# PRAM's Towards Realistic Parallelism: BRAM's

Rolf Niedermeier[1] and Peter Rossmanith[2]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany, niedermr@informatik.uni-tuebingen.de
[2] Fakultät für Informatik, Technische Universität München, Arcisstr. 21, D-80290 München, Fed. Rep. of Germany, rossmani@informatik.tu-muenchen.de

**Abstract.** Due to its many idealizing assumptions, the well-known parallel random access machine (PRAM) is not a very practical model of parallel computation.
As a more realistic model we suggest the BRAM. Here each of the $p$ processors gets a piece of length $n$ of the input, which thus has size $pn$ in total. Access to global memory has to be data-independent, block-wise, and has to obey the owner restriction. Assuming different global memory sizes, BRAM's are suitable for modeling various parallel computers ranging from bounded degree networks to completely connected parallel machines, while abstracting from architectural details.
We present optimal BRAM algorithms requiring different global memory sizes and different numbers of block communications for the longest common subsequence and the sorting problem.

## 1 Introduction

It is a "well-known pragmatic rule that any parallel machine can be used efficiently, provided it is used to solve large enough problem sizes" [12]. The purpose of this paper is to specify the above claim more precisely and to put it into a formal framework drawn from parallel complexity theory. Herein, we make use of the most favorite model in parallel algorithm and complexity theory, the parallel random access machine (PRAM). The two features that decide the PRAM's conceptual simplicity (and, thus, its popularity) are its synchronous mode of operation and its unit time access to global shared memory.

The obvious dilemma between theory (PRAM's) and practice (asynchronous, distributed memory machines) is the seemingly large gap between ideal model and real machine. There roughly are two main ways to bridge the gap. The first is to try to preserve the full PRAM on the level of algorithm design and to show up ways how to implement PRAM's on bounded degree networks [16, 18]. The second one is to abandon the intact world of PRAM's and to use models of computation closer to really existing parallel machines [6]. Whereas the first approach still suffers at least from large constant factors for PRAM simulations on existing machines, the second approach destroys the conceptual simplicity of

This research was partially supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 0342, TP A4 "KLARA."

the underlying model and makes it harder or even impossible to create a real complexity theory upon it. So we subsequently try to find a middle-way between both the extremes.

A typical, existing and foreseeable parallel computer is characterized by a relatively small number of processors compared to a relatively large number of data-items per processor [6]. In our eyes, the deficiency of newly presented models is that they either lack *conceptual simplicity* or they stick to close to the classical PRAM model and do not take into consideration an important side condition like small number of processors paired with large number of data-items. So our main goal is to present a new model without the above deficiencies. In this sense, the contribution of this paper is first of all to methodology in parallel computation and only second to concrete algorithm design.

To introduce our new model of a parallel machine, called *BRAM*[1] (Block-RAM), our general plan of attack is to use known concepts from the PRAM world, to combine them in a new way, and to add a further demand concerning the input/output convention. More precisely, the BRAM model, which is formally introduced in Section 3, evolves informally speaking by putting together the concept of *data-independent* (or oblivious) computations [15] with the *owner read, owner write* restriction for PRAM's [17]. In addition, we assume that *each* of the $p$ processors of a BRAM holds $n$ data-items of an input of total size $pn$ in its *local memory* and that the number of global memory cells may range from $p$ to $p^2$. Note that, for example, BRAM's with global memory size $O(p)$ in essence model bounded degree networks of processors and that BRAM's with global memory size $p^2$ in fact model completely connected parallel machines. Thus, in a sense, BRAM's model parallel computers ranging from bounded degree to completely connected networks from a more abstract point of view, ignoring architectural details.

