Engineering a Distributed Histogram Sort

Roger Kowalewski
Institute of Informatics
Ludwig-Maximilians-Universität
Munich, Germany
kowalewski@nm.ifi.lmu.de

Pascal Jungblut
Institute of Informatics
Ludwig-Maximilians-Universität
Munich, Germany
jungblut@nm.ifi.lmu.de

Karl Fürlinger
Institute of Informatics
Ludwig-Maximilians-Universität
Munich, Germany
fuerling@nm.ifi.lmu.de

***Abstract*—Sorting is one of the most critical non-numerical algorithms and covers use cases in a wide spectrum of scientific applications. Although we can build upon excellent research over the last decades, scaling to thousands of processing units on modern many-core architectures reveals a gap between theory and practice. We adopt ideas of the well-known quickselect and sample sort algorithms to minimize data movement. Our evaluation demonstrates that we can keep up with recently proposed distribution sort algorithms in large-scale experiments, without any assumptions on the input keys. Additionally, our implementation outperforms an efficient multi-threaded merge sort on a single node. Our implementation is based on a C++ PGAS approach with an STL-like interface and can easily be integrated into many application codes. As part of the presented experiments, we further reveal challenges with multi-threaded MPI and one-sided communication.**

***Index Terms*—Sorting, Searching, Combinatorial Algorithms, PGAS**

I. Introduction

Research in High Performance Computing (HPC) is still dominated by numerical methods. However, there is an ongoing shift towards data-intensive applications pushing the focus towards combinatorial problems. Sorting is one of the most critical non-numerical algorithms and serves as a basic building block for a wide spectrum of scientific applications. Parallel sparse matrix computations can benefit from it [2]. Irregular applications, like *N-Body* particle simulations, can achieve load balancing through space filling curves (e.g., Morton Order) by sorting n-dimensional coordinates according to a projection into the 1-dimensional space [3], [4]. *Big Data* applications, which are receiving increasing attention in the HPC area, provide another huge amount of use cases for sorting. One notable example is the Google PageRank algorithm.

In this paper we study the problem of sorting a vector of N elements partitioned among P processors. In the general case, all partitions except potentially the last one, are of equal size ($\sim N/P$). The output invariant requires a partition on processor p_i to be a sorted permutation of input elements and no element may be larger than any other element on processor p_{i+1}. Furthermore each processor p_i should end up with at most $N(1+\epsilon)/P$ elements, where ϵ is a load balancing threshold. In the extreme case we have a strict *perfect partitioning* condition where each processor needs to have the exact same number of elements in the sorted output as in the input sequence ($\epsilon = 0$). According to our experience this is not often required in scientific applications but preferred due to simplicity and productivity in the application itself.

Key to achieving high performance is obviously to minimize communication. This applies not only to distributed memory but to modern shared memory architectures as well. Current supercomputers facilitate nodes with heterogeneous computation and memory capabilities organized in a growing number of NUMA domains to manage parallelism. Recent research suggests that the network interconnect (NIC) is not necessarily the sole bottleneck anymore. While it takes approximately $10-15ns$ for a CPU to fetch one 64 byte cache line from the L3 cache [5] a NIC with 400 GBit/s bandwidth can transmit one message of the same size every 1.3ns [6]. This correlates with NUMA effects making the challenge more difficult. Depending on the data distribution, sorting is subject to a high fraction of data movement and the more we communicate across NUMA boundaries the more negative the resulting performance impact becomes.

Sorting algorithms are a well studied problem in computer science and although we can build upon excellent prior work we see a gap between theory and practice. In particular, if we consider the architectural trends as mentioned before. Exploiting the full potential of today's supercomputers requires efficient use of thousands of cores and minimizing data movement, not only for performance reasons but also to reduce the energy consumption. We engineer a practical distributed sorting algorithm achieving a parallel efficiency of ≈ 0.6 on over 3500 cores which is a fair result if we consider the high communication volume. In contrast to other work we do not pose any assumptions on the input elements regarding data distribution, data type, partition density (sparseness) or the number of processors. Our implementation is based on a partitioned global address space model (PGAS) which is not only least on par with current state of the art algorithms on distributed memory but outperforms high-performance multi-

We would like to thank the members of the DASH project team and gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) in the SmartDASH project. We further thank the authors of paper [1] for providing us a copy of the Charm++ implementation source code.

threaded shared memory libraries as well. We guarantee by design to move data only once to mitigate the impact of NUMA effects as described earlier. Since we rely only on C++14 and MPI-3 functionality it should be portable to any HPC cluster and can be easily integrated into scientific applications.

The remainder of this paper is organized as follows. Section II describes the problem of sorting in detail to set the stage for this paper. Section III summarizes current state of the art parallel sorting algorithms and discusses how they differ from our approach. Section IV continues with basic building blocks and focuses in particular on the selection algorithm used in our sorting algorithm. Section V elaborates our algorithm in detail. We primarily focus on our distributed selection approach and discuss further optimization strategies which we studied to improve the scalability of the overall sorting algorithm. Section VI conducts strong and weak scaling studies on up to 3584 cores to demonstrate the algorithmic performance. Section VII concludes this paper and discusses future work.

II. Preliminaries

Let X be a set of N keys evenly partitioned among P processors, thus, each processor *owns* $n_i \sim N/P$ keys. We further assume there are no duplicate keys which can technically be achieved in a straightforward manner. Sorting is equivalent to permuting all keys over a binary relation in set X. A binary relation is a predicate which we can successfully apply to any ordered pair (x, y) drawn from X. Applying this recursively enables us to determine the rank of an element $I(x) = k$ with x as the k-th order statistic in X. Assuming our predicate is *less than* (i.e., $<$) the output invariant after sorting guarantees that for any two subsequent elements $x, y \in X$

$$x < y \Leftrightarrow I(x) < I(y).$$

Scientific applications usually require a balanced load to maximize performance. Given a load balance threshold ϵ, *local balancing* means that in the sorted sequence each processor P_i owns at most $N(1+\epsilon)/P$ keys. This does not always result in a *globally balanced* load which is an even stronger guarantee. Given a vector of splitters S of length $P+1$ with $S(0) = 0$ and $S(P) = N$.

Definition 1: For all $i \in \{1..P\}$ we have to determine splitter S_i to partition the input sequence into P subsequences such that

$$\frac{Ni}{P} - \frac{N\epsilon}{2P} \leq I(s_i) \leq \frac{Ni}{P} + \frac{N\epsilon}{2P}$$

Determining these splitters boils down to the *k-way selection* problem. Suppose we have two processors, each having the same number of keys. We need to find a pivot r with rank $I(r) = \frac{N}{2}$ to partition the global range into two subranges. Keys in the left partition end up on one processor and the right partition ends up on the other, respectively. After moving all keys to the correct place each processor sorts the local subsequence resulting in a globally sorted sequence.

