PaKman: A Scalable Algorithm for Generating Genomic Contigs on Distributed Memory Machines

Priyanka Ghosh, Sriram Krishnamoorthy, *Senior Member, IEEE*, and Ananth Kalyanaraman, *Senior Member, IEEE*

Abstract—*De novo* genome assembly is a fundamental problem in the field of bioinformatics, that aims to assemble the DNA sequence of an unknown genome from numerous short DNA fragments (aka reads) obtained from it. With the advent of high-throughput sequencing technologies, billions of reads can be generated in a matter of hours, necessitating efficient parallelization of the assembly process. While multiple parallel solutions have been proposed in the past, conducting a large-scale assembly at scale remains a challenging problem because of the inherent complexities associated with data movement, and irregular access footprints of memory and I/O operations. In this article, we present a novel algorithm, called *PaKman*, to address the problem of performing large-scale genome assemblies on a distributed memory parallel computer. Our approach focuses on improving performance through a combination of novel data structures and algorithmic strategies for reducing the communication and I/O footprint during the assembly process. *PaKman* presents a solution for the two most time-consuming phases in the full genome assembly pipeline, namely, *k-mer counting* and *contig generation*. A key aspect of our algorithm is its graph data structure (PaK-Graph), which comprises fat nodes (or what we call "macro-nodes") that reduce the communication burden during contig generation. We present an extensive performance and qualitative evaluation of our algorithm across a wide range of genomes (varying in both size and species group), including comparisons to other state-of-the-art parallel assemblers. Our results demonstrate the ability to achieve near-linear speedups on up to 16K cores (tested) on the NERSC Cori supercomputer; perform better than or comparable to other state-of-the-art distributed memory and shared memory tools in terms of performance while delivering comparable (if not better) quality; and reduce time to solution significantly. For instance, *PaKman* is able to generate a high-quality set of assembled contigs for complex genomes such as the human and bread wheat genomes in under a minute on 16K cores. In addition, *PaKman* was able to successfully process a 3.1 TB simulated dataset of one of the largest known genomes (to date)-*Ambystoma mexicanum* (the axolotl), in just over 200 seconds on 16K cores.

Index Terms—Genome assembly, distributed memory, de bruijn graphs, k-mer counting

1 Introduction

De novo genome assembly is a fundamental problem in computational biology. The goal is to assemble the DNA sequence of an unknown (target) genome using the short fragments (called "reads") obtained from it through sequencing technologies. The output is a set of "contigs" that represent contiguous portions of the target genome. Once assembled, the contigs are scaffolded, which is a step of ordering and orienting the contigs while accounting for potential gaps between successive contigs.

The genome assembly problem has been a topic of interest for well over three decades now, and yet the need for new scalable approaches has never been more critical than it is today. The factor driving this need is the continuously evolving landscape in DNA sequencing technology. With the advent of numerous high-throughput sequencing technologies, it has now become possible (even routine) to sequence a genome by running multiple clonal copies of the target or reference genome (i.e., with *coverage* $C \in [10, 100]$), through a wetlab sequencing machine; and generating billions of reads (or hundreds of gigabytes to terabytes of raw data)——all in a matter of hours. For instance, a widely used technology such as Illumina is capable of generating short reads ($\sim$100 bases in length each) with an impressively low error rate (under 1 percent). There is also a new wave of "long read" technologies that are emerging in the community; however their error rates are still too high for wider adoption.

In this paper, we address the problem of generating a set of assembled contigs using short reads sequenced from an unknown target genome. (We do not consider the subsequent contig scaffolding step in this work.) There is a plethora of short read assemblers that have been developed for well over a decade (e.g., [1], [2], [3], [4], [5]). A large fraction of these assemblers use the *de Bruijn graph*, a graph data structure built out of fixed length substrings (of length k) contained in the reads, called k-mers, as their building blocks (vertices). Edges are established between vertices of any two consecutive k-mers in a read (overlapping in $k-1$

- *Priyanka Ghosh and Sriram Krishnamoorthy are with the Advanced Computing, Mathematics, and Data Division, Pacific Northwest National Laboratory, Richland, WA 99354 USA. E-mail: {priyanka.ghosh, sriram}@pnnl.gov.*
- *Ananth Kalyanaraman is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA. E-mail: ananth@wsu.edu.*

Manuscript received 4 May 2020; revised 25 Nov. 2020; accepted 28 Nov. 2020. Date of publication 8 Dec. 2020; date of current version 11 Jan. 2021. (Corresponding author: Priyanka Ghosh.) Recommended for acceptance by B. Ucar. Digital Object Identifier no. 10.1109/TPDS.2020.3043241

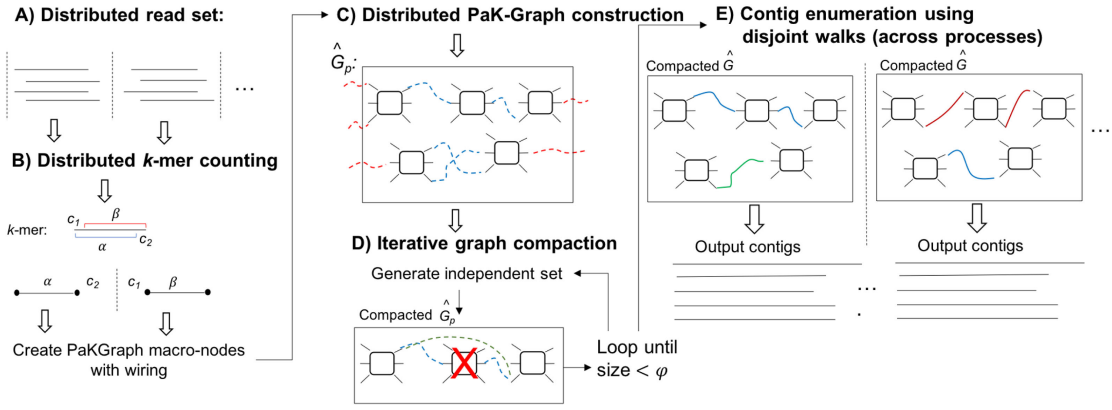


Fig. 1. Schematic illustration of the *PaKman* assembly framework. Dotted arcs: edges in our PaK-Graph (shown only for illustration and not stored in our actual implementation); blue dotted arcs: edges internal to a process; and red dotted arcs cross process boundaries. Solid arcs: walks performed to generate contigs in parallel with replicated copies of the final compacted PaK-Graph as shown. $\hat{G}_p$: portion of PaK-Graph at process p, $\hat{G}$: the global copy.

positions). In contrast to older approaches, de Bruijn graph based approaches have demonstrated greater time-efficiency and high fidelity in genome reconstruction.

Despite their advantages, an efficient parallel implementation of a de Bruijn graph-based method on distributed memory platforms has proven to be challenging for various reasons. The input read set required to construct a de Bruijn graph is typically stored in the file system. In addition, many algorithms use the file system as additional space for intermediate memory-intensive algorithm phases. This, coupled with complicated I/O patterns, can potentially lead to an I/O bottleneck during the graph construction phase.

