Practical Massively Parallel Sorting – Basic Algorithmic Ideas

Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz

Karlsruhe Institute of Technology, Karlsruhe, Germany
{michael.axtmann,bingmann,sanders,christian.schulz}@kit.edu

Abstract. To obtain sorting algorithms that scale to the largest available machines, conventional parallel sorting algorithms cannot be used since they either have prohibitive communication volume or prohibitive critical path length for computation. We outline ideas how to combine a number of basic algorithmic techniques which overcome these bottlenecks.

1 Introduction

Sorting is one of the most fundamental non-numeric algorithms which is needed in a multitude of applications. For example, map-reduce based computations in every step have to group the data items by key values. Or load balancing in supercomputers often uses space-filling curves. This boils down to sorting data by their position on the curve.

We study the problem of sorting n elements on p processing elements (PEs) numbered $1..p$. For simplicity, we will assume that n is divisible by p. The output requirement is that the PEs store a permutation of the input elements such that the elements on each PE are sorted and such that no element on PE i is larger than any elements on PE $i-1$ for some numbering of the PEs (where the numbering can be chosen by the algorithm).

2 Preliminaries and Related Work

2.1 Data Exchange and Model of Computation

A successful realistic model is single-ported bidirectional message passing in a communication network where it is assumed that it plays no significant role who communicates with whom. It is simply assumed that at any point in time, a PE can send to at most one PE and (at the same time) receive from a (possibly different) PE. Sending a message of size ℓ machine words takes time $T_{\text{start}} + \ell T_{\text{word}}$. For simplicity of exposition, we equate the machine word size with the size of a data element to be sorted. We use this model whenever possible. In particular it yields good and realistic bounds for collective communication operations such as broadcast, reduction, and prefix sums. However, for moving the bulk of the data, we get very complex communication patterns where it is difficult to enforce the single-ported requirement.

Our algorithms are bulk synchronous algorithms. These algorithms are often described in the framework of the BSP model [26]. However, at least when considering constant factors in communication delays, it is not clear how to implement the data exchange step of BSP efficiently on a realistic parallel machine. In particular, actual implementations of the BSP model deliver the messages directly using up to p startups. For massively parallel machines this is not scalable enough.

The BSP* model [1] takes this into account by imposing a minimal message size. However, it also charges the cost for the maximal message size for all communications and this would be too expensive for our sorting algorithms. We therefore use our own model: We consider a black box data exchange function $\mathrm{Exch}(P, h, r)$ telling us how long it takes to exchange data on a compact subnetwork of P PEs in such a way that no PE receives or sends more than h words in total and at most r messages in total. Note that all three parameters of the function $\mathrm{Exch}(P, h, r)$ may be essential. Since they model locality of communication, bottleneck communication volume (see also [3,24]) and startups. A lower bound for $\mathrm{Exch}(P, h, r)$ is $hT_{\text{word}} + rT_{\text{start}}$. There are reasons to believe that we can come close to this in practice but we are not aware of actual matching upper bounds. There are offline scheduling algorithms which can deliver the data using time hT_{word} when startup overheads are ignored (using edge coloring of bipartite multi-graphs). However this chops messages into many blocks and also requires us to run a parallel edge-coloring algorithm. This is impractical.

2.2 Sorting Algorithms

Between the 1960s and the early 1990s there has been intensive work on achieving asymptotically fast and efficient parallel sorting algorithms. The "best" of these algorithms, e.g., Cole's celebrated $\mathcal{O}(\log p)$ algorithm [5] have prohibitively large constant factors. Some simpler algorithms with running time $\mathcal{O}(\log^2 p)$, however, contain interesting techniques that are in principle practical. These include parallelizations of well known sequential algorithms like mergesort and quicksort [14]. However, when scaling these algorithms to the largest machines, these algorithms cannot be directly used since all data elements are moved a logarithmic number of times which is prohibitive except for very small inputs.

For sorting large inputs, there are algorithms which have to move the data only once. Sample sort [2], is a generalization of quicksort to $p - 1$ pivots which have to be chosen very carefully in order to keep the work balanced. Sample sort only makes sense for $n \gg p^2$ elements. In particular, it will incur a lower bound of pT_{start} if data exchange is done directly. p-way multiway mergesort [8,25] has the advantage that it gives perfect distribution of the data but has an even larger sequential bottleneck due to an expensive multi-sequence selection operation (see also Section 3.1).

Compromises between these two extremes – high asymptotic scalability but logarithmically many communications versus low scalability but only a single communication – have been considered in the BSP model [26]. Goodrich [10] gives communication efficient sorting algorithms in the BSP model based on multiway merging. However, this algorithm needs a significant constant factor more communications per element than our algorithm. Moreover the BSP model allows arbitrarily fine-grained communication at no additional cost. In particular an implementation of the global data exchange primitive of the BSP that delivers messages directly has a bottleneck of pT_{start} for every global message exchange. In Section 5 we develop a multipass variant of sample sort using an algorithm by Gerbessiotis and Valiant [9] as starting point.

2.3 Multiway Merging

Sequential multiway merging of r sequences with total length N can be done in time $\mathcal{O}(N \log r)$. An efficient practical implementation may use tournament trees [15,21,25]. If r is small enough

this is even cache efficient, i.e., it incurs only $\mathcal{O}(N/B)$ cache faults where B is the cache block size. If r is too large, i.e., $r > M/B$ for cache size M, then a multipass merging algorithm may be advantageous. One could even consider a cache oblivious implementation [4].