Implementing *selection* efficiently is a challenging task especially if we have a large number of processors and skewed or nearly sorted data distributions which is not uncommon in real world problems. Before we examine this in more detail we first need to understand what *efficient* in this regard means.

In literature an efficient algorithm is one which solves the problem at hand with low complexity [7], [8]. Defining complexity depends on the algorithm itself. In selection and sorting complexity is expressed as the number of comparisons because the partitioning step, where all elements are compared to a pivot, incurs the highest cost. Furthermore, we have to distinguish between worst case, average case and best case complexity. The latter is not of particular interest because in general we cannot do better than $O(N)$ for sorting. The average case is which we attribute the highest attention to, because it supports in reasoning about the expected runtime in scientific applications. Finally, the worst case complexity is relevant both in theory and in practice. If an algorithm has unexpectedly higher worst case than average complexity it can be exploited by a malicious user as part of the attack surface.

Another important aspect are inherent constants in algorithms which are not part of the "Big O" but cannot be neglected in practice. As an example, although Quicksort has higher worst case complexity compared to heap sort with $O(N^2)$ vs. $O(N \log N)$ it is preferred in productive implementations due to lower constants which give better performance in the average case. Hardware innovation and architectural changes impact these theoretical models as well as we can confirm in the experimental evaluation. Even if an algorithm has lower complexity, *locality of reference* often outweighs theoretical performance advantages, especially if a high fraction of data movement is involved.

III. Related Work

Before discussing our approach in more detail we summarize prior work. Our algorithm adopts ideas from sample and histogram sort. We subsequently focus on practical sorting algorithms which have experimentally been proven to be scalable on distributed memory machines.

A. Sample Sort

Sample sort is one of the oldest sorting algorithms achieving high performance in distributed memory and extensively studied in literature [9], [10]. It has been adopted in many follow-up research papers including this one. Specifically, it works in three supersteps:

1. **Sampling** Each processor p_i picks a random sample of size s, often called *oversampling ratio*, and sends it to a central processor.
2. **Splitting** The central processor sorts these samples and picks $P-1$ splitters. Each resulting interval between two consecutive splitters is assigned to a single processor. The splitters are broadcast to all processors.
3. **Data Exchange** All processors partition local data according to the received splitters and exchange it with the destination processors. This step denotes a single round of ALL-TO-ALL communication. Upon receiving all chunks, a processor can use any shared memory sorting algorithm (e.g., merge sort) to sort the local subset.

We clearly see that the load balancing quality and performance efficiency depends on the sampling size. It has been shown that with a total sample size of $s = \frac{\ln P}{(1+\epsilon^2)}$ we can achieve almost perfect partitioning with high probability [11]. If ϵ becomes small we may have to perform another load balancing step around the boundaries to satisfy the partitioning conditions, which drastically decreases performance.

Mitigating the heavy dependency on random sampling can be achieved through regular sampling [12]. This approach probes keys from an already sorted list, which is why random sampling is *theoretically* more efficient than regular sampling. However, regular sampling has been shown to achieve better performance in practice due to an almost perfect global balancing [13].

B. Probabilistic Partitioning

Probabilistic partitioning comes close to the idea of sample sort with one major difference. Instead of determining splitters after a single round of sampling, we operate on a vector of splitter candidates which are iteratively refined until the load balance condition is satisfied. While this approach seems to be less efficient, it has several advantages. It is much easier to achieve perfect partitioning since we come closer to the final splitter key with each iteration. Additionally, although determining the splitter may take a number of iterations we can use the already computed information to load balance the data. Before explaining this in more detail we summarize the relevant steps [14], [15], [1].

1) A central processor initially broadcasts a vector of $P-1$ splitter probes to all other processors.
2) Each processor performs a k-way partitioning based on these splitters to obtain a local histogram.
3) All local histograms are reduced to a global histogram on the central processor.
4) The central processor then determines if the cumulative partition sizes satisfy the partitioning conditions. If this is not the case for all splitters, we update the vector of splitter probes and broadcast it again to all other processors before continuing with step 2.
5) Each processor exchanges a portion of the local data with all other processors in such a way that keys in range i are sent to processor i. It is semantically a single round of ALL-TO-ALL communication.
6) Each processors sorts the received chunks which results in a globally sorted sequence.

The iterative process to determine the splitter keys is the most critical step for achieving good performance. We can optimize this using the core idea of sample sort. Instead of histogramming on the whole set of keys we operate on a sample of keys to pick initial guesses and continue with histogramming on this sample [1], [16]. Similar to sample sort we need a sufficiently large number of samples for fast convergence to the final splitters. Axtmann *et al.* show with AMS-sort that a sample of $O(p(\log p + \frac{1}{\epsilon}))$ is required to achieve *locally balanced* partitioning [16] with only one round of histogramming. Extending this to multiple rounds of histogramming achieves a *global balance* with an even smaller sample size. Specifically, a constant number of samples per processor $O(p)$ over $O(\log(\log p/\epsilon))$ suffices [1].

The general idea of histogramming is also a major component in this work. In contrast to the earlier described approaches we do not apply any sampling during the histogramming phase but focus on optimizing the initial splitter guesses. The overhead of sampling each round does not pay off in performance with our case studies.

C. High Performance Sorting Algorithms

Sorting algorithms which have been shown to be scalable to a large number of MPI ranks employ topology-aware optimizations or are even based on different concepts than sample or partition sort.

Sorting networks like Batcher's Bitonic Sort [17] are easy to implement and often used in distributed memory applications. The basic approach is to sort bitonic sequences with an expected theoretical complexity of $O(\log^2 n)$ for $N/P = 1$. An adapted version generalizes it to $O(N \log N)$ for $N/P > 1$ [18]. However, the scalability cannot keep up with sample sort if $N/P \gg 1$ because it transfers data $\log P$ times [10].

The prominent *Quicksort* has been adapted for hypercube topologies by Wagar *et al.* [19]. Each processor partitions the local data portion around a randomly chosen pivot. We subsequently split the processor cube into two subcubes, one for the lower half and the other for the upper half. Both subcubes exchange data according to the pivot and merge received chunks into the local data portion. This procedure continues until we reach a recursion depth of $log(P)$, resulting in a globally sorted sequence.

Hyksort [20] adapts this idea and combines it with histogramming techniques. Instead of looking for one pivot, it partitions the data into k pivots and splits the group of all processors into $k+1$ groups, respectively. It is demonstrably one of the fastest accessible sorting algorithms in distributed memory.

The earlier mentioned AMS-sort [16] additionally employs the technique of overpartitioning to obtain a better pivot selection. Similar to Hyksort, it divides the group of processors into $\sqrt{P}$ subgroups in order to load balance the data in successive steps. The evaluation demonstrated strong scalability up to 2^{15} processors.

