Optimizing High Performance Distributed Memory Parallel Hash Tables for DNA k-mer Counting

Tony C. Pan
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: tpan7@gatech.edu

Sanchit Misra
Parallel Computing Laboratory
Intel Corporation
Bangalore, India
Email: sanchit.misra@intel.com

Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: aluru@cc.gatech.edu

***Abstract*—High-throughput DNA sequencing is the mainstay of modern genomics research. A common operation used in bioinformatic analysis for many applications of high-throughput sequencing is the counting and indexing of fixed length substrings of DNA sequences called k-mers. Counting k-mers is often accomplished via hashing, and distributed memory k-mer counting algorithms for large datasets are memory access and network communication bound. In this work, we present two optimized distributed parallel hash table techniques that utilize cache friendly algorithms for local hashing, overlapped communication and computation to hide communication costs, and vectorized hash functions that are specialized for k-mer and other short key indices. On 4096 cores of the NERSC Cori supercomputer, our implementation completed index construction and query on an approximately 1 TB human genome dataset in just 11.8 seconds and 5.8 seconds, demonstrating speedups of $2.06\times$ and $3.7\times$, respectively, over the previous state-of-the-art distributed memory k-mer counter.**

***Index Terms*—Hash tables, k-mer counting, vectorization, cache-aware optimizations, distributed memory algorithms**

I. Introduction

Wide-spread adoption of next-generation sequencing (NGS) technologies in fields ranging from basic biology and medicine to agriculture and even social sciences, has resulted in the tremendous growth of public and private genomic data collections such as the Cancer Genome Atlas [1], the 1000 Genome Project [2], and the 10K Genome Project [3]. The ubiquity of NGS technology adoption is attributable to multiple orders of magnitude increases in sequencing throughput amidst rapidly decreasing cost. For example, a single Illumina HiSeq X Ten system can sequence over 18,000 human genomes in a single year at less than $1000 per genome, corresponding to approximately 1.6 quadrillion DNA base pairs per year.

The volume and the velocity at which genomes are sequenced continues to push bioinformatics as a *big data* discipline. Efficient and scalable algorithms and implementations are essential for high throughput and low latency analyses including whole genome assembly [4]–[6], sequencing coverage estimation and error correction [7]–[9], variant detection [8] and metagenomic analysis [10].

Supported in part by the National Science Foundation under IIS-1416259 and Intel Parallel Computing Center on Big Data in Biosciences and Public Health.

Central to many of these bioinformatic analyses is k-mer counting, which computes the occurrences of length k substrings, or k-mers, in the complete sequences. k-mer occurrence frequencies reflect the quality of sequencing, coverage variations, and repetitions in the genome. The majority of k-mer counting tools and algorithms are optimized for shared-memory, multi-core systems [11]–[17]. However, distributed memory environments present a ready-path for scaling genomic data analysis, by recruiting additional memory and computational resources as needed, to larger problem sizes and higher performance requirements. While distributed-memory k-mer indexing is sometimes integrated in distributed genome assemblers [5], [6], [18], we previously presented an open source distributed k-mer counting and indexing library, Kmerind [19], with class-leading performance in distributed and shared memory environments.

Exact k-mer counting is usually accomplished via sort-and-count [16] or via hashing [11], [19]. With the hashing approach, the k-mers are inserted into a hash table using a hash function. Each hash table entry contains a k-mer as key and its count as value. In distributed memory parallel environments, each rank reads a subset of input sequences to extracts k-mers, while maintaining a subset of the distributed hash table as a local hash table. The extracted k-mers on a rank are sent to the ranks maintaining the corresponding distributed hash table subsets. The k-mers thus received by a rank are then inserted into its local hash table. Distributed memory k-mer counting spends majority of its time in the following tasks: 1) Hashing the k-mers by each rank, 2) Communicating k-mers between ranks over the network and 3) Inserting k-mers into the local hash tables. The local hash tables are typically much larger than cache sizes and table operations require irregular memory access. Operations on local hash tables are therefore lead to cache misses for nearly every insertion and query and are memory latency-bound. On the other hand, hashing of k-mers is a highly compute intensive task.

In this paper, we present algorithms and designs for high performance distributed hash tables for k-mer counting. With the goal of achieving good performance and scaling on leadership-class supercomputers, we optimized the time consuming tasks at multiple architectural levels – from mitigating

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

network communication costs, to improving memory access performance, to vectorizing computation at each core.

Specifically, we make the following major contributions:

- We designed a hybrid MPI-OpenMP distributed hash table for the k-mer counting application using thread-local sequential hash tables for subset storage.
- We designed a generic interface for overlapping arbitrary computation with communication and applied it to all distributed hash table operations used in k-mer counting.
- We vectorized MurmurHash3 32- and 128-bit hash functions using AVX2 intrinsics for batch-hashing small keys such as k-mers. We also demonstrated the use of CRC32C hardware intrinsic as a hash function for hash tables.
- We improved Robin Hood hashing [20], [21], a hash table collision handling scheme, by using an auxiliary *offset* array to support direct access to elements of a bucket, thus reducing memory bandwidth usage.
- We additionally designed a novel hashing scheme that uses lazy intra-bucket radix sort to manage collisions and reduce memory access and compute requirements.
- We leveraged the batch processing nature of k-mer counting and incorporated software prefetching in both hashing schemes to improve irregular memory access latency.

The impacts of our contributions are multi-faceted. This work represents, to the best of our knowledge, the first instance where Robin Hood hashing has been applied to k-mer counting. We believe this is also the first instance where software prefetching has been used in hash tables for k-mer counting, transforming the insert and query operation from being latency bound to bandwidth bound. Given both of our hashing schemes are designed to lower bandwidth requirements, they perform significantly better than existing hash tables. The hybrid MPI-OpenMP distributed hash table with thread-local storage minimizes communication overhead at high core counts and improves scalability, while avoiding fine grain synchronizations otherwise required by shared, thread-safe data structures.

Our vectorized hash function implementations achieved up to $6.6\times$ speedup over the scalar implementation, while our hashing schemes out-performed Google's dense hash map by $\approx 2\times$ for insertion and $\approx 5\times$ for query. Cumulatively, the optimizations described in this work resulted in performance improvement over Kmerind of $2.05 - 2.9\times$ for k-mer count index construction and $2.6 - 4.4\times$ for query. It performs index construction and query on a ≈ 1 TB human genome dataset in just 11.8 seconds and 5.8 seconds, respectively, using 4096 cores of the Cori supercomputer.

While our vectorized MurmurHash3 hash function implementation and our hash table collision handling algorithms are motivated by the k-mer-counting problem, we expect them to work well generally for applications involving batches of small keys. Our implementation is available at https://github.com/tcpan/kmerhash.

The paper is organized as follows: Section II summarizes previous work on sequential and distributed hash tables. Section III describes our approach to communication-computation overlap for the distributed hash tables while Section IV our Robin Hood and radix sort based hash table collision handling schemes. Section V describes our implementation and the hash function vectorization strategy. In Section VI we examine the performance and scalability of our algorithms, and compare our k-mer counters to the existing tools in shared and distributed memory environments.

A. Problem Statement

A k-mer γ is defined as a length k sequence of characters drawn from the alphabet Σ. The space of all possible k-mers is then defined as $\Gamma = \Sigma^k$. Given a biological sequence $\mathcal{S} = s[0 \ldots (n-1)]$ with characters drawn from Σ, the collection of all k-mers in $\mathcal{S}$ is denoted by $\mathcal{K} = \{s[j \ldots (j+k-1)], 0 \leq j < (n-k)\}$.

K-mer counting computes the number of occurrences ω_l, or frequency, of each k-mer $\gamma_l \in \Gamma$, $0 \leq l < |\Sigma|^k$, in $\mathcal{S}$. The *k-mer frequency spectrum* $\mathcal{F}$ can be viewed as a set of mappings $\mathcal{F} = \{f : \gamma_l \rightarrow \omega_l\}$.

K-mer counting then requires first transforming $\mathcal{S}$ to $\mathcal{K}$ and then reducing $\mathcal{K}$ to $\mathcal{F}$ in Γ space. We note that k-mers in $\mathcal{K}$ are arranged in the same order as $\mathcal{S}$, while Γ and $\mathcal{F}$ typically follow lexicographic ordering established by the alphabet. The difference in ordering of $\mathcal{K}$ and $\mathcal{F}$ necessitates data movement when computing $\mathcal{F}$, and thus communication.