Second, to prune erroneous paths or prepare the output, a parallel implementation that uses a de Bruijn graph should be able to *manipulate* the graph in distributed memory post-construction. This is complicated by the inherently unstructured nature of the graph (as compared to, say, a dense matrix), leading to communication imbalance issues that necessitate specialized optimization strategies.

Finally, generating the output contigs involves performing numerous "walks" along the graph and enumerating the base pairs along the path as a contig. This creates multiple challenges under a distributed memory representation of the graph, requiring frequent coordination. For instance, one needs to ensure the same path is not repeatedly traversed by multiple processes to avoid over-representing a path in the contigs. This requires coordination (e.g., atomic operations) and frequent lookup operations in a complex distributed data-structure that could hamper parallel performance.

Contributions. In this paper, we present the *PaKman* algorithm that addresses the above challenges. The algorithm deviates from the state-of-the-art de Bruijn graph-based methods, and presents a new perspective to addressing the assembly problem. *PaKman* combines MPI I/O, MPI collectives, and a novel graph data structure (which we name *PaK-Graph*) to simplify I/O and communication patterns, and eliminate the need for expensive distributed-memory coordination during the walk phase. We evaluate *PaKman* on both shared and distributed memory platforms, demonstrating near-linear scaling behavior, and observe speedups up to 3.4× over a state-of-the-art distributed memory assembler, and up to 41× over a state-of-the-art shared memory assembler—all while producing comparable quality in output. To summarize, the key contributions are:

- A novel distributed memory data structure that enables contig enumeration with minimal coordination.
- An optimized scalable load-balanced algorithm for k-mer counting. We present two variants for this method–one that uses blocking MPI calls and a second version that utilizes MPI non-blocking collective (or point-to-point) calls in order to overlap the compute and communication bound regions of the phase.
- A novel contig generation algorithm with simplified I/O and communication patterns.
- Extensive evaluation of our algorithm with numerous tests spanning a wide variety of genomes diverse in size, complexity and species type.
- Demonstration and detailed analysis of performance and output quality (in comparison to parallel state-of-the-art methods) on both shared and distributed memory systems.
- Additional results comparing the efficacy of our distributed k-mer counting method with another state-of-the-art scalable implementation.
- Evaluation of our algorithm on one of the largest genomes to be ever sequenced– Mexican axolotl. *PaKman* generates a PaK-Graph with over 37 billion macro-nodes (out of over 294 billion distinct k-mers), while producing an output set of contigs in just over 200 seconds on 16,384 cores.
- Performance portability across various computing platforms.

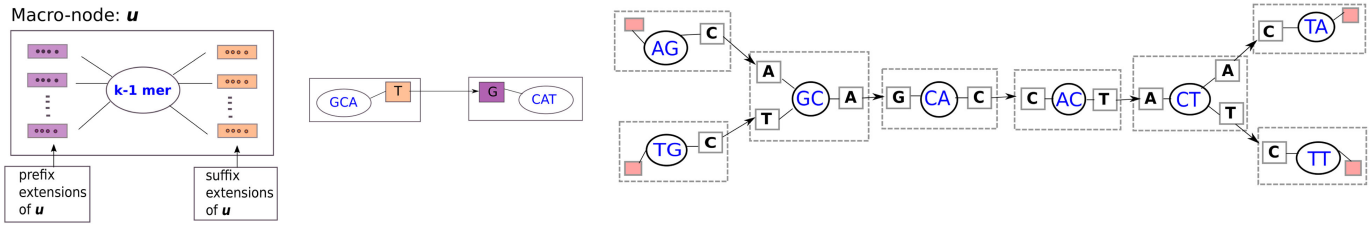
A preliminary version of this paper appeared in [6].[1]

2 Overview of Approach

Fig. 1 summarizes the major steps of the proposed *PaKman* framework. Our approach to efficient and scalable genome assembly involves the following key optimizations:

Contiguous single-pass I/O: Each MPI process reads a distinct contiguous portion of the input file (comprised of short reads) in parallel with other processes. After the input

1. *PaKman* will be available at https://github.com/pnnl/pakman



(a) macro-node structure (b) An edge in $\hat{G}$ (c) The PaK-Graph for input reads $\{AGCACTT, TGCACTA\}$ for k=3.

Fig. 2. Macro-node structure and illustration.

is processed in one pass through the file, no further I/O is performed. This minimal I/O requirement, coupled with the simple I/O pattern, makes it easy to efficiently utilize optimized I/O subsystems (parallel file systems, striping optimizations, burst buffers, etc.).

Parallel load-balanced counting of k-mers: The input reads are processed to generate a stream of k-mers and the global count for each k-mer is computed. *PaKman* employs a scalable load-balanced algorithm to construct the k-mer histogram using only three types of MPI collective calls: `MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Alltoallv`. An alternative implementation utilizes non-blocking collective (`MPI_Ialltoall` and `MPI_Ialltoallv`) or point-to-point (`MPI_Isend` and `MPI_Irecv`) communication; in order to overlap the compute and communication bound parts of the phase.

Novel data representation and iterative compaction: PaK-Graph: While input data and the de Bruijn graph might be space intensive, we observe that the final output of the algorithm is only of the order of Megabytes (or a few Gigabytes), depending on the species. This motivated the design of a compact representation of graph representing the k-mer connectivity. Rather than construct a conventional de Bruijn graph, we construct a new type of graph (which we call *PaK-Graph*; defined in Section 3.2), which provides a way to arrive at a compact representation of the k-mers. This graph captures the k-mers and their connectivity in a lossless fashion. While the savings are not significant initially, we iteratively compact the graph further to dramatically reduce its size while preserving the total information. We demonstrate this compaction procedure can be performed efficiently in parallel.

PaK-Graph replication and parallel deterministic walks: We compact the graph until it fits well within the memory available in each node of a distributed memory machine. We observe that the cost of compaction quickly decreases as the data structure shrinks in size, making subsequent compaction operations inexpensive. We exploit this property to sufficiently compact the graph to enable low-cost replication on all compute nodes of the parallel system. Once replicated, each MPI process picks a distinct set of starting points and performs disjoint walks to generate the contigs without any further communication or coordination. We achieve this using a deterministic algorithm to "wire" the paths through each vertex in a PaK-Graph to enable non-redundant walks without further coordination. This approach reduces the often complicated implementation of the walk phase into an embarrasingly parallel procedure (starting from each candidate k-mer) that incurs negligible time.

PaKman leverages algorithmic improvements to enable simplified communication and I/O strategies. Going further, the entire algorithm and its implementation rely only on a small number of MPI I/O and MPI collective operations, greatly simplifying performance portable implementations on new systems. The algorithm is efficient enough to outperform many shared-memory-specific de Bruijn graph based methods on shared memory platforms.

In the rest of the paper, we describe the design and implementation of each aforementioned step in detail and present results evaluating *PaKman* across eight different datasets varying in size and complexity of the genome.