In contrast to these papers our algorithm does not load balance any data during splitter determination. Although there are valid arguments for moving data in successive steps this comes along with a communicator split each iteration in the recursion tree. In MPI this operation takes linear complexity to the communicator size and is a blocking collective operation among all processors. In consequence this harms performance in particular if N/P is small.

IV. Distributed Selection

We continue with a discussion of the *selection* algorithm which is a fundamental building block in our sorting approach. After explaining the sequential and parallel *1-way* selection we generalize it to *k-way* distributed selection.

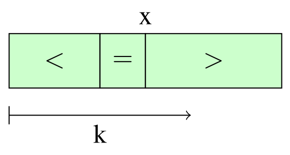


Fig. 1: Position k relative to the partition sizes.

A. Overview

A naive way to obtain the k-th order statistic in a set is first to sort the set and return the element at rank k. However, this results in $O(N \log N)$ complexity. One of the classics in algorithm lectures is *quickselect*, which is in essence a partial *quicksort* and has expected linear time $O(N)$ performance. The underlying technique is effectively *divide-and-conquer*. Let us randomly choose a pivot x. Then, we partition the set around x. If x has rank $I(x) = k$ we are done. Otherwise, we recurse either on the left partition (if $k < I(x)$) or on the right partition (if $k > I(x)$), as illustrated in Fig. 1.

Instead of randomized pivot selection, there exists also a deterministic approach. It has been shown that we can reduce *quickselect*'s worst case complexity from $O(N^2)$ to $O(N)$ using the *median-of-medians* algorithm for smart pivot selection [21]. Although the traditional quickselect is often preferred in practice, it is important to note that we can achieve a linear worst case complexity. Since this algorithm is so fundamental, many efforts have been made to further reduce complexity constants and improve practical performance using a sample of the input sequence for effective median finding [22], [23], [24].

The critical challenge is always to find a *good* pivot which is usually close to the median, as fast as possible. Then we can discard at least half the elements after the first iteration of the selection algorithm. Extending the *median of samples* strategy by assigning weights to each sample leads to the *weighted median* [7].

Definition 2: Given a sequence of elements $x_1, x_2, \ldots, x_n$ with positive and normalized weights $w_1, w_2, \ldots, w_n$ the weighted median satisfies

$$\sum_{x_i < x_k} w_i < \frac{1}{2} \quad \text{and} \quad \sum_{x_i > x_k} w_i \leq \frac{1}{2}$$

Finding the weighted median with quickselect is straightforward. Instead of counting the number of elements in each partition we sum up the weights and recurse on the side that has too much weight.

B. Parallel Selection

Solving the selection problem in parallel is extensively studied in the PRAM model and requires $O(\log n / \log^2 n)$ time using a polynomial number of processors, which is the maximum we can hope for in practice [25], [26], [27]. This model is not applicable to distributed memory machines and we need alternative solutions. Fujiwara *et al.* describe a deterministic algorithm with $O(\min(\log p, \log^2 n))$ communication rounds and $O(n/p)$ local computation per round, subject to the condition that $n/p \geq p^c$ for any constant $c > 0$. The number of communication rounds could be further reduced to $O(\log^2 p)$ using regular sampling [28]. Both algorithms are, however, more of theoretical interest and have not shown to be performance efficient in practice. Some algorithms use a *median of partitions* strategy, which is a modified variant of the *median of medians*, for clever pivot selection [29]. This may lead to uneven (or sparse) partition distributions as the algorithm proceeds iteratively on a smaller subset. Performing some load balancing at the end of each step can mitigate this problem, however, at an additional cost of expensive data movement.

The weighted median has the same property as the median of medians – we can discard at least one quarter of the input elements – and data redistribution is not required anymore. Since this algorithm is a fundamental piece in this paper we briefly outline an implementation for distributed memory in Algorithm 1 [30].

Each processor determines the median of its local partition and broadcasts it to all other processors. Median m_i is weighted by the partition's size n_i in order to calculate the weighted median M. All processors perform a 3-way partition around M on their local portion, denoting the lower and upper bound. If the k-th order statistic falls into the middle range (i.e. $L \leq k < U$) we return m. Otherwise, we either recurse on the lower ($k < L$) or upper ($k > L - 1$) half.

The subset X' to scan shrinks in successive steps. If the size becomes too small the communication overhead is larger compared to the remaining compute overhead and we can switch to a single processor calculating the k-th element sequentially.

We analyze this algorithm in terms of computational and communication complexity. By definition of the median calculated in line 4 each partition is bisected to half the number of elements. This implies the *weighted median* reduces the total size of working set X by at least one quarter (i.e. 3-to-1 split) each iteration. It follows that the depth of recursive calls is at most $\log_{4/3} P$ which we simplify to $\log P$. The communication complexity for a single ALLREDUCE is $O(\log P)$ since we send and receive only one element per process. Hence, the overall communication complexity is $O(\log^2 P)$.

Considering the computation complexity both the SELECT and PARTITIONING operations take at most $O(N/P)$ steps. Taking the communication rounds into account gives an overall complexity of $O(\log P(N/P))$ which confirms the expected theoretical bound of $O(D \log N)$, where D is the network's diameter [31]. Current state of the art research applies two additional optimizations on this algorithm. Pivot selection is more efficient using sampling [24]. Another modification is to split the group of p processes in phase i into subgroups of size $O(\sqrt{P^{(i)}})$ resulting in even better complexity [32].

V. Distributed Histogram Sort

Given a set X of N keys distributed among P processors. Each processor contributes a local capacity $n_i \sim N/P$. Our distributed sort algorithms works in four supersteps.

Algorithm 1 DSELECT (X, N, P, k)

```
Input: $X$ is an input set of size $N$ distributed among $P$ processors.
1: // Each processor goes through the following steps
2: $p_i \leftarrow$ local partition in processor $i$
3: $n_i \leftarrow$ length of partition $p_i$
4: $m_i \leftarrow$ SELECT$(p_i, n_i/2)$ // median in partition $p_i$
5: $M \leftarrow$ ALLGATHER $m_{1..P}$
6: $M \leftarrow$ normalize medians $m_i \in M$ by $w_i = n_i/N$
7: $m \leftarrow$ SELECT_WEIGHTED$(M)$ // find the weighted median
8: $l_i, u_i \leftarrow$ PARTITION$(p_i, m)$ // 3-way partition around pivot $m$
9: $L, U \leftarrow$ ALLREDUCE$(l_{1..P}, u_{1..P}, +)$
10: if $L \leq k \wedge k < U$ then
11:     return $m$
12: else if $k < L$ then
13:     $N' \leftarrow$ ALLREDUCE$(l_{1..P}, +)$
14:     DSELECT($p_i[0..l_i]$, $N'$, $P$, $k$)
15: else
16:     $N' \leftarrow$ ALLREDUCE$(n_{1..P} - u_{1..P}, +)$
17:     $k' \leftarrow k - U + 1$
18:     DSELECT($p_i[u_i..n_i]$, $N'$, $P$, $k'$)
19: end if
```

Local Sort All processors sort the local portion using a fast shared memory algorithm with an expected $O(n_i \log n_i)$ time complexity.

