Parallelizing Big De Bruijn Graph Construction on Heterogeneous Processors

Shuang Qiu
The Hong Kong University of Science and Technology
sqiuac@cse.ust.hk

Qiong Luo
The Hong Kong University of Science and Technology
luo@cse.ust.hk

Abstract—**De Bruijn graph construction is the first step in de novo assemblers to connect input reads into a complete sequence without a reference genome. This step is both time and memory space consuming. To address this problem, we develop ParaHash, a system that partitions the input data in a compact format, parallelizes the computation on both the CPUs and the GPUs in a single computer, and performs hash-based De Bruijn graph construction. This way, ParaHash utilizes all available processors to assemble big genomes that cannot fit into memory. Furthermore, we analyze the characteristics of genome data to set the hash table size, design concurrent hashing algorithms to handle the inherent multiplicity, and pipeline the data transfer and the computation for further efficiency. Our experiments on real-world genome datasets show that the workload was balanced across heterogeneous processors, and that ParaHash was able to construct billion-node graphs on a single machine with an overall performance up to 20 times faster than the state-of-the-art shared-memory assemblers.**

I. Introduction

The Next Generation Sequencing (NGS) machines produce a vast number of short fragments, called **reads**, from genome samples. Before these reads can be used for whole genome studies, we need to assemble them into a complete genome. De novo assembly is the current method of assembling short reads into a full-length genome sequence, without any reference sequence.

The first step in de novo assembly is to construct a De Bruijn graph on the input reads. A De Bruijn graph is defined on an alphabet, associated with a dimension K. The vertices are all possible length-K strings on the alphabet, and edges are length-(K-1) strings that represent the overlaps between each pair of vertices. An **overlap** between a pair of strings is one string's suffix and the other's prefix. In practice, a De Bruijn graph for de novo assembly is a subgraph of a K-dimensional De Bruijn graph, defined on the alphabet $\Sigma = \{A,C,G,T\}$.

Constructing a De Bruijn graph for a big genome requires hundreds of gigabytes or even more memory, which is quite expensive. As such, some solutions [1] partition and construct big graphs with large distributed memory from many machines. Other approaches, e.g., the Minimum Substring Partitioning [2], seek space efficient representations of the graph. However, the time efficiency of such methods remains unsatisfactory, taking hours to construct a billion-node graph on a commodity machine.

In this paper, we propose to parallelize the De Bruijn graph construction on a single commodity computer, consisting of both CPUs and GPUs, to improve the time performance, while keeping the memory requirement low. To handle big genomes, we choose the Minimum Substring Partitioning algorithm for graph partitioning, because it generates intermediate data in a compact format and reduces disk transfer. For time efficiency, we use hashing for constructing the De Bruijn graph, because it is efficient in merging duplicate vertices and recording adjacencies for each distinct vertex.

Most existing parallel assemblers use separate threads to handle separate hash tables. This approach avoids performance penalty from data contention. However, the concurrency in this approach is limited by the number of hash tables [3]. In contrast, we design a single hash table for concurrent access of all threads. Every entry in the hash table is in the form of <vertex, list of edges>. Potentially, our approach may incur high contention on the hash table; however, with a smart data partitioning scheme and fine-grained treatment of read and write access patterns, we can achieve high concurrency on hashing.

Concurrent hashing has been proposed in [4], but it is not suitable for our scenario where duplicate entries in each bucket need to be counted. Furthermore, non-blocking concurrent hashing relies on the compare-and-swap atomic instructions on the hash entry with machine-word length [5]. As a result, the number of hash entries is limited and conflict may occur for large data sets.

Our concurrent algorithm is based on two observations: (1) the total number of vertices in the graph can be estimated based on the input size; (2) operations in each bucket in the hash table follow a pattern of one-insertion multiple-updates. Therefore, we design fine-grained partial-locking based parallel hashing algorithms for operations on the De Bruijn graph construction. Additionally, the type of our hash table entry is not limited by the machine word size, so that we can implement the algorithms for big genomes on both the CPU and the GPU.

We utilize the data parallelism in both data partitioning and hashing. To further improve the overall performance, we use heterogeneous processors to co-process the computation, and pipeline computation with data transfer. We develop a work-stealing based algorithm to pipeline the input-computation-output for partitioned data processing on heterogeneous processors. This way, we dynamically distribute the workload for both graph partitioning and hashing to the CPU and the GPU based on their processing speeds.

In summary, we make the following contributions:

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.250

IEEE computer society

- We parallelize the Minimum Substring Partitioning (MSP) algorithm for both the GPU and CPU, and further reduce disk IO overhead with data encoding.
- We develop ParaHash - a parallel system that utilizes both the CPU and the GPU, and pipelines the data transfer and computation for further efficiency.
- We design concurrent hash algorithms on both CPU and GPU, and experimentally study the performance with real-world genome datasets. End-to-end comparison shows that ParaHash is up to 20 times faster than the counterparts in the state-of-the-art shared-memory assemblers. Furthermore, it can efficiently construct De Bruijn graphs with billions of vertices on a single machine.

II. Background and Related Work

A. Preliminaries

The input files for assembly are plain text, containing reads that collectively cover almost every DNA **base pair** (**bp**) in a genome [6]. A base pair, e.g., A-T or C-G, is the unit for measuring both the read length and the genome size. The genome size or length, denoted as **Ge**, is the total number of DNA base pairs. We denote the number of reads to be **N** and the length of the read to be **L**.

A length-**K** substring generated from a read is called a **kmer**. A kmer generated from the input reads is called a **candidate** vertex in the De Bruijn graph. Suppose an input file contains N reads with a length L, then these reads produce $N(L-K+1)$ kmers with a length K. All these kmers are candidate vertices. Since the input contains tens of replicates for the original genome, there are many **duplicates** in these candidate vertices, and the duplicates are to be merged into **distinct** ones with duplicity information recorded. For each distinct vertex v, its adjacent vertices are detected by searching all possible length-K strings with their $(K-1)$ bps' suffix or prefix being $v's$ prefix or suffix.

Fig. 1 gives an example of the De Bruijn graph structure. Reads in this example are of length 23, and kmers are of length 5. Each read generates 19 kmers, and they are candidate vertices. The three identical kmers are marked in the rectangles in the reads. As duplicate vertices, they are merged into the second distinct vertex in the graph. Since $TGATG$ overlaps $GATGG$ and $GATGA$, two directed edges are added in the graph to represent the adjacency lists. As the number of occurrences for $TGATG->GATGG$ is two and the number of occurrences for $TGATG->GATGA$ is one, we record the multiplicities of these two edges respectively as two and one. In reality, a DNA sequence has a reverse complement. Therefore, a vertex in the graph is represented by the canonical kmer, i.e., the lexicographically smaller one, of a kmer and its reverse complement. Consequently, the constructed graph is commonly bi-directed [7], [8].

In de novo assembly, reads and kmers are treated as text strings, and characters in the input reads are transformed to characters in the alphabet $\Sigma = \{A, C, G, T\}$. All the unknown DNA bases are transformed to "As" in most assemblers. We denote a read of length L as $S[0, L-1]$. Then a substring starting from position i and ending at position j is $S[i, j]$.