3 Methods

3.1 Notation and Terminology

Let r denote a read of an arbitrary length (denoted by $|r|$) over the DNA alphabet $A = \{a, c, g, t\}$. For ease of exposition, we index the characters in a read from 1. Let $r(i, j)$ denote the substring of length j starting at index i in r, such that $i + j - 1 \leq |r|$. We denote the input set of n reads as $\mathcal{R}$= $\{r_1, r_2, \ldots r_n\}$, and their total length as N (= $\sum_i |r_i|$). We use $\cdot$ operator to denote string concatenation.

A *k-mer* in a read r is a substring of length k in r, for a given $k > 0$; similarly, a (k-1)-mer is a substring of length k-1. We use the term *l-mer* to denote a substring of length l that is significantly shorter than k—e.g., a typical value of k is between 32 and 48, whereas l is between 6 and 10.

3.2 PaK-Graph: An Enhanced String Graph

In this section we introduce a new graph data structure called *PaK-Graph*, that we will use in our parallel algorithm (Section 3.3). Given an input read set $\mathcal{R}$, and positive integer constants k and l such that $l < k$, we define a directed graph $\hat{G}(V, E)$ where V is the set of all "macro-nodes" and E is the set of all edges. We call each vertex in $\hat{G}$ a "macro-node" because of its augmented node structure as defined below.

Macro-Node. Each macro-node in $\hat{G}$ (as shown in Fig. 2a) is defined by a distinct (k-1)-mer present in $\mathcal{R}$. A macro-node $u \in V$ has the following node structure:

- $label(u)$ is the (k-1)-mer corresponding to u.
- $prefix\_extensions(u)$ is a set of arbitrarily long strings, each representing a candidate prefix extension of the (k-1)-mer (for an output contig).
- $suffix\_extensions(u)$ is a set of arbitrarily long strings, each representing a candidate suffix extension of the (k-1)-mer (for an output contig).

An extension with an empty string is called a *terminal* (prefix or suffix) extension.

Edges in $\hat{G}$. Edges in $\hat{G}$ are defined between a suffix extension of one macro-node and a prefix extension of another. Specifically, there exists a directed edge e from a suffix extension x of a macro-node u to a prefix extension y of another macro-node v if and only if $label(u) \cdot x = y \cdot label(v)$. Note that this implies there can be no more than one edge incident on each extension.

Fig. 2a represents a single macro-node identified by a (k-1)-mer. Fig. 2b presents two macro-nodes *GCA* and *CAT* connected by an edge such that, $GCA \cdot T = G \cdot CAT$. Fig. 2c presents an example of a PaK-Graph for a given input of two reads for k=3. The empty extenstions (shown in red) for the macro-nodes *AG*, *TG*, *TA*, and *TT* indicate there exists a terminal prefix extension for nodes *AG* and *TG* and a terminal suffix extension for the nodes *TA* and *TT*.

Initially, there exists one macro-node for every (k-1)-mer in $\mathcal{R}$. As for edges, for each k-mer $k_1 = a_1a_2 \ldots a_k$ in $\mathcal{R}$, an edge is introduced in $\hat{G}$ from the suffix extension with string a_k of the macro-node for $a_1a_2 \ldots a_{k-1}$ to the prefix extension with string a_1 of the macro-node for $a_2a_3 \ldots a_k$. In this initial state, the PaK-Graph is equivalent to the de Bruijn graph constructed for $\mathcal{R}$. However, unlike the traditional de Bruijn graph, each macro-node through its extensions can encode an arbitrarily long path along the de Bruijn graph in a compressed manner. For this reason, our PaK-Graph can be viewed as an enhanced version of the string graph data structure originally introduced by Myers [7].

In the implementation, we only store the set of macro-nodes in $\hat{G}$. Edges are *not* explicitly stored because the suffix/prefix from the extension and the (k-1)-mer can be used to uniquely determine neighboring macro-nodes. Sections 3.3.3–3.3.5 further detail the implementation of $\hat{G}$.

3.3 *PaKman*: Parallel Genome Assembly Algorithm

In this section, we describe *PaKman*, our parallel algorithm for genome assembly. The input is a set of n reads $\mathcal{R}$ (made available as a single multi-sequence FASTA format file) and positive integer parameters k and l ($l < k$). The output is a set of *contigs* representing contiguous portions of the target genome. The number of processes is denoted by $psize$.

The algorithm consists of multiple steps as described below.

3.3.1 Input Reading

The input is loaded from the input file in a distributed manner such that each process receives roughly the same amount of sequence data ($\approx \frac{N}{psize}$ per process). This is achieved by each process performing a `MPI_File_get_size` and subsequently loading its unique chunk of reads using MPI-IO functions (`MPI_File_read_at_all`), such that no read is split among processes. Henceforth, we use $\mathcal{R}_p$ to denote the read set loaded at process p (i.e., $\mathcal{R}_p \subseteq \mathcal{R}$).

3.3.2 k-mer Counting

The goal of this stage is to generate and compute the frequency of all k-mers from $\mathcal{R}$. A well-known approach is to generate all k-mers by simply sliding a window of length k over each read and aggregating counts in a lookup table with 4^k buckets (one for each possible k-mer over the DNA alphabet) [8]. However, the large size of k (≥ 32) makes this simple approach prohibitive in space. Therefore, we use an alternative approach based on *minimizers* [9]. The idea is to use a smaller window length l ($< k$; e.g., $l = 8$) to partition k-mers into buckets, prior to obtaining the global count for each k-mer from each bucket. For parallel processing, each $min\text{-}lmer$ bucket is assigned a distinct owner process. There are several ways to implement this minimizer approach using techniques from MinHashing based principles [10]. In our implementation, we assign a k-mer to the bucket corresponding to a least frequent l-mer occurring within that k-mer (i.e., making it the k-mer's choice of its $min\text{-}lmer$). This way, we can expect (though not guarantee) that consecutive k-mers from the same overlapping region across reads are expected to be assigned to the same destination process bucket, which helps reduce communication later.

Algorithm 1 outlines our k-mer counting procedure. In the first step, each process generates all l-mers from its reads in $\mathcal{R}_p$ and obtains a global count for each l-mer using an `MPI_Allreduce` call. The next step implements the minimizer approach described above. This step involves redistributing the k-mers generated at various processes to their respective $min\text{-}lmer$ buckets using `MPI_Alltoallv`. In our implementation, we perform this task using multiple rounds of communication to scale up to large input sizes (we use a batch size of a 100 million k-mers in all our experiments).

Algorithm 1. k-mer Counting

Input: Input set of reads for each process: $\mathcal{R}_p$, number of processes: $psize$, batch_size: b
Output: A set of distinct k-mers and their corresponding counts.

