Parallel Priority Queue and List Contraction: The BSP Approach*

Alexandros V. Gerbessiotis, Constantinos J. Siniolakis and Alexandre Tiskin

Computing Laboratory, Oxford University, Oxford OX1 3QD, U.K.

Abstract. In this paper we present efficient and practical extensions of the randomized Parallel Priority Queue (PPQ) algorithms of Ranade et al., and efficient randomized and deterministic algorithms for the problem of list contraction on the Bulk-Synchronous Parallel (BSP) model. We also present an experimental study of their performance. We show that our algorithms are communication efficient and achieve small multiplicative constant factors for a wide range of parallel machines.

1 Introduction

We present an architecture independent study of the computation and communication requirements of an efficient Parallel Priority Queue (PPQ) implementation and list contraction algorithms along with an experimental study. The computational model adopted is the Bulk-Synchronous Parallel (BSP) model, proposed by L. G. Valiant [20], which deals explicitly with the notion of communication and synchronization among computational threads. A detailed discussion of the BSP model appears in [20].

The first approach on parallel priority queues is implied in the work of [9], where a parallel randomized algorithm to approximate a priority queue is presented. The algorithm is extended to handle exact priority queues in [18]. Deterministic algorithms for maintaining parallel priority queues are derived in [17]. These approaches fail, however, to deliver optimal performance when directly implemented on the BSP model of computation. The only previously known results on BSP PPQ algorithms appear in [2,5,6]. In this work, which first appeared in [5], we improve upon the communication requirements of previous results on parallel priority queues [2,9,17,18], and extend the BSP algorithms of [6] by presenting efficient and simple (non-pipelined) algorithms. We also show that small multiplicative constant factors can be achieved for a wide range of the BSP parameters. The randomized algorithms we present are based on the ideas of [9] as extended in [18]. In the following discussion several proofs are omitted; a detailed account can be found in [6] and the full version of the paper. For the sake of simplicity, throughout the paper we ignore small irregularities that arise from imperfect matching of integer parameters, e.g., we use the term n/p for $\lceil n/p \rceil$. The term

* The first author was supported in part by EPSRC(UK) under grant GR/K16999, the second author was supported in part by a Bodossaki Foundation Graduate Scholarship and the third author was supported in part by ESPRIT Basic Research Project 9072 (GEPPCOM).

slackness refers to the ratio of the problem size n over the number of processors p.

We state results on BSP algorithms for the fundamental operations of *parallel-prefix* and *selection*. The proofs of these results can be found in [5,7]. These operations are auxiliary routines for the algorithms to be presented in this work. Sometimes, we shall write $T_{ppf}(p)$ for $T^1_{ppf}(p)$.

Lemma 1. *There exists a p-processor BSP algorithm for realizing n disjoint parallel-prefix operations, each of size p, over an associative operator, that for any integer $2 \leq t \leq p$, requires time at most (the bracketed notation $[S]$ evaluates to 1 if statement S is true and 0 otherwise) $T^n_{ppf}(p) = O(n + g\ n + L) \cdot [n > 1] + \log_t(p/n)\ O(t + g\ t + L) \cdot [n < p]$.*

Lemma 2. *For any constant $\rho > 1$, there exists a randomized p-processor BSP selection algorithm that, with probability $1 - O(n^{1-\rho})$, determines the c-th smallest, $c \leq \lceil n/2 \rceil$, of n keys that for any integers n and $p \leq n$, requires time*

$$T_{sel}(n,c,p) = \begin{cases} \frac{n+c}{p} + O(\frac{\sqrt{cp \lg n} + n^{\frac{2}{3}+\delta} \lg^{\frac{4}{3}} n}{p} + g\ \frac{n^{\frac{2}{3}+\delta} \lg^{\frac{1}{3}} n}{p} + L + T_{ppf}(p)) & p \leq n^{\frac{2}{3}+\zeta} \\ \frac{n+c}{p} + O(\frac{\sqrt{cp \lg n} + p \lg n}{p} + g + L + T_{ppf}(p)) & \text{otherwise,} \end{cases}$$

where ζ and δ are any arbitrarily small constants such that $0 < \zeta < \delta < 1/3$.

Definition 3. Let $\hat{g}$ be the value of g in lemma 2 such that $\hat{g} = o(n^{1/3-\delta}/\lg^{1/3} n)$ if $p \leq n^{2/3+\zeta}$, and $\hat{g} = o(n/(p \lg p))$ otherwise, for any arbitrarily small constants ζ and δ such that $0 < \zeta < \delta < 1/3$.

