HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures

Hari Sundar
The University of Texas at Austin, Austin, TX 78712
hsundar@gmail.com

Dhairya Malhotra
The University of Texas at Austin, Austin, TX 78712
dhairya.malhotra@gmail.com

George Biros
The University of Texas at Austin, Austin, TX 78712
gbiros@acm.org

ABSTRACT

In this paper, we present HykSort, an optimized comparison sort for distributed memory architectures that attains more than 2× improvement over bitonic sort and samplesort. The algorithm is based on the hypercube quicksort, but instead of a binary recursion, we perform a k-way recursion in which the pivots are selected accurately with an iterative parallel select algorithm. The single-node sort is performed using a vectorized and multithreaded merge sort. The advantages of HykSort are lower communication costs, better load balancing, and avoidance of $\mathcal{O}(p)$-collective communication primitives. We also present a staged-communication samplesort, which is more robust than the original samplesort for large core counts. We conduct an experimental study in which we compare hypercube sort, bitonic sort, the original samplesort, the staged samplesort, and HykSort. We report weak and strong scaling results and study the effect of the grain size. It turns out that no single algorithm performs best and a hybridization strategy is necessary. As a highlight of our study, on our largest experiment on 262,144 AMD cores of the CRAY XK7 "*Titan*" platform at the Oak Ridge National Laboratory we sorted 8 trillion 32-bit integer keys in 37 seconds achieving 0.9TB/s effective throughput.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and Searching*

Keywords

Sorting; Parallel Algorithms; shared-memory parallelism; distributed-memory parallelism; hypercube; quicksort; samplesort; bitonic sort

1. INTRODUCTION

The design and implementation of parallel algorithms for comparison-based sort is a well studied subject. Here we are concerned with the practical issue of scaling comparison sort algorithms to large core counts. Comparison sort is used in many data-intensive scientific computing and data analytics codes (e.g., [10, 17, 18]) and it is critical that we develop efficient sorting algorithms that can handle petabyte-sized datasets efficiently on existing and near-future architectures. To our knowledge, no parallel sort algorithm has been scaled to the core counts and problem sizes we are considering in this article.

Given an array A with N keys and an order (comparison) relation, we will like to sort the elements of A in ascending order. In a distributed memory machine with p tasks[1] every task is assigned an N/p-sized block of A. Upon completion, task i will have the i_{th} block of the sorted array. Designing and implementing an efficient sorting algorithm that can scale to thousands of cores is difficult since it requires irregular data access, communication, and load-balance.

The most popular algorithm used in actual parallel sort implementations is SampleSort [2]: Given p tasks, we reshuffle the elements of A in p buckets so that keys in the i_{th} bucket are smaller or equal than keys in the $(i+1)_{th}$ bucket. Bucket i is assigned to task i and once we have reshuffled A, each task i can sort its keys locally. The challenge is to obtain good load-balancing so that every task i has roughly the same number of keys after reshuffling while minimizing communication costs.

SampleSort reshuffles A by estimating the boundaries for each bucket using $p-1$ keys, which we call the "*splitters*". This can be done for example, by sampling a subset of keys in A, sorting them (say, by gathering these to the task 0 and using a sequential sort) and selecting the splitters from that set. Once these $p-1$ splitters have been selected, a global all-to-all data exchange redistributes the keys of every task to their correct bucket. An additional local sort is invoked to finalize the output array.

SampleSort is well understood. However, its performance is quite sensitive to the selection of splitters, which can result in load imbalance. Most important, the all-to-all key redistribution scales linearly with the number of tasks and can congest the network. As a result SampleSort may scale suboptimally, especially when the communication volume approaches the available hardware limits (Figure 1).

Contributions.

Our algorithm is based on well-known and well-tested algorithmic techniques. SampleSort was proposed in [2]. We

[1]In the MPI context "*a task*" (or process) corresponds to an MPI rank, which may be a core, a socket, or a node.

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

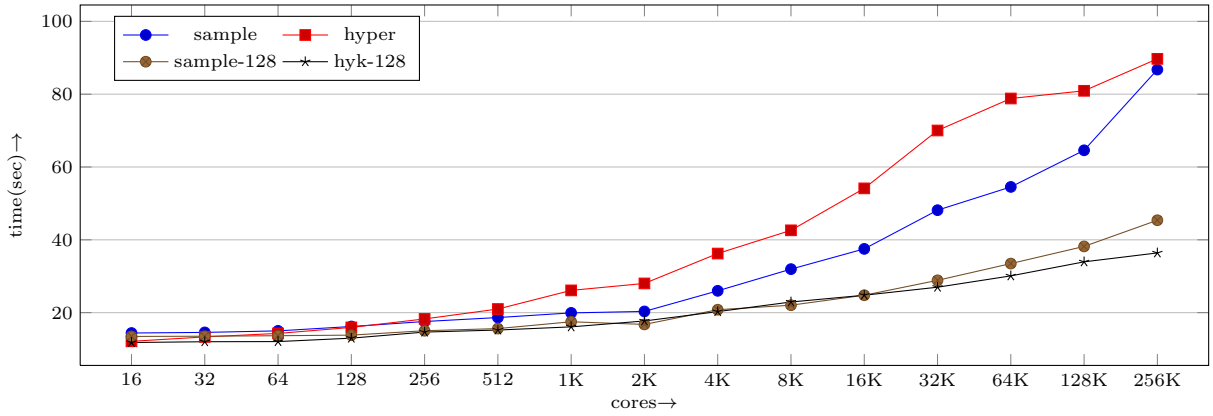

Figure 1: *As a highlight, we present a weak scalability study of our new* HykSort *algorithm against* SampleSort *(combined with bitonic sort to sort the splitters) and* HyperQuickSort *algorithms. We also compare with an improved k-way* SampleSort*. The grain size is 32M (32-bit) integers per core. For the largest run we sort 8.5 trillion integers (34TB) in 37 seconds on 262,144 cores, a throughput of nearly 0.9TB/sec on Titan. All runs were performed using* 2 MPI *tasks per node with* 8 OpenMP *threads per* MPI *task. We obtain* 2× *improvement over the standard* SampleSort *and* HyperQuickSort *algorithms and an 15% improvement over our modified samplesort algorithm.*

estimate our splitters using an approximate parallel selection algorithm first proposed in [16]. Our main contribution is the design of a load-balanced sort algorithm that does not involve any collective communications other than synchronization. In detail:

- **Method:** We present an algorithm that addresses the issues of splitter selection, load balancing, and bucket-reshuffling in samplesort. Our main contribution is the following simple idea: instead of splitting the array A in p buckets, we split it in $k < p$ buckets and then we recurse for every bucket. Indeed, if $k = 2$, our algorithm can be viewed as a distributed memory QuickSort (see Figure 2). At every recursion step, each task communicates only with another k tasks. (For $k = 2$ this resembles the hypercube QuickSort.) This communication is staged and at every stage we use only point-to-point exchanges. Finally, after every recursion step, we load balance so that every task has N/p keys in its working block at all stages of the algorithm.
- **Complexity analysis and implementation:** We analyze the overall scheme and provide complexity estimates. Our implementation combines MPI along with OpenMP. Also we have developed a vectorized multithreaded MergeSort, which we have used for the single socket sorting when we run experiments with one MPI task per socket. Furthermore instead of using a plain all-to-all collective with samplesort, we have implemented our own staged collective communication to avoid congestion.
- **Experimental evaluation:** We conduct an experimental study in which we compare the HykSort with different variants of SampleSort and QuickSort. As a highlight, we present results on up to 262,144 cores on Titan in which we sorted 8.8 trillion (arbitrarily distributed) integers (a 32TB dataset) in less than 37 seconds, achieving near 0.9TB/sec throughput.