Initialize $lmer\_frequency$ buffer of size 4^l
for *each* $r \in R_p$ **do**
  for *each* l-mer $i \in r$ **do**
    Increment $lmer\_frequency[i]$ /*Update l-mer frequency*/
  end
end
`MPI_Allreduce` to compute the global counts for all l-mers
Initialize buffer $kmers\_per\_proc$ of size $psize$
$num\_kmers \leftarrow 0$
for *each read* $r \in R_p$ **do**
  $min\_lmer \leftarrow 0$; $min\_lmer\_count \leftarrow \infty$
  for *each* k-mer $k \in r$ **do**
    for *each* l-mer $i \in k$ **do**
      if $lmer\_frequency[i] < min\_lmer\_count$ **then**
        $min\_lmer \leftarrow i$
        $min\_lmer\_count \leftarrow lmer\_frequency[i]$
      end
    end
    $target\_id \leftarrow$ retrieve the process id based of min_lmer
    Insert k in $kmers\_per\_proc[target\_id]$
    $num\_kmers \leftarrow num\_kmers + 1$
  end
  if $num\_kmers > b$ **then**
    Update $kmer\_list$ by transferring k-mers using `MPI_Alltoallv`
    reset $num\_kmers \leftarrow 0$
  end
end
return $kmer\_list$

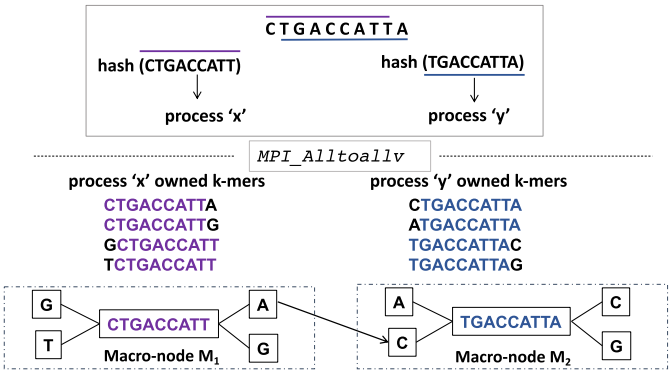


Fig. 3. Redistributing k-mer *CTGACCATTA* (with k=10) to two processes; serving as a prefix (k-1)-mer in macro-node M_2 and as a suffix (k-1)-mer in macro-node M_1.

At the end of this step, all processes have a set of distinct k-mers and their respective global counts. Then, we perform a simple threshold-based pruning: we remove k-mers that have a count below a certain threshold τ. Such k-mers are deemed "poor quality" from the assembly perspective. We determine τ by plotting a k-mer frequency histogram for a fixed number of top buckets (say h)—obtaining the global counts using an `MPI_Allreduce`—and setting τ to the minimum over those h bucket counts. Parameter h is tunable and is set to 20 in our experiments.

Since the k-mer counting method is compute intensive by nature, we further extended our implementation to account for overlapping the computation with the multiple rounds of communication, in order to further improve the performance and scalability of this phase. Results for our k-mer counting phase utilizing MPI non-blocking collective (or point-to-point) communication is described in Section 4.2.2.

3.3.3 PaK-Graph Construction: k-mer Distribution

We now describe the distributed construction of the initial PaK-Graph, involving just a single `MPI_Alltoallv` communication. At the end of this step, each process p will hold a distinct portion $\hat{G}_p$ (subset of macro-nodes) of the initial $\hat{G}$.

Prior to constructing the PaK-Graph , we need to redistribute the k-mers because we need each k-mer in two places—one corresponding to the macro-node of its prefix (k-1)-mer and another to the suffix (k-1)-mer (as shown in Fig. 2). (If both (k-1)-mers are identical, then the k-mer is needed only in one place.) We identify the process id that will act as the owner for each macro-node using a linear congruential hash function for the macro-node's corresponding (k-1)-mer; Subsequently, using an `MPI_Alltoallv` call, the set of k-mers are redistributed among the process space such that all k-mers corresponding a given macro-node are collected on a single process. At this point, each process p has a list of tuples $\mathcal{K}_p = \{kmer, count\}$ that will serve as the input to generate its $\hat{G}_p$.

Fig. 3 provides an example illustrating the redistribution of a given k-mer to two separate processes and thereafter contributing to two distinct macro-nodes.

3.3.4 PaK-Graph Construction: Macro-Nodes

Algorithm 2 shows the steps to build $\hat{G}$ on each process p using $\mathcal{K}_p$. We make a couple of key observations here. First, a process p constructs a macro-node only if its k-mer falls in its domain (using the hash function). Second, as noted in Section 3.2, the edges of $\hat{G}$ are *not* explicitly stored; instead, the extensions on either side of a macro-node are sufficient to capture all the information pertaining to its edges. However, how do we know if a particular extension exists or not (without communicating)? To answer this question, consider a valid prefix extension $c \cdot x'$, where $c \in A$ and x' is the (k-1)-mer corresponding to the macro-node under construction. Then, $c \cdot x'$ must be a k-mer that is also represented in $\mathcal{K}_p$ (as a result of the initial `MPI_Alltoallv`). This is the advantage of initially communicating k-mers to construct the local macro-nodes. In other words, the algorithm becomes communication-free at this step because all necessary information for macro-node construction is available from $\mathcal{K}_p$.

Algorithm 2. Construct a PaK-Graph of macro-nodes

Input: Input set of tuples (k-mer, k-mer_count): $\mathcal{K}_p$ at process p, Alphabet A, Coverage C
Output: The local $\hat{G}_p$ at process p

1: **for** *each* $x \in \mathcal{K}_p$ **do**
2: **for** *each (k-1)-mer* $x' \in x$ **do**
3: **if** p *is the owner for* x' *and* $x' \notin \hat{G}_p$ **then**
4: Create macro-node u with *label* x'
5: **for** *each* $c \in A$ **do**
 /* `Detect edges: prefix extensions` */
6: **if** $(c \cdot x') \in \mathcal{K}_p$**then**
7: Append c to $u.prefixes$
8: Set $vc \leftarrow ceil(kmer\_count(c \cdot x')/C)$
9: Set $u.prefix\_counts \leftarrow \{kmer\_count(c \cdot x'), vc\}$
10: Set $u.prefix\_terminal \leftarrow$ `false`
11: **end**
 /* `Detect edges: suffix extensions of the form` $x' \cdot c$`; Details omitted due to similar logic as above.` */
12: ...
13: **end**
 /* `set the internal wiring from prefix to suffix extensions for node` u */
14: $wire\_info$ = setup_wiring (u)
15: Append $wire\_info$ to $u.wire\_info$
16: Add u to $\hat{G}$
17: **end**
18: **end**
19: **return** $\hat{G}$
20: **end**

There are two other steps in Algorithm 2 that need further elaboration. First, along with each extension, a list of pairs of the form $\langle kmer\_count, visit\_count \rangle$ is stored; where the *visit_count* represents the number of times that extension can be allowed to be traversed while taking part in contig enumeration (explained later). It is initialized to $\lceil kmer\_count/C \rceil$, where C is the sequencing coverage. At this time, we also determine whether a particular extension is terminal or not.