2.4 Pseudorandom Permutations

During redistribution of data, we will randomize the rearrangement to avoid bad cases. For this we select a pseudo-random permutation, which can be constructed, e.g., by composing three to four Feistel permutations [18,6]. We adapt the description from [6] to our purposes.

So assume we want to compute a permutation $\pi : 0..n-1 \to 0..n-1$. Assume for now that n is a square so that we can represent a number i as a pair (a, b) with $i = a+b\sqrt{n}$. Our permutations are constructed from *Feistel* permutations, i.e., permutations of the form $\pi_f((a, b)) = (b, a+f(b) \bmod \sqrt{n})$ for some pseudorandom mapping $f : 0..\sqrt{n}-1 \to 0..\sqrt{n}-1$. f can be any hash function that behaves reasonably similar to a random function in practice. It is known that a permutation $\pi(x) = \pi_f(\pi_g(\pi_h(\pi_l(x))))$ build by chaining four Feistel permutations is "pseudorandom" in a sense useful for cryptography. The same holds if the innermost and outermost permutation is replaced by an even simpler permutation [20]. In [6], we used just two stages of Feistel-Permutations.

A permutation π' on $0..\left\lceil\sqrt{n}\right\rceil^2 - 1$ can be transformed to a permutation π on $0..n-1$ by iteratively applying π' until a value below n is obtained. Since π' is a permutation, this process must eventually terminate. If π' is random, the expected number of iterations is close to 1 and it is unlikely that more than three iterations are necessary

Since the description of π requires very little state, we can replicate this state over all PEs.

3 Building Blocks

3.1 Multisequence Selection

In its simplest form, given sorted sequences $d_1, \ldots, d_p$ and a rank k, multisequence selection asks for finding an element x with rank k in the union of these sequences. If all elements are different, x also defines positions in the sequences such that there is a total number of k elements to the left of these positions. If several elements have the same key, we can break ties by making them unique. We will do this in such a way that keys are augmented with a kind of a global address as an additional component. Note that this does not require us to actually store this additional data explicitly.

There are several algorithms for multisequence selection, e.g. [8,25]. Here we propose a particularly simple and intuitive method based on an adaptation of the well-known quick-select algorithm [12,19]. We first explain it for a situation where each PE has one sorted list. This algorithm has been on the slides of Sanders' lecture on parallel algorithms since 2008 [22]. Figure 1 gives high level pseudo code. The base case occurs if there is only a single element (and $k = 1$). Otherwise, a random element is selected as a pivot. This can be done in parallel by choosing the same random number between 1 and $\sum_i |d_i|$ on all PEs. Using a prefix sum over the sizes of the sequences, this element can be located easily in time $\mathcal{O}(T_{\text{start}} \log p)$. Where ordinary quickselect has to partition the input doing linear work, we can exploit the sortedness of the sequences to obtain the same

// select element with global rank k
Procedure multiSelect$(d_1, \ldots, d_p, k)$
  if $\sum_{1 \le i \le p} |d_i| = 1$ **then** *// base case*
    return the only nonempty element
  select a pivot v *// e.g. randomly*
  for $i := 1$ **to** p **dopar**
    find j_i such that $d_i[1..j_i] < v$ and $d[j_i + 1..] \ge v$
  if $\sum_{1 \le i \le p} |j_i| \ge k$ **then**
    return multiSelect$(d_1[1..j_1], \ldots, d_p[1..j_p], k)$
  else
    return multiSelect$(d_1[j_1 + 1..], \ldots, d_p[j_p + 1..], k - \sum_{0 \le i < p} |j_i|)$

Fig. 1. Multisequence selection.

information in time $\mathcal{O}(\log D)$ with $D := \max_i |d_i|$ by doing binary search in parallel on each PE. If items are evenly distributed, we have $D = \Theta(\frac{n}{p})$, and thus only time $\mathcal{O}(\log \frac{n}{p})$ for the search, which partitions all the sequences into two parts. Deciding whether we have to continue searching in the left or the right parts needs a global reduction operations taking time $\mathcal{O}(T_{\text{start}} \log p)$. The expected depth of the recursion is $\mathcal{O}(\log \sum_i |d_i|) = \mathcal{O}(\log n)$ as in ordinary quickselect. Thus, the overall expected running time is $\mathcal{O}((T_{\text{start}} \log p + \log \frac{n}{p}) \log n)$.

In our application we have to perform r simultaneous executions of multi-select on the same input sequences but on r different rank values. The involved collective communication operations will then get a vector of length r as input and their running time using an asymptotically optimal implementation (e.g. [23]) will become $\mathcal{O}(rT_{\text{word}} + T_{\text{start}} \log p)$. Hence, the overall running time of multisequence selection becomes

$$\mathcal{O}((T_{\text{start}} \log p + rT_{\text{word}} + r \log \tfrac{n}{p}) \log n) \quad . \tag{1}$$

3.2 Fast Work Inefficient Sorting

We generalize an algorithm from [13] which may also be considered folklore. In its most simple form, the algorithm arranges the PEs as a square matrix using PE indices from $1..n \times 1..n$. Input element i is assumed to be present at PE (i, i) initially. The elements are first broadcast along rows and columns. Then, PE (i, j) computes the result of comparing elements i and j (0 or 1). Summing these comparison results over row i, yields the rank of element i.