Splitting Partition the local array into P subsequences. We generalize distributed selection to a distributed multiselection algorithm.

Data exchange Each processor exchanges locally sorted subsequences $i \in \{1..P\}$ with peer processor i.

Local Merge Each processor locally merges the received sorted sequences.

In the following we discuss all phases, except the initial local sort which is not of particular interest in this paper.

A. Splitter Determination

We generalize distributed selection (Algorithm 1) to determine the splitters. Instead of finding one pivot we collect multiple pivots (ranks) in a single iteration, one for each *active* range. If a pivot matches a specific rank we do not consider this range anymore and discard it from the set of active ranges. Otherwise, we examine each of the two resulting subranges whether they need to be considered in future iterations and add them to the set of active ranges. The overall procedure is the following.

Definition 3: We define a vector K of size $P+1$ as a prefix sum series over all local capacities

$$K = \sum_{i=1}^{P} n_i.$$

It is implied that $K(0) = 0$ and $K(P) = N$ to define the outer limits.

We need to find a vector of splitters S with ranks K in X to satisfy the partitioning conditions in Definition 1 using the concepts of histogramming and selection. We suppose *perfect partitioning* requirements ($\epsilon = 0$) in the remainder of this section.

Each splitter is represented as a tuple (S_{il}, S_i, S_{iu}), where S_{il} (S_{iu}) denote the lower (upper) bound of splitter S_i. Initially, S_{il} (S_{iu}) equal the minimum (maximum) key in set X. The goal is then to narrow the range $[S_{il}, S_{iu}]$ until S_i satisfies the partitioning condition.

Because partitions $P_{1..p}$ are *locally* sorted we can easily determine the lower and upper bounds (i.e., l_i and u_i) based on binary search in P_i to obtain a local histogram in each iteration. Summing the local histograms over all ranks gives the global histograms defined as the pair (L_i, U_i) of splitter S_i.

Definition 4: A single splitter S_i is successfully determined if L_i and U_i satisfy the following condition.

$$L_i(S_i) < K_{i+1} \wedge K_{i+1} \leq U_i(S_i), \text{where } i \in \{1, 2, \ldots, P\}.$$

If the condition cannot be satisfied we move the splitter S_i either towards S_{il} or S_{iu} as shown in algorithm 2.

Algorithm 2 VALIDATE_SPLITTER(S_i, K, L_i, U_i)

```
Input: $S$ is the set of all splitters, $K$ the set of ranks
1: if $L_i < K_{i+1}$ and $K_{i+1} \leq U_i$ then
2:     return true
3: else if $L_i \geq K_{i+1}$ then
4:     $S_{iu} \leftarrow S_i$
5: else
6:     $S_{il} \leftarrow S_i$
7: end if
```

Algorithm 3 outlines our histogramming algorithm. We first initialize all splitters with the minimum and maximum key over the global key range. Gathering the tuple (min, max) can be implemented as one reduction with time complexity $O(\log P)$. Then, we iteratively determine the splitters. We first collect a *local histogram* through the lower and upper bounds using binary search. Obtaining the global histogram is straightforward with a single ALLREDUCE. Based on the global histogram each process validates the splitters.

A relevant question is the number of iterations until we converge to the final splitters. Two parameters are of particular interest:

1) Key size (i.e., bits per key), and
2) Uniqueness of the keys.

With normally and uniformly distributed keys the number of iterations is bound by the key size. As an example if we sort 64-bit floating point numbers the overall median lies in the range 60–64 iterations. Sorting 32-bit floats can be accomplished in 25–35 iterations. The number of processors does not impact the number of iterations. This behavior is not unsurprising. In each iteration we bisect the key range of possible splitter candidates, i.e. a single bit.

This does not apply well for skewed data distributions or if we are sorting many equal keys. The reason is that narrowing the splitters by bisection does not necessarily reflect a change in the histograms over the key set. We transform all equal keys into globally unique keys to improve the algorithm's stability. A technically straightforward approach is the following: Each key x is defined as a triple (x, y, z) where y and z denote the *processor id* and the local index in the input sequence [16]. However, it comes at a cost to communicate additional metadata during histogramming which we have in common with other sorting algorithms.

Algorithm 3 FIND_SPLITTERS(X, K, P)

```
Input: $X$ is an input set of size $N$ distributed among $P$ processors.
 1: // Each processor goes through the following steps
 2: $p_i \leftarrow$ local partition in processor $i$
 3: $(min, max) \leftarrow$ ALLREDUCE$(p_{i..P})$
 4: $(S_{il}, S_i, S_{iu}) \leftarrow (min, 0, max)$
 5: repeat
 6:     $S_i \leftarrow (S_{il} + S_{iu})/2$
 7:     $(l_i, u_i) \leftarrow$ BINARY_SEARCH$(p_i, S_i)$ // local histogram
 8:     $(L_i, U_i) \leftarrow$ ALLREDUCE$(l_i, u_i, +)$ // global histogram
 9:     VALIDATE_SPLITTER$(S_i, K, L_i, U_i)$
10: until all splitters found
```

For completeness we again analyze the overall complexity which is given by expanding all steps in Algorithm 1. We know from prior analysis that the number of iterations is bound by either $O(\log p)$ or the key size. Finding the local median requires $O(\log n_i)$ because the local portion is already sorted. We assume a constant $O(1)$ complexity for the normalization. Calculating the weighted median has an expected linear complexity of $O(p)$. In the last step we need to binary search over p target ranks. Multiplying all terms results in

$$O(p(\log n_i) + p + p(\log p)) = O(p(\log p) + p(\log n_i))$$

computational complexity per iteration. We additionally need $O(p)$ space on each process for communicating the reduction operations. For the communication overhead we have two broadcasts, each processor sends $O(p)$ data to all others.

We want to highlight that our algorithm does not rely on any restrictions about the key distribution and number of processors to efficiently determine the splitters. Moreover our implementation handles any partitioning of input keys, for example sparse vectors (matrices).

B. Data Exchange

After determining all splitters each processor prepares for the final MPI ALL-TO-ALLV data exchange. Recall that the local range is split into P segments. We have to compute a global permutation matrix L of size $P \times P$ to determine the final partition distribution.