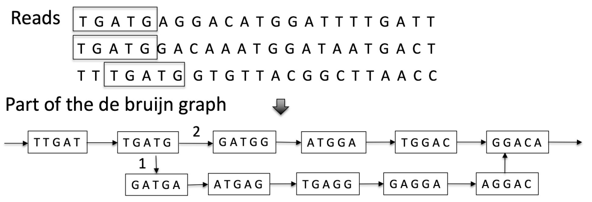


Fig. 1: An Example of De Bruijn Graph Structure

The partial order of reads and kmers is defined on the lexicographical order. We further define three characteristic substrings in the input reads: **P-minimum-substring** (Definition 1), **superkmer** and **minimizer** (Definition 2). The minimizer is commonly used as an identifier for kmers [3], [9], and further utilized in the De Bruijn graph partitioning algorithm [2]. We formalize the De Bruijn graph construction problem in Definition 3.

Definition 1: Let P be a natural number and $1 \le P \le K$. Given a kmer $S[i, i+K-1]$, if a length-P substring $S[j, j+P-1], 0 \le i \le j \le i+K-P$, is the minimum one among all length-P substrings in $S[i, i+K-1]$, then $S[j, j+P-1]$ is the **P-minimum-substring** for kmer $S[i, i+K-1]$.

Definition 2: A **superkmer** is a substring $S[i, j], 0 \le i \le j \le L-1, j-i \ge K$, from a read, in which $S[i, i+K-1], \ldots S[j-K+1, j]$ share a common P-minimum-substring s. s is called the **minimizer** of the superkmer $S[i, j]$.

Definition 3: Given a set of input reads R, the **De Bruijn graph construction** outputs a set of vertex adjacency lists DBG, in which the adjacency between a pair of vertices is defined as a $(K-1)$bp overlap that exists in a read in R. For each vertex v in DBG, each of its adjacent vertex v' has a weight defined as the number of occurrences of $< v, v' >$.

B. De Bruijn Graph Construction

The De Bruijn graph construction is to merge duplicate vertices, and count the number of occurrences for adjacent vertices, recorded as the edge weights. Edge weights are used in determining the traversal paths for assembly. Unfortunately, most standalone De Bruijn graph construction tools omit recording the edge weights [2], [5], [10]; they only count the number of distinct vertices.

A common method for detecting and merging duplicate vertices is hashing. It builds a hash table that absorbs the duplicate vertices into one entry slot in the form of <vertex, list of edges>. Edge weights are also recorded when inserting or updating entries. Intuitively, the hash based algorithm requires a global hash table to record the entire input. However, this table is potentially too big to fit into memory.

Another method of duplicate detection and merging is sort-merge. Kmers and their adjacent vertices are generated as <vertex, edge> pairs, and they are sorted by the kmers. Duplicate vertices are merged and different corresponding edges are appended to the same slot. Substituting the sorting algorithm with merge sort, this approach provides a partition-sort-merge solution that can be processed with multiple passes. However, such multi-pass approach requires multiple inter-partition communication, which is expensive for both disk based and distributed memory based implementations.

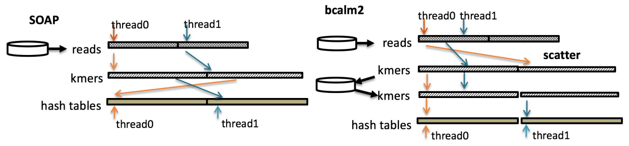


Fig. 2: Comparison of parallel graph construction

C. *Open Challenges in Parallel Graph Construction*

To the best of our knowledge, SOAPdenovo (SOAP) [11] is the most widely-used shared-memory parallel assembler for its efficiency. In the De Bruijn graph construction, SOAP first gets reads from disk and generates all kmers in main memory. When hashing kmers, each thread reads all kmers once and inserts kmers into the thread's local hash table. Since SOAP requires the all hash tables to fit into main memory, memory size becomes a major problem when dealing with big genomes.

To address the big memory assumption, some assemblers utilize distributed memory on many compute nodes [1], [8], [12], [13]. Latest high-performance assemblers, e.g., SWAP-Assembler (SWAP) [1], require clusters with high-bandwidth networks. Such distributed assemblers do not focus on the parallel efficiency on a single machine.

Other solutions explicitly partition the De Bruijn graph to multiple subgraphs, and output subgraphs to disk files. In particular, bcalm2 [3] uses the minimizer-based graph partitioning algorithm to construct and compact the De Bruijn graph. Such partitioning-based method addresses the memory consumption problem to construct big De Bruijn graphs on a single machine. However, they introduce extra graph partitioning work and disk IO overhead, and such overhead leads to considerable performance degradation.

From the state-of-the-art solutions, we observe the following performance issues:

- Parallel assemblers do not fully utilize processor resources on a single machine, especially the heterogeneous many-core processors.
- Existing partitioning based assemblers on a single machine are efficient in memory consumption, but sacrifice the running time performance.

A comparison of the parallel De Bruijn graph construction algorithms is illustrated in Fig. 2. A common strategy in the parallelization is to assign each thread to build a separate local hash table, avoiding write contention in hashing kmers. However, this strategy limits the degree of parallelism, as the number of parallel threads should be no greater than the number of separate hash tables. Due to this reason, work on GPU-based De Bruijn graph construction adopts the sort-merge strategy [14], [15], [16], as there is no concurrent hashing solution to fully exploit the high parallelism in many-core processors.

Related work on concurrent hashing. Recent work optimized the hash join on multi-core CPUs by partitioning input keys and making each partition fit into the cache [17]. Radix and hash algorithms were also parallelized and compared with SIMD (Single Instruction Multiple Data) vectorization [18]. Implementing a hash table on the GPU is even more challenging because of the SIMT (Single Instruction Multiple Threads) execution mechanism of the GPU architecture. A parallel hash table on the GPU was proposed by Alcantara et al. [19]. However, their hash key entry was limited to a machine word size, which is not applicable to the De Bruijn graph. Furthermore, the multiple values for a key in the table were not merged and counted.

Related work on multi-words hashing. Since the kmer length can be set from several base pairs to tens of base pairs, a kmer should be stored in multiple memory words. Inserting or updating a <kmer, list of edges> entry in the hash table causes data contention among concurrent threads. Early work implemented a non-blocking open addressing hash table with mulit-word entries [4]. It provides a single-writer multiple-reader mechanism with assistance of probe bound and state flags. However, this lock-free algorithm cannot count the edges' multiplicities, because it only ensures a successful insertion for duplicate <key, value> pairs, whereas we need to store the number of occurrences of the duplicate <key, value> pairs in the graph construction.

III. PARAHASH

A. *System Overview*

We design a parallel system - ParaHash - to construct big De Bruijn graphs efficiently on a single computer with both multi-core CPUs and the GPUs. We do not assume that the entire graph fits into machine memory. Instead, we use the Minimum Substring Partitioning (MSP) algorithm [2] to first partition the entire graph into subgraphs, and then construct each subgraph concurrently. We use superkmer partitions to represent subgraphs, because it is more space-saving than kmer partitions.