We further note that the storage representation of $\mathcal{F}$ affects the efficiency of its traversal. While Γ grows exponentially with k, genome sizes are limited and therefore $\mathcal{F}$ is sparse for typical k values. Associative data structures such as hash tables are well suited, which typically support the minimal operations of construction (or *insert* and *update*) and *query* of the k-mer counts.

In this paper we focus on efficient data movement in the $\mathcal{K}$ to $\mathcal{F}$ reduction with distributed memory, and on efficient hash functions and hash tables for constructing and querying $\mathcal{F}$.

II. Background

K-mer counting has been studied extensively [11]–[13], [15]–[17], [19], with most of the associated algorithms designed for shared-memory, multi-core systems. More recently, GPUs and FPGAs have been evaluated for this application [17], [22], [23] as well as general purpose hashing [24]. Counting is accomplished mainly through incremental updates to hash tables, while some used sorting and aggregation [16]. A majority of the prior efforts focus on minimizing device and host memory usage through approximating data structures [12], [15] and external memory [16], or efficient utilization of multiple CPU cores and accelerators [11], or both [17], [23].

Kmerind [19] is a k-mer counting and indexing library that instead targets distributed memory systems and compares favorably even against shared memory k-mer counters on high core-count systems. It utilizes a distributed hash table for storing $\mathcal{F}$ and relies on MPI collective communication for data movement and coarse-grain synchronization. It is able to scale to data sizes that exceed the resources of a single machine.

In this work, we significantly improved Kmerind by replacing its core distributed hash table with optimized hash functions and sequential and distributed hash tables, while adopting its templated interfaces and built-in functionalities such as file parsing for compatibility and ease of comparisons.

A. Hash Tables and Collision Handling Strategies

Three factors affect sequential hash table performance: hash function choice, irregular access to a hash table bucket, and element access within a bucket. Hash function performance depends strongly on implementation and hardware capability. Irregular memory access renders hardware prefetching ineffective but software prefetching can serve as the alternative if the memory address can be determined ahead of use.

Intra-bucket data access performance, on the other hand, is an inherent property of the hash table design, namely the collision handling strategy. Collision occurs when multiple *keys* are assigned to the same bucket, necessitating multiple comparisons and memory accesses. High collision rate results in a deviation from the amortized constant time complexity.

Hash table designs have been studied extensively. Cormen, Leiserson, Rivest, and Stein [25] described two basic hash table designs, *chaining* and *open addressing*. Briefly, chaining uses linked list to store collided entries, while open addressing maintains a linear array and employs deterministic *probing* logic to locate an alternative insertion point during collision. Identical probing logic is used during query.

Different probing algorithms exist. Cormen, Leiserson, Rivest, and Stein presented *linear* probing, where the hash table is searched linearly for empty bucket for insertion. Linear probing is sensitive to non-uniform key distribution, suboptimal hash functions, high load factor, and even insertion order, all of which affect performance. Strategies such as quadratic probing and double hashing [25], HopScotch hashing [26], and Robin Hood hashing [20], [21] aim to address these shortcomings. In all these strategies save Robin Hood hashing, key-value tuples from different buckets are interleaved. Probing thus also require comparisons with keys from other buckets, increasing the average number of probes and running time.

The choice of collision handling strategy can even affect memory access patterns for intra-bucket traversal. Double hashing, HopScotch, and quadratic probing introduce irregular memory accesses. Linear probing and Robin Hood require only sequential memory access, thus can benefit from hardware prefetching, maximize cache line utilization, and reduce bandwidth requirement.

III. Distributed Memory Hash Table

We adopt Kmerind's approach and represent $\mathcal{F}$ as a distributed memory hash table. This allows us to encapsulate the parallel algorithm details behind the semantically simple `insert` and `find` operation interfaces.

We first summarize the distributed memory k-mer counting algorithms in Kmerind. Kmerind models a two level distributed hash table. The first level maps k-mers to MPI processes using a hash function, while the second level exists as local hash tables. Different hash functions are used at the two levels to avoid correlated hash values which increases collision in the local hash table. The distributed hash table evenly partitions Γ and therefore $\mathcal{F}$ across P processes. We assume that $\mathcal{K}$ as the input is evenly partitioned across P processes. Kmerind uses Google's dense hash map as the hash table and Farmhash as the hash function.

The distributed hash table insertion algorithm proceeds in 3 steps. Query operation follows the same steps, except an additional communication returns the query results.

1) *permute*: The k-mers are assigned to processes via the top level hash function, and then grouped by process id via radix sort. The input is traversed linearly, but the output array is randomly accessed. The first half of this step is bandwidth and compute bound while the radix sort is latency bound.
2) *alltoallv*: The rearranged k-mers are communicated to the remote processes via `MPI_Alltoallv` personalized collective communication.
3) *local compute*: The received k-mers are inserted into the local hash table.

A. Communication Optimization

A typical `MPI_Alltoallv` implementation internally uses point-to-point communication over P iterations to distribute data from one rank to all others. Its complexity is $O(\tau P + \mu N/P)$ where τ and μ are latency and bandwidth coefficients, respectively, and N/P is the average message size for a processor. As P increases, the latency term becomes dominant and scaling efficiency decreases. To manage the growth of communication latency, we sought to overlap communication and computation, and to reduce P with a hybrid multi-node (MPI) and multi-thread (OpenMP) hash table.

1) Overlapped communication and computation: We overlap communication and computation to hide communication costs during insertion. Rather than waiting for all point-to-point communication iterations to complete, we begin computation as soon as one iteration completes (Algorithm 1). This approach also allows buffer reuse across iterations, thus reducing memory usage.

Query operations can similarly leverage overlapped communication and computation. Once the computation is complete,

Algorithm 1 Alltoallv with overlapping computation

```
1: I[][]: Input key-value array, grouped by process rank
2: O[2][]: Output buffer
3: P: number of processes; r: process rank; C(): computation to perform
4: for i ← 1...(P − 1) do
5:     non-blocking send I[(r + P − i)%P][] to rank (r + P − i)%P
6: end for
7: for i ← 1...(P − 1) do
8:     non-blocking recv O[(r + i)%2][] from rank (r + i)%P
9:     C(O[(r + i − 1)%2][])
10:    wait for non-blocking recv from rank (r + i)%P to complete
11: end for
12: C(O[(r + P − 1)%2][])
13: wait for all non-blocking sends to complete
```

a non-blocking send is issued, for which the matching non-blocking receives can be posted in Lines 4–6.

We designed a set of generic, templated interfaces for overlapped communication and computation, using function objects or functors to encapsulate the computation. Both insert and query operations use these interfaces.

2) Hybrid multi-node and multi-thread: With large P, the latency term dominates in the communication time complexity. In addition, small perturbation during execution due to communication overhead, load imbalance, or system noise can potentially propagate and amplify over P iterations in `MPI_Alltoallv` and our overlapped version (Algorithm 1).

To ameliorate this sensitivity, we reduce P by assigning one MPI process per socket, and for each process spawn as many threads as cores per socket. Each thread instantiates its own local sequential hash table, and we partition Γ across all threads on all processes. This approach maintains independence between local hash tables, thus inter-thread synchronization is minimized. We enabled multi-threading for the *permute* and *local compute* steps in the distributed hash table insertion algorithm, and the computation steps (Lines 9 and 12) in Algorithm 1.

IV. Sequential Hash Tables

In this section we focus on optimizing intra-bucket element access for sequential hash tables. We use the following notations. A hash table element is defined as a key-value pair $w = \langle k, v \rangle$. The table consists of B buckets. The id of a bucket is denoted as b. The hash function associated with the hash table is $H(\cdot)$. For open addressing hash tables, the *ideal* bucket id for an element w is denoted $b_w = H(w.k)\%B$, where $\%$ is the *modulo* operator, to distinguish from the actual bucket position where the element is stored. The *probe distance* r of an element is the number of hash table elements that must be examined before a match is found or a miss is declared. For linear probing, the probe distance is $r = b - b_w$.

A. Robin Hood Hashing

Robin Hood Hashing is an open addressing strategy that minimizes the expected probe distance, and thus the insertion and query times. It was first proposed by Pedro Celis [20], [21]. We briefly describe the basic algorithms.

During insertion, a new element w is swapped with element u at position b if w has a higher probe distance than u, i.e. $(b - b_w) > (b - b_u)$. The insertion operation continues with the swapped element, w', until an empty bucket is found, where w' is then inserted. This strategy results in the spatial ordering of table elements by the ideal bucket ids, resembling sorting.

Query, or *find*, operation is similar algorithmically. Search completes when an element is found, an empty bucket is reached, or the end of the target bucket is reached, $(b - b_w) > (b - b_u)$, indicating an absent element.