Collecting this information first requires the lower and upper bounds (l_i, u_i) of splitter S_i. We exchange these in a single ALL-TO-ALL communication. As a result, processor i knows l_i and u_i of all other processors which have to deliver data to processor i. In *perfect partitioning* or *in-place* scenarios we still may need refinement around the borders. It is guaranteed that up to the lower bound l_i we never exceed the local capacity n_i. There are exactly $K_i - l_i$ excess elements for processor i which have to be filled up. We assign from the remaining contingent $u_i - K_i$ among all processors until the number of elements matches capacity n_i. Algorithm 4 lists the relevant steps.

Algorithm 4 PERMUTATION_MATRIX(L, U, K, n_i, P)

```
Input: $L$ and $U$ result from the ALL-TO-ALL histogram exchange.
1: $s_i \leftarrow$ ACCUMULATE$(l_i, +)$ for $i = 1..P$
2: for all $i$ such that $0 \le i \le P$ and $s_i \ne n_i$ do
3:     $e_i \leftarrow K_i - l_i$ // excess elements on unit $P_i$
4:     $s_i \leftarrow s_i + min(e_i, n_i - s_i)$
5: end for
```

Following this approach allows us to compute permutation matrix L in parallel where the i-th processor is responsible for the i-th row. The eventually refined permutation matrix L needs to be exchanged in another ALL-TO-ALL communication to obtain send counts of processor i to all other processors. Summing up each row locally as a *prefix sum* provides the send displacements.

For the receiving side we need to follow a similar approach. An exclusive scan with operation minus on L gives the receive count. Summing each resulting row locally in another prefix sum provides the receive displacements.

In total, the communication overhead sums to two ALL-TO-ALL collectives with $O(p^2)$ elements to be exchanged, and a single EXCLUSIVE_SCAN. Both collective operations result in a complexity of $O(p \log p)$ to exchange necessary displacements.

Finally, each processor performs a ALL-TO-ALLV data exchange with all communication peers and subsequently merges received subsequences.

C. Local Merge

After receiving all chunks from the other processors, we have to finally merge them into a single sorted sequence. There are two approaches to accomplish this. Either we again sort the full array with an expected time complexity of $O(\frac{N}{P} \log \frac{N}{P})$ using a fast shared memory sort, or we merge it out-of-place in $O(\frac{N}{P} \log P)$ using a binary merge algorithm. All pairwise merges can be performed in parallel. Another technique is a tournament tree [8] where we construct a min (max) heap based on the P chunks. In each round we select the smallest element out of P chunks and push it into a heap of size P. In a tournament tree there is a $O(\log P)$ cost to insert a single element into the heap. If P is small enough tournament trees are even cache efficient since it incurs only $O(N/B)$ cache misses, where B is the length of a single cache line. In contrast, a binary merge requires each element to be merged $O(\log P)$ times. However, we can start merging as soon as

TABLE I: SuperMUC Single Node Specifications.

CPU	2 x E5-2697v3
Memory	64GB (56GB usable)
Network	Infiniband FDR14
Compiler	ICC 18.0.2
MPI library	Intel MPI 2018.2

two chunks are transmitted while a tournament tree requires all chunks in advance.

Similar to sorting there is a excellent prior work, both from a theoretical and practical perspective. However, achieving good performance by overlapping merging (computation) and communication is another challenge. Section VI-E discusses several reasons on this. In this paper and the evaluated implementation we rely on another shared memory sort to "merge" all sequences.

VI. Evaluation and Analysis

We present both weak and strong scaling studies to demonstrate the algorithmic efficiency on distributed and shared memory, respectively. On distributed memory we compare our C++ PGAS implementation against a Charm++ sorting algorithm which is described in a recently published preprint [1]. Our intent was to integrate Hyksort [20] as well. The publicly accessible code[1] first failed to compile. After fixing this we got a runtime error as soon as the globally allocated memory over all MPI ranks exceeded 4GB in total which is obviously too small for our experiments. We are communicating this to the authors.

In all benchmarks we perform *perfect partitioning* ($\epsilon = 0$), serving as a good basis for analyzing the scaling behavior of our sorting algorithm. In shared memory we compete against Intel's Parallel STL implementation which is included in the `icc` compiler suite since major release 2018.

A. Implementation Details

We conducted the experiments on SuperMUC Phase 2 hosted at the Leibnitz Supercomputing Center (LRZ). To better understand the benchmark results we briefly describe our own implementation and architectural details.

1) DASH: Our implementation is integrated into DASH which is a C++14 template library based on the partitioned global address space model (PGAS)[2]. PGAS is a distributed memory abstraction distinguishing between *local* and *remote* partitions in the global address space. Using a one-sided communication interface we can always operate on global data. However, following the *owner computes* model to operate on local data is crucial to achieve performance scalability.

Inspired by the C++ STL concepts we provide containers and algorithms to operate on global data [33]. The main objective is to facilitate the trade-off between performance efficiency and productivity with a zero cost abstraction. While our runtime conceptually supports multiple communication substrates we only use MPI-3 RMA capable libraries in this paper. DASH is lightweight and complements well with other HPC libraries in scientific codes.

A major benefit of PGAS is that we can use fast shared memory semantics if processors communicate intra-node. More specifically, we replace collective communication by fast `memcpy` operations which gives us significant performance benefits as demonstrated in Section VI-D.

2) LRZ Supercomputing Centre: SuperMUC Phase 2 is an island-based computing cluster, each equipped with 512 nodes. However, we were not able to reserve more than one island[3]. Each node has two Intel Xeon E5-2697v3 14-core processors with a nominal frequency of 2.6GHZ and 64GB of memory, although only 56GB are usable due to the operating system. Computation nodes are interconnected in a non-blocking fat tree with Infiniband FDR14 which achieves a peak bisection bandwidth of 5.1 TB/s. Table I summarizes relevant specifications.

We compiled all codes using `icc` 2018 Update 2. As our communication substrate we experimented with two MPI-3 compliant libraries, Intel MPI 2018.2 and IBM POE v1.4. We report only results from Intel MPI as the IBM library does not support MPI-3 shared memory windows which enables us to exploit shared memory semantics in the DASH library.

B. Strong Scaling Analysis

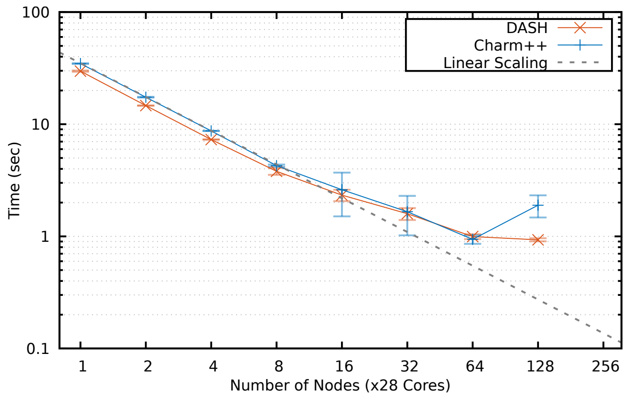
We first compare our sorting algorithm against a Charm++ implementation and discuss subsequently the scaling behavior. For Charm++ we compiled the most recent stable release[4] with MPI-3 and OpenMP support, the same software stack as in DASH. On each node we scheduled 16 MPI ranks, although we have 28 cores available. Both numbers result from limitations in the Charm++ implementation which requires the problem size and the number of processors to be a power of two. Although the Charm++ implementation is prototypical, as noted in the paper [1], we want to emphasize that our implementation does not depend on such constraints. For local sorting both algorithms use a single-threaded C++ STL `sort`.