Limitations.

Our implementation does not support heterogeneous architectures. Efficient sorting implementations exist and can

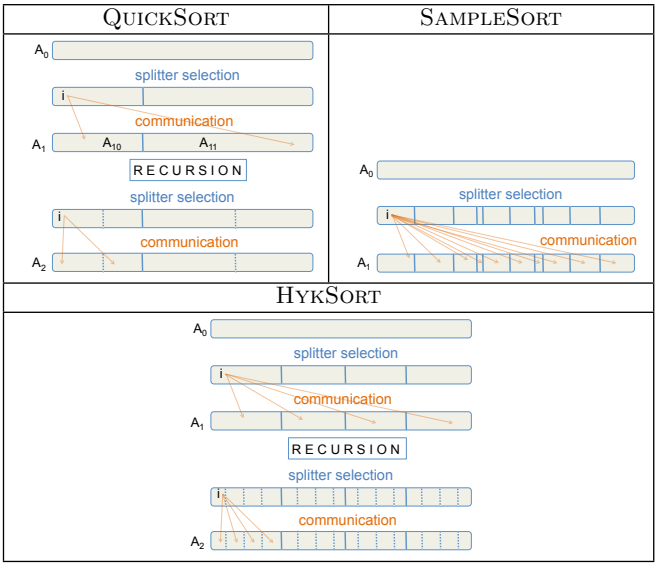

Figure 2: *Here we summarize the basic idea in the proposed method. Given the input array A_0, in* QuickSort *a pivot (splitter) is randomly selected to create two buckets: A_{10} contains all the keys that are strictly less than the pivot and A_{11} the remaining ones. (The first index indicates the recursion level and the second the bucket id.) A hypercube-like communication, here depicted just for MPI task i, can be used to collect the keys from A_0 to the correct bucket (see §3). Then we recurse on each bucket in A_1 to create A_2. The pivot selection and the data exchange can lead to load imbalance between buckets. In* SampleSort*, we use p buckets constructed using $p - 1$ splitters. There is no recursion, but the communication, typically done with an* `MPI_Alltoallv()` *can lead to network congestion (again, we just show communication for task i). In* HykSort*, we use a combination of recursion and multiple splitter selection to control communication costs and load balancing. It is a recursive algorithm in which at every step a small number of splitters is selected—not randomly but as accurately as possible—followed by a staged communication. The new algorithm has provably lower complexity, has predictable behavior, and avoids collective communication primitives.*

be coupled to our code, for example [11, 15]. Although we have tried to optimize our local sort, its performance is subpar to the implementation in [9], which (remarkably for the complexity of sorting) achieves near-memory bandwidth performance. In a nutshell, the efficiency of the local sorts can be significantly improved.

Organization of the paper.

In §2, we discuss the most relevant prior work. In §3, we present the overall methodology. Finally, in §4, we discuss weak and strong scaling results on two different architectures.

2. PRIOR WORK

As we mentioned in the introduction, HykSort is hybrid between a sample sort and a hypercube quick sort. The latter was originally proposed by Wagar [19] and several different algorithmic variants have been proposed since. In our discussion of prior work, we focus our attention to distributed memory algorithms that have been experimentally shown to scale to large core counts and large datasets.

- The SampleSort [2] algorithm was one of the first instances of distributed sort that was adopted extensively in practice[2]. In [3] the authors compare, theoretically and experimentally, different sort algorithms and present results for sorting 132M 64-bit keys on a CM-2 machine. In these experiments SampleSort performs well but the timings do not compare with current architectures.
- Parallel HistogramSort [8, 17] is another variant of SampleSort that estimates the splitters more effectively than the original method. The authors presented one of the largest distributed comparison sort runs (32K cores on BG/P) with 46% efficiency. The algorithm overlaps communication and computation in all stages. During the splitter estimation stage, the iterative estimation of the ranks of the splitters is combined with partial local sorting of the data, by using the splitter candidates as pivots for quicksort. Once the splitters are estimated, the communication of data is staged and overlapped with local merging. In [17] the best throughput was obtained on 16,384 cores of Jaguar XT4 at ORNL for 8M (64-bit) keys per core; the sort took 4.3 seconds achieving a throughput of 0.24TB/sec.
- CloudRAMSort [9] is a recent paper demonstrating good scalability on 256 nodes with shared memory parallelism using `pThreads` and SIMD vectorization. The best results are for sorting 1TB of data (10byte key + 90byte record) in 4.6 secs achieving a throughput of 0.21 TB/sec. They use a variant of the Histogram Sort [8], where the samples are iteratively computed (in parallel) in order to ensure a minimum quality of load-balance across processes. They split the communication of the records into two parts by first communicating the keys followed by the values, during which the keys are merged.
- psort [4] is an efficient implementation of sample sort where the samples are determined using a parallel selection algorithm [16]. They report results for sorting 100 billion (32-bit) integers on 256 processes in under 4 minutes achieving 0.0017 TB/sec throughput.
- TritonSort [13] is the current record holder at the SortBenchmark [7]. It uses a hybrid TCP and `pthreads` implementation of histogram sort using efficient data structures (trie). As with other approaches, they do not sort the local samples initially, relying instead on random samples. The main challenge for scalability is the use of a single coordinator node that computes the global histogram and assigns segments to other nodes. This is the reason that the largest reported run is for 52 nodes. Still this group manages to sort 100TB input data (100-byte tuple, 10-byte key, 10 trillion keys) on 52 nodes achieving a throughput of 0.015TB/sec.
- Although not part of our code, for completeness, we report some representative efforts on accelerators. The performance of various sorting algorithms on GPUs [11] and the Intel Xeon Phi co-processor [15] has also been studied. The key findings from these studies are that radix sort is effective for short keys (less than 32bit) and that a k-way merge ($k = 32, 64, 128$) is more efficient than a 2-way merge on GPUs.

We focused on experimentally verified distributed memory sort algorithms with an emphasis in high performance computing. For more details on the theory of distributed sort see [1, 6].

3. METHODOLOGY

Let us introduce notation for the remaining of the section in Table 1.