Our system consists of two steps. Each step processes the input and output partition by partition. To improve the time performance, we adopt the following two-level optimizations:

(1) At an intra-partition level, we utilize data parallelism and develop parallel implementations on both the CPU and the GPU. Furthermore, we propose a concurrent hashing algorithm for De Bruijn graph construction. To fully utilize the thread parallelism, we design two optimizations for the efficiency of our concurrent hashing: first, the upper bound for the graph size is estimated, such that the costly hash table resizing is avoided; second, a state transfer mechanism is used for the hash entry insertion / update, such that the lock contention on concurrent write / read is reduced by 80% on the real-world datasets.

(2) At an inter-partition level, we develop a work-stealing based queuing algorithm to dynamically distribute computation on partitions to heterogeneous processors. It also pipelines the disk data transfer and the computation. This way, the overall efficiency is further improved, and the workload is balanced among processors.

The workflow in ParaHash is shown in Fig. 3. MSP (Step 1) is the De Bruijn graph partitioning step, and Hashing (Step 2) is the subgraph construction step. The number of partitions is determined by the available memory in the machine and the parameters we set for performance. In Step 1, ParaHash partitions the input file (fastq or fasta) to equal size and

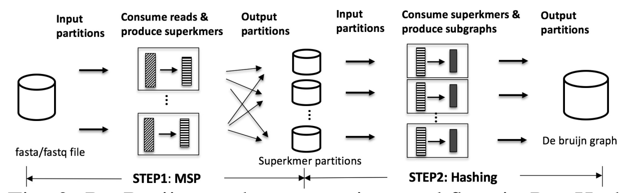


Fig. 3: De Bruijn graph construction workflow in ParaHash

extract reads from the partitions. From each input partition, it generates a subset of superkmers for each of the superkmer partitions, and the superkmer partitions are accumulated as the entire input is processed. In Step 2, it inputs the superkmer partition files, and constructs a subgraph for each partition. Since the cutting edges among subgraphs are maintained during Step 1, all subgraphs generated in Step 2 together constitute the entire De Bruijn graph.

In both steps, an input partition is transferred to the main memory on the CPU. Then an idling CPU or GPU consumes this partition and produces an output partition. This output partition is subsequently transferred to disk by the CPU. Each step finishes its work until all the input partitions are processed and all the output partitions are generated. We use asynchronous data transfer to pipeline the data input / output, the CPU computation, and the GPU computation.

B. De Bruijn Graph Partitioning

In order to partition the graph construction problem into non-overlapping subsets, duplicate vertices, i.e., the equal kmers, must go to the same partition, and the adjacent vertices must be recorded as well. Directly generating and partitioning kmers to disks creates an output with tens of times of the input read size. Recording the adjacency lists for these kmers incurs even more overhead in both the space cost and the data transfer.

The high redundancy in the string vertices from input data is thus exploited to reduce the storage space. A key observation is that, since adjacent kmers overlap, they are likely to share a common P-minimum-substring (Definition 1). Hence a superkmer compacts adjacent kmers from the size $O(MK)$ to size $O(M+K)$, where M is the number of kmers contained in a superkmer. Minimizers [9] were used to partition kmers with a compacted storage.

Therefore, the MSP algorithm in [2] uses superkmer partitions to count distinct vertices for big De Bruijn graph. However, their output superkmers lost information for recording adjacent vertices. As such, the final De Bruijn graph cannot be constructed from the superkmers. To fix this problem, we modify their MSP algorithm by appending two extra base pairs to each superkmer to record the adjacency information in both directions, so that we can construct the entire De Bruijn graph from partitions.

Our MSP-based graph partitioning algorithm partitions the De Bruijn graph based on input reads. We generate superkmers read by read and these superkmers represent pairs of adjacent vertices. Each superkmer is assign an ID and written to the corresponding partition file. The superkmer ID is a value computed from the minimizer's hash bit value with a modulo of the number of partitions. The minimizer of each superkmer is identified by searching and comparing every length-P substring in a read. Since the identical vertices share the same minimizer, all duplicate vertices are finally stored in the same superkmer partition, and their adjacent vertices are appended in the extra base pairs. This way, we partition the entire graph into subgraphs, and each subgraph is stored in a superkmer partition.

To further reduce the output superkmer partition size and the involved disk IO overhead, we encode superkmers with bit values. Since the input reads are transferred to characters in the alphabet Σ, a character in reads or superkmers can be represented with $log_2 4$ bits, i.e., 2 bits. Our encoded output in the MSP step cuts the storage space to about $1/4$ of the size of the non-encoded counterpart in [2].

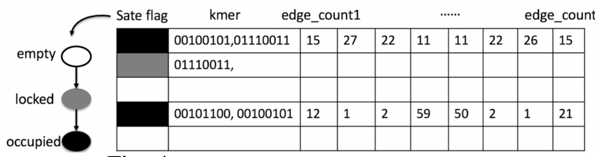
C. Subgraph Construction

We construct the subgraphs from superkmer partitions, and build a hash table for each subgraph construction. When candidate vertices, i.e., kmers, are generated from superkmers, they are indexed by their bit values and a hash function. Duplicate vertices are merged to the same hash table slot with hash entry lookup / insertion / update. The adjacent vertices for each distinct vertex are also inserted and counted in the entry.

To utilize all the available compute cores on the processor for each subgraph construction, we need to develop a concurrent hashing algorithm. However, there are two major concerns that impact the hashing performance. First, resizing a hash table is necessary when the number of elements can not be predetermined; unfortunately, rebuilding the hash table is expensive. Second, the number of contentious data reads and writes intuitively depends on the number of duplicate vertices; it is tens of times of the graph size, i.e., the number of distinct vertices.

To address these problems, we utilize both the characteristic of genome data and the hashing operation patterns in the algorithm. We analyze the expected number of distinct vertices in the graph and allocate enough space for each partitioned hash table, thus the costly hash table resizing is avoided. Furthermore, we limit the number of contentious multi-word operations with the number of distinct vertices, using a state-transfer mechanism. Since the number of distinct vertices is roughly $1/5$ of the entire set, we reduce the contentious lock on the keys by 80% (shown in Table I). Based on these two optimizations, we design the concurrent lookup / insertion / update algorithm, and construct a subgraph with the open-addressing hash table.

1) ***Expected Number of Distinct Vertices:*** An error-free input for the De Bruijn graph construction will produce a graph with the number of distinct vertices to be the same as the genome size. However, the practical input reads contains errors in some base pairs, e.g., the original read $ATT\mathbf{G}GCCA$ is read as $ATT\mathbf{C}GCCA$. Since a single error in a read generates a maximum number of K erroneous kmers, one erroneous base pair results in tens of erroneous kmers. Each erroneous kmer is likely to be a distinct vertex in the graph, and these erroneous vertices can only be filtered by the number of their occurrences



Sate flag	kmer	edge_count1			......				edge_count8
⬛	00100101,01110011	15	27	22	11	11	22	26	15
▨	01110011,								
⬛	00101100, 00100101	12	1	2	59	50	2	1	21

Fig. 4: State transfer in hash table entries

after the graph is constructed. As a larger input brings more erroneous kmers, the number of distinct vertices in the De Bruijn graph is proportional to the big input size.

To estimate the graph size more precisely, we calculate the expected number of distinct vertices. This result helps to estimate the hash table size, which determines the space allocation for the hash tables. Hence the subgraphs can be constructed efficiently without resizing the hash tables.

