the end of round $i < i'$ is at most $\max(n_j/(2^i q_{i+1}), \lg m)$ w.h.p. in m. Assume inductively that at the end of round $i-1$, the number of active items in each set Φ_j is at most $\max(n_j/(2^{i-1}q_i), \lg m)$; the base case can be easily obtained by repeating the first round for a constant number of times. If $n_j/(2^{i-1}q_i) > \lg m$ then the expected number of items that fail is at most $(n_j/(2^{i-1}q_i)) \cdot 2^{-q_i}$. If this expected number is $\Omega(\lg m)$, then by Chernoff bounds (Fact 2.5), the number of items that fail is $O(n_j/(q_i \cdot 2^{i-1}q_{i+1}))$ w.h.p. in m, i.e. no more than $n_j/(2^i q_{i+1})$; if this expected number is $o(\lg m)$, then again by Chernoff bounds, which is defined to be the same as the QRQW work-time presentation except that time is accounted for using the CRQW metric instead of the QRQW metric. An algorithm A in the CRQW work-time presentation obeying the L-spawning model with time t, work n, and number of parallel steps t' can be implemented on a p-processor CRQW PRAM to run in time $O(n/p)$ when $p = O(n/(t + t' \cdot T_{lb}(n, L, \mathcal{M})))$, where $\mathcal{M}$ is the CRQW PRAM model. The proof is similar to the proof of Theorem 3.6, and is omitted.

4 Multiple compaction

In this section we present a logarithmic time, linear work QRQW PRAM algorithm for the *multiple compaction* problem. We start by recalling the definitions of the *compaction* and *linear compaction* problems, which we studied in the context of the QRQW PRAM in [GMR96a].

Compaction Problem. Given an array $A[1..n]$ with k nonzero cells, where k is known but the positions of the k nonzero cells are not known, move the contents of the nonzero cells to the first k locations of array A.

Linear Compaction Problem. Given an input to the compaction problem (i.e. an array $A[1..n]$ with k nonzero cells, where k is known but the positions of the k nonzero cells are not known), move the contents of the nonzero cells to an output array of size $O(k)$.

In [GMR96a] we give a randomized algorithm for linear compaction on the QRQW PRAM that, w.h.p., runs in $O(\sqrt{\lg n})$ time while performing linear work. The same algorithm with an additional simple post-processing step solves the compaction problem in $O(\sqrt{\lg n} + \lg k)$ time and linear work, w.h.p.

The *multiple compaction* problem that we consider in this section is a generalization of the linear compaction problem. The input consists of n items given in an array $A[1..n]$; each item has a *label*, a *count*, and a *pointer*, all from $[1..O(n)]$. The labels partition the items into k sets $\Phi_1, \ldots, \Phi_k$, $k \leq n$, where Φ_j is the set of items labeled with j. For simplicity we will let $k = n$, and allow some of the Φ_j to be empty. The count of an item belonging to Φ_j is an upper bound, $n_j = \mathit{count}(\Phi_j)$, on the number of items in Φ_j, such that $\sum_{j=1}^{n} n_j \leq c \cdot n$ for some constant $c > 0$. Also given is an array $B[1..c'n]$, where $c' \geq 4c$ is a constant. Array B is partitioned into subarrays such that each set Φ_j has a private subarray of size at least $4n_j$; the subarrays are assigned in some arbitrary order. The pointer of an item belonging to a set Φ_j is the starting point in B of the subarray assigned to Φ_j.

Multiple Compaction Problem. Given an input of the form stated in the above paragraph, move each item in array A into a private cell in the subarray for its set in array B.

An important application of multiple compaction is in a randomized CRCW PRAM algorithm for integer sorting [RR89]. In Section 7, we will use the algorithm for multiple compaction given in this section to obtain a logarithmic time, linear work CRQW PRAM algorithm for integer sorting, as well as to obtain efficient QRQW or CRQW algorithms for general sorting and sorting from $U(0, 1)$.

Our main result in this section is a QRQW PRAM algorithm for multiple compaction that runs in $O(\lg n)$ time and linear work w.h.p. as stated in the following theorem.

Theorem 4.1 *The multiple compaction problem can be solved by a* QRQW PRAM *algorithm in $O(\lg n)$ time and linear work w.h.p.*

Proof. We consider two special cases of the multiple compaction problem: In the *heavy multiple compaction* problem, the count of each set is at least $\alpha \cdot \lg^2 n$, for a suitable constant $\alpha > 0$, and in the *light multiple compaction* problem, the count of each set is at most $\alpha \cdot \lg^2 n$. In Section 4.1 we

Theorem 3.6 *Let A be an algorithm in the* QRQW *work-time presentation obeying the L-spawning model with time t and work n, and let t' be the number of parallel steps in A. If Algorithm A is predicted then it can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ if $p = O(n/(t + t' \cdot T_{lb}(n, L, \mathcal{M})))$, where $\mathcal{M}$ is the* QRQW PRAM *model.*

Proof. The processor allocation technique extends the techniques for the 2-spawning model used in [Mat92] for the CRCW PRAM and in [GMR96a] for the QRQW PRAM. Let p be the number of QRQW PRAM processors. Let w_i be the total number of tasks at the beginning of step i of Algorithm A, and let n_i be the approximate bounds on w_i, as defined above. Thus, $\sum_{i=1}^{t'} w_i = n$ and $\sum_{i=1}^{t'} n_i = O(n)$. In order to get an $O(n)$-work implementation on the QRQW PRAM, we keep the invariant that at each step the tasks are evenly distributed among the p processors; i.e. the number of tasks per processor at the beginning of step i is at most cn_i/p, for some constant $c > 0$.

Step i of Algorithm A is implemented as in the algorithm of Theorem 2.3, using the p-processor QRQW PRAM. After step i, each task may spawn at most $L - 1$ new tasks. Therefore, the total number of tasks, n_{i+1}, becomes at most Ln_i, and the number of tasks per processor becomes at most cLn_i/p. A load balancing algorithm is used to redistribute the tasks among the processors so that the number of tasks per processor becomes at most cn_{i+1}/p. If $n_{i+1} \geq n_i/2$, then the maximum normalized load is at most $2cL$, and hence the time for load balancing is at most $T_{lb}(p, 2cL, \mathcal{M})$, which is $O(T_{lb}(p, L, \mathcal{M}))$. So consider the general case where n_{i+1} may drop below $n_i/2$. In such cases, we will add (for the sake of analysis only) dummy tasks to increase n_{i+1} so that the maximum normalized load is at most $2cL$, and then argue that the addition of these dummy tasks increases the time and work bounds by at most a factor of 2 over the original algorithm.

