Ultra Efficient Acceleration for *De Novo* Genome Assembly via Near-Memory Computing

Minxuan Zhou[§]
University of California, San Diego
miz087@eng.ucsd.edu

Lingxi Wu[§]
University of Virginia
lw2ef@virginia.edu

Muzhou Li
University of California, San Diego
mul023@eng.ucsd.edu

Niema Moshiri
University of California, San Diego
niema@ucsd.edu

Kevin Skadron
University of Virginia
skadron@virginia.edu

Tajana Rosing
University of California, San Diego
tajana@ucsd.edu

***Abstract*—*De novo* assembly of genomes for which there is no reference, is essential for novel species discovery and metagenomics. In this work, we accelerate two key performance bottlenecks of DBG-based assembly, graph construction and graph traversal, with a near-data processing (NDP) architecture based on 3D-stacking. The proposed framework distributes key operations across NDP cores to exploit a high degree of parallelism and high memory bandwidth. We propose several optimizations based on domain-specific properties to improve the performance of our design. We integrate the proposed techniques into an existing DBG assembly tool, and our simulation-based evaluation shows that the proposed NDP implementation can improve the performance of graph construction by 33× and traversal by 16× compared to the state-of-the-art.**

I. Introduction

Next Generation Sequencing (NGS) has revolutionized genomics due to the high volume and low cost of sequencing [26]–[28], [56]. A typical NGS system can generate 10TB of short DNA reads (100-300 base pairs) in a single run [4], [58]. For most sequencing experiments in which a high-confidence reference genome is known, the standard workflow is to align these reads against the appropriate reference genome [3], [39], [65]. However, the reference genome is not always available, especially when analyzing unknown species, such as a new virus or bacteria [19], [20], [53], or meta-genome that is sequenced from diverse environmental microbiomes [19], [20], [53]. Even when the reference genome is available (e.g., humans), the reference genome may be missing rare genomic variants of biomedical interest [2], [5], [14], [61]. In these contexts, we must assemble our reads *de novo* (without a reference genome). State-of-the-art *de novo* genome assemblers use the reads to construct a de Bruijn graph (DBG) and subsequently find all maximal non-branching paths of the DBG to produce *contigs* (contiguous segments of the assembled genome) [39], [50], [60], [66]. DBG-based assemblers are both time- and memory-intensive, due to a large amount of sequence data and the explosive number of nodes in the graph, posing significant challenges on conventional computing systems.

Although most DBG-based *de novo* assemblers [30], [38], [41], [60], [66] adopt parallel algorithms to improve performance and scalability, they are always memory-latency bound—the memory access takes up the most portion of execution time. Furthermore, the memory bandwidth requirement of DBG assemblers constantly increases at a linear rate as the degree of parallelism increases, which makes DBG assembly also memory-bandwidth bound in a parallel environment.

Accelerating DBG processing is of paramount importance for several reasons. First, DBG processing is the de facto *de novo* assembler for both large (mammalian-sized) or small (e.g. E.coli) single-cell genome analysis [5], [30], [38], [39], [60], [66], as well as metagenomic studies where a large (up to TBs) mixture of bacterial, viral, and fungal microbiome genomes obtained directly from a human body or an environment needs to be assembled [36], [47]. Second, although primarily developed to assemble the 2nd gen (a.k.a. NGS) reads, DBG processing retain its relevance as the foundation of assembling reads generated by the 3rd gen sequencers [37], [40]. Third, DBG processing is on the critical path of many time-critical genome analysis tasks. In the emerging precision medicine domain, a patient's sample is first sequenced on the NovaSeq instrument in under 48 hours, producing 6 to 12 TB microbiome and human DNA/RNA data. This raw sequence data is then passed through various stages, including the DBG assembly (∼3600 CPU hours) [55]. Finally, the rate of genomes been sequenced is vastly outstripping Moore's law [62], For example, the data volume of unassembled biosequences surpasses that of astronomy, particle physics, and websites such as YouTube and Twitter [15], [54].

Near-data processing (NDP) is an emerging memory-based approach that can provide scalable parallelism and memory bandwidth by integrating massive cores in memory devices [1], [45], [67]–[69]. In this work, we exploit NDP technology to accelerate DBG assembly. We design near-data parallel algorithms for graph construction and graph traversal that exploit the hardware parallelism by distributing data and operations in different memory locations. The near-data parallel implementation enables different NDP cores to process different portions of data simultaneously for performance scaling.

However, naive NDP implementation faces several issues caused by data communication among NDP cores. The graph construction phase requires intensive data movement among NDP cores, because the input sequence and the intermediate

[§]Jointed first authors

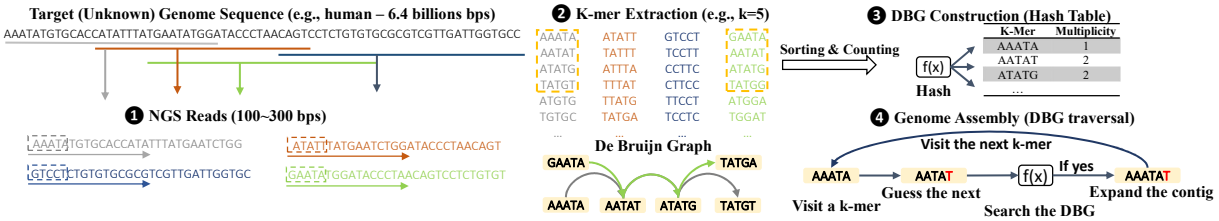


Fig. 1: **The stages in *de novo* genome assembly using de Bruijn graph.**

data structures are distributed with different strategies. Furthermore, during graph traversal, building a contig requires a series of accesses to k-mers (DNA strings of fixed length k) located in different NDP cores. Our evaluation shows that such inter-core data communication can take up to 60% to 75% of the execution. To reduce these overheads, we propose several optimization techniques, based on domain-specific knowledge on genome assembly. In the graph construction phase, we shuffle the distribution of DBG data structures based on the distribution of addresses for copy, using a greedy algorithm to reduce the number of inter-core data movements. Furthermore, we propose message buffering and k-mer compression to reduce the size of data communicated. For graph traversal, we design a speculative contig expansion which parallelizes traversal operations in each core.

We summarize the contributions of this paper as follows:

- This is the first in-memory accelerator for DBG-based *de novo* genome assembly.
- We propose several optimization techniques based on domain-specific knowledge of DBG assembly to reduce the data communication overhead in NDP systems.
- We improve upon a state-of-the-art DBG assembly on a NDP system, and we evaluate our design using a application-level architecture simulator. We compare the performance of the proposed design with the software baseline with real genomes. The results show that the proposed optimization techniques can lead to 33-fold and 16-fold speedup over the software baseline for graph construction and graph traversal, respectively. The performance gap between our NDP-based DBG assembler and a conventional one is expected to grow even wider given larger genome size, as demonstrated in our evaluation. Furthermore, the proposed NDP-based DBG assembler scales well when increasing the system size.

II. Background

A. De Bruijn Graph Genome Assembly

Genome sequencing is the process of determining a segment or the whole DNA sequence of an organism. *De novo* assembly is a key step of genome sequencing, where the short sequenced reads are assembled without using a reference genome [3], [9], [38], [39], [41], [44], [50], [59], [60], [66]. Currently, the most successful *de novo* assembly algorithm is based on de Bruijn graph (DBG).DBG is a form of directed multigraph that stores the overlapping information of k-mers (DNA subsequences of size k) extracted from DNA sequence reads. Each unique k-mer is represented as a node in the graph, and an edge is formed between two nodes if the 'k-1' suffix of the first node exactly matches the 'k-1' prefix of the second node. DBG assembler finds a path that visits each node exactly once to assemble the DNA sequence. The DBGs are of special interest because the assembly algorithm can finish in polynomial time with respect to graph size [50].

Figure 1 shows the full pipeline of DBG-based genome assembly. It takes in NGS short reads sampled in the genome sequence step, and then extracts k-mers from every single position. The de Bruijn graph is built on the coverage relation between k-mers. Unlike a general graph, DBG follows a simple pattern where each node can only have up to four out-going edges and four in-coming edges (four possible nucleobases). Therefore, the most common data structure for DBG is a hash table, which enables efficient traversal on a forward or backward path by searching the next/previous possible k-mers [38], [60], [66]. When a k-mer appears more than once during the graph construction process, they are merged into one node and increase the count as the multiplicity. DBG assembly can build many long sequences, which are called contigs, by traversing the Eulerian path in the DBG. Most DBG assemblers have many common steps, including data loading, error removal, and contig assembly.

The most time-consuming phases in a DBG assembly process are graph construction, which save unique k-mers along with their multiplicity and connectivity from raw input reads to a hash table, and graph traversal, which traverses the graph to connect chain of k-mers as contigs. Based on our experiments on several popular tools [38], [41], [60], [66], graph construction takes 60% of the execution time, and graph traversal for contig assembly takes 30% of execution time. Therefore, this work focuses on accelerating these two phases.