In our benchmark we generate 64-bit unsigned integers, uniformly distributed in the range $[0, 10^9]$ using a Mersenne Twister engine from the C++ STL library. We experimented on a normal distribution as well, however, the Charm++ implementation failed to find the final splitters, i.e., it could not terminate before the job's wall clock limit (fixed to 30 minutes).

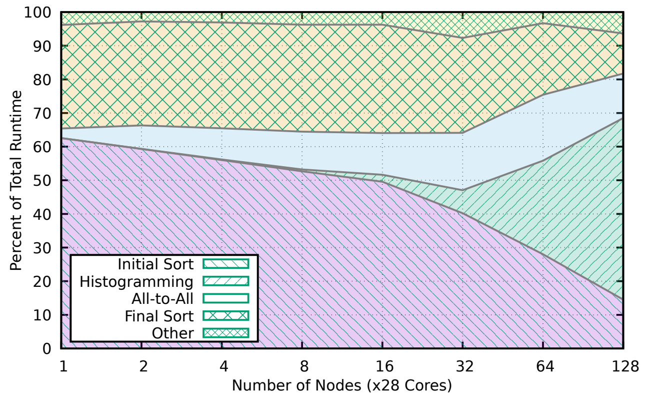
The strong scaling performance results are depicted in Figure 2(a). We always report the median time out of 10 executions along with the 95% confidence interval, excluding an initial warmup run. In Charm++ we can see wider confidence intervals. We attribute this to a volatile histogramming phase which we can see after analyzing generated log files in the Charm++ experiments.

[1]https://github.com/hsundar/usort
[2]https://github.com/dash-project/dash/

[3]Due to maintenance issues.
[4]v6.9.0, http://charmplusplus.org/download/



(a) Median execution time



(b) Strong scaling behavior of `dash::sort`

Fig. 2: Strong scaling study with Charm++ and DASH.

Overall, we observe that both implementations achieve nearly linear speedup with a low number of cores. Starting from 32–64 nodes ($\sim$ 1800 cores) scalability gets worse. DASH still achieves a scaling efficiency of $\approx$ 0.6 on 3500 cores while Charm++ is slightly below.

We further analyze the scaling behavior in DASH. It is clear that the local work load is proportional to the number of processors. With fewer processors the initial and final sort consume most of the time. The communication overhead during histogramming, which takes $\sim$ 30 iterations to find the final splitters, can be almost neglected. This changes if we scale to a higher number of ranks ($>$ 2000). Recall that each iteration requires a collective ALLREDUCE among all processors. Figure 2(b) visualizes the relative fraction of the most relevant algorithm phases in a single run. It clearly identifies histogramming as the bottleneck if we scale up the number of processors. This is not surprising because with 128 nodes (2048 ranks) each rank operates on only 8MB of memory. If we had had the possibility to allocate more memory per node the saturation point would have shifted to a higher processor count. We can further see that the ALL-TO-ALL data exchange overhead is relatively stable which stems from the fact that the communication volume per processor decreases while putting more processors on the workload. The constant overhead to prepare the data exchange, denoted as "Other", can be also neglected.

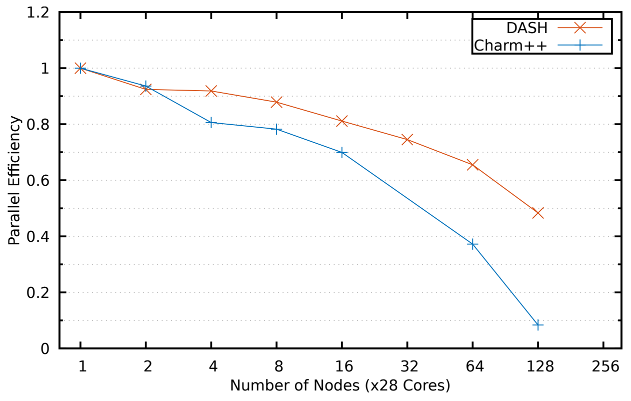
We certainly get a better scaling if we soften the perfect partitioning requirement as the number of histogramming iterations decreases. Nevertheless there is room for improvement. Based on related work and as discussed in Section V we can expect better performance if we split the group of processors into smaller subgroups. Furthermore, if we apply histogramming only on a sampled fraction of the initial input data we gain additional benefits. We evaluate these approaches in further research.

C. Weak Scaling Analysis

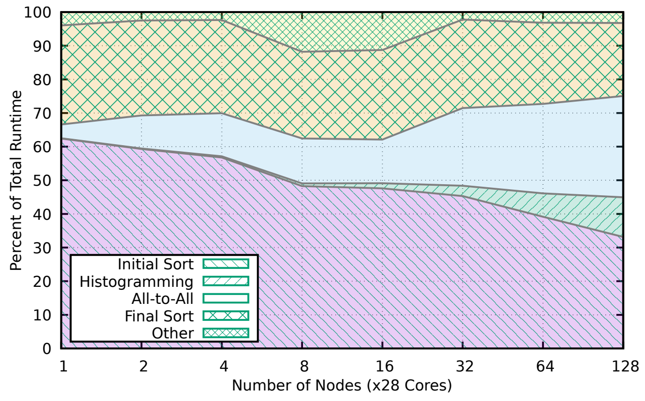
For weak scaling we used the same setup as in strong scaling with a uniform data distribution of 64-bit unsigned integers. However, this time we allocated 2GB of memory per node (128 MB per rank). This is a relatively small memory footprint but we want to prevent the local sort to be the bottleneck in order to understand the scaling efficiency. Let us first discuss what we can expect without looking at the plot. Due to the constant number of histogramming iterations the incidental communication overhead grows proportional to the number of processors. More specifically, we have to communicate more splitters among more ranks. However, in contrast to strong scaling, the communication volume for the ALL-TO-ALL exchange grows exponentially and clearly dominates the algorithmic complexity.