2 BSP Priority Queues

We study the *Parallel Priority Queue* (PPQ) [17] data structure, and formulate an implementation on the BSP model. A PPQ is a data structure for maintaining a collection of (possibly duplicate) items and selecting the n items associated with the smallest values. We simplify the exposition by assuming that all items are unique, e.g., by appending a unique identifier to each item. The following operations are defined on a PPQ Q: INSERT($\{y_1, y_2, \ldots, y_n\}$, Q) for inserting $\{y_1, y_2, \ldots, y_n\}$ into Q; FINDMIN(n, Q) for finding the n smallest items of Q; and DELETEMIN(n, Q) for deleting the n smallest items of Q. The implicit assumption is that there exists a MAKEQUEUE(Q) operation for the construction of an empty queue Q. In contrast to [6,17], our implementation has the advantage of supporting PPQ operations on data sets of varying size, i.e., in [6,17], such operations have to be performed on data sets of predetermined size.

The randomized algorithms we present for supporting the prementioned PPQ operations are based on a straightforward processor-mapping of sequential ordinary (and leftist) heaps, even though other sequential data structures might be supported as well, e.g., relaxed heaps, Fibonacci heaps, search trees. To this end, in the forthcoming analysis of the PPQ algorithms we do not make any assumptions about the underlying local (sequential) priority queues. In order to simplify the exposition, the following notational conventions are employed. Let

$\tau_I(n,m)$ denote the time required to sequentially insert n items into a local priority queue of size m, $\tau_F(n,m)$ the time required to sequentially determine the n smallest items of a local priority queue of size m, $\tau_D(n,m)$ the time required to sequentially delete the n smallest items of a local priority queue of size m, $\tau_C(m)$ the time required to sequentially construct a local priority queue of size m (we note that in general $\tau_C(m) \leq \sum_i \tau_I(1,i)$), and $\tau_M(m_1,m_2)$ the time required to sequentially meld two local priority queues of size m_1 and m_2 into one of size m_1+m_2. The randomized algorithms we present are based on the ideas of [9] as extended in [18]. Let us outline the BSP variant of these algorithms. The PPQ is maintained as a collection of disjoint local (sequential) priority queues. For the insertion of n new items (n/p per processor), each processor sends its n/p items to uniformly at random chosen processors, where they are inserted into their local priority queues. The procedure to manage the insert operation on a PPQ Q is INSERT($\{y_1,y_2,\ldots,y_n\}$, Q) and is outlined in figure 1. The following lemma bounds the number of items received by each processor as a result of the random distribution to processors (line 2) of procedure INSERT.

INSERT ($\{y_1,y_2,\ldots,y_n\}$, Q)
1. denote by Y set $\{y_1,y_2,\ldots,y_n\}$;
2. *distribute* randomly the items of Y among the p processors ;
3. for each processor k, $0 \leq k \leq p-1$, in parallel
4. do INSERT_SEQ(Y^k, Q^k) ;

Figure 1. Procedure INSERT.

Lemma 4. *Let in procedure* INSERT $Y = \bigcup_k Y^k$ *and* $Q = \bigcup_k Q^k$, $0 \leq k \leq p-1$. *Then, for all* k, *with probability* $1 - n^{-\Omega(1)}$, *the following holds (*m *denotes the total number of items in the queue prior to the insertion).*
(i) If $\frac{n}{p} = O(\lg n)$ *then* $|Y^k| = \min\{O(\frac{n}{p}\frac{\lg n}{\lg\lg n}), O(\lg n)\}$, *else* $|Y^k| = (1 + O(\sqrt{\frac{p\lg n}{n}}))\frac{n}{p}$.
(ii) If $\frac{m}{p} = O(\lg n)$ *then* $|Q^k| = \min\{O(\frac{m}{p}\frac{\lg n}{\lg\lg n}), O(\lg n)\}$, *else* $|Q^k| = (1 + O(\sqrt{\frac{p\lg n}{m}}))\frac{m}{p}$.