The generalization works for a rectangular $a \times b$ array of processors where $a = \mathcal{O}(\sqrt{p})$ and $b = \mathcal{O}(\sqrt{p})$. In particular, when $p = 2^P$ is a power of two then $a = 2^{\lceil P/2 \rceil}$ and $b = 2^{\lfloor P/2 \rfloor}$. Initially there are n elements uniformly distributed over the PEs, i.e. each PE has at most $\lceil n/p \rceil$ elements as inputs. These are first sorted locally in time $\mathcal{O}(1 + \frac{n}{p} \log \frac{n}{p})$.

Then the locally sorted elements are gossiped (allGather) along both rows and columns (see Figure 2), making sure that the received elements are sorted. This can be achieved in time

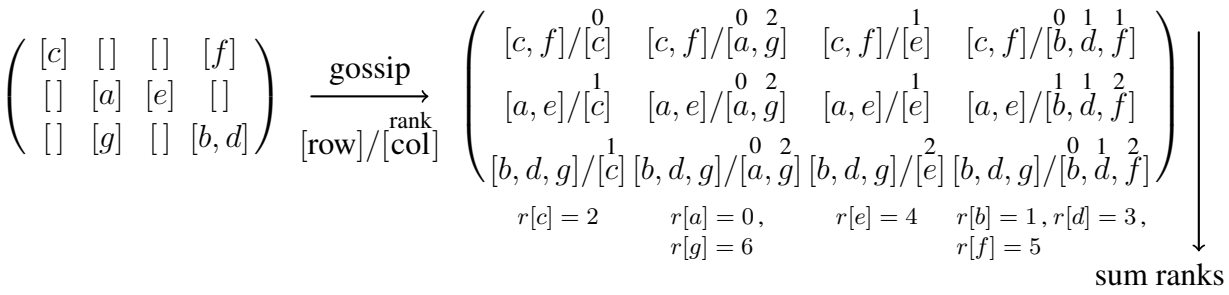

Fig. 2. Example calculations done during fast work inefficient sorting algorithm on a 3×4 array of processors.

$\mathcal{O}(T_{\text{start}} \log p + T_{\text{word}} \frac{n}{\sqrt{p}})$. For example, if the number of participating PEs is a power of two we can use the well known hypercube algorithm for gossiping (e.g., [16]). The only modification is that received sorted sequences are not simply concatenated but merged.[1]

Elements received from column i are then ranked with respect to the elements received from row j. This can be done in time $\mathcal{O}(\frac{n}{\sqrt{p}})$ by merging these two sequences. Summing these local ranks along rows then yields the global rank of each element. If desired, this information can then be used for routing the input elements in such a way that a globally sorted output is achieved. In our applications this is often not necessary because we want to extract elements with certain specified ranks as a sample. Either way, we get overall execution time

$$\mathcal{O}\left(T_{\text{start}} \log p + T_{\text{word}} \frac{n}{\sqrt{p}} + \frac{n}{p} \log \frac{n}{p}\right) \quad . \tag{2}$$

Note that for n polynomial in p this bound is $\mathcal{O}(T_{\text{start}} \log p + T_{\text{word}} \frac{n}{\sqrt{p}})$. This restrictions is fulfilled for all reasonable applications of this sorting algorithm.

4 Generalizing Multiway Mergesort (RLM-Sort)

We subdivide the PEs into "natural" groups of size p' on which we want to recurse. Asymptotically, $r := p/p'$ around $\sqrt[k]{p}$ is a good choice if we want to make k levels of recursion. However, we may also fix p' based on architectural properties. For example, in a cluster of many-core machines, we might chose p' as the number of cores in one node. Similarly, if the network has a natural hierarchy, we will adapt p' to that situation. For example, if PEs within a rack are more tightly connected than inter-rack connections, we may choose p' to be the number of PEs within a rack. Other networks, e.g., meshes or tori have less pronounced cutting points. However it still makes sense to map groups to subnetworks with nice properties, e.g., nearly cubic subnetworks. For simplicity, we will assume that p is divisible by p', and that $r = \Theta(\sqrt[k]{p})$.

There are several ways to use this generalization. We will describe a method we call "recurse last" (see Figure 3) that needs to communicate the data only k times and avoids problems with many small messages. Every PE sorts locally first. Then each of these p sorted sequences is partitioned into r pieces in such a way that the sum of these piece sizes is n/r for each of these

[1] For general p, we can also use a gather algorithm along a binary tree and finally broadcast the result.