A	input array (distributed)
B	output array (ascending order, local block)
comm	MPI communicator
p	number of MPI tasks in *comm*
N	global number of keys in A
$n = N/p$	local number of keys in A
k	number of splitters ($k \neq p$)
s	the splitters
t_w	interconnect slowness (1/bandwidth)
t_s	interconnect latency
t_c	intranode memory slowness (1/ RAM bandwidth)
p_r	the task id (its MPI Rank)
A_r	array to be sorted (local block)
M_s^r	MPI message from s (send) task to r (recv) task
partner	indicates task id in point-to-point exchanges
Functions	
`Length`(B)	number elements in array B
`Rank`(s, B)	`Length`($\{B_i : B_i < s\}$)
`Sort`	local, non-MPI, sort
`BitonicSort`	parallel biotonic sort [6]
log	base-2 logarithm

Table 1: *Here we summarize the notation.* `Sort` *can be any sequential sort (in our experiments, when we have one thread for each MPI task, we use* `std::sort`*). Note the* `Rank`(s, B)*, is equal to the number of keys in B that are strictly smaller than s.*

We start our discussion presenting Wagar's [19] QuickSort on a hypercube, assuming that the input array has been locally sorted using a sequential sort. We select a pivot (or a single splitter) using the median of the root task, and proceed as in the sequential case by using the pivot to split the keys into two buckets and then recursing. If n and

[2]The idea of using splitters, first appeared in [14].

Algorithm 3.1 HypecubeQuickSort

Input: A_r (locally sorted), n, N, $comm$; assume $p = 2^d$
Output: B_r (sorted)
1: **if** $p = 1$, $B_r \leftarrow A_r$, **return**
2: **if** $p_r = 0$, $m_0 \leftarrow$ `Median`(A)
3: `MPI_Bcast`$(m_0, 0)$
4: $A_L \leftarrow \{A_r[i] : A_r[i] < m_0\}$, $A_U \leftarrow \{A_r[i] : A_r[i] \geq m_0\}$
5: $partner \leftarrow (p_r + p/2) \mod p$
6: **if** $p_r < partner$
7: $C \leftarrow$`MPI_Recv()`, `MPI_Send`(A_U), $A_r \leftarrow A_L$
8: **else**
9: $C \leftarrow$`MPI_Recv()`, `MPI_Send`(A_L), $A_r \leftarrow A_U$
10: $A_r \leftarrow$ `Merge`(A_r, C) ▷ $\mathcal{O}(n)$
11: $comm' \leftarrow$ `MPI_Comm_split`$(comm)$ ▷ $\mathcal{O}(\log p)$
12: $n' \leftarrow$ `Length`(A_r)
13: $N' \leftarrow$ `MPI_Allreduce`$(\mathtt{n}', comm')$ ▷ $\mathcal{O}(\log p)$
14: **return** `HypercubeQuickSort`$(A_r, n', N', comm')$

N remain balanced throughout the recursion, the expected complexity is

$$\mathcal{O}\left(t_c \frac{N}{p} \log \frac{N}{p} + t_s \log^2 p + (t_c + t_w)\frac{N}{p} \log p\right),$$

where the first term is the local post-processing sort, the second term is the latency of the collectives due to $\log p$ communication rounds and the last term corresponds to the merge and pairwise data exchange. However, this complexity estimate is highly inaccurate since it doesn't account for two sources of load imbalance. The first one is related to the imbalance between the global sizes of A_L and A_U since they will be unequal if median of A is not chosen as the pivot. Additionally, the pairwise exchanges will lead to imbalance of the number of global keys. As the number of cores increases, these imbalances get amplified. The algorithm therefore, although very simple to state, is not practical. The different approaches summarized in §2 summarize the efforts to address these issues.

One basic approach is to improve the partitioning is by generating not one pivot (or splitter) but $p - 1$ splitters. A further possible improvement is to select the splitters more accurately. These ideas motivate SampleSort, described in Algorithm 3.2.

Algorithm 3.2 SampleSort

Input: A_r (locally sorted), n, N, p
Output: Sorted array B_r ▷ $\mathcal{O}(n \log n)$
1: $s \leftarrow$ `SampleSplitters`(A) ▷ $\mathcal{O}(p\ (\log p)^2)$
2: $d_{i+1} \leftarrow$ `Rank`(s_i, B), $\forall i$ ▷ $\mathcal{O}(p \log n)$
3: $[d_0,\ d_p] \leftarrow [0,\ n]$
4: $M^i_{p_r} \leftarrow \{A_j |\ j \in d_i, ..., d_{i+1} - 1\}$, $\forall i$
5: $B_r \leftarrow$ `Alltoallv`(M_{p_r}) ▷ $\mathcal{O}(n \log p / \log k)$
6: `Sort`(B_r) ▷ $\mathcal{O}(n \log n)$
7: **return**

In *step 1*, `SampleSplitters` is a random sampling approach to choose the splitters and is outlined in Algorithm 3.3. We sample p keys per task that are sorted using a distributed BitonicSort [6]. We select $p - 1$ splitters from the sorted distribution such that every task gets assigned an equal number of keys. In *step 2*, the rank refers to the global rank of the splitter and can be computed using binary search ($\mathcal{O}(p \log n)$) on every task followed by a reduction. In *step 4*, we compute the correct bucket of every key in MPI task i. Here $M^i_{p_r}$ are keys currently owned by task p_r that belong to bucket i and thus, need to be communicated to task i (from task p_r). By using p^2 samples, the expected load-imbalance is no larger than $2n$ [3]. *Step 5* is the all-to-all reshuffling. The standard MPI routine is very expensive (Figure 1). Instead, we're using our own staged all-to-all which we describe in the next section.

Algorithm 3.3 SampleSplitters

Input: A (locally sorted), n, N, p
Output: $p - 1$ splitters s, worst imbalance: factor of 2
1: $r \leftarrow \left\{A_{i\,n/p} |\ i \in 0, ..., p - 1\right\}$
2: `BitonicSort` (r) ▷ $\mathcal{O}(p\ (\log p)^2)$
3: $s \leftarrow$ `All_Gather`(r_{p-1})
4: **return** $\{s_0, s_1, ..., s_{p-2}\}$

3.1 Alltoallv_kway

A major bottleneck in distributed implementations of SampleSort and other similar sorting algorithms is the all-to-all communication using `MPI_Alltoallv`. For all MPI implementations that we have tried, `MPI_Alltoallv` does not scale well beyond 10^4 tasks. In our previous work [10] we used a hypercube scheme to avoid MPI all-to-all communication primitives (that work was in the context of N-body problems, not sorting). In that work we had demonstrated