B. Memory is the bottleneck for DBG processing

We profile three state-of-the-art parallel DBG assemblers, MEGAHIT [38], Abyss [60], and PASHA [41] to shed light on the performance of DBG assembly on CPU. First, DBG often has a large peak memory footprint since each input sequence needs to be decomposed to a set of overlapping k-mers and stored in memory in the graph construction stage. For example, a sequence of 100 bases requires only 200 bits to represent, but its k-merized form takes 4200 bits (k=30). This demands an architecture with a large memory capacity to accommodate the ever-growing genome dataset. Second, due to the highly random memory access pattern, DBG assemblers have a high L2 and L3 cache miss rate (45% to 75.17%), indicating DBG processing does not benefit from a deep memory hierarchy. The cache-unfriendliness of DBG processing causes frequent and slow main memory accesses. Our profiling results suggest, regardless of the thread count, a significant fraction (∼80%) of

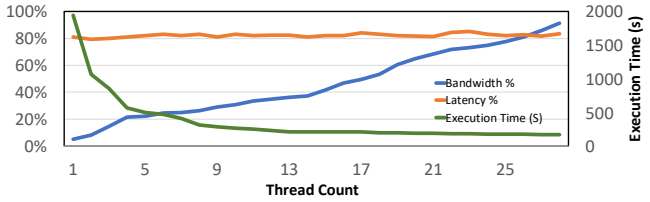


Fig. 2: **Profiling memory bandwidth and latency of MEGAHIT [38] using VTune [29] with workstation configured in Table II. The % latency means the percentage of CPU stall cycles caused by memory access. Constant high-percentage suggests memory latency always dominates the execution time.**

the CPU stall cycles are caused by memory access, shown as the relatively flat orange line in Figure 2. This also means that the computation per data element is too small to mask the high data access latency, and data is not efficiently reused. Third, similar to other data-intensive applications such as machine learning [32], [57], memory bandwidth is a major bottlenecks in scalability and performance of DBG processing. The blue line in Figure 2 plots the memory bandwidth consumption concerning thread count for MEGAHIT. While we observe the execution time decreases as the thread count increases the memory bandwidth consumption also increases, approaching our workstation's limit when all threads are being utilized, shown as the ascending blue line in Figure 2. It suggests that the memory bandwidth will become the bottleneck for a conventional server when we further scale up the parallelism. For instance, the bandwidth requirement is projected to reach 1 TB/s with 512 threads in our experiments. Both the degree of parallelism and the memory bandwidth needs to be increased simultaneously. Finally, increasing the thread count shows a diminishing return in terms of performance improvement, shown as the descending green line in Figure 2. This is because the DBG processing on compute-centric systems generates large amount of off-chip memory requests which cannot be served concurrently due to limited memory channels.

Overall, the above analysis indicates that DBG assemblers favor architectures with a high memory capacity, large parallelism (high core count), low memory latency, and high memory bandwidth. Since there is no evidence suggesting the possibility of compute-bound, a large number of cores with moderate computing power should suffice. Conventional systems cannot fill all requirements. For example, some high-end GPUs can satisfy the parallelism and bandwidth requirements, but they are limited by the onboard memory capacity, leading to frequent data swaps from host to accelerator. In this work, we exploit the emerging near-data processing (NDP) technology to accelerate DBG *de novo* assembly.

C. NDP Systems

NDP is a type of architecture where the data processing unit and storage unit are co-located in a single module. Emerging 3D-stacked DRAMs, such hybrid memory cube (HMC) and high bandwidth memory (HBM), are popular platforms to enable NDP functionality. A 3D-stacked DRAM integrates a logic layer in the memory die, which features

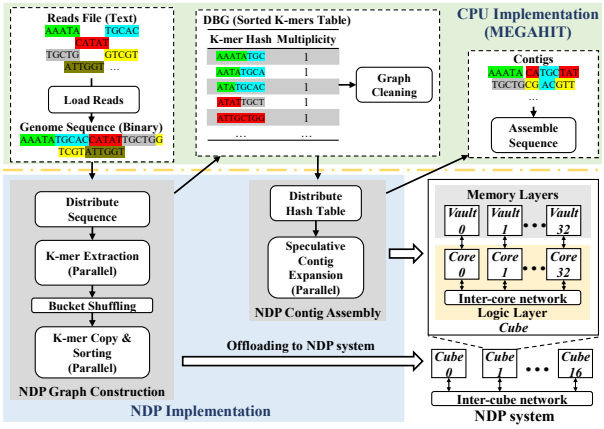


Fig. 3: **The overview of NDP-accelerated DBG assembly.**

highly parallel compute units to leveraging the low access latency and large internal memory bandwidth. Take the HMC as the example, each HMC cube has multiple vertical slices–vaults. The memory layers and the logic layer communicate through fast through-silicon vias (TSVs). There have been various HMC-based NDP systems [1], [12], [52], [67] where a small per-vault core (referred to as a NDP core) is embedded at the logic layer, re-purposing a vault to a near-memory compute unit. The NDP system can scale out by connecting multiple cubes using high-speed serial links to form a network of NDP cores. Scaling out the NDP system simultaneously increase the memory capacity, parallelism, and the aggregated memory bandwidth, which is ideal for parallel genomics workloads with a large memory footprint and high bandwidth demand. In this work, we evaluate the effectiveness of proposed designs in the context of HMC architecture which provides concrete parameters accessible to researchers. However, our optimizations may also be applied seamlessly to other 3D-stacked memories like HBM, which shares a similar degree of parallelism and partition strategy (e.g. channels v.s. vaults) [21], [48].

III. Overview of NDP-accelerated DBG Assembly

We propose our NDP-accelerated DBG assembler by modifying the widely used MEGAHIT tool [38] (Figure 3).

A. DBG Assembly Pipeline

We reuse the interface in MEGAHIT to support the NDP functionality in a general DBG assembly pipeline, which includes read loading, graph construction, contig assembly, etc. We replace the implementation of graph construction and contig assembly, which are performance bottlenecks in the pipeline, with our proposed NDP method.

MEGAHIT uses several intermediate data structures for transitions between different pipeline phases. We do not change these intermediate data structures used in MEGAHIT to keep the general pipeline intact in our implementation. Specifically, the NDP graph construction takes in the binary sequence data generated from the MEGAHIT read loading program, which supports general input formats of genome assembly, including single-end and double-end reads using different sequencing technologies [44], [56]. The NDP-based graph construction generates the sorted k-mers and writes them

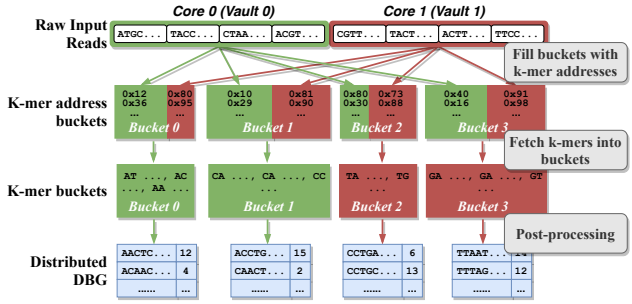


Fig. 4: **The overview of NDP-based DBG construction.**

into files that can be processed by the graph cleaning program in MEGAHIT. Then, the NDP-based contig assembly distributes the cleaned DBG (sorted *k*-mers) in the NDP system and traverses the DBG to build contigs using the proposed techniques. The NDP-based contig assembly generates the contig graph that will then be assembled by the original MEGAHIT implementation for the final sequence.

B. NDP Acceleration

The NDP architecture consists of multiple Hybrid Memory Cubes (HMC), and each HMC connects to the others using an inter-cube network [12], [34]. Each cube's memory is divided into several vertical memory vaults, and each vault is coupled with an integrated processing core which is connected to a memory controller for local vault access. We can schedule parallel applications on NDP systems by exploiting massive NDP cores. NDP system supports remote function calls based on message passing to handle inter-core communication without expensive coherence management [1]. In this work, we propose the implementation of graph construction and graph traversal (contig assembly) on the NDP system with optimization based on domain-specific knowledge.

IV. NDP-based Parallel Graph Construction

Figure 4 illustrates the flow of parallel DBG construction, and Algorithm 1 shows the pseudo code. The DBG is represented as a series of "buckets" distributed among all NDP vaults. The distributed DBG is built through the following steps: (1) Reads distribution. (2) Bucket allocation. (3) *k*-mer address scan. (4) *k*-mer extraction. (5) Post processing. We design an efficient bucket distribution procedure and message buffering and compression to improve the performance by reducing the inter-core communication.

A. NDP parallel graph construction

Input reads are first distributed to all NDP vaults. Then the NDP system sets up several buckets for cores to collaborate without interfering with each other. Building a DBG can be abstracted as putting *k*-mers into different buckets based on their hash values. Each bucket is divided into N non-overlapping partitions (lines 1 to 7), where N is the number of NDP cores. When an NDP core visits the bucket, it is confined to its partition, making concurrent bucket accesses among different cores possible. Then the buckets are assigned to NDP cores (line 8). The distribution of buckets is crucial to the performance of graph construction. Thus we design an optimized bucket mapping scheme, which is described in IV-B.