B. Optimized Robin Hood Hashing

The "classic" Robin Hood hashing strategy spatially groups elements of a bucket together. This decreases the variance of r and eliminates interleaving of buckets, thus reducing the number of elements that must be compared and minimizing memory bandwidth requirement. However, classic Robin Hood begins a search from the ideal bucket, b_w. Elements in the range $[b_w \dots (b_w + r))$ are expected to belong to buckets with ids $b < b_w$. On average, classic Robin Hood must access r elements, compute their hash values, and compare probe distances before arriving at the elements of the desired bucket. We can avoid the extraneous work by storing the distance between b_w and the first element in the bucket.

1) Structure: Our extended Robin Hood hash table, T_{rh}, consists of the tuple $\langle P_{rh}[], I_{rh}[] \rangle$. The $P_{rh}[]$ array stores the elements of the hash table. Each element in $I_{rh}[]$, $I_{rh}[b]$, consists of a tuple ⟨empty, offset⟩. The *empty* field indicates whether bucket b is empty, while the *offset* field contains the probe distance from position b to the first element of bucket b, which is then located at $P_{rh}[b + I_{rh}[b]]$. In the case where $I_{rh}[b]$ is empty and *offset* is not zero, $I_{rh}[b]$ references the insertion position for the first element of bucket b.

Storing the probe distances in the $I_{rh}[]$ array avoids r hash function invocations and memory accesses to $P_{rh}[]$. Consequently computation time and memory bandwidth utilization are reduced compared to classic Robin Hood. We assume that insertion occurs less frequently than queries and therefore optimizations that benefit query operations, such as the use of $I_{rh}[]$, are preferred. As r is expected to be relatively small [21], a small data type, e.g. an 8 bit integer, can be chosen to minimize memory footprint and to increase the cache-resident fraction of $I_{rh}[]$. The sign bit of $I_{rh}[b]$ corresponds to $empty$ while the remaining bits correspond to *offset*.

2) Insert: The optimized Robin Hood hash table insertion algorithm is outlined in Algorithm 2. A search range in $P_{rh}[]$ is first computed in constant time from $I_{rh}[]$ in Line 4. Elements in the bucket are then compared for match in Lines 5–9. When matched, the function terminates. If a match is not found within the bucket, then the input element is inserted into $P_{rh}[]$ in the same manner as in classic Robin Hood Hashing, i.e. via iterative swapping. Since elements from bucket $b + 1$ to p, where p after Line 14 references the last bucket to be modified, are shifted forward, their offsets $I_{rh}[(b + 1) \dots p]$ are incremented (Lines 16–19).

The number of $P_{rh}[]$ elements to shift and $I_{rh}[]$ elements to update can be significant, up to $O(\log(B))$ according to Celis. We limit this shift distance indirectly by using only 7 bits for the offsets in $I_{rh}[]$. Overflow of any $I_{rh}[]$ element during insertion causes the hash table to resize automatically. The small number of bits used for offset values and the contiguous memory accesses imply that $I_{rh}[]$ updates can benefit from hardware prefetching and few cache line loads and stores.

3) Find: Query operations in the optimized Robin Hood hash table follows closely the first part of the Insert algorithm (Algorithm 2 Lines 1–9).

C. Radix-sort Map

Inspired by the fact that Robin hood hashing effectively creates a sorted array of elements according to their bucket

Algorithm 2 Optimized Robin Hood Hashing: Insertion

```
1: T_rh: Robin Hood hash table ⟨P_rh, I_rh⟩
2: H(): hash function; ⟨k, v⟩: key-value pair to insert
3: b ← H(k)%B
4: p ← b + I_rh[b].offset; p1 ← b + 1 + I_rh[b + 1].offset
5: while p < p1 do
6:     if (P_rh[p].k == k) then return
7:     end if
8:     p ← p + 1
9: end while
10: p ← b + I_rh[b].offset
11: while p < B AND NOT (I_rh[p] is empty AND I_rh[p].offset == 0) do
12:     swap(⟨k, v⟩, P_rh[p])
13:     p ← p + 1
14: end while
15: P_rh[p] ← ⟨k, v⟩; I_rh[b].empty ← FALSE
16: while b < p do
17:     I_rh[b + 1].offset ← I_rh[b + 1].offset +1
18:     b ← b + 1
19: end while
```

id, thereby improving locality of all operations, we created a hashing scheme that explicitly sorts the elements according to their bucket id and concurrently manages duplicated elements. The scheme exploits the batch processing nature of the k-mer counting application to achieve good performance.

1) Structure: Our Radix sort based hash table, T_{rs}, consists of the following tuple: $\langle P_{rs}[], C_{rs}[], O_{rs}[], I_{rs}[], D_{rs}, w_{rs}, q_{rs}\rangle$, where, $P_{rs}[]$ is the primary one dimensional array that stores the elements of the hash table. It is divided into bins of equal width, w_{rs}, such that the first w_{rs} entries of $P_{rs}[]$ belong to the bin 0, the next w_{rs} belong to bin 1, and so on. Each bin is responsible for maintaining a fixed number of buckets, q_{rs}, such that the first q_{rs} buckets of the hash table are stored in bin 0, the next q_{rs} in bin 1 and so on. The elements in the bin are maintained in the increasing order of their bucket id. For this reason, we also store the bucket id, b, with each element in addition to the key-value pair $\langle k, v\rangle$. $C_{rs}[d]$ maintains the count of stored elements in bin d. D_{rs} is the total number of bins. Thus, $D_{rs} = B/q_{rs}$. If a particular bin becomes full, any additional elements of the bin are stored in the overflow buffer, $O_{rs}[]$. For each bucket id, b, the corresponding bin id $d = b/q_{rs}$ and $I_{rs}[b]$ stores the position within the bin of the first element of the bucket. Thus, the first element of a bucket is located at $P_{rs}[d \times w_{rs} + I_{rs}[b]]$.

2) Insert: During insertion of an element (Algorithm 3), if the target bin has less than $(w_{rs} - 1)$ elements, we append the element to the bin regardless of whether the key exists in the hash table. Otherwise, we sort the bin elements in increasing order of bucket id and merge and count elements with duplicate keys. If the bin remains full after sorting, we borrow a block from O_{rs} and store any further elements for the bin in that block. We use the last entry of the bin to store the location of the block in O_{rs}. Instead of sorting after each insertion, we only sort when a bin becomes full on insertion. We call this technique *lazy sorting*.

At the end of insertion of a batch of keys, a significant number of bins may contain duplicated keys. Before performing any additional operations, we *finalize insert* (Algorithm 4) by sorting each bin and merging elements with equal keys. We also set the value of $I_{rs}[b]$ for each bucket b.

Algorithm 3 Radix-sort Hashing: Insert

```
1: T_rs: Radix sort based hash table ⟨P_rs, C_rs, O_rs, I_rs⟩
2: H(): hash function; ⟨k, v⟩: key-value pair to insert
3: b ← H(k)%B
4: d ← b/q_rs
5: c ← C_rs[d]
6: if (c == w_rs − 1) then
7:     c ←radixSort(P_rs + d × w_rs, c)
8:     C_rs[d] ← c
9: end if
10: if c == w_rs − 1 then
11:     borrow a block from overflow buffer and store ⟨k, v⟩ there
12: else
13:     P_rs[d × w_rs + c] ← ⟨k, v, b⟩
14: end if
15: C_rs[d] ← c + 1
```

There are several nuances to the design. In practice, the bins are chosen to be small such that 16-bit integers are sufficient for $C_{rs}[]$ and $I_{rs}[]$. In addition, if the D_{rs} is small, $C_{rs}[]$ can fit in cache. Insertion of a new element occurs at the end of a bin thus only one cache line is accessed from memory (for P_{rs}). Moreover, the last active cache line of all recently accessed bins should be in cache. Thus, if D_{rs} is small, many of these accesses result in cache hits. This is in contrast to the Robin Hood scheme where we might need to access two cache lines from memory for inserting each element. We use *radix sort* for sorting the elements in a bin. As the number of different possible values, q_{rs}, is fairly small in practice, a single iteration of *radix sort* is sufficient. If the bin size is smaller than the cache size, then the elements in the bin are loaded into cache completely during their first reads. Subsequent element accesses, which are random during sorting, can be serviced by the cache with low latency. Thus small bin sorting can be highly efficient. On the other hand, larger bins ensure sorting is done less frequently.