Figure 3(a) depicts the weak scaling efficiency. The absolute median execution time for DASH started from $2.3s$ on one node and ended with $4.6s$ if we scale to 128 nodes (3584 cores). As expected, the largest fraction of time is consumed in local sorting and the ALL-TO-ALL data exchange because we have to communicate $\approx$ $256GB$ across the network. Figure 3(b) confirms this. The collective ALLREDUCE of $P-1$ splitters among all processors in histogramming overhead is almost amortized from the data exchange which gives an overall good scalability for DASH. On the other hand, Charm++ cannot keep up with our implementation. Their histogramming algorithm again shows high volatility with running times from $5-25s$, resulting in drastic performance degradation. We cannot see any obvious reason for this and suspect improper sampling in each histogramming round.

Although DASH achieves satisfactory performance we have to address the scalability of the ALL-TO-ALL data communication. One approach is to integrate ALL-TO-ALL and a k-way binary tree merge. Adapting a 1-factor algorithm [34] to merge received chunks in each communication round improves the possible communication-computation overlap as each merge "gives" more time to complete a pending data transfer. Another reason for the bad scalability may be that MPI ALL-TO-ALL communication is more optimized for small messages and not for huge chunks as in this case. Additionally we consider cache-oblivious communication algorithms for intra-



(a) Weak scaling efficiency



(b) Weak scaling behavior of `dash::sort`

Fig. 3: Weak scaling study with Charm++ and DASH.

node communication to reduce the communication time.

D. Shared Memory Benchmarks

Due to the increasing complexity with heterogeneous shared memory architectures PGAS semantics are a natural fit to model NUMA locality. With this motivation in mind we tried not only to be efficient on distributed memory machines but on shared memory as well. While there are many highly optimized sorting implementations, the most frequently used multi-threaded general purpose sorting algorithm is merge sort. Intel provides an efficient tasking implementation based on Intel Thread Building Blocks (TBB) and recently released it to LLVM and GNU for use in C++17 parallel algorithms. We conducted an experiment to study NUMA effects with sorting in shared memory. Since we move data only once being able to keep up with a fast shared-memory implementation is not too ambitious. We compare our sorting algorithm against Intel's Parallel STL implementation. For reference, we include an OpenMP task-based merge sort, provided from Intel, as well.

A single Node on SuperMUC Phase 2 has 4 NUMA domains, each attached with 7 cores (14 hardware threads). For this case study we allocated 64-bit double precisions with $5GB$ total memory. We generated normally distributed numbers in the interval $[-1E6, 1E6]$ using a mean of 0 and standard deviation 1. The benchmark measures strong scaling from 7 to 28 cores ($1-4$ NUMA domains). More specifically, we start by allocating data on 1 NUMA domain and occupy only 7 attached cores. Then we evenly partition data across two NUMA domains and put 14 cores on it for sorting, etc. DASH runs MPI ranks instead of threads and we use `numactl` to guarantee a correct pinning in all measurements. Figure 4 plots the performance results. Note that we utilized hyperthreading (2 threads per core) because both TBB and OpenMP achieved better performance. While we see a performance penalty compared to TBB if we occupy only one NUMA domain we surpass TBB if data needs to be communicated across NUMA boundaries. Again, even the histogramming overhead is amortized through data movement. A surprising insight is that we gain a benefit from hyperthreading with a heavy MPI stack under the hood. Intel TBB merge sort is certainly not an aggressively optimized implementation, however, it does confirm the hypothesis that we can compete with our implementation on shared memory as well.

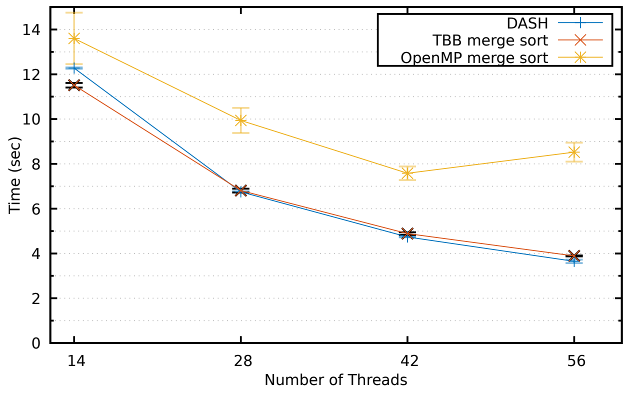


Fig. 4: Shared Memory Benchmarks

E. Discussion

Although the main contribution of this paper is an efficient multi selection algorithm to find the splitters, the evaluation results reveal two primary bottlenecks.

First, the ALL-TO-ALLV data exchange consumes a high fraction of the overall running time, in particular during the weak scaling studies if both the data volume and processor count increases. This is not surprising and recommends to perform some load balancing during histogramming. However, decoupling data movement from splitting does not only simplify the overall implementation. We immediately benefit from future improvements in MPI libraries and we can reuse our distributed selection implementation as a building block in other DASH algorithms, e.g. `dash::nth_element`.

Second, the final merge is still another shared memory sort which is not optimal if we reason with the algorithmic complexities in mind. We discuss both issues and outline possible improvements which we are studying for a more detail paper.

1) All-to-Allv Communication: We can tune our own *all-to-all* algorithm. Constructing send-receive pairs using explicit point-to-point communication is straightforward. Upon receiving at least two chunks we can asynchronously start a merging task and overlap it with the next communication round. Depending on the message length we can utilize specific algorithms to achieve the best performance. For a relatively small N/P we utilize *store-and-forward* algorithms which communicate data in intermediate steps in $\lceil \log(p) \rceil$ rounds. For larger messages we schedule flat handshakes or 1-factorization algorithms [34], [35] to trade off latency and bandwidth bottlenecks. In a PGAS abstraction we have further potential for optimizations especially in shared memory and if two nodes are closely connected to each other. If a pair of processors resides on the same node we do not need to initiate any MPI calls but use fast `memcpy` semantics. A good implementation takes cache efficiency into account to minimize false sharing. For inter-node communication we borrow techniques from studies about hierarchical collectives and utilize our PGAS model to fully overlap communication and merging. A set of dedicated leader cores on a single node is responsibly for communication while the others perform the merging process. Synchronization is performed through a node-local producer-consumer queue which is accessible for all cores. Each pairwise exchange enqueues a communication request and whenever a data chunk is available it enqueues a merge request. This has two advantages. First, it minimizes network congestion as only a set of processors move data across the node boundary. Second, depending on the work load processors can change their role.