In more detail, we partition the steps of Algorithm A into phases, where a *phase* consists of a maximal subsequence of steps for which the n_i's each decrease by more than a factor of 2. Let $\mu_1 = n_1$, and for $i = 2, \ldots, t'$, let $\mu_i = \max\{n_i, \mu_{i-1}/2\}$. For each step i, we add $\max\{0, \mu_i - n_i\}$ dummy tasks. Consider any phase, comprised of steps j through k. Then $\mu_j = n_j$, and $\mu_j, \ldots, \mu_k$ constitute a decreasing geometric series. Thus $\sum_{i=j}^{k} \mu_i < 2n_j$, so adding the dummy tasks increases the time and work bounds for the algorithm by at most a factor of 2.

By Theorem 2.3 and the invariant, the implementation of all steps i, $i = 1, 2, \ldots, t'$, when dummy tasks are included, takes $O(n/p + t)$ time. The implementation of all the load balancing steps when dummy tasks are included adds an additive overhead of $O(t' \cdot T_{lb}(p, L, \mathcal{M}))$. Hence the algorithm runs in time $O(n/p)$ when $p = O(n/(t + t' \cdot T_{lb}(p, L, \mathcal{M}))$. The theorem follows. ■

By Theorem 3.4 we obtain:

Corollary 3.7 *Algorithm A in Theorem 3.6 can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ w.h.p. when $p = O(n/(t + t'\sqrt{\lg n} \lg\lg L + t' \lg L))$.*

In particular,

Corollary 3.8 *Let A be an algorithm in the* QRQW *work-time presentation obeying the L-spawning model with time t and work n, and let t' be the number of parallel steps in A. Then, if $L = 2^{O(\sqrt{\lg n} \lg\lg n)}$, and Algorithm A is predicted, then A can be implemented on a p-processor* QRQW PRAM *to run in time $O(n/p)$ w.h.p. when $p = O(n/(t + t' \cdot \sqrt{\lg n} \lg\lg L))$.*

An application of Corollary 3.8 is given in the next section.

The above results can be extended to algorithms obeying the L-spawning model that are not predicted, if the CRQW PRAM model is used. Specifically, consider a CRQW work-time presentation,

5. We now have a load balancing problem on $O(p)$ super-tasks using p processors, with an initial maximum load of $O(g)$ super-tasks per physical processor and with the initial size of each pointer array being $w_0 = O(g)$. We solve this problem in $O(\lg\lg g)$ stages using the linear processor algorithm given earlier. Since the initial pointer array is as large as the maximum load per processor, we need to be careful about the processing of the pointer arrays and the task distribution step in order to stay within the time and work bounds. We perform this computation as follows: We add the pointer array for each new set of tasks added to a processor as a separate pointer array, and the processors in the team assigned to distribute the tasks in this processor will search serially through the different pointer arrays in this processor to determine the ones that contain its collection of tasks. Since we have at most $O(\sqrt{g})$ processors in a team and $2^{\lg\lg g} = O(\lg g)$ different pointer arrays in any processor at any stage, this step can be performed in $O(\sqrt{g} \cdot \lg g)$ time per stage leading to a total of $O(\sqrt{g} \cdot \lg g \cdot \lg\lg g)$, which is $O(g)$ time for processing the pointer arrays through all stages of the algorithm. At the end of this step, each physical processor has $O(g)$ tasks as required.

It is straightforward to see that the above algorithm runs in time $O(\lg L + T_{lc}(P, \mathcal{M}) \cdot \lg\lg L)$ (for step 3) $+ O(\lg g + T_{lc}(P, \mathcal{M}) \cdot \lg\lg g + g)$ (for step 5), which is $O(\lg L + T_{lc}(n, \mathcal{M}) \cdot \lg\lg L + n/p)$. Finally, if needed, each physical processor P_j can distribute its $O(g)$ tasks in $O(g = n/p)$ time to the $2n/p$ virtual processors in its group by a sequential algorithm. ■

By using the linear compaction algorithm given in [GMR96a], where for $\mathcal{M}$ being a SIMD-QRQW PRAM $T_{lc}(n, \mathcal{M}) = O(\sqrt{\lg n})$ w.h.p., we obtain

Theorem 3.4 *The load balancing problem with maximum normalized load L can be solved by a p-processor* SIMD-QRQW PRAM *algorithm in $O(\sqrt{\lg n} \lg\lg L + \lg L)$ time and linear work w.h.p.*

In particular,

Corollary 3.5 *The load balancing problem with maximum normalized load $L = 2^{O(\sqrt{\lg n} \lg\lg n)}$ can be solved by a p-processor* SIMD-QRQW PRAM *algorithm in $O(\sqrt{\lg n} \lg\lg n)$ time and linear work w.h.p.*

3.3 Application to automatic processor allocation

As mentioned in Section 2, the paper [GMR96a] gave a few examples of general classes of algorithms for which automatic processor allocation techniques can be applied to advantage. Such classes include geometric-decaying algorithms, general task-decaying algorithms, and spawning algorithms. Processor allocation is done by a scheduling scheme using an algorithm for linear compaction.

We show now that load balancing can be used to provide automatic processor allocation to a more general class of algorithms: the L-spawning algorithms. In an *L-spawning model*, at each step each task can spawn at most $L - 1$ more tasks. The total number of tasks may increase or decrease at each step. Thus, the L-spawning model generalizes the spawning model (which is equivalent to the 2-spawning model), as well as the models for task-decaying algorithms and geometric-decaying algorithms. Let w_i be the total number of tasks at the beginning of step i of an L-spawning algorithm A. Similarly to the task-decaying and to the spawning models, an L-spawning algorithm A is *predicted* if an approximate bound on the sequence of work loads $\{w_i\}$ is known in advance. Specifically, if a sequence $\{n_i\}$ is given such that for all i, $n_i \geq w_i$ and $\sum_i n_i = O(\sum_i w_i)$. Furthermore, it is required that for all i, $n_i \leq L \cdot n_{i-1}$.

The time for step 2 is dominated by the broadcasting substep and the time needed to compute the prefix sums on the array of pointers Q_j as well as to construct the array of pointers Q_{j+1} for the next stage. It is straightforward to see that the broadcasting substep can be implemented in $O(\lg u_i)$ time, and the computations on array Q_j can be performed in $O(\lg w_i)$ time. The overall time for the ith stage is $O(\lg u_i)$ as long as $u_i \geq w_i$, which holds for all but the last $\Theta(\lg \lg \lg L)$ stages of the algorithm. Let $i^+ = \lg \lg L - \lg \lg \lg L$. The time taken by the first i^+ steps of the algorithm is (to within a constant factor)

$$\sum_{i=0}^{i^+-1} \lg u_i = \sum_{i=0}^{i^+-1} \lg(2^{2-1/2^{i-1}} L^{2^{-(i+1)}}) < \sum_{i=0}^{i^+-1} (2 + 2^{-(i+1)} \lg L) < \lg L + 2i^+ \ ,$$

which is $O(\lg L)$. The total running time for the first i^+ stages of the algorithm is therefore $O(\lg L + T_{lc}(n, \mathcal{M}) \lg \lg L)$, using n processors.