Clearly, using overflow buffer is a lot more expensive and should be avoided as much as possible. Given a uniform hash function and a sufficiently large q_{rs}, the number of elements per bin is expected to be uniformly distributed by the Law of Large Numbers, despite any non-uniformity in the number of elements in each bucket. Therefore, with a large enough w_{rs}, we should be able to avoid overflow at the cost of decreased hash table load and higher memory foot print. In our implementation, we specify the parameters in such a way that $O_{rs}[]$ is never used when minimizing memory usage. In practice, we have yet to encounter a use case where $O_{rs}[]$ is

Algorithm 4 Radix-sort Hashing: Finalize insert

```
1: T_rs: Radix sort based hash table ⟨P_rs, C_rs, O_rs, I_rs⟩
2: for each bin d do
3:     C_rs[d] ←radixSort(P_rs + d × w_rs, C_rs[d])
4:     for each bucket b in bin d do
5:         I_rs[b] ← starting position of bucket b in bin d
6:     end for
7: end for
```

utilized. We therefore do not detail the usage of $O_{rs}[]$.

3) Find: In order to *find* a key (Algorithm 5), we identify the positions in $P_{rs}[]$ of the corresponding bucket at $P_{rs}[d \times w_{rs} + I_{rs}[b]]$. We then linearly scan the bucket for a match.

V. Implementation Level Optimizations

The k-mer counting application, and likely other big data analytic problems, are often amenable to batch mode processing. We leverage this fact to enable vectorized hash value computation, cardinality estimation, and software prefetching for irregular memory access in the hash table.

For both Robin Hood and Radix-sort hash tables, we choose B and q_{rs} as powers of 2 so that single-cycle bitwise *shift* and *and* are used instead of *division* and *modulo* operations.

A. Hash Function Vectorization

The choice of hash functions can have significant impacts for sequential and distributed memory hash table performances. Uniform distribution of hash values improves load balance for distribute hash tables and thus the communication and parallel computation time, and reduces collision rate thus increases sequential hash table performance. Computational performance of the hash function itself is also important.

For k-mer counting, the keys to be processed are numerous and limited in size, rarely exceeding 64 bytes in length and often can fit in 8 to 16 bytes. Well known and well behaving hash functions such as MurmurHash3 and Google FarmHash are designed for hashing single long key efficiently, however.

We optimized MurmurHash3 32- and 128-bit hash functions for batch hashing short keys using AVX and AVX2 SIMD intrinsics. MurmurHash3 was chosen due to its algorithmic simplicity. The vectorized functions hash 8 k-mers concurrently, iteratively processing 32-bit blocks from each. We additionally implemented a 32-bit hash function using the CRC32C hardware instruction for use with local hash tables, where B is unlikely to exceed 2^{32}. While CRC32C shows reasonable performance for our hash tables, detailed analysis of its hash value distribution is beyond the scope of this work.

B. Cardinality Estimation

Hash table resizing is an expensive operation, as elements are reinserted into the expanded table. We estimate the number of unique k-mers and resize the table ahead of batch insertion.

Algorithm 5 Radix-sort Hashing: Find

```
1: T_rs: Radix sort based hash table ⟨P_rs, C_rs, O_rs, I_rs⟩
2: H(): hash function; k: key to find
3: b ← H(k)%B
4: d_0 ← b/q_rs, d_1 ← (b + 1)/q_rs
5: lo ← I_rs[b]
6: if (d_0 == d_1) then hi ← I_rs[b + 1]
7: else hi ← C_rs[d_0]
8: end if
9: for j ← lo ... hi do
10:     e ← P_rs[d_0 × w_rs + j]
11:     if (e.k == k) then return (d_0 × w_rs + j)
12:     end if
13: end for
14: return −1
```

We implemented HyperLogLog++ [27] as its use of 64-bit hash values supports larger genomes and k values. We use a batch mode vectorized MurmurHash3 hash function in HyperLogLog++ updates, and reuse the hash values during table insertion. Precision defaults to 12-bits, corresponding to a 4096-bin register with 8-bit bins that fit easily in L1 cache during batch update. The estimator is integrated into both Robin Hood and Radix-sort hash tables.

C. Software Prefetching

While individual hash table *insert* and *find* operations incur irregular memory access, with batch mode processing, future memory accesses can be computed thus software prefetching can be employed to reduce or hide memory access latencies.

For each operation, we compute the hash values and bucket ids b_w in batch. The software prefetching intrinsics are issued some iterations ahead of a bucket or bin's actual use. The number of iteration is referred to as the prefetch distance. In both Robin Hood and Radix-sort hashing schemes, the arrays $P_{rh/rs}[]$ and $I_{rh/rs}[]$ are prefetched via this approach. The *permute* step of distributed hash table operations likewise benefits from software prefetching.

We began with a problem that suffers from random memory access and poor data locality, and requires large amounts of data to be read from memory. We formulated new algorithms that improves data locality and reduces memory IO requirement, while using software prefetching to convert the latency bound problem to one of bandwidth bound. We also vectorized the hash function computation and restricted hash table size to powers of 2 to avoid expensive arithmetic operations.

VI. Results

1) Experimental setup: Table I details the datasets used in our studies. The shared-memory experiments, including the comparison to existing tools, were conducted on a single-node quad-socket Intel® Xeon®[1] CPU E7-8870 v3 (Haswell) system with 18 cores per socket running at 2.1 GHz and with 1TB of DDR4 memory. All binaries were compiled using GCC 5.3.0 and OpenMPI 1.10.2. We conducted our multinode experiments on the Cori supercomputer (Phase I partition). Each Phase I node has 128 GB of host memory and a dual-socket Intel® Xeon® CPU E5-2698 v3 (Haswell) with 16 cores per socket running at 2.3 GHz. The nodes are connected with Cray Aries interconnect with Dragonfly topology with 5.625 TB/s global bandwidth. We use cray-mpich 7.6.0 and ICC 18.0.0. Shared-memory experiments were not conducted on a Cori node due to memory requirements for the larger datasets.

A. Hash Function Comparisons

We compare the performance of our vectorized implementations of MurmurHash3 32- and 128-bit hash functions with the

[1]Intel and Xeon trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Other names and brands may be claimed as the property of others. ©Intel Corporation. Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

TABLE I
EXPERIMENTAL DATASETS USED FOR ALL EVALUATIONS. WHERE APPLICABLE, ACCESSION NUMBERS FOR NCBI ARE PROVIDED.

Id	Organism	File Count	File Size (Gbytes)	Source	Accession/ Notes
R1	F. vesca	11	14.1	NCBI	SRA020125
R2	G. gallus	12	115.9	NCBI	SRA030220
R3	H. sapiens	48	424.5	NCBI	ERA015743
R4	B. impatiens	8	151	GAGE [28]	Bumble Bee
R5	H. sapiens	6	957	NCBI	SRP003680
G1	H. sapiens	1	2.9	1000 Genome GRCh37 ref.	assembled
G2	P. abies	1	12.4	Congenie.org	assembled

corresponding scalar implementations for different key sizes in Fig. 1. The keys were randomly generated to simulate encoded DNA sequences. As typical length of a k-mer (i.e., value of k) is less than 100, our hash function performance test used keys with power-of-2 lengths up to 64 bytes, sufficient for $k = 256$ with DNA 2-bit encoding and $k = 128$ for DNA-IUPAC using 4-bit per base encoding.

Our vectorized implementations showed up to 6.6× speedup at key length of 4 bytes over the corresponding scalar implementations. At the more common 8 byte key length, the 128-bit vectorized hash function was 3.4× faster than scalar, and the 32-bit function was 5× faster than scalar. As key size increases towards cache line size and beyond, the overhead of reorganizing data across multiple cache lines for vectorization offsets any performance gains. CRC32C was compared to MurmurHash3 32-bit scalar implementation. It was up to 8.1× faster but its hash value uniformity has not been demonstrated hence it is used with local hash tables only.

B. Hashing Schemes

1) Setting the parameters values: We set the parameters of the hashing schemes - Robin Hood hashing (RH) and Radix-sort hashing (RS) - to empirically determined optimal values near the approximate best theoretical values.