***Property** 1:* Assume that sequencing error occurrences are independent from each other. Given a sequencing error rate, the event that the number of errors occurs in a read follows a Poisson distribution. Then the expected number of vertices in a De Bruijn graph is $\Theta(\frac{\lambda}{4}LN + Ge)$, where λ is the average number of errors in a read. (See Proof in Appendix.)

In practice, the average number of errors in a read is one or two bps ($\lambda = 1,2$) [20]. Intuitively, the expected number of distinct vertices can be as large as $\Theta(LN)$, when $K << L$. With this upper bound, we estimate the graph size from the input data, and further estimate the subgraph size from each superkmer partition size. Therefore, we reduce the expected hash table size by half, when $\lambda = 2$.

*2) **Hash Table Lookup / Insertion / Update**:* For each superkmer in a partition, we generate multiple <kmer, edge> pairs according to the superkmer length, and insert the <kmer, edge> pairs in the hash table. Our result outputs the adjacency list in a form of <vertex, list of edges>, where the vertex is the key and the list of edges is an array to store the multiplicities for edges, i.e., the number of occurrence for each pair of adjacent vertices. Since there are $(K-1)$ overlapping characters between two adjacent vertices, only the rightmost or leftmost character on the destination vertex is needed to represent the edge, and this character is used as the array index.

*3) **State transfer mechanism for partial locking**:* To insert or update a vertex in the graph involves memory reads / writes with multiple words. In this situation, the memory should be locked each time a read or write occurs. It incurs performance penalty in that memory is accessed sequentially by threads. However, we can reduce the lock contention significantly by locking partial data structure, i.e., the vertex field, only once for its entire life. This partial data structure is exclusively written once, but concurrently read tens of times for updating the rest of the data structure, i.e., the list of edges.

We develop a state transfer mechanism for concurrent lookup / insertion / update operations. More specifically, we use a state flag named *occupancy* with three possible values $\{empty, locked, occupied\}$. If a thread accesses an entry with $occupancy = empty$, the slot is empty and ready for insertion. Then the thread changes *occupancy* atomically to *locked*. If this atomicCAS on *occupancy* succeeds, other threads trying to access the kmer are blocked. Until the thread finishes its insertion of the kmer, it changes *occupancy* to *occupied*. A kmer with an $occupancy = occupied$ will never be modified and thus is concurrently read for the rest in the hash table. Threads looking up entries with an identical hash value will access the same entry slot, and be scheduled by the state valued *occupancy*. Progress of this subset of threads are temporarily blocked on the entry. Other threads in the program on other hash entries will make progress independently.

Fig. 4 shows an example to illustrate the state transfer in inserting and updating entries in the hash table. The three states are marked with white (empty), gray (locked), and black (occupied). In this example, the machine word size is 8-bit, and the kmer occupies two words. The entry slot in the hash table with black state flag indicates that this slot is occupied with <kmer, list of edges>. Accessing the kmer, e.g., "00100101,01110011", is concurrent read. When a thread holding a <kmer, edge> visits this slot, it first compares its kmer with the kmer in the entry; if they are equal, it updates the edge_count lists; otherwise, it continues to visit other entry slots until it finds an empty slot (in white state), or finds an entry holding the same kmer. Concurrent updates on the edge_counts lists are supported by atomic increment. In comparison, the entry slot in gray indicates that the kmer is locked, and the kmer value is not finished with write from some thread. Other threads that read this entry are blocked until the state changes to black.

D. Parallel Implementations

To fully utilize the processors on the heterogeneous architecture, we implement the algorithms on both the CPU and the GPU. Since the number of parallel threads on the GPU are much greater than the CPU, we adopt different data parallelism granularity for their implementation. Specifically, we use one CPU thread to access a group of data elements (reads or superkmers) that are located nearby in memory, but use each GPU thread to handle each data elements. For the GPU implementation, shared memory is utilized if the data for a block of threads is accessed multiple times. In the implementation of the partial locking algorithm, we use fast atomic operations supported on both the CPU and the GPU.

In Step 1, identifying minimizers is of $O(LKP)$ computation complexity for each read, and memory access on each read is aligned by the read length. Hence we take advantage of the GPU's high parallelism and memory bandwidth, and offload computing superkmer ids and offsets in reads to the GPU. On the other hand, because generating superkmers involves memory movement of irregular lengths, we leave this part of work to the CPU. However, offloading work to GPUs incurs data transfer overhead. To reduce such overhead, we implement the MSP computation on the GPUs based on encoded bit values for the reads, kmers and superkmers. This way, we reduce the data size transferred between the CPU and the GPU, and the string processing on the GPU becomes faster due to encoding.

In Step 2, inserting or updating <vertex, edge> pairs in the hash table exposes parallelism for massive threads. However, implementing the concurrent hashing algorithm on GPU suffers performance penalty in thread divergence and random memory access. The processing unit on the GPU is the stream multiprocessors (SMs), and a warp of threads executes the same instruction simultaneously. Different threads assigned with different kmers within a warp diverge to different walk length when visiting the hash table slots. Memory access to the hash entries from threads within a warp can not be coalesced. Hence the memory bandwidth benefit is limited in hashing on the GPU.

Parallel hashing techniques on the GPU focus on optimizing the memory access locality [19], [21]. In our De Bruijn graph construction, the graph partitioning approach provides a straightforward method to optimize the memory access locality in hashing. Since the subgraph size distribution is controlled by the minimizer length (shown in Section V-B1), we can use the number of superkmer partitions to set the subgraph size, and thus set the hash table size. Therefore, the memory access is localized by configuring the hash table size. Such optimization works for the implementation on the multi-core CPU as well.

E. Co-processing and Pipelining

Using the GPU to assemble big genomes is still an open problem, because the string processing and graph structure processing involve intensive and irregular memory access. Advantages underlying the GPU hardware cannot be fully exploited in such applications [22]. Therefore, we do not offload the major workloads to the GPU, but use both the CPU and the GPU as co-processors.

Furthermore, pipelining is efficent in overlapping disk transfer with computation, as well as the work between the host and the device [14], [23]. To fully exploit the advantages of our co-processing workflow, we design an execution pipeline to overlap the input data transfer, the computation on processors and the output data transfer. The pipeline consists of three stages: input partitions, consuming and producing, and output partitions. The first stage transfers a partition of input data from the disk and parses it to the required format for the second stage. Then the idling CPU or GPU consumes this input partition and produces an output partition. The third stage parses each output partition to the required format and transfers it to the disk.

As shown in Fig. 5, both Step 1 and Step 2 in our system are decomposed into three stages, and these stages are pipelined. Step 1 consists of extracting read partitions, superkmer generation, and superkmer output. Step 2 consists of superkmer partition loading, hashing, and subgraph output. Processor 0 or Processor 1 gets an input partition based on the input data transfer speed and its computation speed.