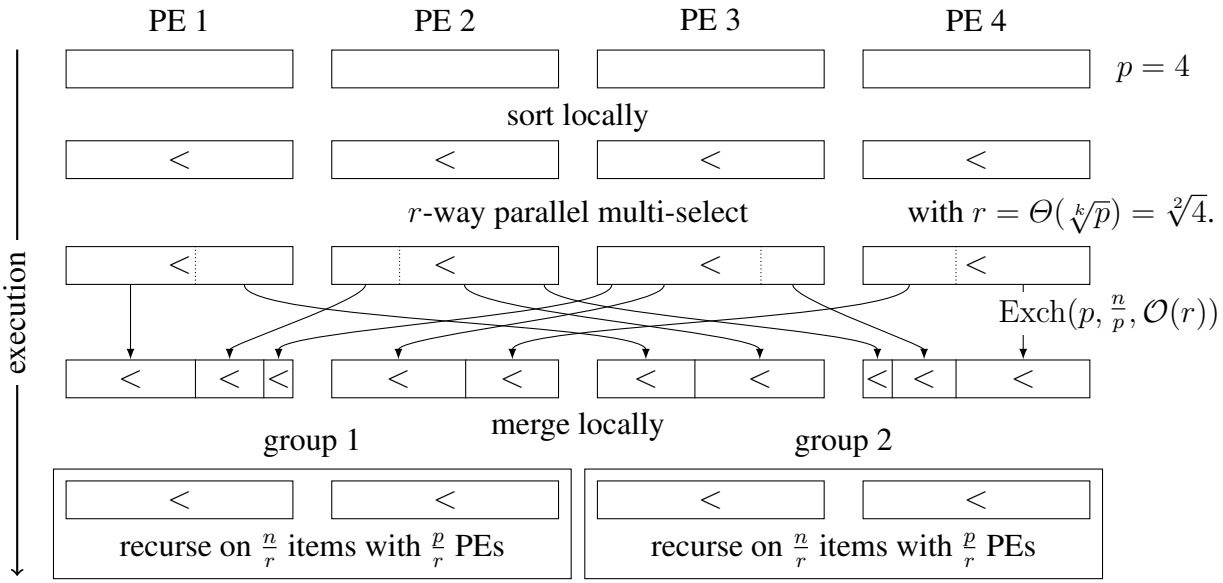

Fig. 3. Algorithm schema of Recurse Last Parallel Multiway Mergesort

r resulting *parts*. In contrast to the single pass algorithm, we now run only r multisequence selections in parallel and thus reduce the bottleneck due to multisequence selection by a factor of p'.

Now we have to move the data to the responsible groups. We explain in Section 4.1 how this is possible using time close to $\mathrm{Exch}(p, \frac{n}{p}, \mathcal{O}(\sqrt[k]{p}))$.

Now group i stores elements which are no larger than any element in group $i-1$ and it suffices to recurse within each group. However, we do not want to ignore the information already available – each PE stores not an entirely unsorted array but a number of sorted sequences. This information is easy to use though – we merge these sequences locally first and obtain locally sorted data which can then be subjected to the next round of splitting.

Theorem 1. *Recurse last parallel multiway mergesort (RLM-sort) with $k = \mathcal{O}(1)$ levels of recursion can be implemented to run in time*

$$\mathcal{O}\left(\left(T_{\text{start}} \log p + \sqrt[k]{p}\, T_{\text{word}} + \sqrt[k]{p} \log \tfrac{n}{p} + \tfrac{n}{p}\right) \log n\right) + \sum_{i=1}^{k} \mathrm{Exch}\left(p^{\frac{i}{k}}, \tfrac{n}{p}, \mathcal{O}(\sqrt[k]{p})\right) \quad . \qquad (3)$$

Proof. The time of the first r-way parallel multi-select, equation (1) with $r = \Theta(\sqrt[k]{p})$, dominates both the time of locally sorting, $\mathcal{O}(\frac{n}{p} \log \frac{n}{p})$, and of the parallel multi-selects in the $k-1$ recursive levels.

In level i of the recursion we have r^i independent groups containing $\frac{p}{r^i} = \frac{p}{p^{i/k}} = p^{1-\frac{i}{k}}$ PEs each. An exchange within the group in level i costs $\mathrm{Exch}(p^{1-i/k}, \frac{n}{p}, \mathcal{O}(\frac{r}{r^i}))$ time. Since all independent exchanges are performed simultaneously, we only need to sum over the k recursive levels, which yields the second term of equation (3). ∎

Equation (3) is a fairly complicated expression but using some reasonable assumptions we can simplify it. If all communications are equally important, the sum becomes $k\mathrm{Exch}(p, \frac{n}{p}, \mathcal{O}(\sqrt[k]{p}))$ –