Algorithm 3.4 SendRecv_kway

Input: $comm$, p (w.l.g., assume $p = mk$) p_r of current task in $comm$, messages $\{M^j_i |\ i \in I_j\}$ destined for task $p_j, \forall j \in comm$.
Output: new communicator $comm'$, $p' = m = p/k$, messages $\{R^j_i |\ i \in \hat{I}_j\}$ destined for task $p'_j \in comm'$ $\forall j$.
1: $R \leftarrow \emptyset$
2: $color \leftarrow \lfloor k\, p_r / p \rfloor$ ▷ for splitting $comm$
3: **parallel for** $i \in 1, ..., k - 1$ **do** ▷ asynchronous receive
4: $p_{recv} \leftarrow m\,((color - i) \mod k) + (p_r \mod m)$
5: $R \leftarrow R \cup$ `MPI_Irecv`$(p_{recv}, comm)$
6: **end for**
7: **for** $i \in 1, ..., k - 1$ **do** ▷ staged sends
8: $p_{recv} \leftarrow m\,((color - i) \mod k) + (p_r \mod m)$
9: $p_{send} \leftarrow m\,((color + i) \mod k) + (p_r \mod m)$
10: `MPI_Issend`$(M^s|_{m\,i \leq s < m\,(i+1)}, p_{send}, comm)$
11: `MPI_WaitRecv`(p_{recv})
12: **end for**
13: `MPI_WaitAll`()
14: $comm' \leftarrow$ `MPI_Comm_split`$(color, comm)$
15: **return** $[R, comm']$

scalability to $\mathcal{O}(10^5)$ cores. In every step of the hypercube communication pattern, each task sends and receives data from one other task and splits the problem into two smaller ones. The communication completes in $\log p$ steps, with $\mathcal{O}(N)$ data transfer in each step. Here we generalize this scheme to k-way splitting of the problem in each step (Algorithm 3.5). This reduces the number of steps by a factor of $\log k$ and the total data transfer from $\mathcal{O}(N \log p)$ to $\mathcal{O}(N \log p / \log k)$. Replacing the original `MPI_Alltoallv` in

SampleSort with this implementation allowed us to improve its scalability as illustrated in Figure 1. In each stage

Algorithm 3.5 Alltoallv_kway

Input: *comm*, p (w.l.g., assume $p = k^l$), p_r of current task in *comm*, message $M^i_{p_r}$ destined for task $p_i \ \ \forall i$
Output: message $M^{p_r}_i$ received from task $p_i \ \ \forall i$
1: **while** $p > 1$ **do** ▷ iterations: $\mathcal{O}(\log p / \log k)$
2: $[M, comm] \leftarrow$ `SendRecv_kway`$(\{M^j_i\}, comm)$
3: **end while**
4: **return** M

of the k-way communication, each task communicates with k other tasks. Sending all of these k messages asynchronously leads to network congestion. Therefore, we have used a staged communication scheme where we send a new message only upon receiving a message as described in [17]. In the extreme case of $k = p$, this method reduces to the scheme where every task communicates with every other task with $\mathcal{O}(p^2)$ total messages sent/received. This scheme worked well up to a few thousand tasks, but becomes costly for very large task counts.

Note that the load imbalance during an intermediate stage of the algorithm can be as large as a factor of $\sqrt{p}$. However, this level of load imbalance is not observed in random distributions. If a significant load imbalance is detected following a communication stage, we redistribute the data for those sub-problems.

3.2 Selecting Splitters using Parallel Select

As discussed, HypercubeQuickSort, HistogramSort, Sample Sort and other splitter based sorting algorithms all rely on efficient determination of accurate splitters. For HypercubeQuickSort, in each stage, a single splitter is determined, usually by choosing the median on any one task. However, this is not a reliable method and an error of ϵN in the rank of the splitter can lead to the final load on a task to be as large as $\mathcal{O}((1+\epsilon)^{\log p} n)$. In SampleSort, a set of $p-1$ splitters is generated by taking p evenly spaced samples in each locally sorted array.

In Algorithm 3.6, we use a technique also used in histogram sort. However, instead of computing $p-1$ splitters, which becomes costly for large p, we only compute $k < p$ splitters: 1) Select $\beta\, k$ samples from the entire data (across all tasks) and collect these on all tasks. 2) Sort samples and determine ranks locally using binary search. 3) Perform an `MPI_AllReduce` to determine the global ranks of the samples. 4) Select pairs of samples with ranks just greater and just smaller than the expected global rank for each of the k splitters. 5) Repeat steps $1-4$ by taking β samples in the narrowed range for each splitter and iterating till the range is within a tolerance value. The number of samples β must be such that the number of iterations needed is not very high and also the cost of each iteration is small. In our experiments, $\beta \in [20, 40]$ worked well.

3.3 HykSort

We improve upon the original Hypercube QuickSort (Algorithm 3.1), by using Algorithm 3.6 to select splitters and by generalizing the 2-way splitting to a k-way splitting combined with Algorithm 3.5. We presort the local array and following data-exchanges, we locally merge the k

Algorithm 3.6 ParallelSelect

Input: A_r (locally sorted), n, N, $R\,[0, ..., k-1]$ (expected global ranks), N_ϵ global rank tolerance, $\beta \in [20, 40]$.
Output: global splitters $S \subset A$ with approximate global ranks $R\,[0, ..., k-1]$
1: $R^{start} \leftarrow [0, ..., 0]$ ▷ Start of range for sampling splitters
2: $R^{end} \leftarrow [n, ..., n]$ ▷ End of range for sampling splitters
3: $n_s \leftarrow [\beta/p, ..., \beta/p]$ ▷ #of local samples (each splitter)
4: $N_{err} \leftarrow N_\epsilon + 1$
5: **while** $N_{err} > N_\epsilon$ **do**
6: $Q' \leftarrow A_r\left[\texttt{rand}\left(n_s, \left(R^{start}, R^{end}\right)\right)\right]$
7: $Q \leftarrow \texttt{Sort}(\texttt{All\_Gather}(\hat{Q}'))$ ▷ $\mathcal{O}(k \log p + k \log k)$
8: $R^{loc} \leftarrow \texttt{Rank}(Q, A_r)$ ▷ $\mathcal{O}(k \log n)$
9: $R^{glb} \leftarrow \texttt{All\_Reduce}\left(R^{loc}\right)$ ▷ $\mathcal{O}(k \log p)$
10: $I[i] \leftarrow \text{argmin}_j |R^{glb}[j] - R[i]| \ \ \forall i$
11: $N_{err} \leftarrow \max \left|R^{glb} - R\,[I]\right|$
12: $R^{start} \leftarrow R^{loc}[I-1]$
13: $R^{end} \leftarrow R^{loc}[I+1]$
14: $n_s \leftarrow \beta \frac{R^{end} - R^{start}}{R^{glb}[I+1] - R^{glb}[I-1]}$
15: **end while**
16: **return** $S \leftarrow Q[I]$

segments to maintain locally sorted arrays. Following [17], we overlap communication to receive the new keys with the merging of the keys with the local array. The overall algorithm is described in Algorithm 3.7.

3.3.1 SIMD Optimized Merge

For merging two sorted sequences we implemented a bitonic merge kernel using SIMD instructions as outlined in [5]. Specifically, we used SSE C intrinsics for integers and floats and AVX C intrinsics for the float and double data types.

Additional shuffle instructions are needed after the second level of the sorting network due to the nature of the data movement and lack of a general SSE shuffle operation which captures all possible permutations. The code for a 4-wide merging network using AVX instructions follows the same logic but extra complications arise due to the higher width which requires the use of instructions that permute in 128-bit chunks and those which permute across the 128-bit boundary, in addition to the shuffle instructions.

3.3.2 OpenMP Merge Sort

We add OpenMP parallelism by performing the local sorting using an OpenMP merge sort algorithm. We partition the array into equal parts and use `std::sort` to sort each part independently using OpenMP. Next, we merge the parts using the merge tree algorithm (Algorithm 3.7 lines: 16-20). Each merge is performed using an OpenMP parallel merge algorithm 3.8, which compute $p-1$ splitters to split A and B, then merges each section independently to C.