To dynamically distribute partitions to the CPU and the GPU, we develop a work-stealing based pipelining algorithm that uses an input queue and an output queue to synchronize the data input, the computation and the data output. We use a shared variable *srv* pointing the tail of the input queue, and a shared variable *cns* pointing to the head of the input queue. *srv* is incremented only by the thread that inputs partitions and read by the threads that distribute partitions to processors. This way, inputting partitions is synchronized with the consuming

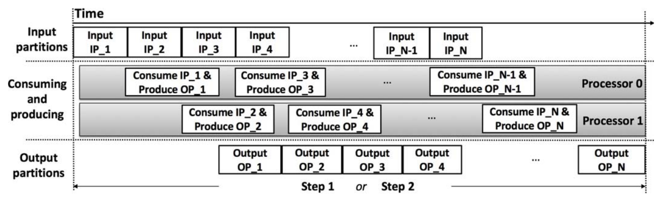


Fig. 5: Time line for pipelined co-processing

and producing from the processors. To fix the consuming order of different processors, we use *cns* to assign a queuing id to the processor. When an input partition is available ($srv \geq cns$), it is assigned to the processor according to its unique queuing id. When an output partition is produced and placed in the output queue, a shared variable *prd* for the output queue is incremented. Output data transfer is synchronized with this *prd*. Only when $prd \geq wrt$, where *wrt* points to the head of the output queue, the output partitions are dequeued and transfered to disk.

IV. Performance Model

Since ParaHash works on partitioned data in each step, setting parameters for a balanced partitioning is critical in performance. We focus on the input data (superkmer partitions) distribution in Step 2, because it is determined by the MSP algorithm. Furthermore, the superkmer partition size distribution decides the average hash table size for the hashing step, and the hash table size impacts the hashing performance.

Therefore, we introduce two parameters in the system settings, i.e., the number of superkmer partitions, and the minimizer length. Section IV-A investigates the performance impact of the parameters in ParaHash. Section IV-B proposes a model to estimate the performance of our co-processing and pipelining.

A. System parameters

Minimizer length *P*. In Step 1, the workload is uniformly distributed by partitioning input into equal size. In Step 2, the superkmer partition size is determined by the MSP algorithm. Since the minimizer is identified by the lexicographical string comparison, a larger value of P decreases the probability that adjacent kmers share a common P-minimum-substring. Correspondingly, when the number of superkmers increases, the average length for a single superkmer decreases. Therefore, setting a suitable value of P determines both the average superkmer partition size and the variance in the sizes of partitions.

Number of superkmer partitions. A larger number of superkmer partitions gives a smaller average superkmer partition size. With a fixed minimizer length, the number of superkmer partitions determines the average superkmer partition size.

Hash table size. ParaHash partitions the hash table of the entire graph by the superkmer partitions. Since the hash table size in a partition affects the memory access locality in hashing, as well as the data contention, it decides the concurrent hashing performance. Property 1 provides an estimation for the average hash table size. Note that $O(LN) = O((L-K+1+K-1)N) = O(N_{kmer})$, where N_{kmer} is the number of kmers generated by all input reads. Since the number of kmers N^i_{kmer}

in the i-th superkmer partition determines the hash table size for the subgraph, we use it with a load ratio α to set the hash table size as $\frac{\lambda}{4\alpha}N^i_{kmer}$.

B. Performance modeling

To justify the evaluation of the overall performance for the system, we develop a performance model to determine the major components. This model defines the following variables: the overall elapsed time (T), the elapsed time for a single step in the system (T^i), the CPU / GPU computation time ($T^i_{CPU/GPU}$), the input / output transfer time (T^i_{input} / T^i_{output}). We assume that with asynchronous data transfer, the CPU computation, the GPU computation, and the input / output transfer time are independent with each other.

In ParaHash, the total elapsed time is $T = \sum_i T_{step_i}, i = 1, 2$. With our execution pipeline, each input partition will generate an output partition. We denote the number of input / output partitions in Step i to be n_i, and use $\frac{1}{n_i}T^i_{input}$ and $\frac{1}{n_i}T^i_{output}$ to estimate the data input and output time for a partition. Then the elapsed time for Step i is estimated as follows:

$$\begin{aligned}
&T^i = Max\{T^i_{CPU}, T^i_{GPU}, T^i_{I/O}\} + \frac{1}{n_i}(T^i_{input} + T^i_{output}), \text{where}\\
&T^i_{CPU} = T^i_{CPU\_compute},\\
&T^i_{GPU} = T^i_{GPU\_compute} + T^i_{DH\_transfer},\\
&T^i_{I/O} = \frac{n_i - 1}{n_i} Max\{T^i_{input}, T^i_{output}\}.
\end{aligned} \tag{1}$$

We use T_{GPU} to denote the maximum elapsed time of all GPU devices, when the number of GPUs (N_{GPU}) is greater than 1. We do not overlap the device computation ($T_{GPU\_compute}$) and the host and device data transfer ($T_{DH\_transfer}$) due to the limited amount of device memory. Since we overlap the CPU computation, the GPU computation and the input / output transfer, the elapsed time T^i is estimated as the maximum of T^i_{CPU}, T^i_{GPU} and $T^i_{I/O}$. We further overlap the input and output with multiple partitions in both steps, thus the $T^i_{I/O}$ depends on the maximum between input transfer and output transfer.

To evaluate the scalability with respect to processors in our system, we estimate the elapsed time under the following two conditions:

Case 1. $T_{I/O} << Min\{T_{only\_CPU}, T_{single\_GPU}\}$.
On a single machine, we only consider the increment of the number of GPUs. We use $1/T^i_{only\_CPU}$ and $1/T^i_{single\_GPU}$ to estimate the CPU and single GPU processing speeds in Step i. Then the ideal elapsed time for Step i can be estimated as

$$\frac{1}{1/T^i_{only\_CPU} + N_{GPU}/T^i_{single\_GPU}}. \tag{2}$$

Case 2. $T_{I/O} > Max\{T_{only\_CPU}, T_{single\_GPU}\}$.
Under this condition, the overall elapsed time for Step i is estimated as $T_{I/O} + \frac{1}{n_i}(T^i_{input} + T^i_{output})$.

V. Evaluation

First, we evaluate the performance of our concurrent hashing, under the impact of the hash table size. Then we study the efficiency of the co-processing and pipelining in our system. Furthermore, we experimentally evaluate our work in comparison with the state-of-the-art shared-memory assemblers, i.e., SOAP [11] and bcalm2 [3]. Finally, we compare the measured performance of our system with the performance model.

TABLE I: Test datasets properties

Genome	Human Chr14	Bumblebee
Fastq file size (GB)	9.4	92
Read length (bp)	101	124
# Reads (Million)	37	303
Genome size (Mbp)	88	250
# Distinct vertices (Million)	452	4,951
# Duplicate vertices (Million)	2,725	29,391

A. Experimental Setup

Data sets. We use the two largest genome datasets in the GAGE project [24]. Table I shows the properties of *Human Chr14* and *Bumblebee*. We use the number of distinct vertices ("# Distinct vertices") to measure the graph size. Results obtained from the output graph show that the graph size of *Bumblebee* is about ten times larger than that of *Human Chr14*.

Hardware configuration. The experiments are conducted on a machine with two 2.20GHz Intel Xeon E5-2660 CPUs (10 cores on each CPU) and two Nvidia Tesla K40m GPUs. Memory on the host is 64GB, and memory on each device is 12GB. To evaluate the system performance with our performance model, we use the memory cached file, i.e., a file that resides in memory, to simulate *Case 1* condition on $HumanChr14$, and use the disk file for *Case 2* condition on *Bumblebee* dataset.