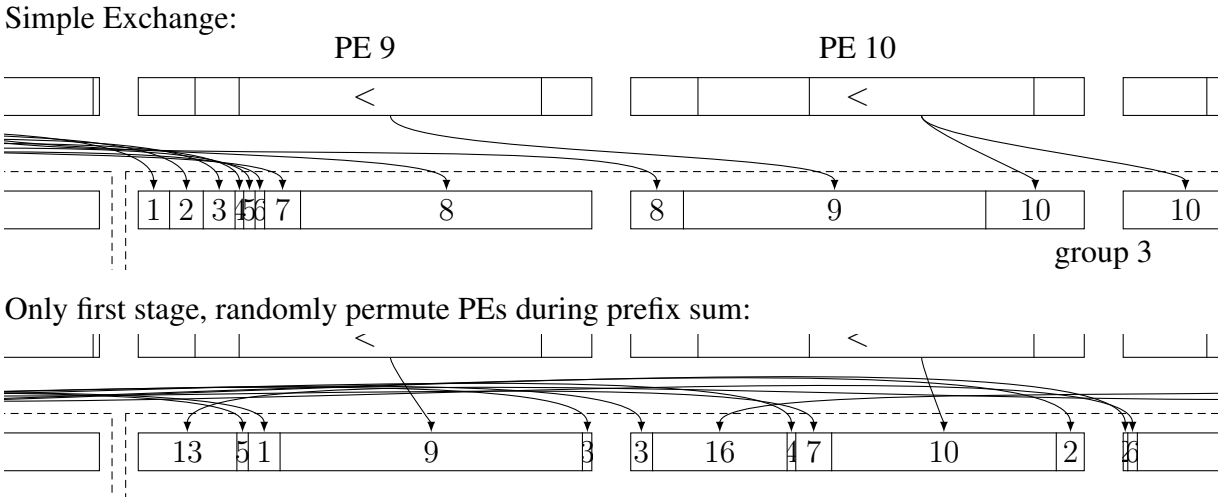

Fig. 4. Exchange schema without and with first stage: permutation of PEs

we have k message exchanges involving all the data but we limit the number of startups to $\mathcal{O}(\sqrt[k]{p})$. On the other hand, on mesh or torus networks, the first (global) exchange will dominate the cost and we get $(1+o(1))\mathrm{Exch}(p, \frac{n}{p}, \mathcal{O}(\sqrt[k]{p}))$ for the sum.

We can assume that n is bounded by a polynomial in p – otherwise, a traditional single-phase multi-way mergesort would be a better algorithm. This implies that $\log n = \Theta(\log p)$ and the $\Omega(\sqrt[k]{p})$ startups hidden in the $\mathrm{Exch}()$ term will dominate the $\mathcal{O}(\log^2 p)$ startups in the remaining algorithm. Furthermore, if $n = \omega(p^{1+1/k} \log p)$ then $n/p = \omega(p^{\frac{1}{k}} \log p)$, and the term $\Omega(T_{\text{word}} \frac{n}{p})$ hidden in the data exchange term dominates the term $\mathcal{O}(T_{\text{word}} p^{\frac{1}{k}} \log n)$. Thus Equation (3) simplifies to $\mathcal{O}(\frac{n}{p} \log n)$ (essentially the time for internal sorting) plus $(1+o(1))$ times the data exchange term. In other words, RLM-sort has an isoefficiency function [16] $\mathcal{O}(p^{1+1/k} \log p)$ if the interconnection network is not a bottleneck. In contrast, all known algorithms with a single data exchange have isoefficiency function at least $\Omega(p^2)$.

4.1 Delivering Data to the Right Place

We first explain the basic idea and then discuss how to refine it in order to avoid possible bad cases. The basic idea is to compute a prefix sum over the piece sizes – this is a vector-valued prefix sum with vector length r. As a result, each piece is labeled with a range of positions within the part it belongs to. Positions are numbers between 1 and $n' := n/r$. Adding n' times the part number, we obtain global positions. Global positions are translated to PE numbers by dividing them by n/p. This way, each PE sends and receives exactly n/p elements. Moreover, each piece is sent to one or two target PEs responsible for handling it in the recursion. Thereby, each PE *sends* at most $2r$ messages for the data exchange. Unfortunately, the number of *received messages*, although the same on the average, may vary widely in the worst case. There are inputs where some PEs have to *receive* $\Omega(p)$ very small pieces. This happens when many consecutively numbered PEs send only very small pieces of data (see PE 9 in the top of Figure 4). One way to handle this situation is to

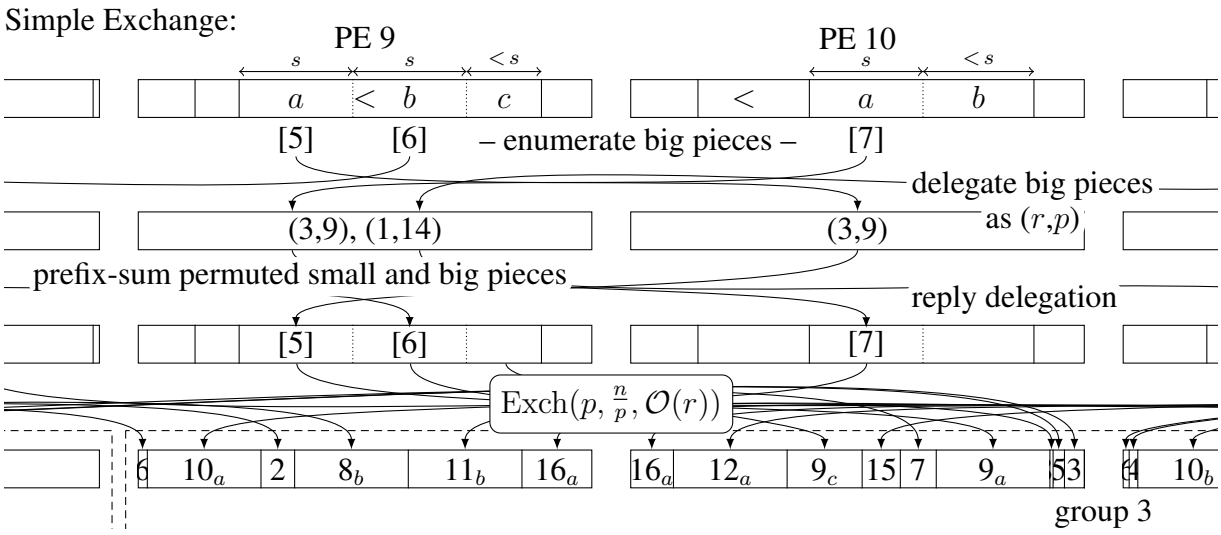

Fig. 5. Exchange schema with second stage

modify the prefix sum to also account for the startup overhead of a piece of data and (possibly) the cost of r-way merging (needed later). A disadvantage of this approach is that PEs which receive few large pieces will need more space. This overhead can be kept small however. For example, if we are willing to allow a times more startups on PEs which receive only small pieces, this will imply at most a factor $1 + 1/a$ more space on PEs which receive only large pieces. Also, there is no need for uneven data distribution in the last level of recursion – when $r = p$, each PE sends and receives exactly one piece and the final output will be perfectly balanced.