Proof. (Omitted). It follows by way of Chernoff bounds [12]. □

The random distribution of items to processors (line 2) takes time $g|Y^k| + L$, with probability $1 - n^{-\Omega(1)}$. The insertion of the $|Y^k|$ items in the local priority queues (line 4) requires time $\tau_I(|Y^k|,|Q^k|) + L$. Thus, insertion can be implemented efficiently by employing this simple randomized scheme in time $\tau_I(|Y^k|,|Q^k|) + g|Y^k| + 2L$. For the deletion of the n smallest items, the following (rather more complicated) procedure is employed (see also [18]). The $2n/p$ smallest items in each processor are duplicated. Let X denote the set of these $2n$ items. By employing selection, the n-th smallest item e of X is determined and broadcast. Let R^k denote the set of items in processor p^k, $0 \leq k \leq p-1$, that are not larger than e. By employing selection, the n-th smallest item e^* from $\bigcup_k R^k$ is determined and broadcast to all processors. Finally, the set S of the n smallest items (the items smaller than e^*) is removed from the local priority queues and returned as the result of the delete-min operation. The procedure to manage the deletion operation on a PPQ Q is DELETEMIN(n, Q) and is outlined in figure 2. A find-min operation is implemented similarly. The following lemma bounds the total number of items and the number of items per processor that are smaller than e (lines 6-10) of procedure DELETEMIN.

DELETEMIN (n, Q)
1. **if** $|Q| = n$ **then**
2. **return** $S^k = \text{DELETEMIN\_SEQ}(|Q^k|, Q^k)$;
3. **else**
4. **for each** processor k, $0 \leq k \leq p-1$, in parallel
5. **do** $X^k = \{\text{FINDMIN\_SEQ}(2n/p, Q^k)\}$;
6. **let** $R^k = \{\}$;
7. **let** $e = \text{SELECT}(\bigcup_k X^k, n)$;
8. **while** $\text{FINDMIN\_SEQ}(1, Q^k) \leq e$
9. **let** $x = \text{DELETEMIN\_SEQ}(1, Q^k)$;
10. **let** $R^k = R^k \bigcup \{x\}$;
11. **let** $S^k = \{\}$;
12. **let** $e^* = \text{SELECT}(\bigcup_k R^k, n)$;
13. **while** $\text{FINDMIN\_SEQ}(1, Q^k) \leq e^*$
14. **let** $x = \text{DELETEMIN\_SEQ}(1, Q^k)$;
15. **let** $S^k = S^k \bigcup \{x\}$;
16. **return** S^k ;

Figure 2. Procedure DELETEMIN

Lemma 5. *Let in procedure* DELETEMIN $R = \bigcup_k R^k$, $0 \leq k \leq p-1$. *Then,* $|R| < 1.15n$ *with probability at least* $1 - 2^{-\Omega(n)}$. *Moreover, for all* $0 \leq k \leq p-1$, *with probability* $1 - n^{-\Omega(1)}$, *the following holds.*

$$|R^k| \leq \begin{cases} \min\left\{O\left(\frac{1.15n}{p}\frac{\lg n}{\lg\lg n}\right), O(\lg n)\right\} & \text{if } \frac{n}{p} = O(\lg n) \\ (1 + O(\sqrt{\frac{p \lg n}{1.15n}}))\frac{1.15n}{p} & \text{otherwise.} \end{cases}$$

Proof. (Omitted). It follows from Chernoff bounds and Azuma inequality [12]. □

The determination of the $2n/p$ smallest items per local priority queue (line 5) in procedure DELETEMIN requires time $\tau_F(2n/p, |Q^k|) + L$. The median selection process (line 7) takes time $T_{sel}(2n, n, p)$. The sequential find-min and delete-min operations per local priority queue (lines 8-10) involve $|R^k|$ items, with probability $1 - n^{-\Omega(1)}$, and therefore require time bounded above by $\tau_F(1, |Q^k|) + (\tau_F(1, |Q^k|) + \tau_D(1, |Q^k|))|R^k| + L$. Accordingly, the selection process (line 12) takes time $T_{sel}(|R^k|, n, p)$. By employing probabilistic arguments similar to those of lemma 4 we can establish that the sequential find-min and delete-min operations per local priority queue (lines 13-15) involve $|Y^k|$ items, with probability $1 - n^{-\Omega(1)}$, and therefore require time $\tau_F(1, |Q^k|) + (\tau_F(1, |Q^k|) + \tau_D(1, |Q^k|))|Y^k| + L$. Finally, the local operations (lines 6,11,19) take time $O(1) + 3L$. Thus, deletion can be implemented by this simple randomized scheme in approximate time $\left(\tau_F(1, |Q^k|) + \tau_D(1, |Q^k|)\right)\left(|R^k| + |Y^k|\right) + O(L) + T_{sel}(2n, n, p) + T_{sel}(|R^k|, n, p)$.