3.3.5 PaK-Graph Construction: Wiring

Next, we compute a "wiring table" that holds the mapping from each prefix extension of the macro-node to a corresponding suffix extension. Algorithm 3 walks through the essential steps of the macro-node wiring procedure. We explain the

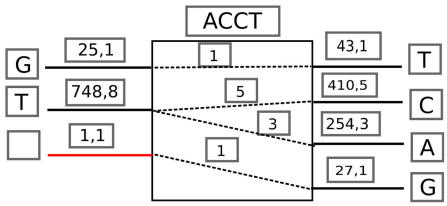


Fig. 4. macro-node wiring illustration for (k-1)-mer *ACCT*. The pair $< kmer\_count, visit\_count >$ labels each extension. The red prefix extension denotes a terminal prefix.

main idea of the algorithm using the simple example in Fig. 4, which shows a macro-node for the (k-1)-mer *ACCT* ($k = 5$). A more detailed description of the wiring table contents is presented in Section 3.4.1.

Initially, this macro-node contains two prefix and four suffix extensions (nonterminal), corresponding to a group of six k-mers. We first calculate the sum of all the prefix (pc) and suffix (sc) $visit\_counts$. If the suffix (prefix) total count exceeds the prefix (suffix) total count, we introduce a new terminal extension on the prefix (suffix) side (shown in red), with the tuple $\langle 1, |sc - pc| \rangle$ as shown. Subsequently, we construct a wiring table that connects each prefix extension to one or more suffix extensions (i.e., a fan-out). Note that multiple prefix extensions may also connect to one suffix extension (i.e., a fan-in). These wiring decisions are made based on the visit counts using a *greedy heuristic*. For instance, the prefix extension corresponding to T that has the maximum visit count (8) is considered first. This extension greedily selects the top available suffix extensions whose total visit counts become greater or equal to its own visit count—effectively selecting the suffixes C and A as shown. Ties are broken arbitrarily (albeit deterministically). This procedure is repeated until all extensions have been exhaustively wired.

Invariant 1. *For every macro-node , the sum of all visit counts of the prefix extensions will exactly match the sum of all visit counts on the suffix side. Steps* $3 - 20$ *in Algorithm 3 preserves this property which holds true in every instance of a given macro-node.*

Establishing a deterministic wiring strategy as described above helps us ensure that during traversal of the macro-nodes (in the contig generation phase), each walk is carried out in a coordination-free/disjoint manner—instilling maximum concurrency in the process (Section 3.3.9).

This simple greedy strategy in wiring is also motivated by its impact on the quality of the output contig. Intuitively the (initial) visit count of an extension represents the number of distinct locations that particular k-mer (obtained by concatenating that extension with the (k-1)-mer) is expected to be present along the genome. Consequently, k-mers that are adjacent to this k-mer can also be expected (but not guaranteed) to occur with approximately the same frequency (e.g., if a k-mer ACCAG is present 10 times, then k-mers that represent one character extensions such as CCAGT or TACCA (if they exist) can also be expected to occur with similar frequency (without guarantee)). This is the intuitive reason behind the greedy strategy in wiring. When tested with k=32 (and l = 8), we observed the $min\text{-}lmer$ (calculated based on k-mer frequency) across consecutive k-mers for a given read, to change once every forty base pairs on average, thus offering further validation for our method. We note here that our wiring strategy is amenable for extension to incorporate other qualitative information such as paired-end reads. If such information is made available, it could potentially have a larger positive impact on assembly quality. This is part of our future work.

Algorithm 3. Wiring Algorithm: $setup\_wiring()$

Input: A $macro\_node$: Mn
Output: Updated $macro\_node$: Mn with $wire\_info$
1: Initialize sc=0, pc=0
2: Initialize $null\_sid$=-1, $null\_pid$=-1
/* Calculate sum of all suffix visit counts of Mn */
3: **for** *each* $i \in Mn.suffixes$ **do**
4: update $sc \leftarrow sc + Mn.suffix\_counts[i].vc$
5: **end**
/* Calculate sum of all prefix visit counts of Mn */
6: **for** *each* $i \in Mn.prefixes$ **do**
7: update $pc \leftarrow pc + Mn.prefix\_counts[i].vc$
8: **end**
/* initialize and assign value to null suffix of Mn */
9: **for** *each* $i \in Mn.suffixes$ **do**
10: **if** $Mn.suffixes[i].size() == 0$ **then**
11: Set $null\_sid \leftarrow i$
12: Set $Mn.suffix\_counts[i] \leftarrow \{1, (pc - sc)\}$
13: **end**
14: **end**
/* initialize and assign value to null prefix of Mn */
15: **for** *each* $i \in Mn.prefixes$ **do**
16: **if** $Mn.prefixes[i].size() == 0$ **then**
17: Set $null\_pid \leftarrow i$
18: Set $Mn.prefix\_counts[i] \leftarrow \{1, (sc - pc)\}$
19: **end**
20: **end**
21: $leftover \leftarrow sc + Mn.suffix\_counts[null\_sid].vc$
/* Initialize a wiring table for a macro-node, to hold information pertaining to every suffix connected to a given prefix */
22: Initialize
$Mn.wireinfo \leftarrow (len(Mn.prefixes) + len(Mn.suffixes) + 1)$
/* Initialize an array to maintain the offsets within each suffix edge */
23: Initialize $offset\_in\_suffix \leftarrow len(Mn.suffixes)$
24: **while** $leftover > 0$ **do**
25: set $largest\_pid \leftarrow$ prefix with largest visit count
26: set $largest\_sid \leftarrow$ suffix with largest visit count
27: $count \leftarrow min(prefix[largest\_pid], suffix[largest\_sid])$
28: update $Mn.wireinfo[pid] \leftarrow (largest\_sid, offset\_in\_suffix[largest\_sid], count)$
29: Decrement $leftover \leftarrow leftover - count$
30: Increment $offset\_in\_suffix[largest\_sid] += count$
31: **end**

3.3.6 Contig Generation: Generate Independent Set

Using the initial $\hat{G}$, we initiate an iterative process of compacting the PaK-Graph until the total number of macro-nodes across all processes reduces to the extent that the entire graph will fit in the memory of each compute node. In our experiments, we set this threshold ψ to 100K macro-nodes.

Algorithm 4 describes the major steps of the iterative process. The main idea of the algorithm is to identify numerous macro-nodes for removal, remove them in a way that their information is captured in the macro-nodes that survive, and iterate with the compacted graph. In the interest of space, we include detailed algorithmic pseudocodes for the individual functions (*Generate_independent_set*, *iterate_and_pack_Mnode* and *serialize_and_transfer*) in supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.3043241; and instead summarize the main ideas in text alongside an illustration (Fig. 5).

We formulate the problem of identifying macro-nodes to be removed as one of identifying an independent set I of macro-nodes in $\hat{G}$. An independent set is a set of vertices in which no two are adjacent to one another. To identify an I corresponding to a $\hat{G}$, we use a simple distributed scheme in which each macro-node selects itself as part of the output set if and only if it contains the lexicographically largest (k-1)-mer among all its immediate neighbors. We devised this simple scheme because it enables each macro-node to make a strictly local decision without having to communicate with any of its neighbors. Surprisingly, we found this simple scheme also yields significant compaction. Specifically, in our experiments, we found the reduction in the number of macro-nodes between successive iterations ranged from $\sim 25-28\%$, over the first few iterations. Such a sustained reduction would imply a $\mathcal{O}(\log(m/\psi))$ number of iterations required to converge, where m is the number of macro-nodes in the initial $\hat{G}$.