Another way to limit the number of received pieces is to use randomization. We describe how to do this while keeping the data perfectly balanced. Also note that this may be a more portable strategy since it does not need a detailed model of the tradeoff between startup overheads and bandwidth. We describe this approach in two stages where already the first, rather simple stage gives a significant improvement. The first stage is to choose the PE-numbering used for the prefix sum as a (pseudo)random permutation (see Section 2.4). However, it can be shown that if all but p' pieces are very small, this would still imply a logarithmic factor higher startup overheads for the data exchange at some PEs.

The second stage adds more randomization and invests some additional communication. The idea is to break large pieces into several smaller pieces. A piece whose size x exceeds a limit s is broken into $\lfloor x/s \rfloor$ pieces of size s and one piece of size $x \bmod s$. We set $s := an/rp$ to be a times the average piece size n/rp where a is a tuning parameter to be chosen later. The resulting small pieces (size below s) stay where they are and the random permutation of the PE numbers takes care of their random placement. The large pieces are delegated to another (random) PE using a further random permutation. This is achieved by enumerating them globally over all parts using a prefix sum. Suppose there are K large pieces, then we use a pseudorandom permutation $\pi : 0..K-1 \to 0..K-1$ to delegate piece i to PE $1 + \pi(i) \bmod p$. Note that this assignment only entails to tell PE j about the origin of this piece and its target group – there is no need to move the actual elements at this point. In Figure 5, we denote the delegation tuples with origin PE p and target group r as (r, p).

Next, for each part, a PE permutes its small pieces and delegated large pieces randomly. Only then, a prefix sum is used to enumerate the elements in each part. The ranges of numbers assigned to the pieces are then communicated back to the PEs actually holding the data and we continue as in the basic approach – computing target PEs based on the received ranges of numbers.

Lemma 1. *The two stage approach needs time* $\mathcal{O}(T_{\text{start}} \log p + rT_{\text{word}}) + 2\text{Exch}(p, \mathcal{O}(r/a), \lceil r/a \rceil)$ *to assign data to target PEs.*

Proof. Each PE will produce at most $\frac{n/p}{s} = \frac{n/p}{an/rp} = r/a$ large pieces. Overall, there will be at most $\frac{n}{s} = \frac{n}{an/rp} = pr/a$ large pieces. The random mapping will delegate at most $\lceil r/a \rceil$ of these messages to each PE with high probability. Since each delegation and notification message has constant size, $2\text{Exch}(p, \mathcal{O}(r/a), \lceil r/a \rceil)$ accounts for the resulting communication costs. All involved prefix sums are vector valued prefix sum with vector length r and can thus be implemented to run in time $\mathcal{O}(T_{\text{start}} \log p + rT_{\text{word}})$. This term also covers the local computations. ∎

Lemma 2. *No PE sends more than* $2r(1 + 1/a)$ *messages during the main data exchange of one phase of RLM-sort. Moreover, the total number of messages for a single part is at most* $p(1+1/r+1/a)$.

Proof. As shown above, each PE produces at most $r(1 + 1/a)$ pieces, each of which may be split into at most two messages. For one part, there are at most p small pieces and $\frac{n/r}{an/rp} = p/a$ large pieces. At most $p/r - 1 < p/r$ pieces can be split because their assigned range of element numbers intersects the ranges of responsibility of two PEs. Overall, we get $p(1 + 1/r + 1/a)$ messages per part. ∎

Lemma 3. *Assuming that our pseudorandom permutations behave like truly random permutations, with probability* $1 - \mathcal{O}(1/p)$*, no PE receives more than* $1 + 2r(1 + 1/a)$ *messages during one phase of RLM-sort for some value of* $a \in \Theta(\sqrt{r/\log p})$.

Proof. Let $m \leq p(1 + 1/a)$ denote the number of pieces generated for part x. It suffices to prove that the probability that any of the PEs responsible for it receives more than $1 + 2mr/p \leq 2r(1 + 1/a)$ messages is at most $1/rp$ for an appropriate constant. We now abstract from the actual implementation of data assignment by observing that the net effect of our randomization is to produce a random permutation of the pieces involved in each part. In this abstraction, the "bad" event can only occur if the permutation produces $2mr/p$ consecutive pieces of total size at most n/p. More formally, let $X_1,\ldots,X_m$ denote the piece sizes. The X_i are random variables with range $[0, \frac{an}{rp}]$ and $\sum_i X_i = n/r$. The randomness stems from the random permutation determining the ordering. Unfortunately, the X_i are not independent of each other. However, they are negatively associated [7], i.e., if one variable is large, then a different variable tends to be smaller. In this situation, Chernoff-Hoeffding bounds for the probability that a sum deviates from its expectation still apply. Now for a fixed j, consider $X := \sum_{j \leq i < j+2mr/p} X_i$. It suffices to show that $\mathbf{P}[X < n/p] \leq 1/rpm$ – in that case, the probability that the bad event occurs for some j is at most $1/rp$. We have $\mathbf{E}[X] = 2n/p$ which differs by $t := n/p$ from the bound marking a bad event. Hoeffding's inequality then assures that the probability of the bad event is at most

$$\mathbf{P}[X < n/p] \leq 2e^{-\frac{2t^2}{\frac{2mr}{r} \cdot \left(\frac{an}{rp}\right)^2}} = 2e^{-\frac{pr}{ma^2}} \leq 2e^{-\frac{pr}{a^2+1}} .$$