One prospect of the BRAM is that it allows the use of sequential algorithms in the design of parallel ones in a systematic way. Often we are able to *prove* the *work-optimality* of our algorithms. In addition, algorithms designed for the BRAM, due to their static communication pattern known at compile time, due to their owner read, owner write communication protocol inter alia avoiding contention effects, and due to their small number of blocked communications enabling the bypassing of latency effects and requiring only simple synchronization mechanisms and a strong use of *locality*, should allow an easy and efficient mapping to existing distributed memory machines. Nevertheless the BRAM still provides enough conceptual simplicity necessary to build a structural theory of BRAM algorithms.

As to the algorithmic problems studied in this work, let us only mention that the sorting problem can be solved work-optimally with a speedup factor between $p/2$ and $p/3$ compared to the best sequential algorithm.

Due to the lack of space several details had to be omitted. They will be in the full paper.

[1] Van Emde Boas [19] uses the terminus "BRAM" in order to refer to a kind of Boolean RAM, which should not lead to any confusion with our model.

## 2 Preliminaries

A *PRAM* (Parallel Random Access Machine) is a set of Random Access Machines, called processors, that work *synchronously* and communicate via a *global, shared memory.* Each computation step takes *unit time* regardless whether it performs a local or a global (i.e., remote) operation. Each processor has a local memory. The input to a PRAM computation is initially given in global memory. A PRAM processor is identified by its processor number. We call processors *neighbors* if their processor numbers differ by one.

The most restricted PRAM variant with respect to access possibilities to shared memory is the *owner mechanism* introduced by Dymond and Ruzzo [7]. Later on, OROW-PRAM's (Owner read, Owner write) were introduced as a still more restricted variation of EREW-PRAM's [17]. Here, a uniquely determined read-owner, resp. write-owner, processor is assigned to each global memory cell. The read owner is the only processor with read and the write owner is the only processor with write permission for this cell. More formally, there are functions called $write\text{-}owner(i, n)$ resp. $read\text{-}owner(i, n)$ that map global memory cells into the set of processors and if $j = write\text{-}owner(i, n)$, then $j$ is the only processor allowed to write into cell $i$. Both functions have to be easily computable, that is, for example, by a logarithmically space bounded Turing machine. In this way, global memory is deteriorated to a set of directed channels between pairs of processors and, in general, we may view OROW-PRAM's just as completely connected, synchronous processor networks.

Recently, the concept of data-independence for PRAM's was studied from a structural complexity theoretic point of view [15]. The central idea behind *data-independence*, also called *obliviousness* by other authors, is the demand for control and communication structures that are independent of the input. Thus, e.g., the global memory access pattern, that is, the addresses used, may only depend on the *size* of the input to the PRAM computation, but not on the concrete input word. The communication structure is static along these lines. The demand for data-independent communication appears to be an important prerequisite for the efficient implementation of PRAM algorithms on distributed memory machines [9]. Note that data-independence allows to optimize parallel algorithms at compile time.

The most important resource bounds for PRAM's are the number of processors and the running time. We call a parallel algorithm *work-optimal* if it performs up to a constant factor the same amount of *work* (that is, the number of processors multiplied with the running time) as the best sequential algorithm.

In the rest of this paper for the ease of presentation we always assume that any occurring fractions or roots of integers shall yield integer values.

## 3 The Model: BRAM's

The two main points of criticism against the PRAM model concern its synchronous mode of operation and its unit time access to global shared memory.

Further on, it is often criticized that PRAM's do not impose any structure on the communication pattern, that most PRAM algorithms are excessively fine-grained, that PRAM's do not allow to model locality in computations, that even for EREW-PRAM's contention may occur during memory access to the same memory bank, and so on [4, 6, 11]. On the other hand, exactly its *conceptual simplicity* is the reason why the PRAM is so popular.

The idea behind the introduction of the BRAM model as an adherent of the PRAM is that we still want to preserve the two most important points of the PRAM's simplicity (synchronism and shared memory), but put forward some natural requirements for communication structure and protocol (data-independent, OROW), for input size (*each* processor gets locally $n$ input items) and the global memory size (at most quadratic in the number of processors). Altogether, we may define the BRAM model using known concepts from the world of PRAM's.