Algorithm 1: Pseudo code for building distributed De Bruijn Graph on a NDP system.

```
   input : Distributed raw read data - reads[num_reads]
           → cores[num_cores].seq_from
           → cores[num_cores].seq_to
   input : num_bucket
   output: de Bruijn graph table - dbg[num_kmers]
   /* Calculate the size and partition for each bucket */
1  #ndp_parallel_for
2  for c ← 1 to num_cores do
3      for seq ← cores[c].seq_from to cores[c].seq_to do
4          for kmer: seq do
5              b = hash(kmer)%num_buckets;
6              cores[c].bucket_size[b] + +;
7              buckets[b].size + +;
   /* Distributed buckets to NDP cores                    */
8  assign_buckets(buckets, cores);
   /* Copy k-mer addresses into buckets                   */
9  #ndp_parallel_for
10 for c ← 1 to num_cores do
11     for seq ← cores[c].seq_from to cores[c].seq_to do
12         for kmer: seq do
13             b = hash(kmer)%num_buckets;
14             buckets[b].addresses.add(&kmer);
   /* Copy k-mers from address                            */
15 #ndp_parallel_for
16 for c ← 1 to num_cores do
17     for bucket: cores[c].buckets do
18         for kmer_addr: bucket.addresses do
19             target_core = find_core(kmer_addr);
20             target_core.copy(kmer_addr, bucket.kmers);
   /* Bucket post-processing: sorting, remove redundancy,
      calculate multiplicity, etc.                        */
21 #ndp_parallel_for
22 for c ← 1 to num_cores do
23     for bucket: cores[c].buckets do
24         post_process(bucket);
25         dbg.add(bucket);
```

Next, a batch of buckets are selected in each iteration, and NDP cores fill those buckets with *k*-mer addresses by scanning its local reads (line 10 to 14). This is because decomposing reads into *k*-mers inflates the size of the input dataset by a factor of (n-k+1)*k/n (n = read length, k = K-mer size), processing all buckets simultaneously results in peak memory explosion. After addresses are filled for all buckets, each NDP core takes the ownership of its buckets by copying *k*-mers based on the addresses gathered from the previous step (lines 15 to 20). The two-pass creation of the buckets for DBG construction is superior than pushing the k-mers directly into the buckets for several reasons: the algorithm iteratively selects a batch of buckets to process, cores may have unbalanced amounts of k-mers belonging to the current buckets. If a single-pass paradigm is adopted, some cores will be busy "pushing" K-mers into the network to the destination buckets while other cores are idle. Furthermore, the "pushing" has a sequential-reads/random-writes pattern, incurring low cache locality. Since K-mer addresses (8-byte) are smaller than the actual K-mers (32-byte to 128-byte), "pushing" addresses incurs a smaller penalty. In the second-pass, cores fill their buckets with K-mers, and since buckets are roughly the same size in each batch, cores have balanced workloads. The second pass has a sequential-reads/sequential-writes pattern.

The *k*-mer may be stored in a remote core that requires

a remote function call to copy the data. Since we evenly distribute sequences to NDP cores in the original order, we can easily locate the target to send the remote function. This step suffers from massive fine-grained communication overhead since many *k*-mers are from reads distributed in remote vaults.

Finally, a post-processing stage (line 21 to 25) is involved to reduce bucket size, since many *k*-mers (as many as 80% [16]) are redundant due to the deep sequencing coverage and repeat patterns in genomes [16]. A common practice is to sort the *k*-mers in a bucket, allowing us to obtain the multiplicity (number of occurrences) of each *k*-mer as a helpful by-product.

B. Bucket Distribution

To design a good bucket distribution scheme, one needs to consider the origins of *k*-mers in a bucket. Figure 5 shows an example of two buckets and three NDP cores (vaults). A large portion of read partition 0 (red) is hashed into bucket 0; thus, co-locating bucket 0 with read partition 0 can significantly reduce the number of remote *k*-mer fetch requests. Similarly, bucket 1 has a high concentration of *k*-mers from the read partition 1 (green), so it is more suitable to be assigned to the vault 1. There is anywhere between 29% to 40% reduction of messages over a naive random bucket mapping if the origins of *k*-mers are considered. One possible explanation for such a phenomenon is that real genomes often contain many regions of repeat patterns. For example, about 8% of the human genome consists of so-called tandem repeats, which are low complexity short sequences that occur multiple times in a row (e.g. "CAGCAGCAG...") [13]. The commonly adopted hashing schemes that operate on the binary form of a *k*-mer pattern will inevitably try to fit *k*-mers obtained from these repeats into a small group of buckets.

However, simply reducing the number of messages passed among the NDP cores may not be the optimal solution, as it fails to consider the non-uniform latency of switching a packet in some networks. For example, in a mesh-style network, the latency of switching a packet is correlated to the distance between two nodes, since a packet arrives at its destination through a series of hops, and each hop adds a certain amount of additional latency. Figure 6 illustrate a situation where a message-reduction-based bucket shuffling strategy does not work well. Suppose buckets have roughly equal amounts of *k*-mers that need to be fetched from each remote vault in an NDP system with a mesh NOC. The total amount of remote messages generated is the same regardless of the bucket location. However, when the hop count per message is considered, it is a poor choice to put this bucket at the four corner vaults. For example, if each remote vault contributes 10 *k*-mers into the bucket, then a bucket generate $10\times1\times2+10\times2\times3+10\times3\times4+10\times4\times3+10\times5\times2+10\times6\times1=480$ total message hops at vault 0, 320 at vault 5, and 400 at vault 13. Therefore, total message hops should be considered if we strive to reduce inter-core communication costs.

The slowest core limits the run time of the parallel graph construction, and the inter-core communication takes the majority of the execution time. Thus the optimal bucket mapping is the one that generates the least communication for the slowest core. For an NDP core, the communication cost of

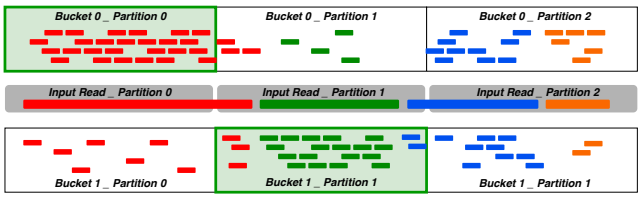


Fig. 5: **Bucket shuffle based on the origins of *k*-mers.**

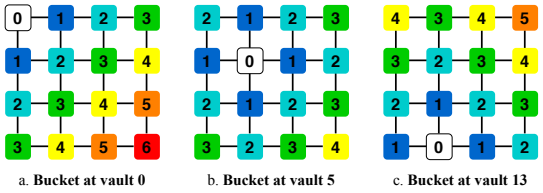


Fig. 6: **Hop distance from the source core (in white) to different remote cores (in color).**

processing its share of buckets can be approximated as the total message hops needed to fetch all *k*-mers. However, to find such optimal bucket mapping is infeasible. Let's consider a simpler case where the number of buckets mapped to each core is the same. With 65536 buckets and 512 NDP cores (vaults), the number of possible mappings that need to be checked is $\binom{65536}{128}\times\binom{65408}{128}\times\binom{65280}{128}\times...\times\binom{128}{128}$. A naive heuristic that selects the least amount of communication cost for each bucket can easily suffer from workload imbalance. We describe below a greedy solution that addresses both the run time concern and the imbalance concern.

After each bucket's size is obtained (line 11), all buckets are ranked in descending order based on their sizes and put into a list. The bucket distribution logic runs in a loop in which each iteration selects a batch of n buckets from the list, with n being the number of NDP cores. For an NDP system with fully-connected networks, each bucket is assigned to a vault based on its largest partition. A bucket will be randomly selected if two or more partitions have the same size. The vault that has been assigned with a bucket in this iteration will not be assigned with another one. When a bucket needs to be assigned to an occupied vault, the bucket is assigned to a vault with the second-highest *k*-mer contribution (second highest partition). This process repeats until all buckets are assigned. For an NDP system without a fully-connected network topology, the bucket shuffling step is the same as the above procedure with minor tweaks. Instead of choosing a winning vault for each bucket based on its partition sizes, the bucket is assigned to the vault that generates the smallest hop count. This shuffling implementation adds an insignificant amount of overhead (<1%) and works well in our evaluation.

Each NDP core is provisioned with a table that indexes buckets to their owner vaults/cores. The number of table entries is equal to the number of buckets, which is 65536 in our evaluation. Each entry has $\log_2 65536 = 16$ bits to represent bucket IDs, and additional bits to represent core IDs (9 bits for 512 cores). The table adds 1.2% total storage overhead per HMC cube. Searching this table is a constant time operation since the bucket index is the hash value of a *k*-mer.

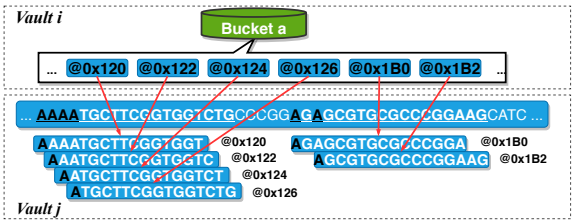


Fig. 7: **Message compression by leveraging the overlapping bases of consecutive *k*-mers.**

C. Message Buffering and k-mer Compression

1) Message Buffering: In the graph construction step, each NDP core copies a *k*-mer from a remote vault by sending that vault's owner (an NDP core) an extraction request wrapped in a message. The remote NDP core responds to the request immediately by sending the *k*-mer back in another message. This is inefficient since each message's payload can fit multiple *k*-mers, and each request has no dependency on each other. An obvious optimization is to delay the responses and aggregate multiple *k*-mers into one message. At each vault, we provide $n-1$ buffers corresponding to the rest $n-1$ remote cores. The specific buffer parameters are discussed in Section VI.