Software in comparison. We compare the De Bruijn graph construction performance in ParaHash with two shared-memory assemblers. Both SOAP [11] and bcalm2 [3] use multi-threaded parallelization, and bcalm2 is the most memory-efficient in constructing big De Bruijn graphs. We do not compare with distributed assemblers [1], [8], [12], [13], since they focus on using distributed memory. Kmer counters and other GPU based assembly tools are also excluded from the comparison, because kmer counters [2], [5], [14] do not generate the complete De Bruijn graph in the output, and the source code of GPU assemblers [15], [16] are not currently accessible.

Parameter setup. Initially, P ranges from 1 to K (the kmer length); the number of superkmer partitions ranges from 1 to 1000, where the upper limit 1000 is the largest number of file handles allowed on our experimental platform. In the hashing step, we set $\lambda = 2$, $\alpha \in [0.5 - 0.8]$, and calculate N^i_{kmer} from the prefix sum of the superkmer lengths in Partition i. Using P and the number of superkmer partitions (# Partitions), we set the hash table size.

Measurement. Time measurement begins at the time the system reads file, and ends at the time all the subgraphs are constructed in main memory. The time of writing the subgraphs to disk files are excluded in performance comparison with other assemblers, since the constructed graphs are the same for all the systems. Measurement for ParaHash includes the write-out and read-in of the superkmer partitions. We measure the GPU computation time with the host and device data transfer time included.

B. Parameter Setting

We experimentally study the system parameters, i.e., the minimizer length P, and the number of superkmer partitions. The default values are set from the sampled results on *Human Chr14* dataset. Since our system works in a partition-by-partition manner, these parameters work for data sets of

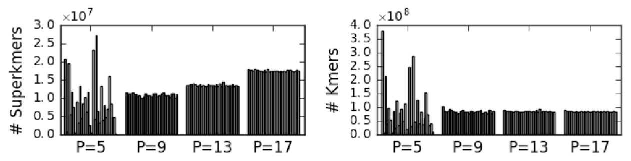


Fig. 6: Distribution of the number of superkmers and kmers. Test dataset is *Human Chr14*; *P* is the minimizer length; the number of superkmer partitions is 32.

TABLE II: Hash table size with *Human Chr14*

NP[1]	16	32	64	128	256	512	960
#Kmers[2]	170	85	43	21	11	5	3
Size[3]	5400	2600	1400	700	320	160	90

[1] The number of superkmer partitions
[2] The number of kmers per partition (Million)
[3] The maximum hash table size per partition (MB)

arbitrary sizes. The subsequent experiments on performance evaluation are conducted with the default settings of the parameters.

*1) **Minimizer Length** P:* We investigate the superkmer partition size distribution with the minimizer length *P* varied, when the number of partitions is fixed. Fig. 6 shows that the variance of partition sizes decreases significantly, when *P* increases from 5 to 17. In contrast, the total number of superkmers increases when *P* increases. This result indicates superkmers are more fragmented with a shorter average length when *P* is larger. When the number of partitions is set, a larger *P* generates a more balanced distribution for the number of kmers / superkmers. It will also result in a smaller value for the upper bound of the hash table sizes. We set *P* to be greater than 11 to get a more balanced partition size distribution in experiments.

Default setting. The default value *P* is 11 for *Human Chr14* and 19 for *Bumblebee*. Kmer length for both datasets is 27.

*2) **Number of Superkmer Partitions**:* Table II shows the maximum hash table size for a single partition for dataset *Human Chr14*. With the number of partitions increases, the maximum size for a single hash table decreases. Furthermore, experiment results in Fig. 7 indicate that a less than 1GB sized hash table ensures good performance in the hashing step. Thus with a fixed value *P*, we use the number of partitions, the input data size (*LN*), and Property 1 to initiate the hash table size.

Default setting. We set the default value to 512 for datasets of several to tens of gigabytes, and set it to 960 for data sets with 100GB or more.

C. Performance Results

*1) **Concurrent Hashing**:* We evaluate our concurrent hashing in three aspects: the performance on the CPU and the GPU, the scalability of concurrent hashing on the CPU with respect to the number of working threads, and the CPU hashing performance comparison with a state-of-the-art assembler.

With a fixed minimizer length, we study the performance of hashing with the number of superkmer partitions varied. Fig. 7 shows when the hash table size decreases with the increment of the number of partitions, both the CPU hashing (with 20 threads) time and the GPU hashing time decrease. GPU hashing time breakdown in Fig. 8 shows the device and host data transfer time remains constant when the number of partitions varies. This is because the total size of the hash tables is fixed, and the data transfer overhead depends on the total data size. On the other hand, a comparison between Fig. 7 and Fig. 8 shows the gap between the CPU hashing time and the GPU hashing time is nearly the host and device data transfer time, when the number of partitions is larger than 16. It indicates that the hashing performance on the 20-core CPU is comparable to the performance on a Nvidia K40 GPU with 15 SMs, when the memory access is random in hashing.

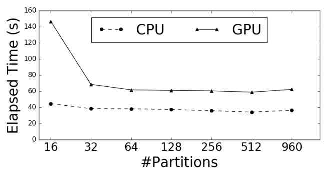


Fig. 7: CPU hashing vs. GPU hashing

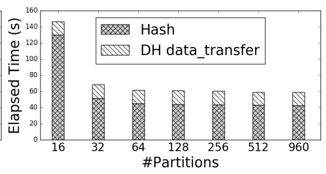


Fig. 8: GPU hashing time breakdown

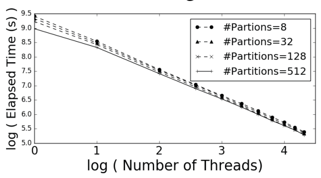


Fig. 9: Concurrent CPU hashing scalability

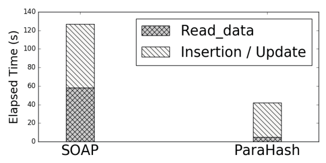


Fig. 10: CPU hashing comparison with time breakdown

To evaluate the scalability of our concurrent hashing, we set the number of threads to be 1, 2, 4, 6, ..., 20, and measure the CPU hashing time. In Fig. 9, when the number of threads is greater than 2, the line fits the function $log(y) = alog(x) + b$, where a and b are constants, x is the number of threads, and y is the elapsed time. By fitting the data points to the function, we find a is close to -1, which means xy is a constant that represents the execution time for the total amount of work. This result shows the scalability of the CPU hashing is close to linear, even with the presence of data contention from multiple threads.

We further compare the CPU hashing performance with SOAP. Since SOAP requires the number of local hash tables to be the same as the number of threads, we set the number of superkmer partitions in ParaHash to 20, and set P = K to directly generate kmers in each partition. We break down the hashing time into reading data time (Read_data) and insertion / update time (Insertion / Update). The Read_data is measured by the time a thread reads its <vertex, edge> entries from main memory to local variables. Fig. 10 shows our hash algorithm is fast both in accessing <vertex, edge> pairs and in inserting or updating the table entries.

*2) **Co-processing and pipelining**:* To evaluate the efficiency of co-processing with both the CPU and the GPU, we first measure the elapsed CPU and GPU time separately, and use it to estimate the CPU and GPU processing speed. Then we evaluate whether the workload is distributed to processors according to their processing speeds. We calculate the ratio of the number of reads (in Step 1) or distinct vertices (in Step 2) processed on the CPU or the GPU to the total number of reads or distinct vertices, to measure the workload on the processors.