Theorem 6. *For any constant* $\rho > 0$, *there exists a BSP algorithm for supporting the PPQ operations* INSERT, FINDMIN, *and* DELETEMIN, *that for all* n, p, L, *and* g *such that* $n/p = \Omega(\lg n)$, $L = O((n \lg(m/p))/(p \lg p))$, *and* $g = O(\lg(m/p) \min\{\hat{g}, 1\})$, *with probability* $1 - O(n^{-\rho})$, *requires time,* $O(n \lg(m/p)/p)$ *per* INSERT, $O(n \lg(m/p)/p)$ *per* FINDMIN, *and* $O(n \lg(m/p)/p)$ *per* DELETEMIN *operation, where* m *refers to the total number of items in the underlying PPQ and* $\hat{g}$ *is as in definition 3.*

Proof. For any constant $\rho > 0$ and $n/p = \Omega(\lg n)$, it follows by lemmas 4 and 5 that $|Y^k| = O(n/p)$ and $|R^k| = O(n/p)$, with probability $1 - O(n^{-\rho})$. Similarly, $|Q^k| = O(m/p)$, with the same probability. For $t = 2$ in lemma 1 and lemma 2 we get $T_{sel}(2n, n, p) + T_{sel}(|R^k|, n, p) = O(n/p)$. By substituting in the analysis of procedures INSERT and DELETEMIN, $O(n \lg (m/p)/p)$ for $\tau_I(O(n/p), O(m/p))$, $\tau_F(O(n/p), O(m/p))$ and $\tau_D(O(n/p), O(m/p))$ [19] we get the desired result. □

We note that we can improve the performance of the INSERT and DELETEMIN operations in several ways. For example, if we substitute in the above result leftist heaps for ordinary ones the following theorem is derived.

Theorem 7. *For any constant $\rho > 0$, there exists a BSP algorithm for supporting the PPQ operations* INSERT, FINDMIN, *and* DELETEMIN, *that for all n, p, L, and g such that $n/p = \Omega(\lg n)$, $L = O(n/(p \lg p))$, and $g = O(\hat{g})$, with probability $1 - O(n^{-\rho})$, requires time, $O(n/p + \lg (m/p))$ per* INSERT, *$O(n/p)$ per* FINDMIN, *and $O(n/p \lg (m/p)/p)$ per* DELETEMIN *operation, where m refers to the total number of items in the underlying PPQ and $\hat{g}$ is as in definition 3.*

The presented PPQ algorithms have been implemented on top of a BSP library. Ordinary and leftist heaps are both implemented as the underlying sequential data structure. Details of the implementation, such as compiler used, compiler options, version of BSP library, and additional experimental results appear in the full version of the paper. In the experiments indicated here we performed a number of INSERT and DELETEMIN operations on heaps of size 256K on a Cray T3D. Data are integers drawn from a uniform distribution. The *generic* nature of our implementations allows for any data type. To this end, a comparison function `compare` is used. The timing results (obtained through a library-supplied timing function) mentioned in the following table are averages over four experiments.

Deletion (Heap Size: 256K)						
	Ordinary Heap			*Leftist Heap*		
n	$p = 1$	$p = 8$	$p = 32$	$p = 1$	$p = 8$	$p = 32$
$2K$	0.081	0.026	0.033	0.110	0.025	0.033
$8K$	0.306	0.059	0.048	0.428	0.064	0.046
$16K$	1.160	0.184	0.084	1.633	0.214	0.078

Insertion (Heap Size: 256K)						
	Ordinary Heap			*Leftist Heap*		
n	$p = 1$	$p = 8$	$p = 32$	$p = 1$	$p = 8$	$p = 32$
$1K$	0.004	0.0016	0.0009	0.009	0.002	0.001
$4K$	0.019	0.0062	0.0019	0.039	0.007	0.002
$16K$	0.093	0.0258	0.0071	0.156	0.026	0.008

Table 1. Execution time for DELETEMIN and INSERT on a Cray T3D.