3.4 Complexity analysis

In the previous section we discussed in detail the various algorithmic variants we have considered in our algorithm design. Here we summarize the complexity estimates assuming a hypercube topology.

- BitonicSort requires no collectives and is very fast for small n (the grain size). The total time is local sort (making the array bitonic) plus exchange comparisons

Algorithm 3.7 HykSort

Input: A_r, $comm$, p (w.l.g., assume $p = mk$), p_r of current task in $comm$,
Output: globally sorted array B

$B \leftarrow$ `LocalSort`(A)
while $p > 1$ **do** ▷ Iters: $\mathcal{O}(\log p/\log k)$
  $N \leftarrow$ `MPI_AllReduce`$(|B|, comm)$
  $s \leftarrow$ `ParallelSelect` $(B, \{i\,N/k \mid i = 1, ..., k-1\})$
  $d_{i+1} \leftarrow$ `Rank`(s_i, B), $\forall i$
  $[d_0,\ d_k] \leftarrow [0,\ n]$
  $color \leftarrow \lfloor k\, p_r/p \rfloor$
  parallel for $i \in 0, ..., k-1$ **do**
    $p_{recv} \leftarrow m\,((color - i) \bmod k) + (p_r \bmod m)$
    $R_i \leftarrow$ `MPI_Irecv`$(p_{recv}, comm)$
  end for
  for $i \in 0, ..., k-1$ **do**
    $p_{recv} \leftarrow m\,((color - i) \bmod k) + (p_r \bmod m)$
    $p_{send} \leftarrow m\,((color + i) \bmod k) + (p_r \bmod m)$
    `MPI_Issend`$(B[d_i, ..., d_{i+1} - 1], p_{send}, comm)$
    $j \leftarrow 2$
    while $i > 0$ *and* $i \bmod j = 0$ **do**
      $R_{i-j} \leftarrow$ `merge`$(R_{i-j}, R_{i-j/2})$
      $j \leftarrow 2j$
    end while
    `MPI_WaitRecv`(p_{recv})
  end for
  `MPI_WaitAll`()
  $B \leftarrow$ `merge`$(R_0, R_{k/2})$
  $comm \leftarrow$ `MPI_Comm_split`$(color, comm)$
  $p_r \leftarrow$ `MPI_Comm_rank`$(comm)$
end while
return B

and point-to-point communication.

$$T_b = \left(t_c \frac{N}{p}\right) \log \frac{N}{p} + \left(t_s + t_w \frac{N}{p}\right) \log^2 p.$$

- The main costs for SampleSort is sorting splitters and collectives for data reshuffling. The latter can lead to load imbalance and network congestion represented by a constant c below.

$$T_s = t_c c \frac{N}{p} \log \frac{N}{p} + (t_s + t_w p) \log^2 p + t_w c \frac{N}{p}.$$

- ParallelSelect for determining k splitters, uses the `All_Gather()` and `All_Reduce()` communication collectives. The main computational cost is in determining the local ranks of the samples using binary search. The number of iterations η depend on the input distribution, the required tolerance $\epsilon = N_\epsilon/N$ and the parameter beta. The expected vaue of η varies as $-\log(\epsilon)/\log(\beta)$ and β must be chosen experimentally to minimize the running time T_{ps}:

$$T_{ps} = \eta\, t_c \beta k \log \frac{N}{p} + \eta(t_s + t_w \beta k) \log p.$$

- AlltoallvKway uses only point-to-point communications with staged message sends and receives, minimizing network congestion. It has $\log p/\log k$ stages with $\mathcal{O}(N/p)$ data transfer and k messages for each task in every stage.

$$T_{a2a} = \left(t_s k + t_w \frac{N}{p}\right) \frac{\log p}{\log k}$$

- HykSort has the same communication pattern as AlltoallvKway. In addition it relies on ParallelSelect to determine splitters. The main computational cost is in the initial local sort and merging k arrays during each iteration.

$$T_{Hk} = t_c \frac{N}{p} \log \frac{N}{p} + \left(t_c \frac{N}{p} + T_{ps}\right) \frac{\log p}{\log k} + T_{a2a}$$

Unlike SampleSort, the complexity of HykSort does not involve any $\mathcal{O}(p)$ terms. (Bitonic sort involves $\mathcal{O}(\log^2 p)$ terms but is quite slow when $N/p \gg 1$.)

Algorithm 3.8 OpenMP_Merge

Input: A, B sorted arrays to merge, p number of OpenMP threads, $\beta \sim 10$.
Output: C merged output.

$S \leftarrow A[0 : \frac{n_A}{\beta p} : n_A] \cup B[0 : \frac{n_B}{\beta p} : n_B]$
$R^A_{i+1} \leftarrow$ `Rank`(S_i, A), $\forall i$ ▷ in parallel
$R^B_{i+1} \leftarrow$ `Rank`(S_i, B), $\forall i$ ▷ in parallel
$R \leftarrow R^A + R^B$
$I_i \leftarrow \text{argmin}_j |R_j - i(n_A + n_B)/p|$, $\forall i = 1, .., p-1$
$d^A \leftarrow [0,\ R^A_I,\ n_A]$
$d^B \leftarrow [0,\ R^B_I,\ n_B]$
$d \leftarrow d^A + d^B$
parallel for $i \in 0, ..., p-1$ **do**
  $C_{d_i\,:\,d_{i+1}-1} \leftarrow$ `SIMD_Merge`$(A_{d^A_i\,:\,d^A_{i+1}-1}, B_{d^B_i\,:\,d^B_{i+1}-1})$
end for
return C

4. EXPERIMENTAL STUDY

In this section we present results demonstrating the scalability and performance of k-way SampleSort and HykSort. We provide details regarding our implementation and the different datasets used for evaluation. Finally, we present weak and strong scalability experiments that demonstrate the performance of our scheme.

All experiments reported in this paper were performed on *Titan*, a Cray XK7 supercomputer at Oak Ridge National Laboratory (ORNL). Titan has a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has with 32GB of memory. It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes. (In addition, all of Titan's 18,688 compute nodes contain an NVIDIA Tesla K20 graphical processing unit, which we don't use currently). In our largest runs we use a total of 262,144 cores.

4.1 Implementation Details

All algorithms described in this work were implemented using C++ using OpenMP and MPI. In addition, SIMD acceleration (`SSE` and `AVX`) was employed to speed up the merge operation. SIMD acceleration was only employed for supported datatypes (`int`, `float` and `double`). The sequential sorting algorithm used in all parallel implementations was the C++ standard library sort (`std::sort`) which implements Introsort [12]. The PGI compiler (`v12.9`) was

Table 2: *Single-core performance, in seconds.*

data	size	Titan
int	128MB	4.81
int	2GB	88.35
treenode	128MB	3.01
treenode	2GB	56.68
100-byte	128MB	1.67
100-byte	2GB	39.62

Table 3: *Single-node performance on Titan. The column header specifies the number of MPI tasks and the number of OpenMP threads per task. In all instances, MPI performs better. See Figure 3 for results at higher node counts.*

	tasks : threads		
key type	16:1	2:8	1:16
int 128mb	0.51	0.61	0.63
int 2GB	9.09	10.65	10.96
treenode 128mb	0.39	0.76	1.13
treenode 2GB	6.4	12.78	18.61
100-byte 128mb	0.33	0.62	1.15
100-byte 2GB	5.54	10.52	15.79

used on Titan with the following optimization flags `-O3 -fastsse -mp`. Three different datatypes were used in our experiments; 32-bit integer keys (`4-byte`), a `56-byte`-key (`treenode`) used for representing octrees [18] and a `100-byte` structure commonly used in sort benchmarks [7]. The `100-byte` structure consists of a `10-byte` key and a `90-byte` payload. For the `int` and `treenode` keys, we generated datasets using uniform and Gaussian distributions.