Algorithm 4. Iterative Algorithm to Compact a PaK-Graph

Input: PaK-Graph $\hat{G}_p$, node_threshold: ψ
Output: Compacted graph

1: Initialize independent set (array) $I \leftarrow \emptyset$
2: $num\_macro\_nodes \leftarrow len(\hat{G}_p)$
3: **while** $num\_macro\_nodes > \psi$ **do**
4: $I \leftarrow Generate\_independent\_set(\hat{G}_p)$
/* For every node $u \in I$, pass u.pred_ext to u's successor and u.succ_ext to u's predecessor, and then delete u. *iterate_and_pack_Mnode* returns the list of neighboring macro-nodes to be modified */
5: $(transfer\_nodeInfo, pcontig\_list) \leftarrow iterate\_and\_pack\_Mnode(I, \hat{G}_p)$
6: $new\_size \leftarrow len(\hat{G}_p) - len(I)$
7: Resize $\hat{G}_p$ to new_size after deleting all $u \in I$ /* Inform all macro-nodes that are neighbors of deleted nodes in I so that they can update their extensions. This is achieved using an MPI_Alltoallv. */
8: $rewire\_nodes\_list \leftarrow serialize\_and\_transfer(transfer\_nodeInfo, \hat{G})$
9: Iterate through the list of modified macro-nodes and re-wire them
10: **end**
11: populate $begin\_kmer\_list \leftarrow$ list of starting points for the walks
12: $global\_\hat{G} \leftarrow MPI\_Allgatherv(\hat{G})$
13: **return** $(global\_\hat{G}, pcontig\_list, begin\_kmer\_list)$

Intuitively, the motivation behind iterative compaction is to compress the graph to a state where the graph can be replicated in the local memory of each compute node and the contig enumeration step can be embarrassingly parallelized. The idea of detecting and using an independent set of macro-nodes to compact the graph at every step ensures that this compaction is achieved in a lossless manner. This is because no two macro-nodes to be removed at an iteration can be adjacent in an independent set, and a macro-node that survives this removal process carries forward the sequence information preserved in the corresponding extensions of the adjacent removed macro-nodes—as illustrated by the example in Fig. 5. This property holds true even in the more complex cases wherein a macro-node to be retained has multiple predecessor and successor macro-nodes (with one or more being part of the independent set); in such a case, our wiring scheme guarantees a deterministic pairing of all the added extensions at the surviving node, thereby resulting in no loss of data (details of the wiring implementation provided in Section 3.4.1). In terms of space complexity, with each compaction step, the removal of macro-nodes generates significant savings in practice, not only owing to the space constant (i.e., overhead) associated with each macro-node, but also by eliminating the redundancy that exists in the representation of k-mers among adjacent nodes in a PaK-Graph (i.e., the (k-1)-mer label and implicit edges).

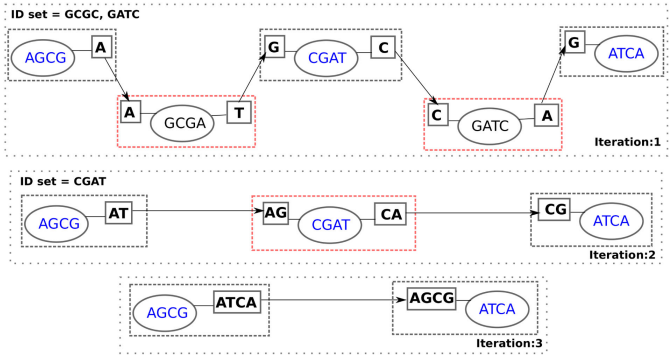


Fig. 5. Illustration of iterative compaction of a five-macro-node PaK-Graph to a two-macro-node one in three iterations.

3.3.7 Contig Generation: Iterate and Pack Nodes

In this step, the impact of removing the macro-nodes that are part of the independent set at each iteration is communicated to the surviving macro-nodes, so that they can update their structure (described in Section 3.3.8). In the first iteration of the example in Fig. 5, once the macro-node corresponding to label *GCGA* is removed, its two corresponding wired prefix-suffix extension pair, namely the macro-nodes for *AGCG* and *CGAT*, need to be informed. If any of these macro-nodes are remote, then the information about this deleted macro-node needs to be communicated. The *iterate and pack* function prepares the data to be communicated and the next step (*serialize and transfer*) performs the communication and macro-node update.

3.3.8 Contig Generation: (de)Serialize and Transfer

In this step for iterative graph compaction, an `MPI_Alltoallv` communication call relays all removed macro-node information to the impacted processes. Subsequently, macro-node information at the impacted processes is updated based

on the macro-nodes removed from $\hat{G}$. Consider again the example in Fig. 5. After removal of the macro-node *GCGA* in iteration 1, the neighboring macro-nodes now become immediate predecessor and successor (akin to the removal of a node in a linked list). Note that such new predecessor-successor relationship is established only between pairs of prefix-suffix extensions that have an entry in the wiring table of the macro-node being removed. As a result of this repacking, the suffix and prefix extensions of the two macro-nodes (*AGCG* and *CGAT*, respectively) should be extended as shown to include the values from removed macro-node. Note that if both a prefix and suffix extension wired pair for the removed macro-node happen to be terminals, then we construct and output the corresponding contig.

It is to be noted that the sizes of the macro-nodes in the buffer $transfer\_nodeInfo$ do not stay constant, owing to the varying lengths of the tuple entries that get communicated. In fact, the extensions tend to grow in size as the number of iterations grows. As a result, we need to serialize the contents of the $transfer\_nodeInfo$ buffer to convert it to a byte stream. We utilize *cereal* [11], a lightweight C++11 serialization library. We create a custom MPI derived datatype to encapsulate the serialized data in the send buffer for `MPI_Alltoallv`. Once the call completes, we deserialize the receive buffer to obtain the list of tuples, which contain the macro-nodes to be updated in $\hat{G}$. Lastly, we add the updated macro-nodes to a buffer ($rewire\_list$); this is to initiate the rewiring of all modified macro-nodes.

3.3.9 Contig Generation: Gather and Walk

As described in Algorithm 4, at the end of the iterative phase, we are left with a total number of macro-nodes $< \psi$ across all processes. At this stage, each process prepares a list of distinct starting points for initiating a walk in the compacted *PaK-Graph*. Entries in the $begin\_kmer\_list$ are identified as the (k-1)-mer of macro-nodes with a terminal prefix (and $visit\_count > 0$). Given that the graph has been sufficiently compacted (such that it can fit the memory of a single node), we initiate a call to `MPI_Allgatherv` to collate and gather all remaining macro nodes from all the processes. Thus each process now effectively receives a copy of compacted $\hat{G}$, i.e., $global\_\hat{G}$.