The left figure in Fig. 11 shows the elapsed time on each processor. It demonstrates that the processing time of different

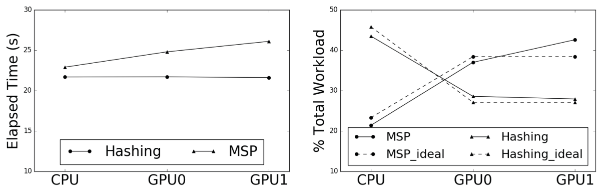


Fig. 11: Workload distribution with co-processing

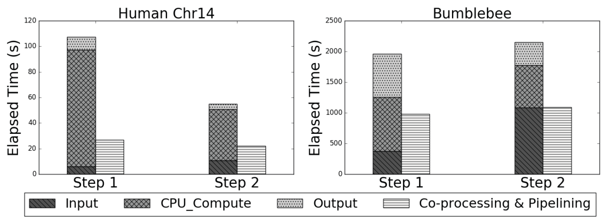


Fig. 12: Time breakdown without pipeline vs. Elapsed time with pipeline

processors in both steps are close to each other. In the right figure, we estimate the ideal number of processed reads or vertices on processors based on their processing speeds, and measure the real numbers. Results show the workload is distributed to processors according to their processing speeds. Since the real distribution (marked in solid line) matches the ideal distribution (marked in dotted line) better in hashing, it indicates that the workload is more balanced in hashing.

The factor that influences the workload distribution in Step 1 and Step 2 is the involvement of the CPU threads in inputting and outputting partitions. In Step 1, the CPU does more input and output data parsing work, e.g., extracting and encoding reads from the input, and encoding superkmers as the output, hence it spends less time in the computation. In contrast, the input and output on the CPU is simply disk transfer in the hashing step, and the workload for computation is more balanced between the CPU and the GPU.

We evaluate the pipelining efficiency in the two steps by comparing the accumulated time of non-pipelined stages with the pipelined elapsed time. Since the data transfer from / to disk overlaps the data parsing on the CPU, we can not measure the disk transfer time separately. Instead, we measure the entire input or output process that involves disk IO. With time breakdown of the three stages (Input, CPU_Compute, and Output), Fig. 12 shows that pipelining significantly improves the overall performance, when the IO time does not dominate the performance (with dataset *Human Chr14*). In the case of *Bumblebee* where the IO time dominates, the elapsed time is saved by half, since we overlap the input and output, and hide the computation in the data transfer.

*3) **Comparison with Other Assemblers**:* We compare the performance of ParaHash with SOAP [25] and bcalm2 [3] in both the running time and peak memory consumption on the host. SOAP and bcalm2 are both run with 20 threads for their best performance. To study the overall scalability of ParaHash with respect to processors, we run ParaHash separately with CPU only (ParaHash-CPU), two GPUs (ParaHash-2GPU), CPU and two GPUs (ParaHash-CPU-2GPU) for the computation. For ParaHash running on both host and devices, the peak memory usage on the host still dominates the memory usage of the system, since the host always maintains a copy of the data on the devices.

TABLE III: Performance comparison with assemblers

Dataset / System	Human Chr14		Bumblebee	
	Time[1]	Memory[2]	Time[1]	Memory[2]
bcalm2[*]	1124	3	18101	5
SOAP	159	16	NA	NA
ParaHash-CPU	132	2	1992	4
ParaHash-2GPU	72	2	**1770**	4
ParaHash-CPU-2GPU	**49**	2	2013	4

[1] Elapsed time (s)
[*] Time measured for bcalm2 includes kmer counting time and the MPHF hashing time for junction kmers
[2] Peak memory usage on the host (GB); the number of intermediate partitions are the same for ParaHash and bcalm2.

Table III shows the performance comparison. In the experiments with *Human Chr14*, the elapsed time of ParaHash reduces with the increment of processors. ParaHash running with all processors is 3 times as fast as SOAP, with the presence of graph partitioning overhead. It is 20 times faster than bcalm2. In the experiments of *Bumblebee*, SOAP cannot run with the hardware configuration, since it require the entire graph to reside in the 64GB main memory. With this data set, ParaHash is 9-10 times faster than bcalm2. For the memory usage, ParaHash is as efficient as bcalm2. Such memory efficiency ensures the scalability with respect to the data size.

*4) **Comparison with Model Estimation**:* To understand the performance of ParaHash, we compare it with the estimated ideal performance. Experiments are conducted with cases whether the disk transfer dominates the performance. We evaluate the time with different processor configurations for the computation, i.e., the consuming and producing stage in the pipeline: CPU-only, offloading to 1 GPU, offloading to 2 GPUs, co-processing with the CPU and 1 GPU, and co-processing with the CPU and 2 GPUs.

$\mathbf{T_{I/O} << Min\{T_{only\_CPU}, T_{single\_GPU}\}}$*:* Our experiment with *Human Chr14* was done with a memory cached file. This way, we got the simulated disk IO bandwidth to be several gigabytes, which is much higher than the computation throughput on a single GPU or the CPU.

Fig. 13 compares the real elapsed time with the estimated ideal time, in both Step 1 and Step 2. We use the best CPU-only and single GPU-only performance as ideal single processor baselines for the computation, and estimate the ideal co-processing elapsed time with Equation (2). Results show when the number of processors increases, the elapsed time decreases according to the processing speeds of different processors. Offloading the computation work to more devices improves the performance better.

$\mathbf{T_{I/O} > Max\{T_{only\_CPU}, T_{single\_GPU}\}}$*:* Table I shows that the graph for *Bumblebee* contains about 5 billion vertices. With each <vertex, list of edges> occupying 32 bytes in memory, it costs 160 GB space. The input file for this dataset is 92GB; the total superkmer partition size is 81GB; and the output De Bruijn graph file is about 20GB (with invalid vertices filtered). Therefore, the disk IO bandwidth becomes the bottleneck.

In Fig. 14, we use Max{CPU_compute, GPU_compute} to denote the maximum computation time on processors, and use Max{Input, Output} to denote the maximum between inputting partitions time and outputting partitions time. We

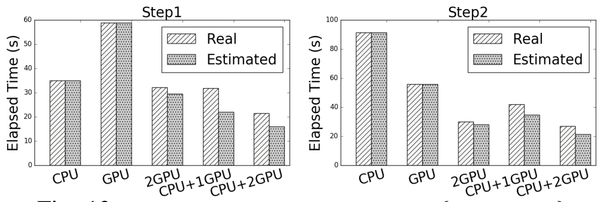


Fig. 13: Real vs. Estimated, $T_{I/O} << Min\{T_{CPU}, T_{GPU}\}$

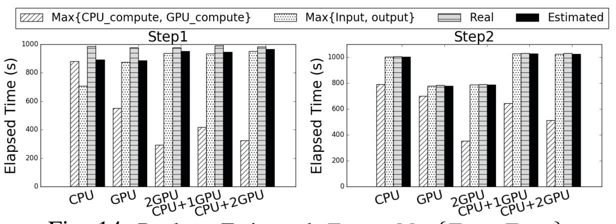


Fig. 14: Real vs. Estimated, $T_{I/O} > Max\{T_{CPU}, T_{GPU}\}$

compare the real elapsed with the ideal estimated time from Equation (1). In Step 1, running with only CPU, the CPU computation time is longer than the input and output time. With the acceleration on GPUs, the input / output time becomes dominant, and the overall time is almost the same as the estimated time. In Step 2 the real elapsed time is nearly the input / output time, with different processor configurations. Furthermore, comparing the computation time with a single GPU configuration between *Human Chr14* and *Bumblebee*, we see the GPU computation time increases linearly with the increment of the total data size, indicating that the GPU computation performance is scalable with the data size. In contrast, the CPU computation performance degrades when it processes input / output for big data size.