As the trace of the operations differs for varying machine configurations (value of p) one can't derive reliable speedup conclusions from the timing results; this is due to the randomly distributed nature of the PPQ and its operations. Another reason speedup conclusions cannot be easily drawn is the effect of caching. Table 1 indicates that the INSERT operation on 32 processors is approximately 5-18 times faster than on one processor, with typical values 10-15. For the DELETEMIN operation comparable performance is achieved. We note that the size of the heaps and the number of elements inserted/deleted per operation has been kept small to emphasize the efficiency and practicality of our methods. For large problem instances, higher and more consistent performance is achieved. The main reason for the adequate performance of our implementations on small problem instances derives from the efficiency of the underlying parallel selection operation. Although it might have been expected, following the theoretical conclusions, that the leftist heap implementation would yield higher performance, particularly for INSERT

operations, this has not been the case in general as ordinary heap operations involve array indexing whereas leftist heap operations involve more complex (and expensive) pointer manipulations.

3 List contraction

This section considers BSP computation on linked lists. The most common problem of this kind is *list ranking*: for each node determine its distance from the head (or tail) of the list (see e.g. [8]). List ranking allows one to solve more general list problems, such as all-prefix sums on a list. Following [10], we consider this class of problems as an abstract *list contraction*: contract the list to a single node, using the operation of merging two adjacent nodes as a primitive. In more concrete settings, this merging operation is implemented by pointer jumping.

On a sequential model of computation the problem can be solved by a trivial algorithm that traverses the list of n nodes in $\Theta(n)$ time. The problem is rather more complicated on parallel models. The easiest way to obtain an efficient parallel list contraction algorithm is by using randomization. A simple randomized list contraction algorithm from [13] is based on the technique of *random mating*. The algorithm proceeds in parallel steps. In each step a node chooses its left or right neighbor by flipping an independent unbiased coin. Then the pairs of nodes that have chosen each other merge. The procedure is repeated until only one node is left. One step of the algorithm reduces the length of the list by about one quarter, thus the expected number of parallel steps is $\Theta(\log n)$.

The amount of work performed by the algorithm in [13] is optimal; however, in the PRAM model the time-processor product is still suboptimal. Many attempts to improve the processor efficiency of randomized list contraction have been made. An algorithm in [14] is time-processor optimal. Although it is slightly suboptimal in time, it performs better in practice than the more sophisticated algorithm in [1], optimal both in time and in the time-processor product.

The problem of optimal efficiency for randomized list contraction is much easier to solve in the BSP model, given sufficient slackness. We assume that the input list is distributed across the processors evenly (n/p elements per processor), but otherwise arbitrarily. The following straightforward implementation of random mating in BSP was suggested in [11].

RANDOMIZED LIST CONTRACTION

We assume $n = \Omega(p^2 \cdot \log p)$. The algorithm proceeds in two stages.

First stage. The list is reduced from n to n/p nodes by repeated rounds of random mating. Each round is implemented by a superstep which merges the mating pairs. An easy analysis along the lines of [10] shows that $O(\log p)$ rounds will suffice with high probability. Since the length of the list is expected to decrease exponentially, the communication cost of the first round, equal to $O(n/p \cdot g)$, dominates all subsequent communication with high probability.

Second stage. The remaining n/p nodes are collected in a single processor. This processor completes the contraction, reducing the list to one node by local computation. The expected communication cost of the whole algorithm is $O(n/p \cdot g)$, and the expected synchronization cost is $O(\log p \cdot l)$. The amount of parallel slack

required can be reduced to $n = \Omega(p \log p)$ by employing more tight probability bounds and a new balancing scheme during the final phases of the algorithm. The algorithm terminates by employing Wyllie's algorithm [8] on a list of size $n/\lg n$.

Another direction of research has been aimed to provide an optimal deterministic algorithm for list contraction. Known efficient deterministic algorithms for PRAM (see e.g. [8,15]) typically involve symmetry breaking by deterministic coin tossing (see [3]). Such algorithms are complicated and often assume non-standard arithmetic capabilities of the computational model, such as bitwise operations on integers. As in the case of randomized algorithms, it is much easier to design an optimal deterministic algorithm for list contraction in the BSP model, provided that the slackness is sufficient.

DETERMINISTIC LIST CONTRACTION

We assume $n \geq p^3 \cdot \log p$. The algorithm proceeds in two stages.

First stage. The list is reduced from n to n/p nodes by repeated rounds of deterministic mating.

Each round starts with reducing all chains of adjacent elements that are local to any particular processor. All nodes that remain after this reduction have both neighbors outside their own processors.