It is not difficult to see that at the end of step i^+, $u_{i^+} = O(\lg L)$ and $w_{i^+} = O(\lg L / \lg \lg L) = O(\lg L)$. Since each processor has a total of $O(\lg L)$ tasks arranged in a collection of $w_{i^+} = O(\lg L)$ arrays, each processor can sequentially collect together all of the tasks in all of its task arrays into a single task array in $O(\lg L)$ time. Now we have a new load balancing problem on n processors with maximum load $O(\lg L)$. We apply steps 1 and 2 repeatedly to this problem until the load balancing is completed. This second phase clearly takes no more time than the first phase. Hence, the overall running time of the algorithm is $O(\lg L + T_{lc}(n, \mathcal{M}) \lg \lg L)$, using n processors.

Finally, each processor can convert its task representation from the array of array format to the single array format in constant time since it has only a constant number of tasks assigned to it at the end of the algorithm.

Reducing the number of processors. It remains to show how to implement the above algorithm (which assumes $2n$ virtual processors) on $p \leq n$ processors with an additive time overhead of $O(n/p)$.

The $2n$ virtual processors are partitioned into p groups of $g = 2n/p$ processors each, and the jth group is assigned to the jth physical processor, $1 \leq j \leq p$. We will combine the tasks in the virtual processors into 'super-tasks' that contain g original tasks (with possibly a few smaller super-tasks) and perform load balancing on these super-tasks. For this, the jth real processor P_j will perform the following computation on the virtual processors in the jth group, $1 \leq j \leq p$:

1. Designate the virtual processors in the jth group whose load is at least $g = 2n/p$ as "heavy processors" and the remaining processors in the jth group as "light processors".

2. For each heavy processor $H_{i,j}$ in the jth group, let its load be m_i. Combine its tasks into super-tasks of size g, with possibly one smaller super-task by setting its new load to be $\lceil m_i/(2n/p) \rceil$, and setting its 'normalizing' factor to be g.

3. Perform load balancing on the super-tasks in the heavy processors using the linear processor algorithm given earlier. This is a load balancing problem on p processors with $O(p)$ super-tasks and an initial maximum load of $O(L)$ super-tasks per processor.

4. At this stage each physical processor has $O(g^2)$ original tasks consisting of a constant number of super-tasks (of size g) from heavy processors and tasks from up to g light processors, each of which has at most g tasks. These tasks are organized in a pointer array of size $O(g)$. Each physical processor processes this pointer array and its array(s) of tasks so that the tasks are once again grouped into super tasks of size g (and possibly one smaller super-task in the jth group), and such that a chunk of r super-tasks, starting with ℓth super-task can be retrieved in constant time, given r and ℓ. This preprocessing can be performed in $O(g)$ time sequentially by each physical processor.

Lemma 3.3 *Let $\mathcal{M}$ be a model at least as strong as the* EREW PRAM. *Then*

$$T_{lb}(n, L, \mathcal{M}) = O(\lg L + T_{lc}(n, \mathcal{M}) \cdot \lg\lg L) \ .$$

Proof. Assume first that the number of available processors is $2n$. We later show how to reduce the number of processors to $n/T_{lb}(n, L, \mathcal{M})$, as required.

Our algorithm is based on a CRCW load balancing algorithm by [Gil91], which consists of $O(\lg\lg L)$ applications of a *dispersal* stage. Each dispersal stage uses a linear compaction algorithm as a main building block.

Let $u_0, u_1, \ldots$ be a sequence defined by $u_{i+1} = 2\sqrt{u_i}$ and $u_0 = \sqrt{L}$. It is straightforward to verify by induction on i that $u_i = 2^{2-1/2^{i-1}} L^{2^{-(i+1)}}$ for $i \geq 1$, and hence u_k becomes constant for $i = O(\lg\lg n)$. For simplicity, we will assume that the numbers $\sqrt{u_i}$, $i = 0, 1, \ldots$, as well as other outcomes of calculations below, are integers; it is straightforward, albeit somewhat tedious, to adapt the setting of parameters and the analysis to handle the general case.

As an invariant, we let u_i^2 be an upper bound on the maximum load among the processors at the beginning of the $(i+1)$st dispersal stage. A processor is said to be *overloaded* if it has at least $2u_i$ tasks. The $(i+1)$st dispersal stage reduces the upper bound on the maximum load per processor to $u_{i+1}^2 = 4u_i$, as follows:

Step 1. The overloaded processors are injectively mapped into an auxiliary array of size $2n/u_i$.

Step 2. For each cell of the auxiliary array there is a team of u_i processors standing by: each of them *adopts* up to $2u_i$ tasks of the overloaded processor that was mapped into this cell, thereby freeing the overloaded processor from all its tasks. Each processor has now at most $2u_i$ old tasks and at most $2u_i$ new tasks. Therefore, the upper bound on the maximum load among the processors becomes $4u_i = u_{i+1}^2$, as required.

Clearly, after $i^* = \lg\lg L$ stages , u_{i^*} is reduced to a constant, and we are done.

Implementation of step 1. An injective mapping is obtained by using a linear compaction algorithm, in $O(T_{lc}(n, \mathcal{M}))$ time. Note that since the total number of tasks is at most $2n$, there are at most n/u_i overloaded processors. The contribution of step 1 to the entire algorithm is therefore $O(T_{lc}(n, \mathcal{M}) \lg\lg L)$ time.

Implementation of step 2. Each processor P_j keeps an array of pointers Q_j to the arrays of tasks which currently belong to the processor. In each stage, the size of this pointer array at most doubles, so in the ith stage, the size of this pointer array is no more than $w_i = g \cdot 2^i$, where g is the initial size of the pointer array. Since the initial size of the pointer array is 1 (by our convention for the input representation), the size of this array in the ith stage is bounded by 2^i for each processor.

Processor P_j also keeps an additional array T_j which represents the prefix sums $T_j[\ell] = \sum_{k=1}^{\ell} t_{j,k}$, $1 \leq \ell \leq w_i$, where $t_{j,k}$ is the number of tasks in the kth task array of processor P_j. The tasks of the ℓth subarray of an overloaded processor P_j are to be adopted by $\lceil t_{j,\ell}/u_i \rceil$ processors. The pointer to the ℓth subarray of P_j is broadcast together with $T_j[\ell - 1]$ and $T_j[\ell]$ to processors P_v, $v \in \{\lceil T_j[\ell-1]/u_i \rceil + 1, \ldots, \lceil T_j[\ell]/u_i \rceil\}$, in the team which is allocated to P_j (here v is the numbering of processors within the team). Each processor can infer from this information the pointer(s) to the subarray(s) of tasks it needs to adopt and hence perform the appropriate updates. Note that an overloaded processor P_j may also be part of a team allocated to another overloaded processor. Therefore, before the above update takes place, each overloaded processor P_j updates both its pointers array Q_j and its prefix sums array T_j to null.