2) k-mer compression: This compression technique is used in conjunction with the message buffering to improve a message payload density. The key observation is that since the *k*-mer addresses are put into a bucket by sequentially sliding a window on input reads (with variable stride lengths), there is an opportunity for data reuse when copying *k*-mers pointed by those addresses. Figure 7 illustrates this idea. A naive way of sending *k*-mers from Vault j to Vault i is to lay them out exactly in the message payload one by one. Suppose the message payload size is 64 bit and 2-bit/base. The uncompressed format allows two *k*-mers to be sent through one message. A more compact representation of those *k*-mers is to copy the entire sequence from the first base of *k*-mer at 0x120 to the last base of *k*-mer at 0x126 (19 bases) and provide a small array of offset pointers to distinguish each *k*-mer. This allows the same message payload to fit four *k*-mers.

The compressibility of *k*-mers in a packet depends on several variables: the number of buckets, size of k, hash function, genome repeat patterns, etc. Deriving a formula to predict the effectiveness of packet compression accurately is out of this project's scope. We empirically evaluated this idea using an E.coli DNA sequence and realistic DBG assembler settings: k=22, 65536 buckets, and the first four bytes of each *k*-mer are hashed. We find that over 20% of consecutive *k*-mer pairs in a packet are overlapped, and the proposed compression technique trims away more than 10% of redundant bases. We also analyze how many bases every overlapping *k*-mer pair shares. The distribution is summarized in Figure 8. The result suggests that each pair of overlapping *k*-mers have a high chance of sharing more than half of their content.

V. NDP-based Parallel Graph Traversal

This section introduces the NDP-based graph traversal. We exploit the NDP system's parallelism to construct contigs and use a speculation mechanism to accelerate contig expansion.

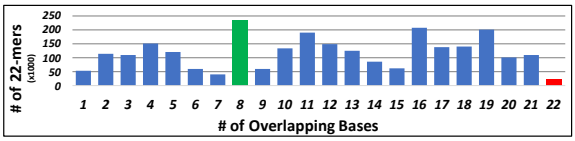


Fig. 8: **The distribution of the number of bases that are overlapped for each consecutive 22-mer pairs.**

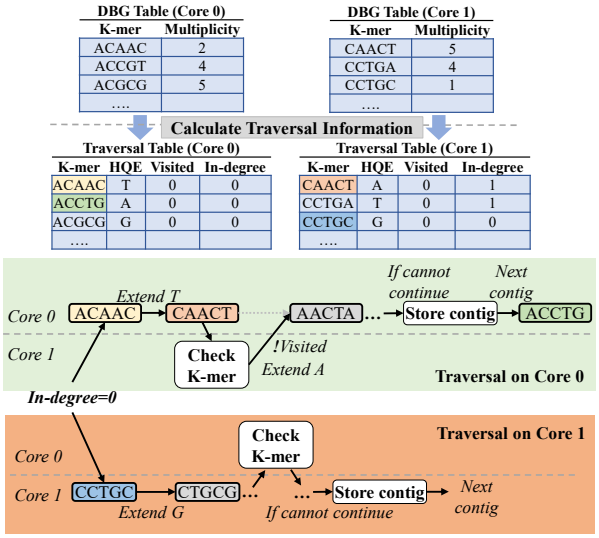


Fig. 9: **The overview of NDP-based graph traversal.**

Algorithm 2: Pseudo code for NDP-based graph traversal (contig assembly).

```
input : Distributed DBG table - dbg[num_kmers]
        → cores[num_cores].kmer_from
        → cores[num_cores.kmer_to
output: Contigs built by traversal - contigs
/* Calculate k-mer information                          */
#ndp_parallel_for
for c ← 1 to num_cores do
    for kmer ← cores[c].kmer_from to cores[c].kmer_to do
        /* Find the high-quality extension (HQE)        */
        for c: ['A','T','G','C'] do
            kmer.HQE = max_multiplicity(kmer[1:] + c);
        /* Update the in-degree of HQE                  */
        target_core = find_core(kmer.HQE);
        target_core.increament(kmer.HQE.in_degree);
/* Parallel contig assembly                             */
#ndp_parallel_for
for c ← 1 to num_cores do
    for kmer ← cores[c].kmer_from to cores[c].kmer_to do
        if kmer.in_degree == 0 then
            contig = kmer;          // Initiate a contig
            target_core = find_core(kmer.HQE);
            while !target_core.get(kmer.HQE.visited) do
                contig.expand(kmer.HQE);
                kmer = kmer.HQE;
                target_core = find_core(kmer.HQE);
            contigs.add(contig);
```

A. NDP Parallel Graph Traversal

Figure 9 shows the high-level flow for NDP-based graph traversal, and Algorithm 2 shows the pseudo-code.

1) Data Initialization: The input of graph traversal is the DBG data (hash table) generated in the graph construction phase. The hashing is supported in the general-purpose NDP cores. We use a leveled hashing scheme to resolve conflicts.

We distribute the DBG (hash table) over different NDP cores. As discussed in Section IV, the DBG is divided into buckets, each of which is stored in a core.

2) Information Calculation: To efficiently construct contigs, we need to calculate *k*-mer information used during the traversal. Such information includes the high- quality extensions (HQE) and in-degree of each *k*-mer. High-quality extension (HQE) is the most likely extension for each *k*-mer. DBG assemblers use HQE to remove forward *k*-mers, which are introduced by read errors [16]. We point out that the graph traversal step uses HQEs to generate contigs (long sequences without branches), instead of the whole sequence. If a K-mer has multiple HQEs, the assembler stops extending the current contig because the K-mer may be a branch. The branches caused by repeated DNA patterns will be considered after the traversal phase to assembly the full DNA sequence. Furthermore, in-degree is used to filter the start *k*-mer for each contig to remove redundant traversal. Specifically, the DBG assembler only constructs a contig from a *k*-mer with no in-coming edges. This method avoids assembling the same contig by different cores.

Each NDP core processes the information for its allocated *k*-mer independently (line 1 - 7). Each core sequentially processes *k*-mers and checks all 4 possible bases that can extend the *k*-mer (line 4 - 5). The HQE of each *k*-mer is determined by the base that leads a *k*-mer with the highest multiplicity. Then, the core checks the pre-loaded bucket table to locate the core that handles the HQE *k*-mer (line 6), and increases the in-degree of the HQE *k*-mer in the target core.

3) Parallel Contig Construction: The next step is to assemble contigs by graph traversal, where each NDP core constructs contigs independently by selecting a local *k*-mer as the first segment of a contig (line 8 - 17). As mentioned previously, each NDP core only selects *k*-mers without in-coming edges and expands the contig in one direction to avoid redundant work (line 11). To extend a contig, the source core, which is the core constructing the contig, checks the availability of the HQE of the current *k*-mer in the target core. If the *k*-mer is stored in the local vault, the source core searches its DBG table. Otherwise, the source core uses a remote function call on the target core to check the availability of HQE.

The result of *k*-mer expansion depends on two facts: 1) whether the *k*-mer exists, and 2) whether another contig has already included the *k*-mer. If the *k*-mer exists, the target core checks the "visited" tag of the *k*-mer to determine whether the *k*-mer has been used or not. If the source core receives a response from the target core that the *k*-mer is available for the extension, the source core uses the HQE to extend the current contig. Otherwise, the source core adds the current contig to the result ($contigs$) and selects another local *k*-mer as the seed for the next contig construction.

B. Speculative Contig Expansion

The graph traversal phase also suffers from inefficient inter-core communications, especially during the contig expansion where the source NDP core needs to send the query to a remote core and wait for the remote core responses to search the requested *k*-mer in the *k*-mer table. All these operations, including bi-directional communication and the search, are in the critical path of the contig expansion. Based on our experiments, the contig expansion would spend 70% of its time on inter-core communication. Therefore, it is important to reduce this time to achieve the full potential of NDP systems.

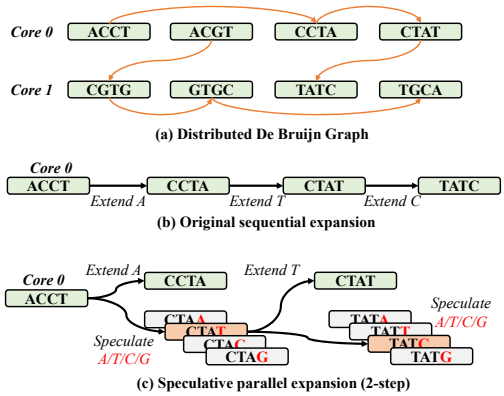


Fig. 10: **The speculative search optimization.**

1) Optimization Overview: We propose a speculative contig expansion shown in Figure 10. In the speculative contig expansion, each NDP core searches multiple steps ahead, instead of only the HQE. The speculation's insight is to hide the latency of *k*-mer query by parallelizing subsequent operations.