**Definition 1.** A $p$-processor-*BRAM (Block-RAM)* is an OROW-PRAM with $p$ processors obeying the following restrictions:

1. The size of global shared memory lies between $p$ and $p^2$.
2. The input of total size $pn$ is equally distributed among the local memory banks of the $p$ processors, that is, *each* processor has local input of size $n$.
3. The communication structure is data-independent, it only depends on $p$ and $n$.

The term "block" refers to two important characteristics of BRAM's — the partitioning of the input into $p$ pieces each of size $n$ and the use of blocking techniques for communication. So blocking of communications means that data items are only exchanged in large portions between processors and generally *not* item-wise.

We chose the owner read, owner write (OROW) mechanism with a number of memory cells lying between $p$ and $p^2$, because this exactly allows to model a broad range of parallel computers, going from bounded degree to completely connected networks. Eventually, the OROW restriction compared to other conflict resolution rules like EREW, in particular prevents contention conflicts due to the "channel character" of its communication mechanism.

The requirement for a local input piece of size $n$ has many faces. First, it enables a good modeling of locality and of the fact that in most practical situations the size of the input exceeds the number of processors by large. It also models the observation that one advantage of using parallel machines is that they increase the total main memory size at hand. Our input size demands also lead in a natural way to the use of relatively few communications with relatively large data blocks to be transferred. So latency problems might be done away with. Finally, it also makes a direct comparison with sequential algorithms possible. We can use the best sequential algorithms during local computations. Since we may use sequential algorithms in some sense as parameters, we can often say that our parallel algorithms are *provably* work-optimal up to a certain constant factor of small size (e.g., in the case of sorting, see Section 6).

At last, the request for data-independent communications, as pointed out in the previous section, puts the missing structure on PRAM communications and admits optimizations at compile time [9, 15].

Typical BRAM algorithms are clearly separated in global communication and local computation phases. The blocking of communications and the in general small number of communication phases facilitates the implementation of BRAM algorithms on existing asynchronous distributed memory machines. In particular, synchronization costs remain quite modest. A useful, additionally restricting demand for BRAM algorithms also might be that the local computation time has to be significantly greater than the communication time (measured in the size of the data to be transferred). Several of our BRAM algorithms possess this property, as the subsequent sections show.

There is good reason to assume that the BRAM model may still be used to build a structural theory in analogy to the one for PRAM's. This is due to the definition of BRAM's in the line of known PRAM concepts and the use of *only* two parameters, number of processors $p$ and local input size $n$.

## 4 Related Work

There is a lot of literature dealing with more practical models of parallel computation. See the papers of Chin [4] and Heywood and Leopold [11] for recent surveys. We only mention some papers with close relations to our work.

Perhaps the closest relationship is with Gottlieb and Kruskal [10]. They also study the phenomenon of a more efficient use of parallel machines through enlargement of problem sizes. For example, they introduce the so-called supersaturation limit, which, informally speaking, asks how the relation of input size to number of processors has to be in order to get an asymptotically optimal speedup of $\Theta(p)$. If there exists a supersaturation limit, they call a problem *completely parallelizable.* In three tables they present lower and upper bounds for several problems. Gottlieb and Kruskal seemingly do not use parallelization by using sequential algorithms.

Kruskal, Rudolph, and Snir [12] started to build a complexity theory of efficient parallel algorithms. They emphasize speedup *and* work-optimality as two crucial aspects of parallel algorithm design. Using the conventional PRAM model, they compare the performance of a parallel algorithm with the best sequential algorithm. As there is not always a provably best sequential algorithm known, it is not possible to formally define complexity classes with notions like reducibility and completeness. Moreover, they do not work under the proviso that the input size exceeds the number of processors by large.

Aggarwal, Chandra, and Snir introduced Block-PRAM's (BPRAM's, not to be confused with out BRAM's) [1] and Local-memory PRAM's (LPRAM's) [2] to model latency restrictions and locality aspects. Their basis models are "conventional" EREW- resp. CREW-PRAM's. In their framework, the input is still given in global memory and they do not include the use of sequential algorithms.