4.2 Single-Node and MPI vs OpenMP Performance

In this section we report single-core performance on different datatypes, as a point of reference for the rest of the experiments. As previously mentioned, we use the C++ `std::sort` method as the sequential sort. We report the time to sort the different datatypes in Table 2.

We also consider single-node performance on Titan. Specifically, we make the choice between using pure MPI based parallelization and a MPI-OpenMP hybrid code. On both platforms, we test three configurations; pure MPI (16 MPI tasks), hybrid (2 MPI tasks, 8 OpenMP threads per task) and pure OpenMP (1 MPI task, 16 OpenMP threads). These results are summarized in Table 3. As can be seen, in all cases the pure MPI configuration is the most efficient. Although running a single MPI task per core is most efficient on a single node and at small node—and consequently small core—counts, the communication costs start to dominate at larger node counts. The difference between the two approaches is shown in Figure 3 for a grain size of 2GB per node using integer keys and the SampleSort algorithm. As can be seen, the communication costs are significantly lower by using the hybrid approach and using one MPI task per socket (i.e., two MPI tasks per node on Titan). For this reason, we used the hybrid approach for all scalability results reported in this paper.

4.3 Parallel Scalability

To study the parallel performance of our implementation, we performed strong (fixed-size) and weak (isogranular) scalability experiments. We report statistics for several stages of the algorithm. For every stage we report the maximum time for that stage across all MPI tasks. We evaluated the scalability of HykSort and the k-way SampleSort and compared them against the the classical SampleSort and Hypercube QuickSort. For the SampleSort algorithm we report the times for the following stages;

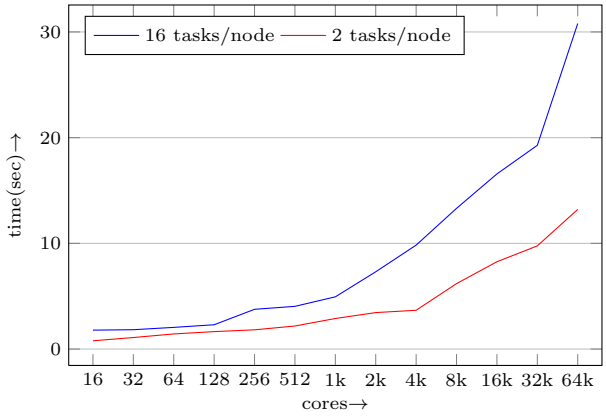

Figure 3: *Effect of number of MPI tasks per node on the SampleSort communication (`all2all`) stage on Titan. Both approaches use a `128-way all2all`, which was the fastest for $8 \leq k \leq 256$. A grain size of 512M keys of type `int` (2GB) per node was used in both cases.*

- **sequential sort** the time spent in local sequential sorts,
- **splitters** the time spent in sampling, ranking, sorting and selecting the splitters, and
- **all2all** the time spent in the final exchange of data.

For HykSort, we report the times for the following stages;

- **sequential sort** the time spent in initial sequential sort,
- **splitters** the time spent in determining the k-splitters, cumulative over all stages, and
- **exchange-merge** the time spent in the overlapped k-way exchange and merging of sub-arrays.

Optimal k.

To select the best value of k, we performed several weak scaling studies. In these studies, we focussed on the `all2all` stage for SampleSort and the combined data exchange and merge for HykSort since these are the only stages that depend on k. The best performance for both approaches on Titan was obtained for $k = 128$ and is plotted in Figures 4 and 5 for values of $k = 64, 128$ and 256. It is important to note that the optimal k will depend on the network characteristics and will be specific to the cluster. In our experiments, we did not observe sensitivity to the datatype on the value of k.

Strong Scalability.

To assess the strong scalability of HykSort, especially at large core counts, we used a problem size of 1TB and increased the core count from 8K to 256K (a 32 fold increase) on Titan. For integer keys this reduced the number of keys per core from 32 million to one million. The strong scalability experiments were carried out for both HykSort as

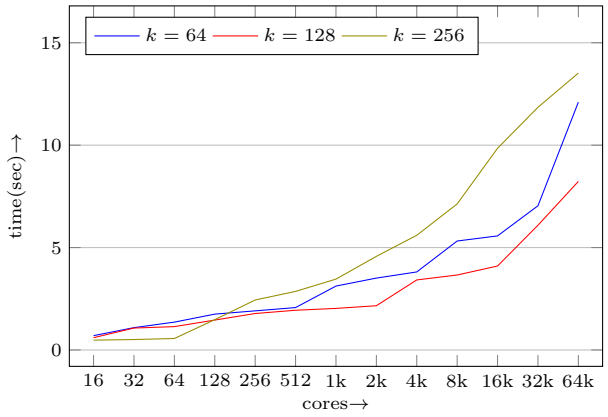

Figure 4: *Effect of different k values on the weak scalability of the* HykSort `exchange data` *stage on Titan. A grain size of 38M* `treenode` *keys (2GB) keys per node was used in all cases. The plots correspond to the communication part only. In addition, since we overlap communication with the merging stage, only the part where the task is waiting for data is measured.*

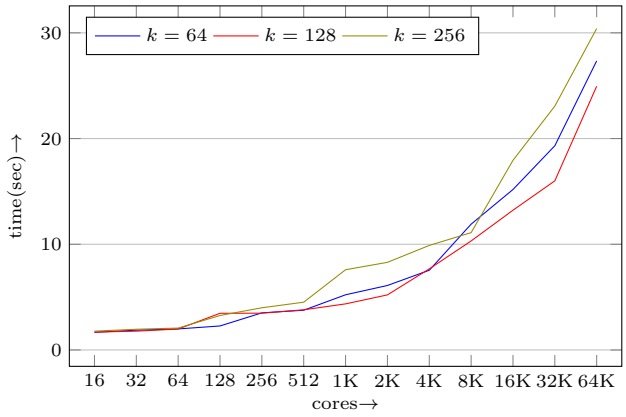

Figure 5: *Effect of different k on weak scalability of the* SampleSort *communication (*`all2all`*) stage on Titan. A grain size of 512M* `int` *(2GB) keys per node was used in all cases. The plots correspond to the communication part only.*