Unlike the current contig that has the information of HQE, we do not know what will be in future steps for a query if we successfully extend the current contig with the HQE. The NDP core needs to search for all possible *k*-mers in the speculative steps to guarantee the speculative contig expansion's correctness. The number of possible *k*-mers is 4^{n-1}, where n is the number of speculative steps. During speculative search, an NDP core calculates hash values and sends search requests to target cores for all possible *k*-mers in the next n steps.

2) Operation Combining: An $n-step$ speculative search can achieve up to $O(n)\times$ performance improvement over the default one-step expansion. However, the speculation would introduce significant overhead without any optimization because of more data communications and operations for searching all possible *k*-mers. The number of messages that are generated could grow exponentially while the performance improvement is always linear as we increase the speculation depth. Thanks to DNA sequences' nature, we can significantly reduce the speculation overhead by combining speculation for similar *k*-mers into a single move. All possibles *k*-mers in a speculation step share the most significant bases. Therefore, these *k*-mers are stored in a contiguous memory location in the sorted *k*-mers table (bucket). We may only need to send one message for all possible *k*-mers in a speculative step since the same target core handles these *k*-mers. The target core can quickly access continuous memory addresses by utilizing the data cache in the core. For example, in Figure 10(c), Core 0 may store all four 1-step speculative k-mers ($CTA\{A,T,C,G\}$), and Core 1 may store all sixteen 2-step speculative k-mers ($TA\{A,T,C,G\}\{A,T,C,G\}$) based on the range of hash table. In this case, a two-step speculation only requires 2 messages (1 per core). It is possible that k-mers in a speculation step are stored across cores, requiring

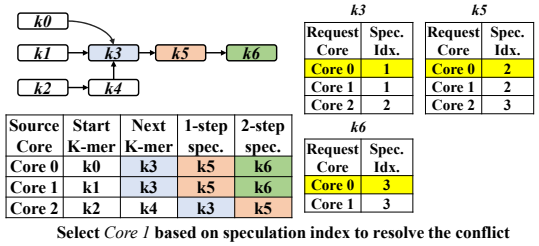


Fig. 11: **Resolving speculation conflicts.**

TABLE I: Programming Interface

Operation	Remote Function Call
Copy *k*-mer	get(id, A func, A addr, A ret, S ret_size)
Set Data	put(id, A func, A addr, S size)
Request *k*-mer	put(id, A func, A addr, S size)
Get Buffered *k*-mers	get(id, A func, A ret, S ret_size)
Search *k*-mer	get(id, A func, V hash, A ret, S ret_size)

multiple messages. In our experiments, we only observe a trivial amount of speculations (up to 5-step) requiring multiple messages in a single step because of the large data size.

3) Conflict Resolution: Another issue with speculative contig expansion is that we need a more complex mechanism to resolve the conflicts between cores that simultaneously access specific *k*-mers. Once an NDP core receives results of all possible *k*-mers in the next n steps, it tries to extend its current contig by checking the HQEs and corresponding "visited" tags of *k*-mers sequentially. It needs to send messages to cores handling the extended k- mers to avoid redundant work (setting the "visited" tags). However, without an efficient mechanism, the overhead of synchronization can eliminate the benefit of speculation. Figure 11 shows the example of synchronizing three cores in 2-step speculation.

To efficiently resolve conflicts, we propose a lightweight mechanism, nearest source assignment, to solve the conflict in the target core. Specifically, each source core extends all speculative *k*-mers locally as further as possible and then sends the confirmation messages to all target cores to notify the success of *k*-mer extension. The source code also sends a speculation index, which is the position of the *k*-mer in the speculation path, along with each message. The target core receives confirmation messages from different source cores on the same *k*-mer. It picks the source core, which sends the smallest speculation index in the message as the core to use the *k*-mer for expansion. If multiple cores send the same speculation index, the target core picks the core with the smallest core index to break the equality. This mechanism can effectively resolve the conflicts because different contigs will follow the same path when they conflict on the same *k*-mer. Therefore, the nearest source assignment can avoid potential deadlocks in a continuous *k*-mer path.

VI. Software and Hardware Support

This section discusses the implementation of software and hardware to support the proposed ideas.

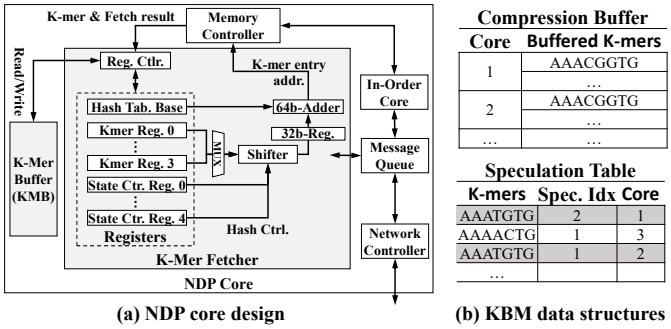
A. Programming Interface

We utilize the message passing and remote function call in Tesseract [1] as the programming interface for its versatile programming interface and lightweight hardware support for message-passing (i.e., message queue). Table I lists implementations of key operations required in NDP-based DBG assembly. We use the blocking (get) or non-blocking (put) remote function call to implement different operations, where the remote function call is based on a message passing mechanism. Specifically, copying a *k*-mer from a remote call requires a blocking remote function call (get), where the parameters require the target core, the address of target *k*-mer, the address of return value, and the size of return value. We use A, S, and V to represent the address type, the size type, and the value type respectively. However, the blocking get function cannot support our proposed buffering and compression mechanism. Therefore, we propose a request operation that uses the non-blocking put to notify a target call to store the target *k*-mer in the message buffer. During the execution, each core calls the request function for each *k*-mer while maintaining a counter for the number of messages that have been requested for remote cores. When the counter is equal to the buffer size (introduced in Section VI-B), the core calls a get function to the target core to get all buffered *k*-mers. The target core compresses the buffered *k*-mers and sends the results back to the source core. After the blocking get function call, the source core resets the counter for a specific target core and continue execution.

To enable programmers to implement NDP-based DBG assemblers, we need a framework that combines the parallel computing and the proposed programming interface based on message passing. Since we have no access to the real NDP hardware, we use a simulation-based method to emulate the proposed NDP assembler. In our implementation, we simulate OpenMP-based programs in Sniper simulator [8], which uses Pin-tool [43] as the front-end to generate simulation statistics for multi-core architectures. We insert specialized APIs using Sniper's magic instruction in the OpenMP program so that Sniper can recognize the message-passing based NDP operations. We implement the simulation logic for different message-passing functions using Sniper's synchronous and asynchronous timing models to generate the final simulation results for NDP architectures. Future work can follow a similar scheme to realize the proposed assembler in a real NDP hardware. For example, the framework can extend the syntax of widely used parallel programming APIs (e.g., OpenMP) to include function calls of message-passing, and implement a specialized runtime to schedule operations on NDP cores.

B. Hardware Support

We propose several lightweight hardware components inside each core in our NDP architecture to support the NDP functionality. Similar to Tesseract [1], each core uses a message queue and a network controller to process remote function calls based on message passing. In addition, we add two lightweight hardware components, a k-mer fetcher (KMF) and a k-mer buffer (KMB) to support the proposed optimizations. Figure 12(a) shows the architecture of the proposed NDP core. Specifically, the k-mer fetcher (KMF) is the unit which we can offload operations for the proposed optimizations from the NDP core. KMF can decode the potential memory addresses of k-mers based on the hash value, and generate memory commands directly to the memory controller. The

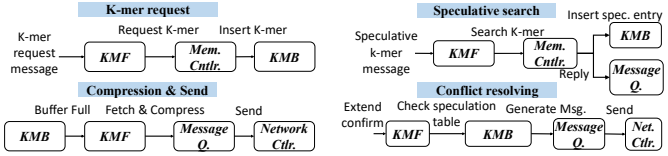


(a) NDP core design (b) KBM data structures

Component	Parameters
Shifter	2 cycles/load&shift
Adder	64-bit 4-pipelined, 4 cycles/add
K-mer buffer	64KB SRAM, 64B/block, 1 cycle/access

(c) Parameters of customized hardware components

Fig. 12: **Hardware support for NDP-based DBG assembly.**



(a) Operations offloading during *graph construction*. (b) Operations offloading during *graph traversal*.

Fig. 13: **Operations in hardware components.**

KMF contains several 64-bit hardware registers to store the working information during the k-mer fetching, including the base address of the hash table (1 register), the k-mer data (4 registers), and state control information (4 registers). To generate the address for k-mer fetching, KMF first loads the base address of the hash table from the in-order core. Then, KMF uses a shifter to generate the offset of a k-mer by concatenating different bits of k-mer registers. The offset is stored in a 32-bit register, which is added with the base address in a 64-bit adder. KMF then sends the generated address to the memory controller and receives k-mer data in the k-mer registers for future operations (e.g., writing to the k-mer buffer. The k-mer buffer (KMB) stores k-mer related data which is configured to different formats for graph construction and graph traversal, as shown in Figure 12(b). During the graph construction, in order to support the k-mer compression, KMB acts as a compression buffer which stores the requested k-mers grouped by the requester core. During the graph traversal, KMB is organized as a speculation table which stores the searched k-mer, requester core, and the speculation index for conflict resolving as illustrated in Section V. Figure 12(c) shows key parameters of the proposed hardware components used in our evaluation. We implement the components of KMF using Verilog HDL and synthesize the design on Synopsys Design Compiler. The synthesized design is placed and routed using Synopsys IC Compiler. The KMB parameters are estimated using the analytical tool CACTI-3DD [11] on 22nm technology node.