This should be smaller than $1/rp$. Solving the resulting relation for a yields

$$a \leq \frac{1}{2}\left(\sqrt{1+\frac{r}{\ln \frac{rp}{2}}}-1\right) \quad .$$

■

Note that Lemma 3 implies that with high probability both the number of sent and received messages during data exchange will be close to $2r$ and the number of message startups for delegating pieces (see Lemma 1) will be $o(r)$. Hence, we have shown that handling worst case inputs by our algorithm adds only lower order cost terms compared to the simple variant (plain prefix sums without any randomization) on average case inputs. In contrast, applying the simple approach to worst case inputs directly completely ruins performance.

We summarize the result in the following theorem:

Theorem 2. *Data delivery of $r \times p$ pieces to r parts can be implemented to run in time*

$$\mathcal{O}(T_{\text{start}} \log p + rT_{\text{word}}) + (1+o(1))\text{Exch}(p, \tfrac{n}{p}, 2r)$$

with high probability.

A Deterministic Solution. The basic idea is to distribute the pieces in a two phase process. In the first phase, small pieces of size at most $n/2pr$ are enumerated using a prefix sum. Small piece i of part j is assigned to PE $\lfloor i/r \rfloor$ of part j. This way, all small pieces are assigned without having to split them and no receiving PE gets more than half its final load.

In the second phase, the remaining (large) pieces are assigned taking the residual capacity of the PEs into account. Conceptually, we enumerate the unassigned elements on the one hand and the unassigned slots able to take them on the other hand and then map element i to slot i. To implement this, we compute a prefix sum of the residual capacities of the receiving PEs on the one hand and the sizes of the unassigned pieces of the other hand. This yields two sorted sequences X and Y which have to be merged in order to locate the destination PEs of each unassigned piece. Since each PE has residual capacity at least half the size of even the largest pieces, no piece is split between more than three PEs. On the other hand, since only large pieces are assigned in the second phase, no PE can receive more than $2r$ pieces. Thus, overall, no PE sends or receives more than $\mathcal{O}(r)$ pieces in this situation.

There are several ways to implement this approach. We outline one that is conceptually simple and asymptotically efficient. PE i sends the description of its piece for part j to PE $\lfloor i/r \rfloor$ of group j. This can be done in time $\text{Exch}(p, \mathcal{O}(r), r)$. Now each group produces an assignment of its pieces independently, i.e., each part consisting of p/r PEs assigns p pieces – r on each PE. The first phase is easy to implement using prefix sums. The difficult part in the second phase is merging the two sorted sequences X and Y. Here one can adapt and simplify the work efficient parallel merging algorithm for EREW PRAMs from [11]. Essentially, one first merges the p' elements of X with a deterministic sample of subsequence of Y – we include the prefix sum for the first large piece on each PE into Y. This merging operation can be done in time $\mathcal{O}(T_{\text{start}} \log p')$ using Batcher's merging network. Then each elements of X has to be located within the $\leq r$ local elements of

Y on one particular PE. Since it is impossible that $\leq r$ pieces (of total size $\leq rn/p$) fill more than $2r$ PEs (of residual capacity $> n/2p$). Each PE will have to locate only $\mathcal{O}(r)$ elements. This can be done using local merging in time $\mathcal{O}(r)$. In other words, the special properties of the considered sequence make it unnecessary to perform the contention resolution measures making [11] somewhat complicated. Overall, we get the following deterministic result.

Theorem 3. *Data delivery of $r \times p$ pieces to r parts can be implemented to run in time*

$$\mathcal{O}(T_{\text{start}} \log p + rT_{\text{word}}) + \text{Exch}(p, \mathcal{O}(r), \mathcal{O}(r)) + \text{Exch}(p, \tfrac{n}{p}, \mathcal{O}(r)) \ .$$

5 Adaptive Multipass Sample Sort (AMS-Sort)

A good starting point is the multi-pass sample sort algorithm by Gerbessiotis and Valiant [9]. However they use centralized sorting of the sample and their data redistribution may lead to some processors receiving $\Omega(p)$ messages. We improve on this algorithms in several ways to achieve a truly scalable algorithm. First, we sort the sample using fast parallel sorting. Second, we give a scalable parallel adaptation of the idea of overpartitioning [17] in order to reduce the sample size needed for good load balance.