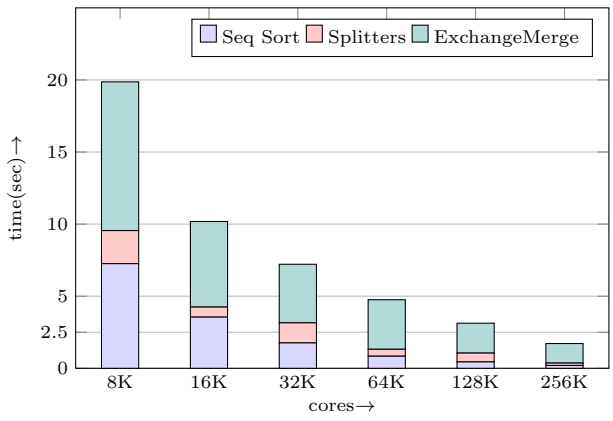

Figure 6: *Strong scalability using 128-way* HykSort *with integer datatype, with a fixed problem size of 1TB. All runs were performed using* 2 MPI *tasks per node with* 8 OpenMP *threads per* MPI *task. The 32 fold increase in core count reduces the number of keys from 32 million per core to just 1 million per core. We obtain a speedup of about* 12x *at 262,144 cores.*

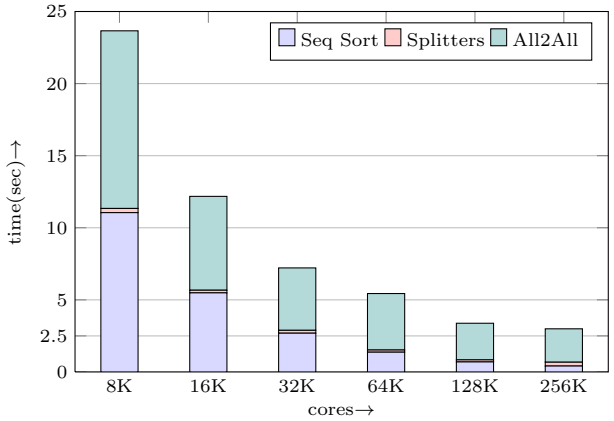

Figure 7: *Strong scalability using 128-way* SampleSort *with integer datatype, with a fixed problem size of 1TB. All runs were performed using* 2 MPI *tasks per node with* 8 OpenMP *threads per* MPI *task. The 32 fold increase in core count reduces the number of keys from 32 million per core to just 1 million per core. The scalability is lower than that obtained using* HykSort *for the same problem, with a speedup of about* 8x *at 256K cores. The sequential sort takes longer compared compared with* HykSort *because of the need to do two sequential sorts as opposed to just one for* HykSort.

well as the k-way SampleSort. These results are shown in Figure 6 and 7. For HykSort we obtain very good scalability all the way till 256K cores resulting in a 12x speedup. For SampleSort we obtain slightly lower speedup of 8x, as the cost of sorting splitters using BitonicSort increases at 256K cores.

Weak Scalability.

Weak scalability tests were performed using the integer and `100-byte` datatypes. For both cases, we choose a grain size of 128MB per core resulting in a total problem size of 34TB on 262,144 cores. This results in a total of 8.5 trillion integer keys and 350 billion 100-byte keys that need to be sorted. The results for the integer keys are reported in Figures 8 and 9. Using HykSort we are able to sort the 8.5 trillion keys in about 37 seconds for a throughput of approximately 0.9TB/s. The throughput using the 128-way SampleSort algorithm is slightly lower at 0.75TB/s. We see that the cost of sorting the splitters using BitonicSort starts to become significant at large core counts. The results for sorting the 100-byte keys are similar and are reported in Figures 10 and 11. With the larger datatype, the effects of load-imbalance are more pronounced and can be observed in the scalability of both HykSort and SampleSort. We are currently working on incorporating load balancing into these algorithms which should help maintain a high throughput with large datatypes at large core counts.

5. CONCLUSIONS

We have presented two new algorithms for distributed memory comparison sort. The first algorithm is a k-way all-to-all staged communication that can be used in the classical SampleSort sort algorithm to accelerate the data exchange by avoiding network congestion. The second algorithm, HykSort combines accurate splitter selection with a k-way recursion to minimize load imbalance and communica-

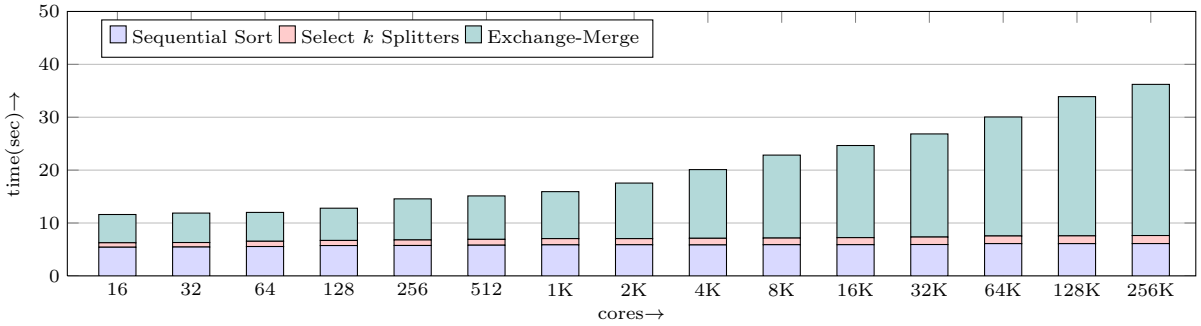

Figure 8: *Weak scaling for* 128*-way* HykSort *from 16 to 262,144 cores with a grain size of* 128MB *per core using the* int *datatype. All runs were performed using* 2 MPI *tasks per node with* 8 OpenMP *threads per* MPI *task. A total of 8.5 trillion keys (*32TB*) were sorted in about 37 seconds, for a throughput of* 0.9TB/s*. Note that the problem size increases* 16,384x *during this experiment from* 2GB *to* 32TB*.*

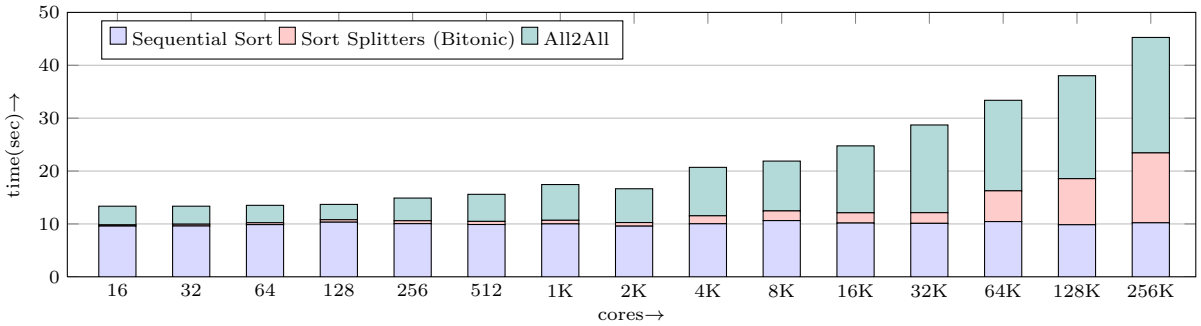

Figure 9: *Weak scaling for* 128*-way* SampleSort *from 16 to 262,144 cores with a grain size of* 128MB *per core using the* int *datatype. A total of 8.5 trillion keys (*32TB*) were sorted in about 45 seconds, for a throughput of* 0.75TB/s*. The lower throughput compared with* HykSort *is mainly due to the poor scalability of* BitonicSort *at large core counts. The sequential sort takes longer compared compared with* HykSort *because of the need to do two sequential sorts as opposed to just one for* HykSort*. Note that the problem size increases* 16,384x *during this experiment from* 2GB *to* 32TB*.*