Culler *et al.* [6] presented the LogP model as a more realistic model of parallel computation. They say that "parallel algorithms will need to be developed under the assumption of a large number of data elements per processor." The LogP model, however, is quite far away from the PRAM model and has as many as four main parameters: number of processors and memory modules, per-processor communication bandwidth, communication delay or latency, and per-processor communication overhead. The large number of parameters makes algorithm design and analysis rather difficult. Culler *et al.* also emphasize that current PRAM simulation techniques on existing parallel machines suffer from large constant factors, context switching costs, and hardware support requests for synchronization.

Heywood and Leopold [11] underline the importance of input and output, which is often neglected in algorithm design. Further on, they criticize that the PRAM imposes no structure on communication and that it assumes a "worst case" scenario for locality. Heywood and Leopold also stress that the LogP model only is designed for short term because it has not enough conceptual simplicity. According to them a computational model needs to offer the opportunity to use locality, but also has to provide synchronous communication and modularity.

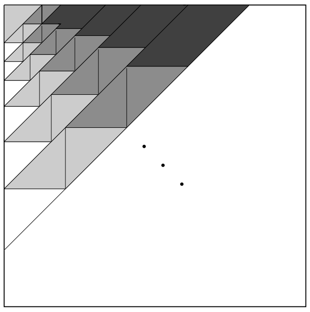
Vitányi [22] analyzes and discusses in detail the interplay between locality of computation, communication, and physical realization of multicomputers. In particular, he provides lower bounds on the average interconnect length of physical embeddings of various parallel architectures. As a consequence, he asks for "*realistic* formal models of nonsequential computation." So he concludes that "mesh-connected architectures may be the ultimate solution for interconnecting the extremely large (in numbers) computer complexes of the future."

In an announcement for a workshop on suggesting computer science agendas for high-performance computing [21], several requirements are enumerated. Here, questions and problems like latency, synchronization, and portability are of major importance. One of the focal questions there is how the transition from serial to parallel computation may occur.

## 5 Longest Common Subsequence

The longest common subsequence problem is, given two sequences of same length, to find a maximum length common subsequence of both sequences. The longest common subsequence problem possesses a well-known, quadratic time dynamic programming solution. To the authors' knowledge only for some special cases more efficient sequential solutions are known. The decisive fact we will also make use of here is that to compute the longest common subsequence, one basically has to compute the entries of a table $c[0..n, 0..n]$. The only thing of interest for the time being is that table entries $c[0, i]$ and $c[i, 0]$, $i \geq 0$, are initialized with zero and that for $i, j > 0$, entry $c[i, j]$ only depends on the entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$ and whether the $i$th element of the first sequence is the same as the $j$th element of the second sequence.

We now briefly discuss a BRAM parallelization of the dynamic programming approach that exhibits the fundamental importance of how to partition the algorithm's input, output and intermediate data structures in order to enable blocked communications and bounded degree communication, i.e., communication only with neighboring processors.



Our modus operandi is to exhaust table $c$ by using right angled triangles of increasing size, having two equal sides each time. The pairs of triangles represent the areas of local computations. In general, each processor is assigned to a pair of triangles each time. The right-hand picture shows a partitioning of table $c$ for four processors. The advantage of this approach is that it allows for local computations and thus also for the blocking of communications. Due to the lack of space, we only state the result.

**Theorem 2.** *The longest common subsequence of two strings of length $pn$ each can be computed by a $p$-processor BRAM in time $O(pn^2)$ using $2p$ global memory cells and performing $O(p \log p)$ blocked communications per processor.*

## 6 Sorting

Cole [5] gave a work-optimal merge-sort algorithm that sorts in $O(\log n)$ time using $n$ processors of an EREW-PRAM. From a purely theoretical point of view, this result is the best possible. From a more practical point of view, large constants and the model of computation ("full" EREW-PRAM) are still disturbing factors. We present an efficient, work-optimal sorting algorithm for BRAM's. For this purpose, we make use of Kunde's technique of sorting with all-to-all mappings on grids of processors [13]. To combine a certain number of grid processors into sub-grids called blocks and to sort "locally" within blocks is a basic idea of Kunde. In our case a single processor and its local memory bank play the rôle of a block.