The final phase of the contig generation algorithm involves the traversal (or walk) across the nodes in $global\_\hat{G}$. As described in Algorithm 5, we begin enumerating a contig for each entry in $begin\_kmer\_list$. Each MPI process initializes a contig and appends to it the terminal prefix extension followed by the macro-node, (k-1)-mer, and then initiates a walk, wherein it looks up its corresponding suffix extension in the wiring table and appends it to the contig. If the suffix extension is not terminal, the process continues the walk in a recursive fashion until a terminal suffix is encountered, at which time the walk is completed and the contig is returned as output. The detailed algorithmic pseudocode for the *walk* function is described in Algorithm 6.

We illustrate the walking algorithm in Fig. 6, wherein we depict the contigs enumerated across three macro-nodes. The numbers on each wire represent the $visit\_count$ for the corresponding prefix/suffix extensions and the tuple (in brackets) indicates the: {offset in suffix, count} on the wire.

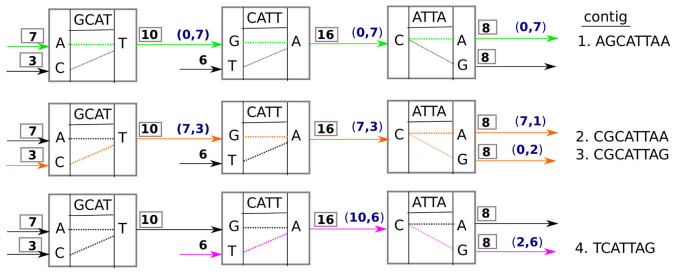


Fig. 6. Walk algorithm illustration.

The walking algorithm ensures that at no point during the walk, will the same sub-range in a wire be walked more than once. As seen in the case of contigs 1 and 2, the edge *GCATT* connecting the first and second node is traversed as part of both contigs. However, since they traverse separate ranges within the wire, both walks are disjoint and can occur concurrently.

Invariant 2. *If we have two or more paths that reach a particular macro-node with different ranges; then they shall never reach any other macro-node with intersecting ranges. The steps of Algorithm 6 preserves this property.*

Lastly, we summarize the properties of the walking algorithm: a) Any walk will start at a terminal prefix and end at a terminal suffix; b) Every walk will terminate, and two walk's starting from two different terminal prefixes will be guaranteed to be disjoint; c) There might be instances where a walk may traverse the same node multiple times owing to presence of repeat regions; and d) The algorithm does not guarantee that all repeat regions will be reachable from a terminal prefix, and thus covered as part of the contigs.

Algorithm 5. *Walk* Algorithm to Generate Final Contigs

Input: $global\_\hat{G}$: compacted *PaK-Graph*, $begin\_kmer\_list$
Output: Final set of contigs
1: **for** *each entry* $b \in begin\_kmer\_list$ **do**
2: $mn \leftarrow$ find b in $global\_\hat{G}$
3: **for** *each extension* $i \in len(mn.Prefixes)$ **do**
4: **if** i *is terminal* **then**
5: $prefix\_id \leftarrow i$
6: $freq \leftarrow mn.prefix\_counts[i]$
7: Initialize contig c
8: Append $mn.prefixes[i]$ to c
9: Append $mn.$(k-1)-mer to c
/* A walk in $global\_\hat{G}$ terminates when a suffix terminal is encountered. The final contig is returned by the output function within the call to walk */
10: walk($c, freq, 0, mn, prefix\_id$)
11: **end**
12: **end**
13: **end**

3.4 Algorithm Properties

3.4.1 Macro-Node Wiring Table Implementation

As defined in Section 3.3.5, there exists for every macro-node: a) a set of prefix extensions b) a set of suffix extensions and c) a wiring table. For every prefix/suffix extension there exists a corresponding tuple $<$k-mer count, visit count$>$, wherein

the k-mer count represents the number of occurrences of the concluding k-mer (for the given extension) in the input data, and the visit count is calculated as $\lceil kmer\_count/C \rceil$, where C is the sequencing coverage. The visit count determines the number of times that extension can be traversed during the 'walk' phase at the time of contig generation.

Algorithm 6. Algorithm for the $walk()$ Function

Input: Partial contig: c, incoming frequency: $freq$, incoming prefix edge offset: $offset\_in\_prefix$, macro_node: mn, prefix edge index: pid
Output: list of contigs

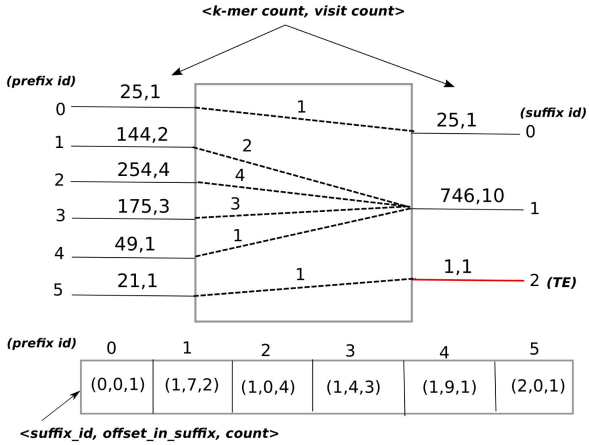
1: $partial\_contig\_size \leftarrow len(c)$
2: $freq\_remaining \leftarrow freq$
3: $internal\_off = 0 \leftarrow$ keeps track of which entry within len ($mn.wire\_info[pid]$) to continue walking from
4: **for** *each entry* $i \in mn.wire\_info[pid]$ **do**
5: $sid \leftarrow mn.wire\_info[pid][i].suff\_id$
6: $sz \leftarrow mn.wire\_info[pid][i].count$
7: $off\_in\_suffix \leftarrow mn.wire\_info[pid][i].suff\_off$
/* $internal\_off$ is always initialized to zero and incremented by min_freq */
8: **if** $internal\_off$ *for* i *is exhausted* **then**
9: continue
10: **end**
11: **if** $offset\_in\_prefix > internal\_off$
12: Set $off\_in\_wire \leftarrow offset\_in\_prefix - internal\_off$
13: **end**
14: Set $next\_off \leftarrow off\_in\_suffix + off\_in\_wire$
15: Set $freq\_in\_wire \leftarrow min(freq\_remaining, (sz - off\_in\_wire))$
16: Append $Mn.Suffixes[sid]$ to c
17: **if** $mn.suffix\_terminal[sid]$ *is terminal* **then**
/* output contig when walk reaches terminal suffix */
18: output(c)
19: **end**
20: **else**
21: Lookup the next macro_node $next\_mn$
22: $next\_prefix\_id \leftarrow prefix\_id$ of $next\_mn$
23: walk(c, $freq\_in\_wire$, $next\_off$, $next\_mn$, $next\_prefix\_id$)
24: **end**
25: Decrement $freq\_remaining -= freq\_in\_wire$
26: Resize c to $partial\_contig\_size$
27: Increment $internal\_off \leftarrow mn.wire\_info[pid][i].count$
28: **end**

The wiring table designates a mapping strategy, assigning each prefix extension to a corresponding suffix extension. Algorithm 3 enumerates the wiring strategy in greater detail. In this section we elaborate on the structure of the wiring table entries, and provide additional implementation specific details on how the wiring decisions influence the paths traversed in the PaK-Graph, during the 'walk' phase at the time of contig generation.