Figure 13(a) and Figure 13(b) show the operations offloaded from the NDP core to the lightweight components during graph construction and graph traversal phases. During graph construction, KMF of each core handles k-mer requests to the local k-mer from other cores. When receiving a k-mer request message (with the address), KMF generates memory commands to the memory controller which will fetch the k-mer. KMF then stores the k-mer to the corresponding entry of compression buffer (stored in KMB). If all entries of the requester core is full in the compression buffer, KMF compresses all k-mers requested by the requester core and generate a compression message. During graph traversal, KMF handles speculative search requests from other cores. Since the speculative search may request a non-existing k-mer, KMF generates memory commands for search operation in the local hash table based on the k-mer's hash value. If the k-mer exists, KMF inserts the k-mer with the requester information (e.g., speculation ID) in the speculation table. No matter whether the k-mer exists or not, KMF sends a message to the requester core about the search result. If a requester core wants to confirm the extension with a speculation k-mer, KMF fetches all entries about the requested k-mer in the speculation table, and resolves the conflict based on the algorithm illustrated in Section V. As compared to the pure-software implementation, hardware-assisted optimizations reduces data movements between the memory and the in-order core. Furthermore, the added hardware components can directly communicate with memory controller and message interface to reduce the latency of optimization in the critical path.

TABLE II: Workstation and NDP Configuration

CPU Model	Intel(R) Xeon(R) E5-2658 v4
Core/ Thread/ Frequency	14/ 28/ 2.30 - 2.80 (GHz)
L1/L2/L3 Cache	32 (KB) / 256 (KB) / 35 (MB)
Main Memory	DDR4-2400 MHz, 54GB/s
Memory Organization	32GB / 2 Channels / 2 Ranks
HMC 2.0 Organization	8 DRAM layers, 8Gb/layer, 8GB/cube, 32 vaults, internal (external) bandwidth: 512GB/s (480GB/s)
NDP cores	1 GHz, single-issue, in-order, 32 KB I$ and D$, LRU, 80 mW, 0.51 mm^2
HMC Memory	tCK = 1.6 ns, tRAS = 22.4 ns, tRCD = 11.2 ns, tCAS = 11.2 ns, tWR = 14.4 ns, tRP = 11.2 ns
NOC Configuration	Crossbar network, 64 KB/message payload
Inter-cube Network	2 cycles/hop, 64 bits/cycle, 2D-Mesh (default) / DragonFly / Fully-connected

VII. Experimental Setup

A. Simulation

We emulate the execution of our NDP-based DBG assembler using multi-threading supported by OpenMP [10]. Specifically, we create a thread for each NDP core and manually assign different tasks and data structures to threads. The proposed NDP system, including all DRAM vaults and NDP cores, is modeled in Sniper [8] according to parameters reported in Table II. The parameters are gathered from previously published work [1], [12], [52], [67], data-sheet for commercial products [45], and simulation in Cacti [63] and McPAT [63]. We use a Pin-tool [43] front-end to tag NDP data structures' addresses in the simulation. Therefore, Sniper can recognize and operate on these NDP data structures using NDP-specific models of remote function call based on message passing. We use Ramulator [35] to model the memory behaviors since Sniper lacks a detailed memory model. We use Cacti [63] to simulate the performance and power of customized buffers at 32nm technology. Each NDP core takes

TABLE III: Genome Datasets

Genome Name	Size
Escherichia coli O157 (E-Coli)	5,528,445 bp
Homo sapiens chromosome 3 (Human)	198,295,559 bp
Ananas comosus cultivar (pineapple)	24,880,688 bp

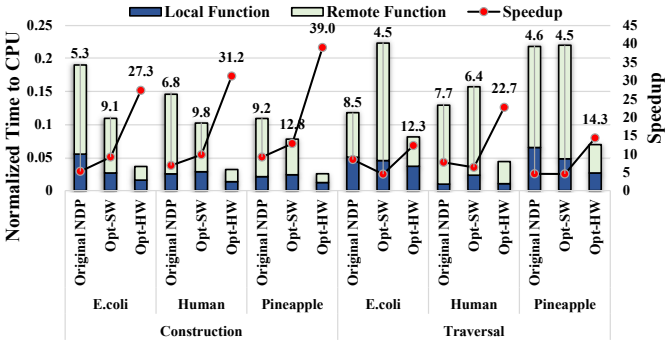


Fig. 14: **Performance comparison with the baseline on graph construction and graph traversal.**

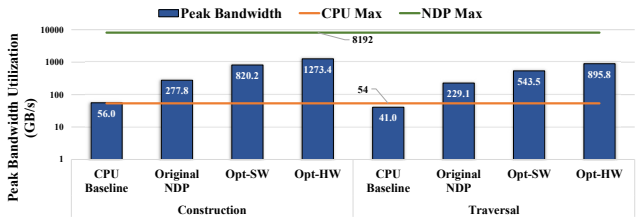


Fig. 15: **Memory bandwidth utilization for `Human` genome.**

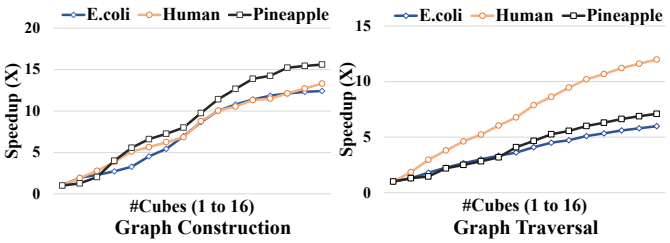


Fig. 16: **Scalability results from 1-cube to 16-cube.**

$0.51mm^2$ chip area and 32 NDP cores only consume 7.2% of the chip area available in the HMC logic layer ($226mm^2$ [1]).

B. Baseline System

The baseline performance is measured from MEGAHIT [38] running on a workstation configured in Table II. We should note that CPUs are the predominant platform for DBG assembly, instead of GPUs, and MEGAHIT is one of the fastest implementations [30], [38], [41], [60] that is capable of assembling a large genome in parallel. We do not compare to GPU, since all of the GPU-based DBG assemblers we find are deprecated [33], [38], [42] due to lack of support and performance. In fact, we are informed by the authors of MEGAHIT that the GPU-implemented MEGAHIT is slower and harder to use than its CPU counterpart.

C. Workloads

We test DNA sequences from three species downloaded from GenBank [6] as shown in Table III. We use a next-generation sequencing read simulator [23] to generate NGS reads using Illumina technology [26]–[28]. We set the fold of coverage, length of reads, and mean size of DNA fragments to 20, 150, and 200 to generate sufficient simulation data.

VIII. Results

A. Overall Performance Analysis

Figure 14 shows the results of a comparison between the 16-cube NDP system and the CPU baseline. We compare the performance of the optimized NDP implementation (`Opt-HW`) to the `CPU baseline` and the NDP implementation without optimizations (`Original NDP`). `Opt-SW` represents software-implemented optimizations All results are normalized to the `CPU baseline` and we break the total execution time into the time on local functions and remote functions.

Comparison to CPU and original NDP. On the one hand, the original NDP is 7.1× and 6.9× faster than the CPU baseline on graph construction and graph traversal. This result indicates the simply parallel NDP solution does not fully utilize the hardware, considering the number of cores in the NDP system is much larger than that in the CPU baseline. On the other hand, the optimized solution is 32.5× and 16.4× faster than the CPU baseline for graph construction and graph traversal, respectively. The performance improvement provided by `Opt-HW` over `Original NDP` results from the reduced inter-core communication caused by the proposed optimization techniques, including bucket shuffling, message buffer and compression, and speculative contig expansion.

Comparison to software-implemented optimizations. The result shows that the performance improvement for graph construction is more significant than that for graph traversal. It is because the de Bruijn graph has a random structure that may cause cores to have unbalanced workloads. It is also consistent with the observation that the NDP solution performs better on a large genome than a small genome. In general, the proposed techniques perform better on large genomes that exhibit high-degree parallelism and sufficient per-core workload to exploit the parallelism of NDP hardware. `Opt-HW` outperforms `Opt-SW` by 3.1× and 3.2× on average for graph construction and graph traversal respectively. The performance gain provided by the hardware-implemented optimizations results from the reduction of memory accesses and updates for the optimization data structures.

Bandwidth utilization. Figure 15 shows the memory utilization for different systems running graph construction and graph traversal on *Human* genome. We get the CPU-baseline result from VTune [29] and NDP configurations from simulation. The results show: (1) NDP solutions, which run 512 parallel threads, require significantly more bandwidth than the CPU baseline maximum bandwidth. (2) The proposed optimization can increase the memory bandwidth utilization because of the better performance than the NDP baseline.