After that, a complete weighted digraph with p vertices is constructed. Each vertex of the graph represents a processor. The weight of the edge from v_1 to v_2 is defined as the number of nodes in the processor represented by v_1 with their right neighbors in the processor represented by v_2. The graph is used to mark each processor either "right-looking" or "left-looking". Each node inherits the mark from its processor. Let m be the number of nodes before the current round. The marks are assigned in such a way that the number of adjacent pairs of nodes looking at each other is at least $m/4$. Such a marking always exists and can be easily computed from the graph by a greedy algorithm in sequential time $O(p^2)$.

Each round is implemented by three supersteps. In the first superstep each processor computes its local part of the graph, namely the weights of its own outcoming edges. After that, the whole graph is collected in a single processor. This processor computes the marks and tells each processor its mark. In the second superstep each node chooses its left or right neighbor, according to the mark received by its own processor. Then the pairs of nodes that have chosen each other merge together. The processor to hold the merged pair is chosen arbitrarily between the two processors holding the original nodes. After the second superstep, the total number of nodes becomes at most $3m/4$, but the distribution of these nodes across the processors may not be even. Therefore, it is necessary to perform the third superstep, which redistributes the nodes so that each processor receives at most $3m/4p$ nodes. This completes the round.

The total number of rounds necessary to reduce the list to n/p nodes is $O(\log p)$. Since the length of the list decreases exponentially, the communication cost of the first round, equal to $O(n/p \cdot g)$, dominates all subsequent communication with high probability.

Second stage. The remaining n/p nodes are collected in a single processor. This processor completes the contraction, reducing the list to one node by local

computation. The communication cost of the whole algorithm is $O(n/p \cdot g)$, and the synchronization cost is $O(\log p \cdot l)$.

Both the randomized and the deterministic list contraction algorithms can be used to solve the problem of tree contraction in the BSP model. The tree is partitioned across the processors by an algorithm from [4], using list contraction on the Euler tour of the tree. Each processor contracts its part of the tree locally, then the remaining nodes are collected in a single processor, and the contraction is completed by a local computation. The asymptotic BSP cost of this method of tree contraction is the same as of the list contraction algorithms above.

References

1. R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list ranking. *Information Processing Letters*, 33(5):269–273, 1990.
2. A. Baumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of Euro-Par'96*, LNCS 1124, Springer Verlag, August 1996.
3. R. Cole and U. Vishkin. Deterministic coin tossing with application to optimal list ranking. *Information and Control*, 70(1):32–53, 1986.
4. H. Gazit, G. L. Miller, and Shang-Hua Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., *Concurrent Computations: Algorithms, Architecture and Technology*, pp. 139–156, 1988.
5. A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Tech. Report PRG-TR-14-96, Comp. Laboratory, Oxford University, June 1996.
6. A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proc. of the 1996 Int'l Conf. on Parallel and Distrib. Proc. Techniques and Applications*, USA, August 9-11, 1996.
7. A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Tech. Report PRG-TR-23-96, Comp. Laboratory, Oxford University, October 1996.
8. J. JáJá. *An Introduction to Parallel Algorithms*. Addison–Wesley, 1992.
9. R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and-branch and bound. *Journal of the ACM*, 40(3):765-789, 1993.
10. C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.
11. W. F. McColl, 1996. Private communication.
12. C. J. H. McDiarmid. On the method of bounded differences. *Surveys in Combinatorics*, (J. Siemons, ed.), Volume 141,pp. 148-188, CUP 1989.
13. G. L. Miller and J. F. Reif. Parallel tree contraction and its applications. In *26th IEEE FOCS*, pages 478–489, October 1985.
14. M. Reid-Miller. List ranking and list scan on the Cray C–90. In *Sixth Annual Symposium on Parallel Algorithms and Architectures*, pages 104–113, 1994.
15. M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.
16. R. Motwani and P. Raghavan. *Randomized Algorithms*. CUP, 1996.
17. M. C. Pinotti and G. Pucci. Parallel priority queues. *IPL*, 40:33-40, 1990.
18. A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Proceedings of the 6-th IEEE SPDP*, pp. 97-103, 1994.
19. R. E. Tarjan. *Data structures and network algorithms*. SIAM, 1983.
20. L. G. Valiant. A bridging model for parallel computation. *CACM*, 33:103-111,1990.