VI. Conclusion

With the advances in sequencing technology, more and more genome-wide studies will become feasible and assemblers will have to deal with various kinds of big genomes efficiently. To construct De Bruijn graphs for big genomes, we partition the entire graph into subgraphs, generate and output compact partition data, and take it as input for subgraph construction. This way, we are able to construct graphs with billions of vertices on a single commodity computer. Furthermore, we utilize heterogeneous processors on the single computer to perform both steps for efficiency.

To improve the in-memory parallel processing, we design concurrent hash tables on partitioned superkmers so that the massive number of CPU / GPU threads can perform hash-based De Bruijn graph construction efficiently. We develop a work-stealing based algorithm to dynamically distribute computation to processors. Hence pipelining is further employed to overlap the data transfer and computation on processors. When constructing medium-size genomes, e.g., *Human Chr14*, ParaHash is up to 20 times faster than the state-of-the-art shared-memory assemblers. When constructing big-size genomes, e.g., *Bumblebee*, the overall elapsed time is reduced by the overlapped input, output and computation. ParaHash is 9-10 times faster than the state-of-the-art assembler, with only several-gigabyte memory requirement.

We also study the co-processing performance under the condition whether the disk transfer bandwidth dominates computation throughput in the hardware configuration. Results show that increasing the number of processors improves the overall performance, when the computation cost dominates. Workload is also balanced among all available processors. Experiments also indicate that increasing the number of devices to offload computation achieves nearly ideal performance improvement. Finally, the overall elapsed time is approximately the disk transfer time when the disk IO bandwidth becomes the bottleneck with big genome dataset.

VII. Acknowledgement

This work was partly supported by Grants 616012 and 617509 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

Appendix
Proof of Property 1

Proof: Suppose random variable X is the number of errors that occur in a read $S[0,L-1]$; Y is the number of erroneous kmers generated by the read. Then the expected number of erroneous vertices generated by a read is

$$\begin{aligned} E(Y) &= E(E(Y|X)) = \sum_j E(Y|X=j)P(X=j) \\ &= \sum_j \sum_i iP(Y=i|X=j)P(X=j) \\ &\leq \lambda(\sum_i iP(Y=i|X=1)) \end{aligned} \quad (3)$$

Suppose that an error occurs at any position in the read $S[0,K-1]$ with an equal probability $\frac{1}{L}$. At some position in a read, i.e., $S[i], i=0,...,L-1$, the number of kmers that covers $S[i]$ decides the number of erroneous kmers. If $0 \leq K \leq \lceil\frac{L+1}{2}\rceil$, we have

$$P(Y=K|X=1) = \frac{L-2(K-1)}{L}$$

$$P(Y=m|X=1) = \frac{2}{L}, 1 \leq m \leq K-1.$$

Then

$$\sum_i iP(Y=i|X=1) = K\frac{L-2K+2}{L} + \sum_{m=1}^{K-1} m\frac{2}{L} \leq \Theta(\frac{L}{4}) \quad (4)$$

If $\lfloor\frac{L+1}{2}\rfloor \leq K \leq L$, we have

$$P(Y=L-K+1|X=1) = \frac{2K-L}{L}$$

$$P(Y=m|X=1) = \frac{2}{L}, 1 \leq m \leq L-K.$$

Then

$$\sum_i iP(Y=i|X=1) = (L-K+1)\frac{2K-L}{L} + \sum_{m=1}^{L-K} m\frac{2}{L} \leq \Theta(\frac{L}{4}) \quad (5)$$

Applying (4) or (5) in (3), we get

$$E(Y) = E(E(Y|X)) \leq \Theta(\frac{\lambda L}{4})$$

If $\frac{L+1}{2} < K < L$, $E(Y)$ is less than the result in the case of $0 \leq K \leq \frac{L+1}{2}$.

Therefore, the expected number of erroneous kmers generated by N reads from a Ge-size genome is bounded by $\Theta(\frac{\lambda}{4}LN + Ge)$. ■

References

[1] J. Meng, S. Seo, P. Balaji, Y. Wei, B. Wang, and S. Feng, "Swap-assembler 2: Optimization of de novo genome assembler at extreme scale," in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016, pp. 195–204.

[2] Y. Li, P. Kamousi, F. Han, S. Yang, X. Yan, and S. Suri, "Memory efficient minimum substring partitioning," in *Proceedings of the VLDB Endowment*, vol. 6, no. 3. VLDB Endowment, 2013, pp. 169–180.

[3] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[4] C. Purcell and T. Harris, "Non-blocking hashtables with open addressing," in *International Symposium on Distributed Computing*. Springer, 2005, pp. 108–121.

[5] 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.

[6] M. C. Wendl, "A general coverage theory for shotgun dna sequencing," *Journal of Computational Biology*, vol. 13, no. 6, pp. 1177–1196, 2006.

[7] B. G. Jackson and S. Aluru, "Parallel construction of bidirected string graphs for genome assembly," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 346–353.

[8] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru, "Parallel de novo assembly of large genomes from high-throughput short reads," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[9] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[10] 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.

[11] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome research*, vol. 20, no. 2, pp. 265–272, 2010.

[12] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 437–448.

[13] 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.

[14] M. Lu, Q. Luo, B. Wang, J. Wu, and J. Zhao, "Gpu-accelerated bidirected de bruijn graph construction for genome assembly," in *Web Technologies and Applications*. Springer, 2013, pp. 51–62.

[15] S. F. Mahmood and H. Rangwala, "Gpu-euler: Sequence assembly using gpgpu," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 2011, pp. 153–160.

[16] A. Garg, A. Jain, and K. Paul, "Ggake: Gpu based genome assembly using k-mer extension," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 1105–1112.

[17] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.

[18] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1493–1508.

[19] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 154, 2009.

[20] X. V. Wang, N. Blades, J. Ding, R. Sultana, and G. Parmigiani, "Estimation of sequencing error rates in short reads," *BMC bioinformatics*, vol. 13, no. 1, p. 1, 2012.

[21] R. Bordawekar, "Evaluation of parallel hashing techniques."

[22] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber, "Multi-level parallel query execution framework for cpu and gpu," in *East European Conference on Advances in Databases and Information Systems*. Springer, 2013, pp. 330–343.

[23] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 325–336.

[24] 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 *et al.*, "Gage: A critical evaluation of genome assemblies and assembly algorithms," *Genome research*, vol. 22, no. 3, pp. 557–567, 2012.

[25] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *Gigascience*, vol. 1, no. 1, p. 18, 2012.