*Algorithm 1 (Sorting)*

1. Locally sort each processor's $n$ items.
2. Distribute the sorted data items among all processors as "equally" as possible: Assume that the items of the $i$th processor ($1 \leq i \leq p$) are called $x_1^i, \ldots, x_n^i$. Then $x_j^i$ goes to the $k$th processor, where $k = 1 + (i + j) \bmod p$.
3. Locally sort each processor's $n$ items.
4. Send all items stored in a processor's local memory from cell $((i-1)n/p)+1$ to cell $in/p + p$ to the $i$th processor.
5. Locally sort each processor's $n + p^2$ items.
6. Finally throw out overlaps of sorted arrays of neighboring processors.

In Algorithm 1 local sorting predominates the overall complexity. So we may sort $pn$ data-items in parallel in nearly the same (up to a constant factor between 2 and 3) time as we need to sort $n$ data-items sequentially.

**Theorem 3.** *Sorting $pn$ items can be done by a $p$-processor-BRAM in time $2t(n) + O((n + p^2)\log p)$, using $p^2$ global memory cells and performing $2p + 2$ blocked communications per processor, where $t(n)$ denotes the time needed to sort $n$ elements sequentially.*

*Proof.* (Sketch) Since we mainly transferred Kunde's sorting methodology for grids [13] to the BRAM model, the correctness of Algorithm 1 follows similar to there. With respect to the underlying BRAM model, note that in Algorithm 1 communication partners are statically determined, they only depend on the number of processors $p$. Even the size of the data blocks to be transferred only depends on $n$ and $p$. Clearly, one communication channel between each pair of processors is sufficient.

Let us shortly analyze the time and communication complexity of Algorithm 1. The first and third step simply can be done using some sequential sorting algorithm of time complexity $t(n)$. The second and fourth step consist of all-to-all inter-processor data-exchange using blocked communications in a static (data-independent) fashion. In the second step we transmit $pn$ data-items and in the fourth step we transmit $pn + p^2$ data-items. So we transmit in these two steps $2pn + p^2$ items. The sixth step simply needs the exchange of constant size messages between neighboring processors.

The fifth step could also be done using a local sorting as in the first and third step. But it is also possible to make use of the fact that the given data is already "pre-sorted." Remember that each processor gets a sorted sub-array of size $n/p + p$ from each other processor. The idea is to keep a sorted "minimum-list" of size $p$, where in the beginning we put the first element of each of the above sub-arrays. To get all data-items sorted, we do the following: Take away the first element of the minimum-list and subsequently insert into the minimum-list the currently first element from the sub-array where the taken element came from. Doing this, use insertion sort. Then again take away the new first element of the minimum-array and so on. Altogether, this takes time $O((n + p^2)\log p)$ with a small constant, which, for $p = o(\sqrt{n})$, beats the application of general local sorting in step 5.

In total, we get the following bounds for Algorithm 1: The per-processor computation costs are $2t(n) + O((n + p^2)\log p))$ and the per-processor communication costs are $2n + p^2 + O(1)$. Altogether that yields time complexity $2t(n) + O((n + p^2)\log p)$. □

In Algorithm 1 the computation costs are predominant compared to the communication costs, because sequential sorting needs time $\Theta(n \log n)$. It is easy to see that the memory space needed on principle is bounded by $p(n + p^2)$ provided that we use a sequential algorithm that sorts in place. Under the common assumption of a very small number of processors compared to the number of data-items that means that Algorithm 1 "nearly sorts in place."