For RS, we set B to the expected number of unique keys. We set w_{rs} to 4096 so as to have approximately the largest bins that fit in cache. Thus, we ensure nearly the smallest possible frequency of sorting while still guaranteeing fast *radix sort* performance. We set q_{rs} to 2048, maintaining sufficient space

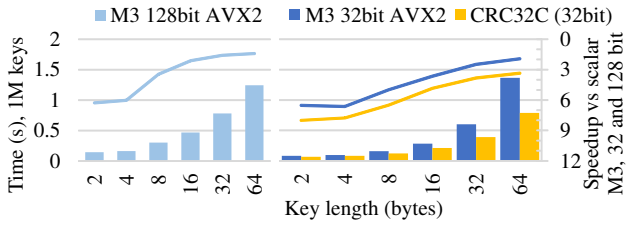
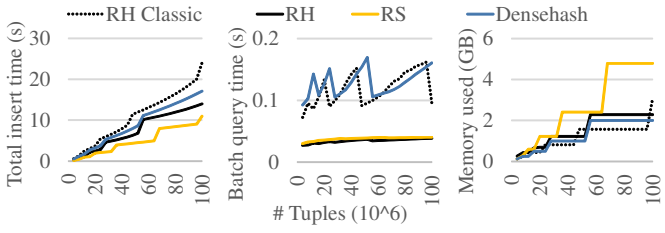


Fig. 1. Time to hash 1 million keys (shown as bars) using AVX2 vectorized MurmurHash3 (M3) 128-bit (left) and 32-bit (right) hash functions, and the speedups versus their respective scalar implementations (shown as lines). Hardware assisted CRC32C based hashing is shown with its speedup values computed relative to the times for the 32-bit scalar MurmurHash3 function.



(a) Insert (b) Find (c) Memory Usage

Fig. 2. Performance of different hashing schemes as hash table sizes increased in increments of 4 million random integer tuples ⟨64-bit key, 32-bit value⟩. Total insertion time, total memory usage, and time to find 4 million keys are reported for each table size. Densehash uses quadratic probing. RH Classic implementation is from https://github.com/martinus/robin-hood-hashing.

for *lazy sorting* to be effective without allowing too much empty space. We set the *prefetch distance* to 8 for $I_{rs}[]$ and 8 and 16 for $P_{rs}[]$ for *insert* and other operations respectively. Prefetching for $I_{rs}[]$ must be completed first, as positions in $P_{rs}[]$ are computed from $I_{rs}[]$ entries.

For RH, all hash table operations begin with accessing $I_{rh}[b]$ for bucket b, followed by conditionally comparing $P_{rh}[b + I_{rh}[b]]$ if $I_{rh}[b]$ indicates an occupied bucket. We therefore set the prefetch distances of $P_{rh}[]$ as 8 and $I_{rh}[]$ at twice that. The load factor is inversely proportional to memory requirement and throughput. We experimented with multiple values of the load factor between 0.5 and 0.9 and found the throughput to be nearly the same for values between 0.5 and 0.8, and decreased for values greater than 0.8. Hence, we picked the value of 0.8 to get nearly the maximum performance with minimum memory footprint.

2) Sequential Comparison of various hashing schemes: Fig. 2 compares the sequential performance of our hashing schemes with the classic Robin Hood (RH Classic) hashing and Google dense hash map (Densehash) as table size grows from repeated insertion. In order to isolate the comparison to only the hashing scheme, we chose the scalar MurmurHash3 128-bit as the hash function, since only our hashing schemes support vectorized hash functions. For RS, for the *insert* plot we performed *finalize insert* only at the end. For the *find* plot, we performed *finalize insert* at the end of every iteration and then performed *find*. Randomly generated 64-bit integers were used as keys and 32-bit integers as values.

In Fig. 2a, the sudden jumps in insertion time were due to hash table resizing and corresponded to the jumps in Fig. 2c. For *insert*, RS significantly out-performed all other implementations, achieving speedups of up to 3.5×, 2.2× and 3.2× over RH Classic, RH and Densehash. RH was up to 2.6× and 2.3× faster than RH Classic and Densehash. For *find*, RS and RH respectively achieved speedups of 4× and 4.3× over Densehash, and 4.3× and 4.7× over RH Classic. The times consumed showed only a small dependence on hash table size and table load in contrast to RH Classic and Densehash. RS consumed significantly higher memory than the other three schemes. The memory consumption of RH, Google dense hash map and RH classic were similar to each other with RH classic

TABLE II
NUMBER OF LOAD MICRO-OPERATIONS AND MISS RATE FOR L1, L2, AND L3 CACHE FOR INSERTING 90 MILLION RANDOM DNA 31-MERS WITH AVERAGE REPEAT RATE OF 8. MURMURHASH3 128-BIT WAS USED.

	Load μOp (10^9)			Miss (%)		
	L1	L2	L3	L1	L2	L3
Densehash	13.05	0.40	0.40	3.09	99.41	90.11
RH Classic	10.35	0.22	0.21	2.06	96.68	83.14
RH	20.67	0.13	0.12	0.62	89.78	82.32
RS	15.12	0.21	0.13	1.43	59.67	71.27

needing the least amount in most cases.

Table II illustrates the effectiveness of RH and RS hashing schemes. The experiment inserted 90 million random DNA 31-mers with an average duplication rate of 8 into Densehash, RH Classic, RH, and RS hash tables configured with 128-bit MurmurHash3. The simulation parameters were chosen to mirror the per-processor k-mer counts when dataset R4 is partitioned to 512 processors. Both RH and RS posted significantly more L1 load operations as well as significantly lower L1 miss rate than Densehash and RH Classic. RH in particular had the lowest L1 miss rate of 0.62%, corresponding to 128 million L1 misses, or equivalently, L2 load operation counts. Higher L1 miss rate in RS was offset by lower L2 miss rates, however, bringing the L3 load operation count to par when compared to that for RH. Densehash and RH Classic L2 load operation counts remained higher than RH and RS. The higher L2 miss rate in Densehash and RH Classic (and to a lesser degree, RH) suggests that the L2 cache was not effective for these schemes. The lower L3 miss rate for RH and RS, coupled with significantly lower L3 load operation counts, contributed to the higher RH and RS performance. Table II demonstrates that generally the Robin Hood hashing scheme reduces cache loads and misses, and RH and RS further improves cache utilization. The *find* operation showed similar cache performance.

C. Speedup with respect to optimizations on a single node

Fig. 3 compares the performance of Kmerind [19] (KI) with RH and RS based k-mer count indices. For RH and RS hashing schemes, each bar includes the cumulative effects of all the optimizations listed to its left. The left-most bar represents the performance using scalar MurmurHash3 128-bit hash function for *Transform, Permute* and *Local compute* stages and with *software prefetching* OFF. The next bar shows the effects of activating *software prefetching*. The subsequent bar shows the impact of vectorized MurmurHash3 128-bit hash function. The last bar shows the performance when the vectorized MurmurHash3 128-bit hash function was replaced with the CRC32C hash function for the *Local compute* stage.

The performance of *Transform, Permute* and *Local compute* stages in *insert* improved with the use of *software prefetching* and *vectorization*. *Software prefetching* improved insertion time by 58.1% for RS, while vectorization resulted in an additional 18.7% improvement. CRC32C further improved the performance by 8.5%. For RH, the incremental improvements were 48.7%, 18.0%, and 3.8% for insertion.

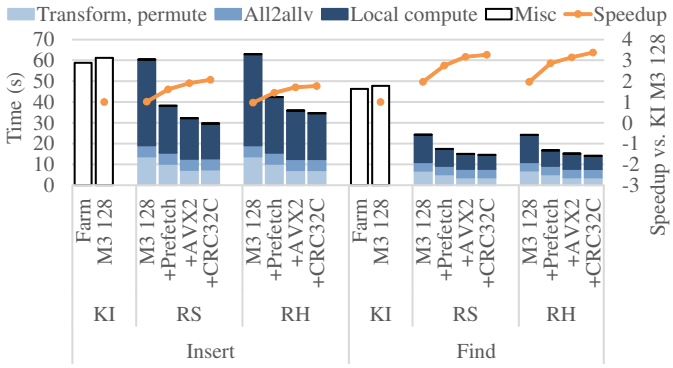


Fig. 3. Effects of optimizations on hash table operations on a single node. M3 refers to MurmurHash3. The line plots depict speedups relative to Kmerind operations with M3 128. MPI ranks: 64, dataset: 1/2 of R4, $k = 31$.

For the *find* operation, the hashing schemes alone accounted for 96.4% and 97.0% improvement over KI for RS and RH, respectively. *Software prefetching* produced an additional 40.0% and 45.1% improvement respectively for RS and RH. Hash function vectorization and CRC32C accounted for another 15.3% and 3.2% improvement for RS, and 10.2% and 7.2% for RH. Overall, RS was 2.1× and 3.3× faster than KI for *insert* and *find* operations, while RH was 1.8× and 3.4× faster than KI for *insert* and *find* respectively. In subsequent experiments, KI used Farmhash, and RS and RH used software prefetching, AVX2 MurmurHash3 128-bit (for *Transform, Permute*), and CRC32C (for *Local compute*).