But back to our version of multi-pass sample sort. As in RLM-sort, our intention is to split the PEs into r groups of size $p' = p/r$ each, such that each group processes elements with consecutive ranks. To achieve this, we choose a random sample of size ar where the *oversampling factor* a is a tuning parameter. If $ar > p$ every PE chooses ar/p random elements from its local elements. Otherwise, the PEs are divided into ar groups of size p/ar and one PE within each group chooses a sample element. The overall sample of size ar is sorted using a fast sorting algorithm. This could be a scalable and work efficient algorithm like parallel quicksort. In this case one chooses $ar \geq p$ and gets time $\mathcal{O}(\frac{ar}{p}(T_{\text{word}} \log p + \log(ar)) + T_{\text{start}} \log^2 p)$. For small samples it might be better to use the fast inefficient algorithm from Section 3.2. Here, we get execution time $\mathcal{O}(\frac{ar}{\sqrt{p}}(T_{\text{word}} + \log \frac{ar}{\sqrt{p}}) + T_{\text{start}} \log p)$.

From the sorted sample, we choose $br - 1$ splitter elements with equidistant rank. The *overpartitioning factor* b is a tuning parameter. These splitters are broadcast to all PEs. This is possible in time $\mathcal{O}(T_{\text{word}} br + T_{\text{start}} \log p)$.

Then every PE partitions its local data into br *buckets* corresponding to these splitters. This takes time $\mathcal{O}(\frac{n}{p} \log(br))$.

Using a global (all-)reduction, we then determine global bucket sizes in time $\mathcal{O}(T_{\text{word}} br + T_{\text{start}} \log p)$. These can be used to assign buckets to PE-groups in a load balanced way: Given an upper bound L on the number of elements per PE-group, the PEs[2] scan through the array of bucket sizes and skip to the next PE group when the total load would exceed L. Using binary search on L this finds an optimal value for L in time $\mathcal{O}(br \log n)$.

Conjecture 1. We can achieve $L = (1+\epsilon)\frac{n}{r}$ with high probability choosing appropriate $b = \Omega(1/\epsilon)$ and $a = \Omega(b \log n)$.

[2] Each PE performs the same computation here.

Within a PE group that gets assigned n' elements, each PE is assigned $\lceil n'/p \rceil$ or $\lfloor n'/p \rfloor$ elements. For simplicity of exposition we will omit the rounding from now on. Using the methodology from Section 4.1, we can now use a prefix and deliver the data using a data exchange step (time $\mathcal{O}(T_{\text{start}} \log p + rT_{\text{word}}) + \text{Exch}\,(p, (1+o(1))L, (2+o(1))r)$).

Finally, we recurse on the PE groups similar to Section 4. Within the recursion it can be exploited that the elements are already partitioned into br buckets.

We get the following overall execution time for one pass:

Conjecture 2. One pass of AMS-sort works in time

$$\mathcal{O}\left(T_{\text{start}} \log p + T_{\text{word}} \frac{r}{\epsilon}\left(1 + \frac{\log n}{\sqrt{p}}\right)\right) + \text{Exch}(p, (1+\epsilon)\tfrac{n}{p}, \mathcal{O}(r)) \quad . \tag{4}$$

In comparison, the algorithm of Gerbessiotis and Valiant [9] would have $\frac{r}{\epsilon} \to \frac{r}{\epsilon^2}$ and $\mathcal{O}(r) \to p$ in that expression, i.e., we would need a larger sample for the same load balance and we would have more startup overheads.

Using a similar argument as for MLS-Sort, we get an isoefficiency function of $p^{1+1/k}$ for $r = \sqrt[k]{p}$. This is a factor $\log p$ better than for MLS-sort and is an indication that AMS-sort might be the better algorithm – in particular if some imbalance in the output is OK and if the inputs are rather small.

6 Conclusion

We have shown how practical parallel sorting algorithms like multi-way mergesort and sample sort can be generalized so that they scale on massively parallel machines without incurring a large additional amount of communication volume. Future work includes efficient implementations of our ideas and proofs of our conjectures.

Acknowledgments: We would like to thank Christian Siebert for valuable discussions.

References

1. A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Algorithms ESA'95*, pages 17–30. Springer, 1995.
2. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *3rd Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.
3. Shekhar Borkar. Exascale computing – a fact or a fiction? Keynote presentation at IPDPS 2013, Boston, May 2013.
4. G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.
5. R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.
6. R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, pages 195–208, Toulouse, 2004.
7. Devdatt Dubhashi, Volker Priebe, and Desh Ranjan. Negative dependence through the FKG inequality. Research Report MPI-I-96-1-020, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, August 1996.
8. P. J. Varman et al. Merging multiple lists on hierarchical-memory multiprocessors. *J. Par. & Distr. Comp.*, 12(2):171–177, 1991.
9. A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

10. M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.
11. T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW-PRAM. *Information Processing Letters*, 33:181–185, 1989.
12. C. A. R. Hoare. Algorithm 65 (find). *Communication of the ACM*, 4(7):321–322, 1961.
13. Michael Ikkert, Tim Kieritz, and Peter Sanders. Parallele algorithmen. course notes, October 2009.
14. J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.
15. D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
16. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
17. H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 46–56, Cape May, New Jersey, 1994.
18. Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, April 1988.
19. K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.
20. M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 12(1):29–66, 1999.
21. P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
22. P. Sanders. Course on Parallel Algorithms, lecture notes, 2008. http://algo2.iti.kit.edu/sanders/courses/paralg08/.
23. P. Sanders, J. Speck, and J. Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.
24. Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *IEEE Int. Conf. on Big Data*, 2013.
25. J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th Euro-Par*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.
26. L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1994.