2) Parallel k-way Merging: Our final merge on the received subsequences from the data exchange was initially based on a parallel binary merge tree as described in Section V-C. However, we faced unexpected low performance in the final merge step with large data volumes. For this reason we conducted separate experiments to understand the performance in more detail. We assume an almost perfectly load-balanced scenario where all chunks have the same size (i.e. $O(\frac{N}{P^2})$) and values are drawn from a uniform distribution. We implemented our own k-way binary merge using OpenMP tasks and *GNU Parallel* provides a multi-threaded k-way merge routine using tournament trees. The baseline is a parallel merge sort from the Intel Parallel STL which is a task based merge sort implemented with TBB tasking. We executed the benchmarks on a single node of SuperMUC Phase 2. The problem size is the same as in our sorting experiments with 16GB 32-bit integer keys while varying the number of threads and the number of chunks. Our experiments show similar issues among all merging algorithms. Scheduling only two threads achieves a notable speedup with fewer large chunks. However, the trend changes if we merge many but relatively small chunks. Scheduling many threads for merging many small chunks causes drastic performance degradation due to a high fraction of cache misses. Processing many merge tasks in parallel with another parallel sort clearly outperforms merging. We leave this problem as an open question for future work and may consider a cache oblivious merge algorithm [36].

VII. Conclusion

We have discussed building blocks to engineer a performance efficient sorting algorithm based on histogramming and selection. Our implementation shows good scalability on parallel machines with a large processor count. Compared to other algorithms we do not pose any assumptions on the number of ranks, the globally allocated memory volume or the key distribution. Performance measurements reveal that we can keep up with recently published high performance sorting algorithms. We get the best result if P and N are not too much out of kilter which agrees with measurements in related work. Furthermore we put efforts to keep the implementation as efficient as possible which is why we provide notable performance on shared memory architectures as well. We are aware of edge cases where our implementation does not provide the best performance, for example if N/P is very small. However if we keep the general purpose motivation in mind our algorithm is on par with current state-of-the-art. The algorithm's interface is in accordance with C++ `std::sort` and can be easily integrated into a scientific code base which requires efficient sorting. Additionally, we can handle sparse data structures where a fraction of all processors do not contribute local elements. This is useful for example in numerical algorithms to load balance sparse matrices [2].

In future work we have to scale to more processing entities, which we could not do for this paper due to maintenance issues at the LRZ supercomputing center. We further address the overhead of histogramming in strong scaling. We see the most potential in efficient sampling mechanisms to reduce the number of histogramming rounds, while reducing the group size of communicating ranks at the same time. Another problem remains the *all-to-all* data exchange. We are studying and optimizing this in the scope of another paper which is already in progress.

References

[1] V. Harsh, L. Kale, and E. Solomonik, "Histogram sort with sampling," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '19. New York, NY, USA: ACM, 2019, pp. 201–212.

[2] V. Shah and J. R. Gilbert, "Sparse matrices in Matlab* P: Design and implementation," in *International Conference on High-Performance Computing*. Springer, 2004, pp. 144–155.

[3] M. S. Warren and J. K. Salmon, "A Parallel Hashed Oct-Tree N-body Algorithm," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 12–21.

[4] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and rasiosity," *Journal of Parallel and Distributed Computing*, vol. 27, no. 2, pp. 118–141, 1995.

[5] D. Molka, D. Hackenberg, and R. Schöne, "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014.

[6] E. Alliance, "2018 ethernet roadmap," `http://bit.ly/EAEthernetRoadmap18` (Jan 2018).

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, ser. The MIT Press. MIT Press, 2009.

[8] D. E. Knuth, *The Art of Computer Programming*, 3rd ed., ser. Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., 1998, vol. 1, (book).

[9] W. D. Frazer and A. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM)*, vol. 17, no. 3, pp. 496–507, 1970.

[10] 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 *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*. ACM, 1991, pp. 3–16.

[11] S. Seshadri and J. F. Naughton, "Sampling issues in parallel database systems," in *International Conference on Extending Database Technology*. Springer, 1992, pp. 328–343.

[12] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.

[13] D. R. Helman, J. JáJá, and D. A. Bader, "A new deterministic parallel sorting algorithm with an experimental evaluation," *Journal of Experimental Algorithmics (JEA)*, vol. 3, p. 4, 1998.

[14] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," in *Parallel and distributed information systems, 1991., proceedings of the first international conference on*. IEEE, 1991, pp. 280–291.

[15] E. Solomonik and L. V. Kale, "Highly scalable parallel sorting," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[16] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, "Practical massively parallel sorting," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 13–23.

[17] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[18] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 216–228, 1989.

[19] B. Wagar, "Hyperquicksort: A fast sorting algorithm for hypercubes," *Hypercube Multiprocessors*, vol. 1987, pp. 292–299, 1987.

[20] H. Sundar, D. Malhotra, and G. Biros, "Hyksort: a new variant of hypercube quicksort on distributed memory architectures," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 293–302.

[21] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.

[22] R. W. Floyd and R. L. Rivest, "Expected Time Bounds for Selection," *Commun. ACM*, vol. 18, no. 3, pp. 165–172, Mar. 1975.

[23] A. Rauh and G. R. Arce, "Optimal Pivot Selection in Fast Weighted Median Search," *IEEE Transactions on Signal Processing*, vol. 60, no. 8, pp. 4108–4117, Aug 2012.

[24] C. Martínez and S. Roura, "Optimal sampling strategies in quicksort and quickselect," *SIAM Journal on Computing*, vol. 31, no. 3, pp. 683–705, 2001.

[25] P. F. Dietz and R. Ramant, "Very fast optimal parallel algorithms for heap construction," in *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*. IEEE, 1994, pp. 514–521.

[26] S. Chaudhuri, T. Hagerup, and R. Raman, "Approximate and exact deterministic parallel selection," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1993, pp. 352–361.

[27] R. J. Cole, "An optimally efficient selection algorithm," *Information Processing Letters*, vol. 26, no. 6, pp. 295–299, 1988.

[28] A. Tiskin, "Parallel selection by regular sampling," in *European Conference on Parallel Processing*. Springer, 2010, pp. 393–399.

[29] I. Al-Furiah, S. Aluru, S. Goil, and S. Ranka, "Practical algorithms for selection on coarse-grained parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 813–824, 1997.

[30] E. Saukas and S. Song, "A note on parallel selection on coarse-grained multicomputers," *Algorithmica*, vol. 24, no. 3-4, pp. 371–380, 1999.

[31] L. Shrira, N. Francez, and M. Rodeh, "Distributed k-selection: From a sequential to a distributed algorithm," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983, pp. 143–153.

[32] F. Kuhn, T. Locher, and R. Wattenhofer, "Tight bounds for distributed selection," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2007, pp. 145–153.

[33] K. Fuerlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms," in *2016 IEEE 18th International Conference on High Performance Computing and Communications*, Dec 2016.

[34] P. Sanders and J. L. Träff, "The hierarchical factor algorithm for all-to-all communication," in *Euro-Par 2002 Parallel Processing*, B. Monien and R. Feldmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 799–803.

[35] D. S. Scott, "Efficient all-to-all communication patterns in hypercube and mesh topologies," in *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, April 1991, pp. 398–403.

[36] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.