D. Comparison to existing k-mer counters on a single node

We compared our performance with existing k-mer counters, the majority of which are built for shared memory systems. We include KMC3 [16], Gerbil [17] (GB), and KI in our comparisons as they represent the most performant k-mer counters current available. KMC3 and GB were run with default parameters, up to 512 GB memory, and 1 thread per core. KI, RH, and RS assigned 1 MPI process per core.

File IO and memory access are two important factors in k-mer counting scalability. Fig. 4 compares the strong scaling performance of various tools on total time inclusive of file IO. Scaling efficiency of KMC3 and GB degraded rapidly, reaching only 0.27 at 64 threads in both cases. This is attributable to heavy use of file IO by these tools and synchronization costs.

KI, RH and RS scaled significantly better at 0.65, 0.49, and 0.56 respectively. However, they were hindered by the file IO scaling efficiency (0.36-0.38 at 64 cores) as well. Excluding file IO, KI, RH, and RS have scaling efficiencies of 0.76, 0.56, and 0.69 at 64 cores, respectively. Note that optimizing file IO is beyond the scope of this paper.

Given that these implementations are memory bandwidth bound and RS requires lower memory bandwidth than RH, RS scales better. While KI was significantly slower than RH and RS at 64 cores, it achieved better scaling due to significantly worse performance at lower core count, attributable to a memory latency bound implementation.

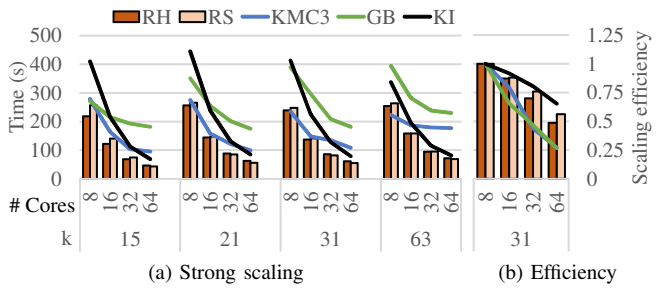
TABLE III
COMPARISON OF TIME CONSUMED (IN SECONDS) BY VARIOUS k-MER COUNTERS ON VARIOUS DATASETS. THE "COUNTING" ROWS EXCLUDE FILE IO TIME. PHYSICAL CORES: 64, $k = 31$.

	K	R1	R2	R3	G1	G2
Total Time (s)	JF	127.84	347.28	1465.86	132.65	329.45
	KMC3	31.80	99.41	455.96	31.03	102.15
	GB	34.83	184.31	696.62	1235.82	153.09
	KI	23.97	77.90	269.96	21.01	70.59
	RH	21.13	66.34	239.59	18.17	61.68
	RS	19.30	58.44	231.90	17.71	61.32
Counting	JF	27.95	215.21	1201.46	14.74	63.93
	KI	12.53	52.06	190.29	8.42	27.84
	RH	9.67	40.58	160.69	5.73	22.12
	RS	8.08	35.22	162.05	5.36	21.74

Next, we compare the performance of these tools on a set of larger datasets (Table III). We included additionally JellyFish [11] as it is a well known k-mer counter. The results show that we obtained significantly better performance, $\approx$15% improvement over the previous state-of-the-art, Kmerind (KI), for most all datasets tested. Compared to KMC3, RS and RH demonstrated at least a speedup of 1.5× and up to 2× for RS with dataset R3. In addition, we observed that nearly all the improvements are in the counting time rather than the file IO times when compared to KI. For smaller read sets, R1 and R2, RS further showed a 10∼20% advantage over RH.

E. Multi-node Scaling

1) Strong scaling: We compare the strong scaling performance of KI with various versions of RS and RH in the left half of Fig. 5. At lower core counts, the benefit of overlapping *Local compute* and communication is evident, while the hybrid MPI-OpenMP version is handicapped by a significantly higher *Wait* time attributable to large message sizes, $O(N/p)$ instead of $O(N/cp)$, where p is the socket count and c is the core count per socket. At lower total core count, a single communicating thread per socket in the hybrid configuration was not sufficient to saturate the network bandwidth. MPI operations in the hybrid MPI-OpenMP case also process messages c times larger than with MPI only. At higher core counts, however, the hybrid version performed well as it reduced the impact of the increased network latency



(a) Strong scaling (b) Efficiency

Fig. 4. Comparison of strong scaling performance of KMC3, GB, KI, RH, and RS on 8–64 shared memory cores. Varying k shows the effects of changing key size. Parallel efficiencies are shown for $k = 31$ as the efficiency behaviors for different k values are similar. Dataset: R2

TABLE IV
L3 CACHE LOADS AND MISS RATE AS A FUNCTION OF PER-PROCESS ELEMENT COUNT (N/p, IN MILLIONS) IN STRONG SCALING SCENARIO. HASH FUNCTION: 128-BIT MURMURHASH3.

	Load μOp (10^6)				Miss rate (%)			
N/p	90	45	22.5	11.3	90	45	22.5	11.3
KI	403.1	197.1	101.6	53.4	90.1	83.2	70.1	51.0
RH	116.9	51.2	28.1	13.2	82.3	80.2	60.7	48.5
RS	129.1	54.9	30.4	13.1	71.3	68.9	51.8	53.0

overhead. For the MPI-only version with overlap, *Wait* time was significantly higher at large core-counts due to reasons discussed in Section III.

As the core count increased in this strong scaling experiment, per-processor element count decreased such that a greater proportion of the hash table became cached. Table IV shows this effect with a simulated hash table insertion experiment that scaled from 90 million DNA 31-mers, corresponding to per-core elements counts for dataset R4 partitioned to 512 cores, down to 11.3 million k-mers per-core for the 4096 core case. The L3 cache miss rates decreased for all hash tables in the simulation as the element counts decreased. Since RH and RS required significantly fewer L3 loads while demonstrating lower L3 miss rate, their memory access latency penalties were lower than that for KI.

The better cache behavior, along with the lower latency overhead of the hybrid version (+MT) at high total core counts, contributed to higher and even super-linear scaling efficiencies in almost all cases. Overall, we achieved significantly higher performance compared to KI, a speedup over KI of up to 2.6× and 4.4× for *insert* and *find* using the overlapped communication and computation strategy at 512 cores. At 4096 cores, the hybrid implementation produced speedups of 2.15× and 2.6× for *insert* and *find*. The differences in the scaling behaviors at low and high core counts suggest that a strategy to dynamically choose an implementation based on core count and network performance. For *insertion*, that point occurs between 1024 and 2048 cores, whereas for *find* the inflection point is close to 4096 cores.

The performance of our implementation relies on balanced communication volume and computation as stated in Section V-A. We encourage load balance through equi-partitioning of the input k-mers and the use of hash functions with approximately uniform output. With R5, MurmurHash 3 and 512 cores, each core is assigned an average of 195.5 million input k-mers with a standard deviation of 1.48 million, or 0.76%. At 4096 cores, the standard deviation was 513.6 thousand, 2.1% of the average of 24.5 million k-mers.

2) Weak scaling: We needed to use a small dataset for strong scaling experiments as we were bound by the available memory at lower node counts, resulting in strong scaling experiments consuming less than 2 seconds in *Local compute* using 4096 cores. For runs this short, wasted cycles in wait time can be overrepresented. Thus, we perform weak scaling experiments using much larger dataset at 4096 cores as shown in the right half of Fig. 5. The results confirm that the wait