$m = \omega(n)$ our load balancing algorithm continues to have the time bound of $O(\lg L + \sqrt{\lg n} \cdot \lg\lg L)$, but the output representation of tasks will be an array of 'super-tasks', each of size $\lceil m/n \rceil$, where each super-task is represented by a pointer into the input task arrays.

3.1 A lower bound

In this section we show that the load balancing problem requires $\Omega(\lg L)$ time on the QRQW PRAM, where L is the maximum load on any processor. The lower bound uses the following lower bound on the 'broadcasting' problem, which is given in [GMR96a].

Theorem 3.1 ([GMR96a]) *Any deterministic or randomized algorithm that broadcasts the value of a bit to any subset of k processors in a* QRQW PRAM *requires expected time $\Omega(\lg k)$, regardless of the number of processors used.*

We now present our lower bound for the load balancing problem.

Theorem 3.2 *Any deterministic or probabilistic* QRQW PRAM *algorithm for the load balancing problem with maximum initial load L requires $\Omega(\lg L)$ time regardless of the number of processors used.*

Proof. Let the load balancing algorithm guarantee that each processor has at most $c(1 + m/n)$ tasks, for a suitable constant $c \geq 1$. Our proof is based on showing a constant time EREW PRAM reduction from the problem of broadcasting the value of a bit to any subset of $(1/c) \cdot L$ processors out of a total of n processors to the following load balancing problem: one processor P has L tasks, and the remaining $n - 1$ processors have 0 tasks. If the value of the bit to be broadcast is 0 then the L tasks are located in an array starting at memory location $n + 1$; if the value of the bit to be broadcast is 1 then the L tasks are located in an array starting at memory location $2n + 1$. All of the tasks are 'dummy' tasks, with constant size representation. This reduction can be implemented in constant time by having the ith processor enter the task representation for the ith dummy task to the array starting at location $n + 1$ and to the array starting at location $2n + 1$. Processor P initializes the pointer to the array of task representations to $n + 1$ or $2n + 1$ depending on whether its bit value is 0 or 1, and sets its load to be L.

The solution to the above load balancing problem consists of a subset $\mathcal{S}$ of at least L/c processors, each receiving a pointer to a subarray consisting of at most c tasks. These subarrays are either in the block of memory between $n+1$ and $2n$ or between $2n+1$ and $3n$. Depending on which range the pointer lies, each of the processors in $\mathcal{S}$ can determine whether the value b of the bit in processor P is 0 or 1. Hence by Theorem 3.1 it follows that the load balancing problem requires $\Omega((1/c) \cdot \lg L)$ expected time, i.e. $\Omega(\lg L)$ expected time. ∎

3.2 An algorithm

Let $T_{lb}(n, L, \mathcal{M})$ be the time needed to solve the load balancing problem of size n with maximum normalized load L, using linear work on a model $\mathcal{M}$. By Theorem 3.2, if $\mathcal{M}$ is a QRQW PRAM, then $T_{lb}(n, L, \mathcal{M}) = \Omega(\lg L)$.

A problem related to load balancing is the previously studied *linear compaction* problem: Consider an array of size n with k nonempty cells, with k known. The linear compaction problem is to move the contents of the non-empty cells to an output array of $O(k)$ cells. Let $T_{lc}(n, \mathcal{M})$ be the time for solving the linear compaction problem of size n, using n processors on a model $\mathcal{M}$. Our load balancing algorithm is primarily based on repeated applications of a linear compaction algorithm:

Fact 2.5 *$Pr\{X \geq \beta E[X]\} \leq e^{(1-1/\beta-\ln\beta)\beta E[X]}$, for all $\beta > 1$.*

A convenient corollary to this Chernoff bound is the following (see, e.g. [GMR96a]):

Observation 2.6 *Let X be a binomial random variable. For all $f = O(\lg n)$, if $E[X] \leq 1/2^f$, then $X = O(\lg n/f)$ w.h.p. Furthermore, if $E[X] \leq 1$ then $X = O(\lg n/\lg\lg n)$ w.h.p.*

3 Load balancing

Let m independent tasks be distributed among n virtual processors, and let L be the maximum number of tasks (i.e. the maximum "load") on any of the processors. In the *load balancing* problem, the input to each processor P_i consists of m_i, the number of tasks allocated to this processor (its "load"), together with a pointer to an array of task representations; no other information about the global partition is available, except for m and L. The load balancing problem asks for a redistribution of the tasks among the processors so that each processor has $O(1 + m/n)$ tasks.

Our load balancing algorithms will use a more general representation for the tasks during the course of the computation. In this representation, which we call the *array of arrays* format, the tasks assigned to each processor are specified by an array of pointers to arrays of tasks, so that each task is in exactly one of those task arrays. The format specified for the input to the load balancing problem is a specific instance of the array of arrays format in which the array of pointers contains only one element. Note that if the input is specified in the more general array of arrays format, then we can convert it into the prescribed input format in $O(\lg L)$ time with $O(m)$ work as follows: We convert the task arrays into linked lists. We then link these linked lists for the different arrays for a given processor into a single linked list. Both of these steps can be performed in constant time and $O(m)$ work over all processors. We then perform list ranking on the linked list for each processor, and transfer the tasks in the linked list into an array of suitable size. This can be performed in $O(\lg L)$ time and $O(m)$ work. In view of this conversion procedure, we assume, for convenience, that the input is in the form prescribed above.

We note the following property of the array representation for tasks. Given the array representation for tasks for each processor as specified above for the input, a given processor P_i can acquire a block of k tasks assigned to processor P_j starting at a given location r in P_j's task representation in constant time, given the values of i, k and r. If P_j's task representation is the array of arrays format, then P_i can access a block of k tasks starting at position r of the sth array of P_i in constant time, given the values of i, k, r and s.

We will assume that $m \leq 2n$, and that $L \leq n$. This assumption is justified below, by showing a constant time reduction from the general load balancing problem.

Consider a general load balancing problem. The tasks at each processor P_i can be grouped into super-tasks of $\lceil m/n \rceil$ tasks each, with possibly one smaller super-task. The number of super-tasks per processor is $\lceil m_i/\lceil m/n \rceil \rceil$. Therefore, the total number of super-tasks is $\sum_{i=1}^{n} \lceil m_i/\lceil m/n \rceil \rceil \leq 2n$ and the maximum load per processor is $\lceil m_i/\lceil m/n \rceil \rceil \leq n$. A load balancing algorithm for the super-tasks will allocate a constant number of super-tasks per processor. Therefore, the number of tasks allocated per processor will be $O(\lceil m/n \rceil)$, as required. We refer to the maximum load in the new input, $\lceil m_i/\lceil m/n \rceil \rceil$ as the *normalized maximum load*.