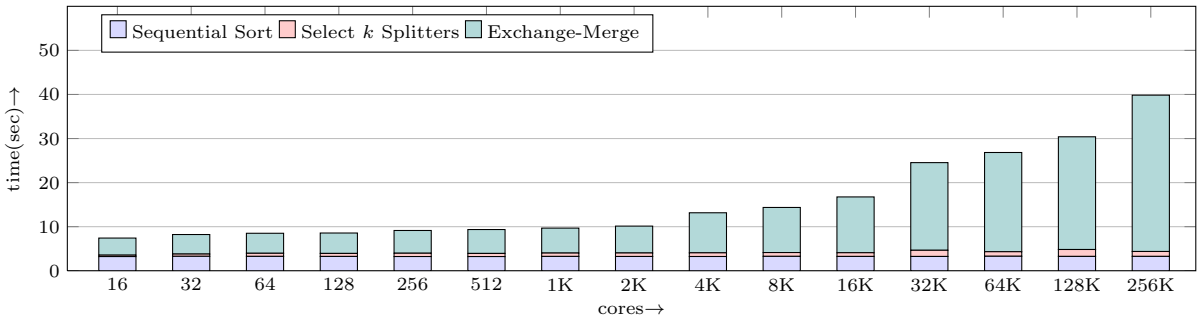

Figure 10: *Weak scaling for* 128*-way* HykSort *from 16 to 262,144 cores with a grain size of* 128MB *per core using the* 100-byte *datatype. A total of 351 billion keys (*32TB*) were sorted in about 40 seconds, for a throughput of* 0.85TB/s*. Note that the problem size increases* 16,384x *during this experiment from* 2GB *to* 32TB*.*

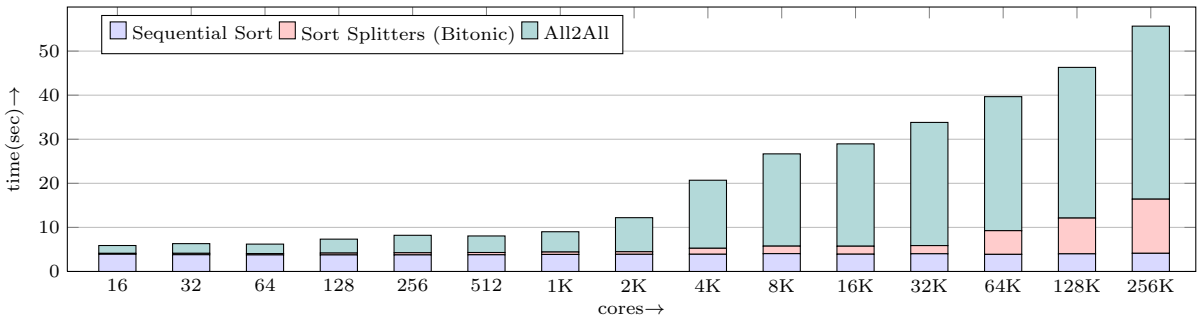

Figure 11: *Weak scaling for* 128*-way* SampleSort *from 16 to 262,144 cores with a grain size of* 128MB *per core using the* 100-byte *datatype. A total of 351 billion keys (*32TB*) were sorted in about 55 seconds, for a throughput of* 0.63TB/s*. The lower throughput compared with* HykSort *is due to the poor scalability of* BitonicSort *at large core counts as well as due to load-balance issues arising with the distribution of the* 100-byte *keys. Note that the problem size increases* 16,384x *during this experiment from* 2GB *to* 32TB*.*

tion. Our design avoids `MPI_Alltoall` and `MPI_Alltoallv` exchanges. The only remaining issue is the load imbalance introduced; not due to uneven bucket size, but due to variations in the local grain size due to the data exchanges. With large k and as the recursion progresses this is less of an issue but it is still present. The fix is easy: we need to add a load balancing step after every data exchange. This repartitioning can be done again without any all-to-all collectives. But even without repartitioning, our proposed design improves significantly over the state of the art by achieving 0.9TB/sec throughput for the `int` dataset. The source code for all algorithms described in this paper is available at `http://padas.ices.utexas.edu/`.

Moving forward, the main improvements in our implementation will be the inclusion of accelerators and algorithmic redesign to enable fault tolerance without excessive costs. Another possibility is that of tuning the code based on machine characteristics, number of MPI tasks, and the problem size. For example, for very small grain sizes bitonic sort is expected to perform very well. Also the k parameter can be selected by building a more accurate performance model.

Acknowledgment

This research was partially supported by AFOSR grant FA9550-12-10484; DOE grants DE-FG02-08ER2585 and DE-SC0009286; and NSF grants OCI-1047980, OCI-1029022 and OCI-0923710. Computing time on ORNL's Titan system was provided by an allocation from the Oak Ridge Leadership Computing Facility. We will also like to thank Dr. Judith Hill of ORNL for her assistance with porting our codes to Titan.

6. REFERENCES

[1] P. Berthomé, A. Ferreira, BM Maggs, S. Perennes, and CG Plaxton. Sorting-based selection algorithms for hypercubic networks. *Algorithmica*, 26(2):237–254, 2000.

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

[3] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[4] D.R. Cheng, V.B. Shah, J.R. Gilbert, and A. Edelman. A novel parallel sorting algorithm for contemporary architectures. *Int. J. Parallel Prog*, pages 1–12, 2007.

[5] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, August 2008.

[6] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison Wesley, second edition, 2003.

[7] Jim Gray. Sort benchmark home page. `http://sortbenchmark.org/`, April 2013.

[8] L.V. Kale and S. Krishnan. A comparison based parallel sorting algorithm. In *International Conference on Parallel Processing, 1993*, volume 3, pages 196–200. IEEE, 1993.

[9] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *Proceedings of the 2012 international conference on Management of Data*, pages 841–850. ACM, 2012.

[10] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Communications of the ACM*, 55(5):101–109, May 2012.

[11] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[12] D.R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[13] A. Rasmussen, G. Porter, M. Conley, H.V. Madhyastha, R.N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *USENIX NSDI*, volume 11, 2011.

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

[15] N. Satish, C. Kim, J. Chhugani, A.D. Nguyen, V.W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs, GPUs and intel MIC architectures. Technical report, Technical report, Intel, 2010.

[16] E.L.G. Saukas and SW Song. Efficient selection algorithms on distributed memory computers. In *SC98, IEEE/ACM Conference on Supercomputing, 1998*, pages 20–20. IEEE, 1998.

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

[18] Hari Sundar, Rahul S. Sampath, and George Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

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