Gottlieb and Kruskal [10, Table III] state that sorting is completely parallelizable (cf. Section 4) on bounded degree networks if for the relation between number $n$ of per-processor local data-items and number $p$ of processors it holds $n = p^{\Omega(\log p)}$. In their terminology, Theorem 3 says that in our framework sorting is completely parallelizable for $n = p^2$. This still is improved in the subsequent Theorem 6. Kruskal, Rudolph, and Snir [12, pages 106–107] point out that based on a bitonic sorting network one can get an efficient sorting algorithm that is optimal provided that $\log^2 p = O(\log n)$ in their setting. It is thus in their class $ANC$ (almost efficient $NC$ fast). Translated into our setting that means that they get a completely parallelizable sorting algorithm under the same conditions as already Gottlieb and Kruskal [10] do.

Note that although our BRAM used has unbounded degree, that is, global memory size $p^2$, due to the small number of large, blocked communications these data exchanges can also be performed on bounded degree networks without time loss. In essence, this already follows from Kunde's realization of all-to-all mappings on grids of processors [13]. On the other hand, observe that in a sense Ajtai, Komlós, and Szemerédi's $NC^1$ sorting network [3] on principle shows that sorting is completely parallelizable on bounded degree networks even for $n = 1$. This result, however, is of purely theoretical interest since it at least suffers from huge constant factors (also cf. [12]).

**Corollary 4.** *For $p = O(\sqrt{n})$ sorting $pn$ items can be done by a $p$-processor-BRAM in time $O(n \log n)$, which is work-optimal up to a constant factor close to* 3.

For integer sorting (that is, the values of the items are drawn from a polynomial range), we still get work-optimality for $p = O(\sqrt{n})$.

**Corollary 5.** *If the items are drawn from a polynomial range, then sorting $pn$ items can be done by a $p$-processor-BRAM in time $O(n + p^2)$.*

The following result owing to its larger constants has a somewhat less practical flavor. It shows that we can even increase the number of processors up to an arbitrary polynomial in $n$ and still get an asymptotically work-optimal parallel sorting algorithm for BRAM's. In Gottlieb's and Kruskal's terms, it says that sorting is completely parallelizable for $n = p^\epsilon$ for any $\epsilon > 0$.

**Theorem 6.** *For $p = n^{O(1)}$ sorting $pn$ items can be done by a $p$-processor-BRAM in time $O(n \log n)$, which is work-optimal.*

*Proof.* (Sketch) In order to increase the number of processors in comparison to Corollary 4, the basic idea is to make use of an "hierarchy of localities." For the time being, assume that we want to improve Corollary 4 from $p = O(\sqrt{n})$ to $p = O(n)$. Then partition the $p$ processors (and, thus, the input) into $\sqrt{p}$ groups of $\sqrt{p}$ processors each time. Each group of processors shall play the rôle of a single processor in Algorithm 1. To sort within such a group, we again make use of Algorithm 1, but now really with individual processors. We now need 9 applications of a sequential sorting algorithm.

The step from $p = O(n)$ to $p = n^{O(1)}$ is an easy generalization of the above method. The number of applications of sequential sorting algorithms and communication phases grows exponentially with the degree of the polynomial. □

Recently, it has been shown that *average case* sorting on grids for uniformly distributed inputs can be done with only one instead of two all-to-all mappings [14]. This implies that for average case sorting on BRAM's we can save one local sorting and one global communication phase (i.e., steps 1 and 2 in Algorithm 1) and thus Theorem 3 improves to time $t(n) + O((n + p^2)\log p)$.

## 7 Conclusion and Open Questions

The purpose of this paper was to introduce the BRAM model as a natural adherent of the PRAM. The two main points of interest in the model formation were *conceptual simplicity* on the one hand and *realistic modeling* of most aspects of existing parallel machines on the other hand. Some of the advantages of the BRAM model we see are as follows. *Locality* of computation is a natural part of BRAM computations, leading to the possible (re)use of sequential algorithms, sometimes enabling results of provable work-optimality. The clear separation into local computation and into in general few phases of blocked, data-independent communications simplifies the transfer to asynchronous distributed memory machines with communication delay. BRAM's take into account that one aspect of parallel computers is the increase of main memory size and thus larger problems may be solved exclusively within the main memories of a parallel machine, whereas a sequential machine possibly had to make use of secondary storage (think of thrashing effects!).