In this section we show that $\Omega(\lg L)$ time is required to solve the load balancing problem with maximum load L on a QRQW PRAM. We then present a QRQW PRAM algorithm for this problem on n processors with $m = O(n)$ tasks that runs in time $O(\lg L + \sqrt{\lg n} \cdot \lg\lg L)$. The $\sqrt{\lg n}$ term in the time bound arises from the use of an algorithm for the 'linear compaction' problem, for which we use the QRQW PRAM algorithm in [GMR96a], which runs in $O(\sqrt{\lg n})$ time w.h.p. In the case when

In the previous paper, we showed that the work-time framework is well-suited to the QRQW PRAM. In the QRQW work-time presentation, a parallel algorithm is described as a sequence of steps, where each step may include any number of concurrent read, compute, or write operations. In this context, the *work* is defined to be the total number of operations, and the *time* is defined to be the sum over all steps of the maximum contention of the step. Then Brent's scheduling principle [Bre74] can be applied to give a QRQW PRAM algorithm running in $O(work/p + time)$ time on p processors:

Theorem 2.3 ([GMR96a]) *Assume processor allocation is free. Any algorithm in the* QRQW *work-time presentation with x operations and t time (t is the sum of the maximum contention at each step) runs in at most $x/p + t$ time on a p-processor* QRQW PRAM.

We further showed a number of general scenarios under which automatic techniques can be used to efficiently handle processor allocation issues. Consider, for instance, *geometric-decaying* algorithms, in which the sequence of work loads (i.e. operations per step), $\{w_i\}$, is upper bounded by a decreasing geometric series, and each task at step i was appointed by one task at the preceding step $i-1$. For this scenario, we have shown a technique for automatic processor allocation that yields the following result:

Theorem 2.4 ([GMR96a]) *Let A be a geometric-decaying algorithm in a* QRQW *work-time presentation with time t and work n. Then Algorithm A can be implemented on a p-processor* QRQW PRAM *in time $O(n/p)$ w.h.p. if $p = O(n/(t + \sqrt{\lg n}\lg\lg n))$*

For ease of exposition, most of the QRQW algorithms in this paper are presented using the QRQW work-time framework; Theorem 2.4 is used as appropriate.

Among the algorithmic results in our previous paper [GMR96a] are sublogarithmic time randomized algorithms on the queue-write PRAM model for two problems for which the fastest algorithm known on the corresponding exclusive-write PRAM model takes $\Theta(\lg n)$ time. The two results are an $O(\lg n/\lg\lg n)$ time, linear work w.h.p. SIMD-CRQW PRAM algorithm for computing the OR of n bits and an $O(\sqrt{\lg n})$ time, linear work w.h.p. SIMD-QRQW PRAM algorithm for the linear compaction problem.

In addition, we present an $\Omega(\lg n)$ expected time lower bound on a QRQW PRAM with an unbounded number of processors for the problem of broadcasting the contents of a given memory location to n memory locations.

2.3 Probability facts and notations

A *Las Vegas* algorithm is a randomized algorithm that always outputs a correct answer, and obtains the stated bounds with some stated probability. All of the randomized algorithms in this paper are Las Vegas algorithms, obtaining the stated QRQW PRAM bounds with high probability. Recall that a probabilistic event occurs *with high probability (w.h.p.),* if, for any prespecified constant $\delta > 0$, it occurs with probability $1 - 1/n^\delta$, where n is the size of the input. Thus, we say a randomized algorithm runs in $O(f(n))$ time w.h.p. if for every prespecified constant $\delta > 0$, there is a constant c such that for all $n \geq 1$, the algorithm runs in $c \cdot f(n)$ steps or less with probability at least $1 - 1/n^\delta$. Often, we can test whether the algorithm has succeeded, and if not repeat it. In this case, it suffices to design an algorithm that succeeds with probability $1 - 1/n^\epsilon$ for some positive constant ϵ, since we can repeat the algorithm δ/ϵ times if necessary, to boost the algorithm success probability to the desired $1 - 1/n^\delta$. With this in mind, we will freely use "with high probability" in this paper to refer to events or bounds that occur with probability $1 - 1/n^\epsilon$ for some positive constant ϵ.

In the results that follow, we apply the following Chernoff bound on the tail of a binomial random variable X ([Lei92], p.168):

Concurrent reads and writes to the same location are permitted in a step. In the case of multiple writers to a location x, an arbitrary write to x succeeds in writing the value present in x at the end of the step.

Definition 2.3 *Consider a* QRQW PRAM *step with maximum contention κ, and let $m = \max_i\{r_i, c_i, w_i\}$ for the step, i.e. the maximum over all processors i of its number of reads, computes, and writes. Then the* **time cost** *for the step is $\max\{m, \kappa\}$. The* **time** *of a* QRQW PRAM *algorithm is the sum of the time costs for its steps. The* **work** *of a* QRQW PRAM *algorithm is its processor-time product.*

This cost measure models, for example, a MIMD machine such as the Tera Computer [ACC$^+$90], in which each processor can have multiple reads/writes in progress at a time, and reads/writes to a location *queue* up and are serviced one at a time. Note that as a pure shared memory model, the QRQW PRAM model is independent of the particular layout of memory on the machine, e.g. the number of memory modules, and can be used to model even cache-based (COMA) machines, e.g. the KSR1 [FBR93], in which the mapping of memory cells to machine nodes varies dynamically as the computation proceeds.

Our previous paper also defined the SIMD-QRQW PRAM model, a restricted version of the QRQW PRAM in which $r_i = c_i = w_i = 1$ for all processors i at each step. This model is suitable for SIMD machines such as the MasPar MP-1 or MP-2, in which each processor can have at most one read/write in progress at a time, reads/writes to a location *queue* up and are serviced one at a time, and all processors await the completion of the slowest read/write in the step before continuing to the next step. Another variant is the CRQW PRAM, in which unlimited concurrent reading is permitted; for this model, the maximum contention for a step is defined to be the maximum over all locations of the number of *writers* to the location. Several of our results in Section 7 are for the CRQW PRAM.

2.2 Previous results

In addition to defining the QRQW models, our previous paper [GMR96a] presented a number of results characterizing the power of the QRQW models relative to other models. For two models, M_1 and M_2, let $M_1 \preceq M_2$ denote that one step of M_1 with time cost $t \geq 1$ can be emulated in $O(t)$ time on M_2 using the same number of processors. We have:

Fact 2.1 ([GMR96a]) EREW PRAM $\preceq$ SIMD-QRQW PRAM $\preceq$ QRQW PRAM $\preceq$ CRQW PRAM $\preceq$ CRCW PRAM.