B. Performance Scalability

Figure 16 shows the performance of DBG assembly on the different number of NDP cubes for three genomes. We scale the system from 1 cube to 16 cubes. The 16-cube NDP system is 12.4× to 15.6× faster than the single-cube system

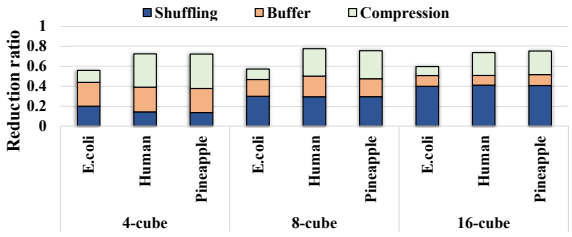


Fig. 17: **The reduction of inter-core message passing provided by optimizations for graph construction.**

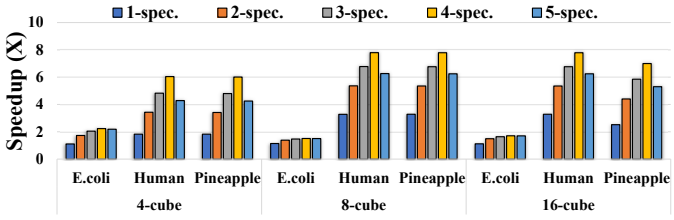


Fig. 18: **The performance comparison among different steps for speculation.**

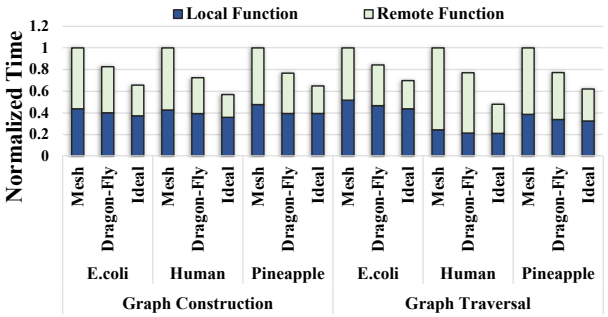


Fig. 19: **The performance comparison among different network structures.**

over three genomes for graph construction. Such results show good scalability of the NDP implementation. The performance of NDP implementation depends on DNA patterns in the dataset. Suppose the dataset has a lot of repeated patterns (e.g., Pineapple). In that case, the NDP implementation has better scalability, because we can significantly reduce the data access time by mapping the buckets of repeated patterns in the same core with the corresponding sequence.

For graph traversal, the 16-cube NDP system is only 6.0×, 12.0×, and 7.1× faster than the single-cube system for E.coli, Human, and Pineapple, respectively. The overall improvement provided by the large systems is much less than that in graph construction. The reason is that graph traversal has more randomness in the workload, thus is less likely to schedule balanced workloads over the NDP cores in the large system. The results over different genomes also show that the NDP implementation has better scalability in the large genome.

C. Inter-core Communication Reduction

Figure 17 shows the effects of the proposed optimization on the reduction of the inter-core message. The reduction ratio is calculated in the order of shuffling, buffering, and compression. Our experimental results show that bucket shuffling can reduce 14% and 40% of inter-core messages in a 4-cube system and a 16-cube system, respectively, over a random bucket mapping scheme. The gap between small systems and large systems results from that small systems have an even distribution of buckets because each core is allocated more sequences than larger systems.

Unlike the bucket shuffling, the message reduction provided by k-mer buffering and compression becomes less when increasing the system size. Specifically, k-mer buffering (compression) reduces 24% (26%) of messages in the 4-cube system while reducing only 10% (15%) of messages in the 16-cube system. This is because large systems schedule fewer messages for each core, so that the opportunity for buffering and compression becomes less than smaller systems. In general, our experiment on the data movement reduction shows that the bucket shuffling and k-mer buffering and compression can work well together to reduce the number of inter-core messages in different sizes of systems.

D. Exploration on Speculation

Figure 18 shows the exploration of the speculation steps for graph traversal. We test different speculation steps and show the speedup over the baseline without any speculation. The result shows that the four-step speculation has the best performance for all workloads on systems of different sizes. Smaller speculation steps may not fully exploit the available memory bandwidth and parallelism of the NDP system, while a larger speculation steps have larger overhead of resolving the conflicts between different NDP cores.

E. Exploration on Network

Because previous works show that the NDP system's interconnect plays a critical role in the performance and energy consumption, we also explore different interconnect structures in the baseline NDP architecture. Figure 19 shows graph construction and graph traversal execution time on three structures: mesh, dragon-fly [34], and an ideal fully-connected network. All results are normalized to the mesh structure. The experiment shows that the dragon-fly network can improve the mesh structure's performance by 1.3× on average, while the performance-optimized ideal network is 1.7× faster than the mesh. However, the ideal network incurs 5.4X and 2.7X higher area overhead than the mesh and dragon-fly configurations.

F. Energy Efficiency

We estimate the energy of NDP system based on the average active cycles of cores and memory and the power values reported in the official product description [45] and previous works [1], [52]. Our results show that the proposed NDP system consumes 28.9 × and 15.0 × less energy for graph construction and graph traversal than CPU. Such energy reduction mainly comes from the faster execution. The average power consumed by NDP is higher than the CPU baseline because of the higher power consumed by the memory layers and the NoC power consumption. However, previous work [1] shows that such power consumption density in the memory chip will not exceed the thermal constraints. In this work, we only added a small storage component with a controller in the original NDP hardware, which has trivial power and area

overhead ($< 2\%$). Therefore, the proposed NDP-based DBG assembler is practical in terms of power and thermal efficiency.

G. Comparison with Other Distributed Algorithms

The DBG processing of large genomes are also deployed on distributed-memory parallel computers using framework such as MPI due to their scalability (large capacity and high core count). Notable distributed-memory assemblers are Ray [7], PASHA [41], YAGA [31], ABySS [60], HipMer [17], and PakMan [18]. This work shares similarities with distributed-memory DBG assemblers at high-level. For example, addressing the communication imbalance issues during the parallel graph construction phase and avoiding traversing the same contig by multiple processing nodes repeatedly. If handled inefficiently, the overhead of orchestrating nodes outweighs the performance benefit of parallelization. These challenges are typically not found in shared-memory DBG assemblers, which primarily focus on optimizing algorithm complexity and assembly quality. However, migrating existing distributed-memory DBG schemes into an NDP system is a complex undertaking. Each node in the distributed-memory system handles multi-threading workloads with large memory footprint, but each NDP core is single-threaded with limited memory capacity. For example, PakMan [18] compresses the DBG into a compact graph with macro-nodes to ensure each compute node can fit the whole compact graph during the graph traversal phase, where each process can concurrently traverse multiple independently paths. Therefore, the optimization for cross-node communication in distributed-memory systems is too coarse-grained in the NDP implementation.

We provide an indirect comparison with one of the state-of-the-art distributed assemblers. As reported in the previous work [18], PaKman offers 9.3× speedup over IDBA-UD [49] with 40-cores@2.2GHz in its MPI-based shared-memory mode. Assuming performance scales linearly with core frequency, and since both PaKman and our work demonstrate linear scaling w.r.t core count, PaKman offers 54.08× speedup over IDBA-UD with 512-cores@1GHz. With 512-cores@1GHz, our work outperforms MEGAHIT by 31.6× which is already 3.5× faster than IDBA-UD (110.6×). This result shows our design is about 2× faster than PaKman even if PaKman can be perfectly mapped to an NDP architecture. Finally, this work leverages software/hardware codesign to speedup DBG assembly. Without the appropriate hardware support (e.g., the latency/bandwidth advantages of PIM and our customized hardware components), the software optimizations alone do not achieve the best results.

IX. Related Work

Non-genome NDP accelerators. There are similar 3D-stacked NDP accelerators for graph processing [1], [67], pointer chasing [22], and large-scale data analytics [12], [51], [52]. Some aspects of these work are similar to ours, such as minimizing communication, optimizing data partitioning, and providing a framework for the proposed architectures. However, these works are not directly applicable to DBG.

PIM bio-accelerators. There are several PIM accelerators for bioinformatics workloads. Wu et al. [64] proposes an in-situ solution which fits minimalist bitwise operation logic inside DRAM chips, and utilizes subarray-level parallelism to support massively parallel *K*-mer matching. GenCache [46] modifies the SRAM chip to support sequence alignment. Medal [25] leverages off-the-shelf DRAM components to build a DNA seeding accelerator. RADAR [24] is a 3D-ReRAM based accelerator for BLAST. AligneR [70] is a ReRAM-based PIM architecture which accelerates the bottleneck stage of genome sequencing. FindeR [71] enhances the FM-Index EPM search throughput in the gnomic sequencing step using commodity ReRAM chips. These works target different stages of genome pipelines. To the best of our knowledge, this is the first PIM-based *De Novo* assembly accelerator.

DBG assemblers. We have compared this work to prior distributed-memory DBG assemblers in Section VIII-G. There has a limited effort of porting de Bruijn graph onto GPU such as [33], [42]. They either focus on only one stage of DBG assembly or only work with small genome. In contrast, we provide comprehensive support for every stage of DBG and work with a much larger genome.