We also studied BRAM algorithms for the matrix multiplication, the selection, and the closest pair problem. Here, we also get BRAM algorithms optimal up to small constant factors. The complexity of the list ranking problem remains open. Also work-optimal BRAM algorithms for several graph problems, e.g., depth-first tree of an undirected graph, (strongly) connected components, transitive closure etc. are of interest.

## References

1. A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proc. of 1st SPAA*, pages 11–21, 1989.
2. A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Comput. Sci.*, 71:3–28, 1990.
3. M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.
4. A. Chin. Complexity models for all-purpose parallel computation. In Gibbons and Spirakis [8], chapter 14, pages 393–404.

5. R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, Aug. 1988.
6. D. Culler et al. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
7. P. W. Dymond and W. L. Ruzzo. Parallel RAMs with owned global memory and deterministic language recognition. In *Proc. of 13th ICALP*, number 226 in LNCS, pages 95–104. Springer-Verlag, 1986.
8. A. Gibbons and P. Spirakis, editors. *Lectures on parallel computation.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1993.
9. D. Gomm, M. Heckner, K.-J. Lange, and G. Riedle. On the design of parallel programs for machines with distributed memory. In A. Bode, editor, *Proc. of 2d EDMCC*, number 487 in LNCS, pages 381–391, Munich, Federal Republic of Germany, Apr. 1991. Springer-Verlag.
10. A. Gottlieb and C. P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *J. ACM*, 31(2):193–209, April 1984.
11. T. Heywood and C. Leopold. Models of parallelism. Technical Report CSR-28-93, The University of Edinburgh, Department of Computer Science, July 1993.
12. C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Comput. Sci.*, 71:95–132, 1990.
13. M. Kunde. Block gossiping on grids and tori: Sorting and routing match the bisection bound deterministically. In T. Lengauer, editor, *Proc. of 1st ESA*, number 726 in LNCS, pages 272–283, Bad Honnef, Federal Republic of Germany, Sept. 1993. Springer-Verlag.
14. M. Kunde, R. Niedermeier, K. Reinhardt, and P. Rossmanith. Optimal Average Case Sorting on Arrays. In E. W. Mayr and C. Puech, editors, *Proc. of 12th STACS*, number 900 in LNCS, pages 503–514. Springer-Verlag, 1995.
15. K.-J. Lange and R. Niedermeier. Data-independences of parallel random access machines. In R. K. Shyamasundar, editor, *Proc. of 13th FST&TCS*, number 761 in LNCS, pages 104–113, Bombay, India, Dec. 1993. Springer-Verlag.
16. W. F. McColl. General purpose parallel computing. In Gibbons and Spirakis [8], chapter 13, pages 337–391.
17. P. Rossmanith. The Owner Concept for PRAMs. In C. Choffrut and M. Jantzen, editors, *Proc. of 8th STACS*, number 480 in LNCS, pages 172–183, Hamburg, Federal Republic of Germany, Feb. 1991. Springer-Verlag.
18. L. G. Valiant. General purpose parallel architectures. In van Leeuwen [20], chapter 18, pages 943–971.
19. P. van Emde Boas. Machine models and simulations. In van Leeuwen [20], chapter 1, pages 1–66.
20. J. van Leeuwen, editor. *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science.* Elsevier, 1990.
21. U. Vishkin. Workshop on "Suggesting computer science agenda(s) for high-performance computing" (Preliminary announcement). Announced via electronic mail on "TheoryNet", January 1994.
22. P. Vitányi. Locality, Communication, and Interconnect Length in Multicomputers. *SIAM J. Comput.*, 17(4):659–672, August 1988.

This article was processed using the LaTeX macro package with LLNCS style