Moreover, we have characterized the relative power of these models as follows:

Theorem 2.2 ([GMR96a]) *The following relations hold:*

1. *There is an $\Omega(\sqrt{\lg n})$ time separation between an* EREW PRAM *with arbitrarily many processors and an n-processor* SIMD-QRQW PRAM.
2. *A* SIMD-QRQW PRAM *can emulate a* QRQW PRAM *to within constant time factors, given sufficiently many extra processors.*
3. *There is an $\Omega(\lg n)$ time separation between a* QRQW PRAM *with arbitrarily many processors and an n-processor* CRQW PRAM.
4. *There is an $\Omega(\lg n / \lg\lg n)$ time separation between a deterministic* CRQW PRAM *with arbitrarily many processors and a deterministic n-processor* CRCW PRAM.

It appears that coordination among processors may occasionally be quite expensive on the QRQW PRAM, as implied by the lower bounds for broadcasting [GMR96a] and load balancing, and should be avoided if at all possible. Fast CRCW PRAM algorithms tend to have very little such coordination, which makes them good candidates as basis for adaptation to QRQW PRAM algorithms. Indeed, one of the main features in the $O(\lg\lg n)$ time CRCW PRAM hashing algorithm [GM94b] which is the basis for our QRQW PRAM algorithm is the "oblivious execution" technique, which allows the computation to proceed without coordination among processors. By contrast, an $O(\lg n)$ time CRCW PRAM hashing algorithm [MV91b] makes extensive use of (semi-)sorting for processor coordination, which on the QRQW PRAM would be both slow and inefficient.

Finally, we remark on the role that randomization plays for our QRQW PRAM algorithms. We recall that the power of the QRQW PRAM model, in comparison with the EREW PRAM model, is in the fact that it is not necessary to schedule the memory accesses explicitly so as to avoid concurrent access. There are two natural ways to leverage on this power. One way is the use of irregular small contention (deterministic) memory accesses, as illustrated in [GMR96a] in the context of the 2-compaction problem. Another way is to use randomization as a technique for random assignment of resources, be it read operations as in the hashing algorithm and in the fat-tree data structure, or write operations as in the linear compaction, multiple compaction, load balancing and random permutation algorithms. This technique has been essentially used in all the algorithms presented in this paper, and has proved to be a simple and effective tool for low-contention parallel algorithms.

The rest of this paper is organized as follows. In Section 2 we review the definition of the QRQW model and some previous results for the model. Then, as indicated above, Sections 3–7 consider load balancing, multiple compaction, generating a random permutation, hashing, and sorting. Finally, Section 8 contains concluding remarks.

The results in this paper appeared in preliminary form in [GMR93, GMR94a, GMR94b].

2 Preliminaries

2.1 The QRQW PRAM model

We begin by reviewing the definition of the QRQW PRAM model [GMR96a].

Definition 2.1 *Consider a single step of a* PRAM, *consisting of a read substep, a compute substep, and a write substep. The* **maximum contention** *of the step is the maximum, over all locations x, of the number of processors reading x or the number of processors writing x. For simplicity in handling a corner case, a step with no reads or writes is defined to have maximum contention 'one'.*

Definition 2.2 *The* **QRQW PRAM** *model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. Processors execute a sequence of synchronous steps, each consisting of the following three substeps:*

1. *Read substep: Each processor i reads r_i shared memory locations, where the locations are known at the beginning of the substep.*
2. *Compute substep: Each processor i performs c_i* RAM *operations, involving only its private state and private memory.*
3. *Write substep: Each processor i writes to w_i shared memory locations (where the locations and values written are known at the beginning of the substep).*

the MasPar MP-1 parallel machine [Mas91]. Recently, the QRQW random permutation algorithm was also implemented on a CRAY J90, and was shown to be considerably faster than the best known (sorting-based) EREW algorithm [BGMZ95].

In Section 6 we present a linear work, $O(\lg n)$ time randomized QRQW PRAM algorithm for constructing a hash table and for parallel membership queries into the table. Our algorithm is based on an $O(\lg\lg n)$ time CRCW algorithm of [GM94b], which uses an oblivious execution technique to keep to minimum the required "bookkeeping" operations. In order to obtain a fast, efficient QRQW algorithm, we replace the polynomial hash functions used in the CRCW algorithm by hash functions [DM90] which have collision behavior that looks quite random. To implement an efficient access to these hash functions, we devise a low-contention QRQW PRAM algorithm which is based on the following simple, yet useful, idea: if a program variable is to be read by k (a priori unknown) processors, then we replace the program variable with k copies of the same value; we then let each of the k processors select one of the copies at random and read the selected copy.

Our sorting algorithms are given in Section 7. We present linear work, $O(\lg n)$ time randomized algorithms for sorting from $U(0,1)$ on the QRQW PRAM, and for integer sorting on the CRQW PRAM. We use the latter result in a fast, efficient emulation of the powerful FETCH&ADD PRAM on the CRQW PRAM. In addition, we adapt the $\sqrt{n}$-sample sort CREW PRAM algorithm of Reischuk [Rei85] to obtain a simple, work-optimal QRQW PRAM algorithm for general sorting. The QRQW algorithm employs a novel *binary search fat-tree* data structure;[5] the added fatness over a traditional binary search tree ensures that, with high probability, each step of the search encounters low contention.

1.2 Techniques for QRQW PRAM algorithms

Important technical issues arise in designing algorithms for the queue models, that are present in neither the concurrent nor the exclusive PRAM models. For example, much of the effort in designing algorithms for the QRQW models is in estimating the maximum contention in a step, and occasionally identifying the number of processors that try to access the same memory address. As one high contention step can dominate the running time of the algorithm, we cannot afford to underestimate the contentions significantly.

There are several techniques for replacing a high contention step with a sequence of a few low contention steps. One such technique is to replace concurrent read operations by local broadcasting steps, as done in the algorithms for load balancing, multiple compaction, and random permutation. Another technique is using larger arrays into which processors are "compacted", so as to reduce the size of collision sets; this is used in the linear compaction algorithm in [GMR96a], as well as in an algorithm for random cyclic permutation. A third important technique is that of duplicating the contents of one or more program variables, and then having each processor access a random copy of such a variable, thereby reducing contention. Algorithms that use this technique include the hashing and the general sorting algorithms.

Some QRQW PRAM algorithms consist of iterations that include a random scatter step, in which processors access a random cell in a linear size array; this is an example of the duplication scheme mentioned above. The maximum contention in such steps is $\Theta(\lg n/\lg\lg n)$ w.h.p., implying that to obtain $O(\lg n)$ time the number of iterations must not exceed $O(\lg\lg n)$. Indeed, some of the $O(\lg n)$ time QRQW PRAM algorithms are based on "highly parallel" CRCW PRAM randomized algorithms, whose running time on the CRCW is w.h.p. $O(\lg\lg n)$ or $O(\lg^* n)$ [Mat92]. Algorithms that use the "doubly-logarithmic paradigm" include those for load balancing, random permutation, and hashing. Algorithms that use the "log-star paradigm" include those for multiple compaction and random cyclic permutation.