X. Conclusions

In this work, we propose a software-hardware co-design for DBG assembly that leverages emerging 3D-stacked memory architectures with high parallelism and bandwidth. We identify graph construction and contig assembly as two bottleneck stages, as they suffer from high communication overhead due to frequent message passing. By exploiting real DNA sequence characteristics, we optimize our design with an effective data partitioning strategy and a message buffering and compression technique to reduce inter-core communication. We also develop a speculation scheme to extend each contig by multiple bases each time tentatively. The optimizations above synergistically offer the combined benefit of speedups and energy savings over the CPU by 24× and 22×. Our NDP-based DBG processing framework can significantly reduce the run time of many critical steps in analyzing human and microbial genomes, which aids in disease diagnosis, precision medicine, vaccine development, and other tasks.

XI. Acknowledgments

We would like to thank the anonymous reviewers for their feedback and suggestions. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. This work was also funded by NSF grants (#1730158, #2100237, #1911095, #2112167, #2052809, #1826967).

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K.Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[2] C. Alkan, B. P. Coe, and E. E. Eichler, "Genome structural variation discovery and genotyping," *Nature Reviews Genetics*, vol. 12, no. 5, pp. 363–376, 2011.

[3] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature genetics*, vol. 41, no. 10, p. 1061, 2009.

[4] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[5] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner, "Spades: a new genome assembly algorithm and its applications to single-cell sequencing," *Journal of computational biology*, vol. 19, no. 5, pp. 455–477, 2012.

[6] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler, "Genbank," *Nucleic acids research*, vol. 28, no. 1, pp. 15–18, 2000.

[7] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, p. 1519–1533, 2010.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.

[9] M. Chaisson, P. Pevzner, and H. Tang, "Fragment assembly with short reads," *Bioinformatics*, vol. 20, no. 13, pp. 2067–2074, 2004.

[10] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[11] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 33–38.

[12] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *ISCA*, 2017.

[13] J. Duitama, A. Zablotskaya, R. Gemayel, A. Jansen, S. Belet, J. R. Vermeesch, K. J. Verstrepen, and G. Froyen, "Large-scale analysis of tandem repeat variability in the human genome," *Nucleic Acids Research*, vol. 42, no. 9, p. 5728–5741, 2014.

[14] L. Feuk, A. R. Carson, and S. W. Scherer, "Structural variation in the human genome," *Nature Reviews Genetics*, vol. 7, no. 2, pp. 85–97, 2006.

[15] R. Gebelhoff, "Sequencing the genome creates so much data we don't know what to do with it," *" The Washington Post*, 2015.

[16] 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 *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 437–448.

[17] 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," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[18] P. Ghosh, S. Krishnamoorthy, and A. Kalyanaraman, "Pakman: Scalable assembly of large genomes on distributed memory machines," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 578–589.

[19] M. Hess, A. Sczyrba, R. Egan, T.-W. Kim, H. Chokhawala, G. Schroth, S. Luo, D. S. Clark, F. Chen, T. Zhang, R. I. Mackie, L. A. Pennacchio, S. G. Tringe, A. Visel, T. Woyke, Z. Wang, and E. M. Rubin, "Metagenomic discovery of biomass-degrading genes and genomes from cow rumen," *Science*, vol. 331, no. 6016, pp. 463–467, 2011.

[20] A. C. Howe, J. K. Jansson, S. A. Malfatti, S. G. Tringe, J. M. Tiedje, and C. T. Brown, "Tackling soil diversity with the assembly of large, complex metagenomes," *Proceedings of the National Academy of Sciences*, vol. 111, no. 13, pp. 4904–4909, 2014.

[21] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 204–216. [Online]. Available: https://doi.org/10.1109/ISCA.2016.27

[22] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," *ICCD*, 2016.

[23] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "Art: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.

[24] W. Huangfu, S. Li, X. Hu, and Y. Xie, "RADAR: A 3D-reRAM Based DNA Alignment Accelerator Architecture," in *DAC*, 2018.

[25] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, , and Y. Xie, "MEDAL: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm," in *MICRO*, 2019.

[26] Illumina, "Miseq system," https://www.illumina.com/systems/sequencing-platforms/miseq.html/.

[27] Illumina, "Nextseq 2000 system," https://www.illumina.com/systems/sequencing-platforms/nextseq-1000-2000.html.

[28] Illumina, "Novaseq 6000 system," https://www.illumina.com/systems/sequencing-platforms/novaseq.html.

[29] Intel, "Intel VTune Amplifier," https://software.intel.com/en-us/vtune, 2019.

[30] S. Jackman, B. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S. Hammond, G. Jahesh, H. Khan, L. Coombe, R. Warren, and I. Birol, "Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter," *Genome Res*, 2017.

[31] 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," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[32] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

[33] A. Jain, A. Garg, and K. Paul, "Gagm: Genome assembly on gpu using mate pairs," *20th Annual International Conference on High Performance Computing*, 2013.

[34] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 145–155.

[35] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[36] M. Kolmogorov, D. M. Bickhart, B. Behsaz, A. Gurevich, M. Rayko, S. B. Shin, K. Kuhn, J. Yuan, E. Polevikov, T. P. Smith *et al.*, "metaflye: scalable long-read metagenome assembly using repeat graphs," *Nature Methods*, vol. 17, no. 11, pp. 1103–1110, 2020.

[37] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, "Assembly of long, error-prone reads using repeat graphs," *Nature biotechnology*, vol. 37, no. 5, pp. 540–546, 2019.

[38] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph," *Bioinformatics*, vol. 31, no. 10, p. 1674–1676, 2015.

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

[40] Y. Lin, J. Yuan, M. Kolmogorov, M. W. Shen, M. Chaisson, and P. A. Pevzner, "Assembly of long error-prone reads using de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 113, no. 52, pp. E8396–E8405, 2016.

[41] 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, 2011.

[42] M. Lu, Q. Luo, B. Wang, J. Wu, and J. Zhao, "Gpu-accelerated bidirected de bruijn graph construction for genome assembly," *Web Technologies and Applications Lecture Notes in Computer Science*, p. 51–62, 2013.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[44] E. R. Mardis, "The impact of next-generation sequencing technology on genetics," *Trends in genetics*, vol. 24, no. 3, pp. 133–141, 2008.

[45] Micron, "Hybrid memory cube specification 2.1," http://hybridmemorycube.org/specification-v2-download/.

[46] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *MICRO*, 2019.

[47] S. Nurk, D. Meleshko, A. Korobeynikov, and P. A. Pevzner, "metaspades: a new versatile metagenomic assembler," *Genome Research*, vol. 27, no. 5, p. 824–834, 2017.

[48] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 41–54.

[49] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Idba-ud: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth," *Bioinformatics*, vol. 28, no. 11, pp. 1420–1428, 2012.

[50] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the national academy of sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[51] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *IEEE Micro*, 2014.

[52] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads."

[53] J. Qin, R. Li, J. Raes, M. Arumugam, K. S. Burgdorf, C. Manichanh, T. Nielsen, N. Pons, F. Levenez, T. Yamada *et al.*, "A human gut microbial gene catalogue established by metagenomic sequencing," *nature*, vol. 464, no. 7285, pp. 59–65, 2010.

[54] G. V. RESEARCH, "Metagenomics market size, share and trends analysis report by product (sequencing and data analytics), by technology (sequencing, function), by application (environmental), and segment forecasts, 2018 - 2025," GRAND VIEW RESEARCH, 2017.

[55] T. S. Rosing, N. Moshiri, and R. Knight, "Acceleration of bioinformatics workloads," *DARPA ERI Summit*, Jul 2020.

[56] S. C. Schuster, "Next-generation sequencing transforms today's biology," *Nature methods*, vol. 5, no. 1, pp. 16–18, 2008.

[57] R. Sharifi and Z. Navabi, "Online profiling for cluster-specific variable rate refreshing in high-density dram systems," in *2017 22nd IEEE European Test Symposium (ETS)*, 2017, pp. 1–6.

[58] J. Shendure, S. Balasubramanian, G. M. Church, W. Gilbert, J. Rogers, J. A. Schloss, and R. H. Waterston, "Dna sequencing at 40: past, present and future," *Nature*, vol. 550, no. 7676, pp. 345–353, 2017.

[59] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[60] J. Simpson, K. Wong, S. Jackman, J. Schein, S. Jones, and I. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome Res*, 2009.

[61] E. L. van Dijk, Y. Jaszczyszyn, D. Naquin, and C. Thermes, "The third revolution in sequencing technology," *Trends in Genetics*, vol. 34, no. 9, pp. 666–681, 2018.

[62] K. A. Wetterstrand, "DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)," https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data, 2018.

[63] S. J. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[64] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, "Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching," in *ISCA*, 2021.

[65] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. S1. Springer, 2013, p. S13.

[66] 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.

[67] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA*, 2018.

[68] M. Zhou, M. Li, M. Imani, and T. Rosing, "Hygraph: Accelerating graph processing with hybrid memory-centric computing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 330–335.

[69] M. Zhou, A. Prodromou, R. Wang, H. Yang, D. Qian, and D. Tullsen, "Temperature-aware dram cache management—relaxing thermal constraints in 3-d systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 1973–1986, 2019.

[70] F. Zokaee, H. R. Zarandi, and L. Jiang, "Aligner: A process-in-memory architecture for short read alignment in rerams," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 237–240, 2018.

[71] F. Zokaee, M. Zhang, and L. Jiang, "Finder: Accelerating fm-index-based exact pattern matching in genomic sequences through reram technology," in *PACT*, 2019.