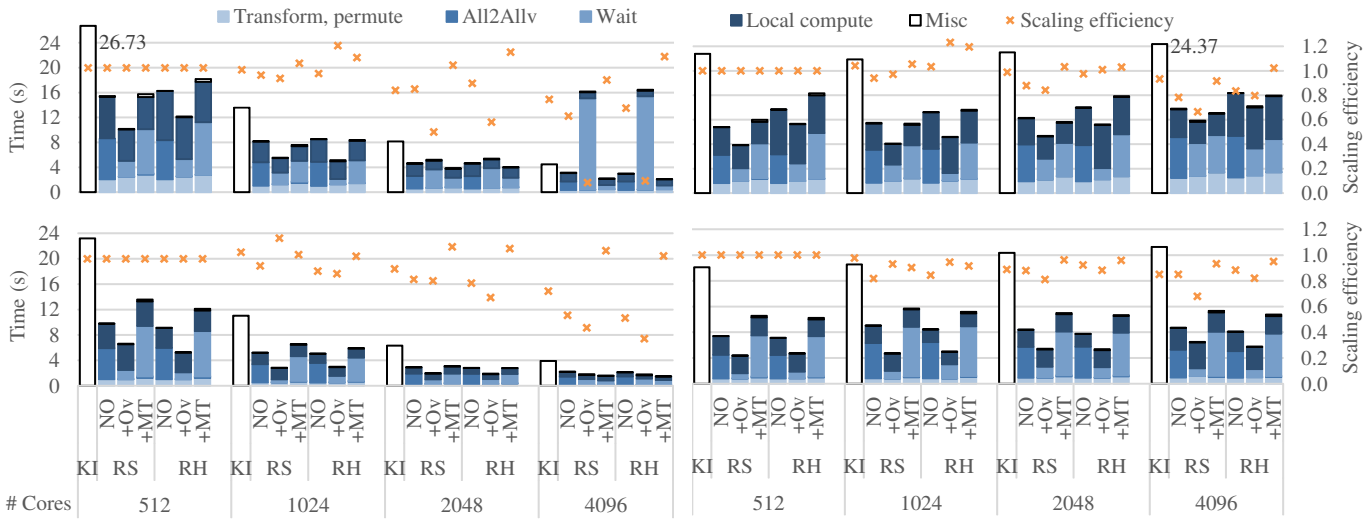


Fig. 5. Scaling experiments on Cori using 16 to 128 nodes. Strong scaling (left column) used dataset R4, weak scaling (right column) used portions of R5. The top row shows *insert* times, and the bottom row shows *find* times. NO and Ov stand for non-overlapped and overlapped communication, respectively. For RS and RH, the left-most bar (NO) corresponds to the best configuration from Fig. 3. The next bar includes overlapped compute and communication (+Ov), and the last bar includes multi-threading (+MT). The markers show the scaling efficiency of each configuration relative to the 512-core run. For the +MT case, we use one MPI rank per socket or NUMA domain, and # cores/socket as # OpenMP threads/rank. For all other cases, # MPI ranks = # cores.

time is overrepresented in the strong scaling results for 4096 cores for the MPI-only version with overlap. Apart from that, the weak scaling results show similar pattern to the strong scaling results. The overlapped communication and computation strategy performs the best. For *insert* and *find*, we achieved a speedup over KI of up to $2.9\times$ and $4.1\times$ at 512 cores and up to $2.06\times$ and $3.7\times$ at 4096 cores.

VII. Conclusion

In this paper we present our work towards optimizing k-mer counting, a critical task in many bioinformatics applications, for distributed memory environments. We optimized tasks at multiple architectural levels, ranging from network communication optimization, to cache utilization and memory access tuning, to SIMD vectorization of computational kernels, in order to create an efficient distributed memory implementation.

We designed our optimizations to take advantage of specific features of k-mers and properties of the k-mer counting task. We vectorized MurmurHash3 to support hashing multiple small keys such as k-mers concurrently. Our AVX2 vectorized hash functions achieved up to $6.6\times$ speedup for hashing k-mers compared to the scalar MurmurHash3 implementation.

The batch-mode nature of k-mer counting allowed us to make effective use of software prefetching, hiding the latencies of irregular memory accesses common for hash tables. We designed and implemented two hashing schemes, Robin Hood and Radix-sort, both aimed to improve cache locality and minimize memory bandwidth usage though through distinct approaches. Our cache friendly hashing schemes addresses both the latency and bandwidth aspects of memory access. With integer keys, our sequential hash tables out-performed Google dense hash map by up to $3.2\times$ for the *insert* operation and $4.3\times$ for the *find* operation.

Finally, we developed a communication primitive for overlapping arbitrary user specified computation with collective all-to-all personalized communication. We also designed a hybrid MPI-OpenMP distributed hash tables with thread-local sequential hash tables that minimizes synchronization costs. Our distributed hash tables have been shown to scale effectively to 4096 cores with high parallel efficiency, and performs index construction and query on a $\approx$1 TB human genome dataset in just 11.8 seconds and 5.8 seconds, respectively, using 4096 cores. Cumulatively, the optimizations contribute to performance improvements over Kmerind of $2.05-2.9\times$ for k-mer count index construction and $2.6-4.4\times$ for query.

As next generation sequencing becomes an increasingly ubiquitous tool in diverse fields of study, the need for low latency and high throughput analysis of bioinformatic data will only increase. Distributed memory algorithms and software offer the potential to address these big data challenges. Our contributions presented in this work, while motivated by k-mer counting in bioinformatics, have broader potential impact for applications that require fast hash functions, cache friendly and memory access efficient hash tables, and high performance and scalable distributed memory parallel hash tables.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. Conflicts of interest: None declared.

References

[1] The Cancer Genome Atlas Research Network, J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The Cancer Genome Atlas Pan-Cancer analysis project," *Nature Genetics*, vol. 45, no. 10, pp. 1113–1120, 2013.

[2] The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[3] K. Koepfli, B. Paten, the Genome 10K Community of Scientists, and S. J. O'Brien, "The Genome 10k Project: A way forward," Feb. 2015.

[4] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[5] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: A parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[6] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "HipMer: An extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 14:1–14:11.

[7] S. Kurtz, A. Narechania, J. C. Stein, and D. Ware, "A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes," *BMC Genomics*, vol. 9, no. 1, p. 517, 2008.

[8] D. R. Kelley, M. C. Schatz, and S. L. Salzberg, "Quake: Quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, p. 1, 2010.

[9] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: Representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[10] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, "CLARK: Fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers," *BMC Genomics*, vol. 16, p. 236, 2015.

[11] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[12] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a Bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 1, 2011.

[13] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.

[14] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent k-mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014.

[15] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown, "These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure," *PloS ONE*, vol. 9, no. 7, p. e101271, 2014.

[16] M. Kokot, M. Długosz, S. Deorowicz, and B. Berger, "KMC 3: Counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, Sep. 2017.

[17] M. Erbert, S. Rechner, and M. Müller-Hannemann, "Gerbil: A fast and memory-efficient k-mer counter with GPU-support," *Algorithms for Molecular Biology*, vol. 12, no. 1, p. 9, Mar. 2017.

[18] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de Bruijn graphs," *BMC Bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[19] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. PP, no. 99, pp. 1–1, 2017.

[20] P. Celis, P.-A. Larson, and J. I. Munro, "Robin Hood hashing," in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, ser. SFCS '85. Washington, DC, USA: IEEE Computer Society, 1985, pp. 281–288.

[21] P. Celis, "Robin Hood hashing," Ph.D. Thesis, Department of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1986, tech report CS-86-14.

[22] "NVBIO," https://nvlabs.github.io/nvbio/, last accessed May 31, 2018.

[23] N. Mcvicar, C. C. Lin, and S. Hauck, "K-mer counting using Bloom filters with an FPGA-attached HMC," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2017, pp. 203–210.

[24] D. Jünger, C. Hundt, and B. Schmidt, "WarpDrive: Massively parallel hashing on multi-GPU nodes," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[26] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing," in *Distributed Computing*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sep. 2008, pp. 350–364.

[27] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: ACM, 2013, pp. 683–692.

[28] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marçais, M. Pop, and J. A. Yorke, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome Research*, vol. 22, no. 3, pp. 557–567, 2012.