[5] The term *fat-tree* was previously used by Leiserson [Lei85] in the context of interconnection networks, to describe a tree that becomes thicker as it gets closer to the root.

Summary of Algorithmic Results

problem	**previous result** (EREW)	**new result** (QRQW)
random permutation	$O(\lg n)$ time, $O(n \lg n)$ work [Hag91] $O(\lg n \lg\lg n)$ time, $O\left(\frac{n \lg n}{\lg\lg n}\right)$ work [AH92] $O\left(\frac{\lg^{1.5} n}{\sqrt{\lg\lg n}}\right)$ time, $O(n\sqrt{\lg n \lg\lg n})$ work [AH92] $O(n^{\epsilon})$ time, constant $\epsilon > 0$, linear work [KRS90]	$O(\lg n)$ time, linear work w.h.p.
multiple compaction	same as above	$O(\lg n)$ time, linear work w.h.p.
sorting from $U(0,1)$	same as above	$O(\lg n)$ time, linear work w.h.p.
parallel hashing	same as above with $\lg^* n$ slowdown [GMV91, MV95]	$O(\lg n)$ time, linear work w.h.p.
load balancing, max load L	$O(\lg n)$ time, linear work [LF80]	$O(\sqrt{\lg n}\lg\lg L + \lg L)$ time, linear work w.h.p.

Table 1: *Fast, efficient low-contention parallel algorithms for several fundamental problems. For the first four problems above, we obtain work-optimal low-contention (*QRQW PRAM*) algorithms running in logarithmic time, whereas the best known work-optimal zero-contention (*EREW PRAM*) algorithms run in polynomial time. For load balancing, we improve upon the* EREW *result whenever the ratio of the maximum to the average load is not too large. The* EREW *results shown are the best known for either deterministic or randomized algorithms. The* EREW *results for the first three problems are obtained by easy reductions to the integer sorting problem. The result for the fourth is obtained using a* CRCW *hashing algorithm and a general simulation of the* CRCW PRAM *on the* EREW PRAM. *The load balancing* EREW PRAM *result is a simple application of a prefix sums algorithm.*

to the load balancing problem, and using the lower bound for broadcasting presented in [GMR96a].

The load balancing algorithm is a useful tool for processor allocation. We use it to obtain an algorithm that automatically handles processor allocation for any algorithm that can be described within certain specifications (such algorithms are called "L-spawning algorithms"). We use this general result in our work-optimal algorithms for the multiple compaction problem and for the problem of generating a random cyclic permutation.

In Section 4 we consider the multiple compaction problem, which has an important application in a CRCW PRAM algorithm for integer sorting [RR89]. We present a linear work, $O(\lg n)$ time randomized QRQW PRAM algorithm, which is quite different than the known CRCW PRAM algorithms for the problem. Some parts of the algorithm follow a general strategy used in a CRCW PRAM algorithm that runs in $O(\lg^* n)$ time [GMV91], and in particular the log-star paradigm [Mat92]. The QRQW PRAM algorithm for multiple compaction has applications for QRQW or CRQW algorithms for integer sorting, general sorting, and sorting from $U(0,1)$.

The problem of generating a random permutation is considered in Section 5. We present a linear work, $O(\lg n)$ time randomized QRQW PRAM algorithm that is essentially the same as the $O(\lg\lg n)$ time CRCW PRAM algorithm of [Gil94], analyzed for the QRQW metric. Two algorithms are presented for the problem of generating random cyclic permutations. A linear work, $O(\lg n \lg^* n/\lg\lg n)$ time randomized algorithm is adapted (with some modifications) from an $O(\lg^* n)$ time CRCW PRAM algorithm of [MV91a]. A faster QRQW PRAM algorithm, which takes $O(\sqrt{\lg n})$ time w.h.p. but uses n processors, is based on the linear compaction algorithm presented in [GMR96a]. The idea behind the algorithm is to use a relatively large array into which processors are "compacted", so that the number of processors accessing the same array location is not too large.

We also demonstrate in Section 5 the efficiency of a QRQW PRAM low-contention random permutation algorithm, compared with the popular EREW algorithm, through several experiments on

The QRQW PRAM is strictly more powerful than the EREW PRAM, while being as efficiently emulated on a BSP or a hypercube-type, non-combining network, and is also a better match for real machines. Hence an important theoretical and practical question is the extent to which fast, work-optimal, low-contention (QRQW) algorithms can be designed for problems for which there are no known fast, work-optimal, zero-contention (EREW) algorithms. This paper considers five such problems — generating a random permutation, multiple compaction, distributive sorting, parallel hashing, and load balancing — and presents fast, work-optimal QRQW PRAM algorithms for these fundamental problems. These results are summarized in Table 1, and are contrasted with the best known EREW PRAM algorithms for the same problems. All of our algorithms are randomized, and are of the "Las Vegas" type; they always output correct results, and obtain the stated bounds with high probability.

Another important question is the extent to which EREW PRAM algorithms can be replaced by QRQW PRAM algorithms that are simpler, and therefore perhaps more appealing for implementation. In this context we would allow the theoretical efficiency of the simpler QRQW PRAM algorithm to be similar or even somewhat inferior to that of the EREW PRAM algorithms as long as the resulting algorithm is simpler. This paper considers such algorithms for the general sorting problem. It presents a QRQW PRAM algorithm that is considerably simpler than the known EREW PRAM algorithms with comparable asymptotic performance. The new algorithm is arguably as simple as the known CRCW PRAM algorithms.

All of the algorithms we present in this paper are randomized, and many of our results are obtained "with high probability" (w.h.p.). A probabilistic event occurs *with high probability (w.h.p.),* if, for any prespecified constant $\delta > 0$, it occurs with probability $1 - 1/n^{\delta}$, where n is the size of the input. Thus, we say a randomized algorithm runs in $O(f(n))$ time w.h.p. if for every prespecified constant $\delta > 0$, there is a constant c such that for all $n \geq 1$, the algorithm runs in $c \cdot f(n)$ steps or less with probability at least $1 - 1/n^{\delta}$.

We provide next a summary of our algorithmic results, and point out a few technical issues that are relevant for QRQW PRAM algorithms.

1.1 Summary of results

Our first results are for the load balancing problem, considered in Section 3. We present a linear work randomized algorithm whose running time is $O(\sqrt{\lg n}\lg\lg L + \lg L)$, where L is the ratio of the maximum to the average load per processor. Our load balancing algorithm is an adaptation of a CRCW PRAM algorithm by Gil [Gil94], which runs in $O(\lg\lg n)$ time w.h.p. Gil's algorithm uses as a subroutine an algorithm for the so-called "renaming" problem. Our low-contention implementation is essentially obtained by substituting this subroutine with a QRQW PRAM algorithm for linear compaction, presented in [GMR96a], and by replacing concurrent read operations executed during bookkeeping steps with local broadcasting steps.