APPENDIX

A. Abstract

We present the artifacts associated with the paper *Optimizing High Performance Distributed Memory Parallel Hash Tables with Application to DNA k-mer Counting*. Two primary categories of artifacts were produced in this work. The first includes the C++ and MPI-based software implementations described in ths paper, structured as a reusable header-only library, and the applications that utilize these library components. The second category of artifacts consists of execution and analysis scripts for the experiments and results described in the paper, as well as the experimental data themselves. Both categories of artifacts are being released publicly. We describe the artifacts, the process to obtain them, and the software installation and invocation procedures in order to facilitate the reproduction the presented experiments and results.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** AVX2 vectorized MurmurHash3 hash functions, optimized Robin Hood hashing algorithm, Radix-sort based hash table, and collective all-to-all communication implementation with overlapped generic computation.
- **Program:** C++ templated header-only libraries for the hash function and hash table implementation, and standalone demonstration application for k-mer counting.
- **Compilation:** CMake, make, and a C++ 11 compiler (GNU C++, Clang, or Intel C++ compiler)
- **Binary:** Sequential and MPI unit tests, and MPI applications
- **Data set:** See Section B5.
- **Run-time environment:** RedHat Enterprise Linux 6.x, Cray Unix
- **Hardware:** 64-bit x86 CPU with AVX2 support, MPI supported interconnection network (InfiniBand, ethernet)
- **Run-time state:** For benchmarks, exclusive reservation of compute resources, and preferably of parallel file systems and interconnection network
- **Execution:** via job schedulers scripts or interactive invocation, using `mpirun` with appropriate arguments
- **Output:** k-mer counts can be saved as binary files. Timing results are written to console and can be redirected to log files.
- **Experiment workflow:** Please see Section D
- **Publicly available?:** Yes

2) How software can be obtained (if available): The k-mer counting application is available from the git repository https://github.com/tcpan/kmerhash under the *sc2018* branch. All software are Licensed under the Apache version 2.0 License. To obtain the software, invoke the command:

```
git clone --recurse-submodules \coderepo --branch sc2018
      --single-branch
```

The git repository https://github.com/tcpan/kmerhash-sc2018-results contains the scripts to run the experiments and extract relevant data from the experimental logs. Additionally, the repository contains the Microsoft Excel spreadsheets used to generate the tables and figures in this paper. The Excel figures are covered by the same IEEE copyright as this paper.

3) Hardware dependencies: Our software has been tested on 64-bit systems and clusters with Intel Haswell, Broadwell, and Skylake CPUs. AVX 2 support is required. Fast interconnect, e.g. QDR Infiniband, is recommended when the software is used in a distributed memory environment.

4) Software dependencies: We compiled and tested our software on Linux platforms. Our software depends on components from the *Kmerind* library and its dependencies. All dependencies are handled as git submodules.

Compilation: The software requires `CMake` version 2.8 or later for configuration and `make` for building. Tested compilers include GNU g++ 4.9.3 and later, Clang++ 3.7 and later, and ICPC 18.0.0. Compiler support for OpenMP is required. OpenMPI, MPICH, MVAPICH, or Cray MPICH headers and libraries are required

Execution: MPI applications in https://github.com/tcpan/kmerhash depends on MPI version 2.0 compliant runtime. OpenMPI, MPICH, MVAPICH, and Cray MPICH have been tested. SLURM job scripts are included in https://github.com/tcpan/kmerhash-sc2018-results for references. For cache performance profiling, Intel VTune is required.

Comparisons: We compared our algorithm implementation to Kmerind, JellyFish, KMC, and Gerbil. The specific versions used can be obtained from:

- Kmerind revision $d9c7c7b$: https://github.com/ParBLiSS/kmerind
- JellyFish 2.2.3: http://www.genome.umd.edu/jellyfish.html
- KMC 3.0: http://sun.aei.polsl.pl/REFRESH/index.php?page=projects&project=kmc&subpage=download
- Gerbil 1.0: https://github.com/uni-halle/gerbil

Data Analysis: The experimental data is primarily analyzed using `grep`, shell scripts, and Microsoft Excel.

5) Datasets: The datasets used for the experiments are listed in Table I in the paper. The datasets can be downloaded from the following URLs using the accession or project numbers listed in the table.

- R1-R3, R5: https://www.ncbi.nlm.nih.gov/sra
- R4: http://gage.cbcb.umd.edu/data
- G1: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13
- G2: ftp://plantgenie.org/Data/ConGenIE/Picea_abies/v1.0/FASTA/GenomeAssemblies/Pabies1.0-genome.fa.gz

For datasets R1, R2, R3, and R5, the `SRA Toolkit` is used to convert from `SRA` to `FASTQ` format.

C. Installation

The following commands are used to configure the software with default settings and to compile the software. The same `cmake` command is executed twice in order to ensure that the parameters are propagated.

```
mkdir build
cd build
cmake {src directory} -DCMAKE_BUILD_TYPE=Release
cmake {src directory} -DCMAKE_BUILD_TYPE=Release
make -j8
```

CMake will attempt to detect the installed C++ compiler and MPI library. To use an alternative compiler or MPI library, the environment variables `CC` and `CXX` can be set. Alternatively, `ccmake` can be used for interactive configuration of the build parameters. The default parameters translates to the following compiler flags: `-march=native -O3 -std=c++11`.

For cache miss rate profiling, Intel VTune and additional CMake parameters are required

```
-DENABLE_VTUNE_PROFILING=ON
-DVTUNE_LIB="absolute path to libittnotify.a"
```

Software used in comparisons, including KMC, Gerbil, and JellyFish, may require additional installation according to the instructions accompanying the respective distributions. Kmerind's k-mer counting application is replicated in our software and no additional installation is required.

D. Experiment workflow

The generated binaries are in `build/bin` directory. A collection of MPI executables are generated, each corresponds to a set of template specializations. The executables are named with the following pattern:

```
{prefix}KmerIndex-{format}-a{alphabet}-k{k}-CANONICAL-{
    map}-COUNT-dtIDEN-dh{distributed_hash}-sh{
    local_hash}
testKmerCounter-{format}-a{alphabet}-k{k}-CANONICAL-{map
    }-COUNT-dtIDEN-dh{distributed_hash}-sh{local_hash}
```

The `testKmerCounter-` variant counts the k-mer incrementally in memory-limited settings. Supported prefix strings include:

- `noPref_`: software prefetching is turned off
- `overlap-`: overlapped communication/computation is turned on
- `test`: overlapped communication/computation is turned off

The executable name parameters have the following allowable values:

- `format`: supports `FASTA` and `FASTQ`
- `alphabet`: 4, 5, and 16 corresponding to DNA, DNA5, and DNA16 respectively.
- k: supports 15, 21, 31, and 63 for scaling experiments. Additional k values can be added by modifying `benchmark/CMakeLists.txt` in the source directory.
- `map`: supports `DENSEHASH`, `BROBINHOOD`, and `RADIXSORT`, as well as the multi-threaded variants `MTROBINHOOD` and `MTRADIXSORT`.
- `distributed_hash`: supports `MURMUR`, `MURMUR32`, `MURMUR64avx`, `MURMUR32avx`, and `FARM`
- `local_hash`: supports all `distributed_hash` values and `CRC32C`.

MPI experiments should account for NUMA memory architecture through binding processes to cores. With OpenMPI we used the command

```
mpirun -np {P} --map-by ppr:{cores}:socket
       --rank-by core --bind-to core
       {executable} [params...]
```

Sequential hash function and hash table benchmarks are

```
benchmark_hashes
benchmark_hashtables_{local_hash}
benchmark_hashtables_grow
```

where `local_hash` is the same set as those used by the distributed k-mer counters.

All binaries support the command line parameters `-h` and `--help` to produce the list of available commandline parameters and their meanings.

SLURM job scripts for all experiments can be found in https://github.com/tcpan/kmerhash-sc2018-results `/scripts/run`. Please see the scripts for example commandline parameters, and use them as templates. The job scripts will likely require customizations for each system, for example the directory names and software installation locations will need to be modified.

E. Evaluation and expected result

The benchmark logging facility captures timing and memory usage data on each MPI process and performs simple aggregation when reporting results.

The benchmarking facility captures the cumulative and elapsed times ("cum_*" and "dur_*"), the current and peak memory ("curr_*" and "peak_*") used, as well as the number of data elements processed ("cnt_*") by each MPI rank for a block of instrumented code. For each measurement, the minimum (min), maximum (max), mean (mean), and standard deviation (stdev) across all MPI ranks are computed then output to console by MPI rank 0 process. The console output can be redirected to a log file. Sequential experiments report identical minimum, maximum, and mean for each measurement, and the standard deviation is 0.

An example log file snippet for the elapsed time is shown below.

```
[TIME] open     header (s)      [,read,]
[TIME] open     dur_min [,30.772840450,]
[TIME] open     dur_max [,30.773302853,]
[TIME] open     dur_mean        [,30.772997589,]
[TIME] open     dur_stdev       [,0.000146621,]
```

The time and memory usage information are extracted from the log files for Robin Hood and Radix-sort hashtable results and Kmerind output, using `grep` and reformatted into tabular form. For KMC3, Gerbil, and JellyFish, https://github.com/tcpan/kmerhash-sc2018-results `scripts/analysis` contains shell scripts to extract and format the relevant timing and memory usage data from their respective log files.

The extracted results are imported into Microsoft Excel for further summarization and visualization. Expected results depend on the CPU, cache memory hierarchy, file system, the network performances, and other environmental factors. Our reported results in the Excel files in https://github.com/tcpan/kmerhash-sc2018-results in https://github.com/tcpan/kmerhash-sc2018-results can serve as references.