For small values of L, our load balancing algorithm can be much faster than the $\Theta(\lg n)$ time, prefix-sum based EREW PRAM algorithm. However, for $L = \Omega(n^{\epsilon})$ with constant $\epsilon > 0$, the $\lg L$ term implies a running time of $O(\lg n)$. In contrast, load balancing on n processors can be performed on a CRCW PRAM in $O(\lg^* n)$ time[4] w.h.p., independent of L [GMV91]. We show that the $\lg L$ term is unavoidable by presenting a lower bound of $\Omega(\lg L)$ expected time on the QRQW PRAM for the load balancing problem. Our lower bound result is based on a reduction from the broadcasting problem

the EREW performs $\Theta(\lg p)$ times more work than the CRCW it emulates. Hence, it cannot be used to obtain EREW PRAM algorithms, much less hypercube algorithms, with linear or near-linear speedups. Similarly, the best known emulations for the CREW PRAM (or ERCW PRAM) on the EREW PRAM (or standard BSP or hypercube) require logarithmic work overhead for logarithmic slowdown or, alternatively, polynomial slowdown for constant work overhead.

[4] The function $\lg^{(j)}(\cdot)$ is defined as the j'th iterate of lg: $\lg^{(1)} x \equiv \lg x$, and for $j > 1$, $\lg^{(j)} x \equiv \lg\lg^{(j-1)} x$. The function $\lg^*(\cdot)$ is defined as $\lg^* x \equiv \min\left\{j : \lg^{(j)} x \leq 2\right\}$.

1 Introduction

The Parallel Random Access Machine (PRAM) model of computation is the most-widely used model for the design and analysis of parallel algorithms (see, e.g. [KR90, JáJ92, Rei93]). The PRAM model consists of a number of processors operating in lock-step and communicating by reading and writing locations in a shared memory. Standard PRAM models can be distinguished by their rules regarding contention for shared memory locations. These rules are generally classified into the *exclusive* read/write rule in which each location can be read or written by at most one processor in each unit-time PRAM step, and the *concurrent* read/write rule in which each location can be read or written by any number of processors in each unit-time PRAM step. These two rules can be applied independently to reads and writes; the resulting models are denoted in the literature as the EREW, CREW, ERCW, and CRCW PRAM models.

In a previous paper [GMR96a], we argued that neither the *exclusive* nor the *concurrent* rules accurately reflect the contention capabilities of most commercial and research machines, and proposed a new PRAM contention rule, the *queue* rule, that permits concurrent reading and writing, but at an appropriate cost:

> *Queue* read/write: Each location can be read or written by any number of processors in each step. Concurrent reads or writes to a location are serviced one-at-a-time.

Thus the worst case time to read or write a location is linear in the number of concurrent readers or writers to the same location.

The queue rule more accurately reflects the contention properties of machines with simple, non-combining interconnection networks than either the exclusive or concurrent rules. The exclusive rule is too strict, and the concurrent rule ignores the large performance penalty of high contention steps. Indeed, for most existing machines, including the CRAY T3D, IBM SP2, Intel Paragon, MasPar MP-1 and MP-2 (global router), MIT J-Machine, nCUBE 2S, Stanford DASH, Tera Computer, and Thinking Machines CM-5 (data network), the contention properties of the machine are well-approximated by the queue-read, queue-write rule. For the Kendall Square KSR1, the contention properties can be approximated by the concurrent-read, queue-write rule.[1]

In [GMR96a] we defined the Queue-Read, Queue-Write (QRQW) PRAM model, a model for the design and analysis of coarsely-synchronized parallel algorithms running on MIMD machines, and investigated some of its capabilities. In particular, we showed that the QRQW PRAM can be effectively emulated on the Bulk-Synchronous Parallel (BSP) model of Valiant [Val90]:

Theorem 1.1 ([GMR96a]) *A p-processor* QRQW PRAM *algorithm running in time t can be emulated on a $(p/\lg p)$-component standard* BSP *model*[2] *in $O(t \lg p)$ time with high probability.*

It follows from Valiant's work [Val90] and Theorem 1.1 that the QRQW PRAM can be emulated in a work-preserving manner on hypercube-type, non-combining networks with only logarithmic slowdown, *even when latency, memory granularity, and synchronization overheads are taken into account.* This matches the best known emulation for the EREW PRAM on these networks given in [Val90]; in contrast, work-preserving emulations for the CRCW PRAM on such networks are only known with *polynomial* slowdown.[3] We refer the reader to [GMR96a] for further details relating the QRQW PRAM to existing models and machines.

[1] In the KSR1, multiple requests to read the same location are combined in the network, so there is no penalty for high contention steps. Note that caches have only a secondary effect on the contention rule; see [GMR96a] for details.

[2] We denote as the *standard* BSP model a particular case studied by Valiant in which the model's throughput parameter, g, is taken to be a constant and its periodicity parameter, L, is taken to be $\Theta(\lg p)$.

[3] Note that the standard $\Theta(\lg p)$ time emulation of CRCW on EREW (see, e.g. [KR90]) is not work-preserving, in that

Efficient Low-Contention Parallel Algorithms

Phillip B. Gibbons
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
gibbons@research.att.com

Yossi Matias
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
matias@research.att.com

Vijaya Ramachandran[*]
Dept. of Computer Sciences
University of Texas at Austin
Austin TX 78712
vlr@cs.utexas.edu

May 21, 1996

Abstract

The queue-read, queue-write (QRQW) parallel random access machine (PRAM) model permits concurrent reading and writing to shared memory locations, but at a cost proportional to the number of readers/writers to any one memory location in a given step. The QRQW PRAM model reflects the contention properties of most commercially available parallel machines more accurately than either the well-studied CRCW PRAM or EREW PRAM models, and can be efficiently emulated with only logarithmic slowdown on hypercube-type non-combining networks.

This paper describes fast, low-contention, work-optimal, randomized QRQW PRAM algorithms for the fundamental problems of load balancing, multiple compaction, generating a random permutation, parallel hashing, and distributive sorting. These logarithmic or sublogarithmic time algorithms considerably improve upon the best known EREW PRAM algorithms for these problems, while avoiding the high-contention steps typical of CRCW PRAM algorithms. An illustrative experiment demonstrates the performance advantage of a new QRQW random permutation algorithm when compared with the popular EREW algorithm. Finally, this paper presents new randomized algorithms for integer sorting and general sorting.

[*] Supported in part by NSF grant CCR-90-23059 and Texas Advanced Research Projects Grants 003658480 and 003658386.