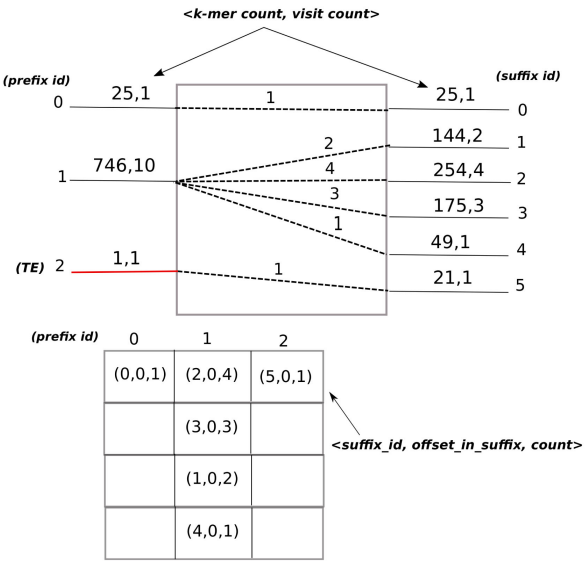
There exists at least one entry for each prefix extension in the wiring table (as shown in steps $21-30$ of Algorithm 3), wherein the structure of an entry is defined by the following tuple: $< suffix\_id, offset\_in\_suffix, count >$.

- $suffix\_id$ is identified by the unique suffix extension id, mapped to a given prefix extension. Wiring decisions are made based on the visit count using a *greedy* heuristic as explained in Section 3.3.5.
- $offset\_in\_suffix$ represents the offset (or sub-range) within the wire to be traversed by the given entry. This value is initialized to zero, and incremented when there exists multiple prefixes mapped to the same suffix extension.
- $count$ represents the frequency or the number of times this path can be traversed, and is initialized to $\sim minimum\{visit\_count$(largest prefix), $visit\_count$ (largest suffix)$\}$. The largest prefix/suffix is determined based on the visit count upon applying the *greedy* heuristic, as detailed in Algorithm 3.

Note that multiple prefix extensions may be connected to one suffix extension (i.e., a fan-in or join) as shown for the macro-node in Fig. 7a. We notice that there exists multiple



(a) Macro-node with wiring table illustrating a fan-in (or join). Note that multiple prefix extensions are mapped to a single suffix.



(b) Macro-node with wiring table illustrating a fan-out (or fork). Note that a single prefix is mapped to multiple suffix extensions.

Fig. 7. Illustration of wiring strategies for two separate macro-nodes with their corresponding wiring tables, for coverage(C)=80x. The wires labeled 'TE' (and in red) imply a Terminal Edge; and the number on the dotted wires represents the value of '*count*'.

entries in the wiring table with $suffix\_id$ set to *1*, thus indicating that multiple prefixes have been mapped to the same suffix (denoted by id *'1'*). However for each of those entries we observe a distinct value for the $offset\_in\_suffix$. The *offset* value thus ensures that there exist separate ranges within a wire, such that multiple disjoint walks can occur concurrently.

Invariant 3. *At the instance of a fan-in (join), two or more different branches (or paths) that connect to the same suffix, will originate from disjoint prefixes. The visit count of an edge on the suffix side will always correspond to the sum of all its incoming edges from the prefix side.*

The same is true in vice versa wherein a single prefix may be connected to multiple suffix extensions (i.e., a fan-out or fork) as shown in Fig. 7b. In this case, we observe multiple entries in the wiring table for a given prefix id (denoted by id *'1'*), wherein each entry is mapped to a different corresponding $suffix\_id$.

Invariant 4. *At the instance of a fan-out (fork), two or more different branches (or paths) that connect to the same prefix, will map to disjoint suffixes. The visit count of the incoming edge on the prefix side will always match the sum of all outgoing edges on the suffix side.*

The wiring strategy described in Algorithm 3 will preserve the properties of Invariants 3 and 4, which holds true throughtout a given path traversed in the PaK-Graph.

3.4.2 PaKman *Walk Algorithm Lemmas*

Lemma 1. *A walk will start at a terminal prefix and always terminate in a terminal suffix.*

Proof. As outlined in Algorithm 5, contig enumeration starts with a begin k-mer, which is identified as a terminal prefix with a corresponding visit count > 0. For every begin k-mer, we retrieve the visit count (from the macro-node) corresponding to the terminal prefix (denoted as *pid*) in the variable *freq*, and subsequently initiate the *walk* function.

The *walk* function as detailed in Algorithm 6, performs a lookup in the wiring table for *pid* and retrieves the corresponding a) *suffix_id*, denoting the suffix mapped to *pid*, and b) *offset_in_suffix*, which specifies the offset in the wire to continue the walk. The walk function is henceforth invoked in a recursive fashion by the algorithm at every instance a lookup is performed, to traverse the next macro-node in the path; and concludes once we span all the branches for a given *pid*. It is to be noted that the *freq* count, ensures that a k-mer is traversed exactly the number equivalent to its corresponding visit count.

By virtue of Invariant 2 we, know that no two paths shall walk in overlapping ranges, therefore in the event that multiple suffixes are mapped to the same *pid*, each walk shall proceed separately, in a disjoint fashion, along the designated offset. As a result a walk may consist of multiple paths (akin to a rooted tree).

By virtue of the deterministic wiring strategy, every path will encounter a terminal suffix when the *freq* count has been exhausted. Thus we can prove that a walk starting at a terminal prefix can lead to multiple terminal suffixes, and thereby enumerate multiple contigs (wherein each traversed path yields a distinct contig). □

Lemma 2. *The walking algorithm is deterministic and data-race free.*

Proof. Irrespective of the number the processes, each walk begins at a distinct terminal prefix of a given macro-node. Each prefix extension is mapped to a corresponding suffix extension using a deterministic wiring strategy as detailed in Algorithm 3. Macro-nodes with updated extensions at the end of each iteration of the compaction phase are rewired in order to guarantee deterministic pairing of all added extensions, thereby ensuring no loss of data. As a result we can establish that all walks on the compacted PaK-Graph are deterministic i.e., the algorithm generates the same set of contigs as output for every run.

In the event that multiple processes walk the same path, the setup of the wiring table (denoted as $wireinfo$ in Algorithm 6) ensures that each process traverses the path at a distinct offset (off_in_suffix as enumerated in Algorithm 6), wherein both the visit count and offset of the subsequent macro-node traversed is uniquely inferred at each level of recursion as seen in lines 14-15, until the walk concludes in a terminal suffix (as seen in lines $17-19$). As a result, we can guarantee that walks proceeding concurrently on the same path will do so without encountering a race-condition, since internally each process will travel at a different offset range. □

3.5 Limitations of the *PaKman* Algorithm

In this section, we discuss some of the implications of our algorithmic choices and point to avenues for further research:

- The performance of the *input reading* phase depends largely on the I/O subsystem. The absence of a good I/O subsystem may lead to bottlenecks in performance, reducing the overall speedup we can obtain for the full computational pipeline. We have studied the impact of I/O related parameters across multiple file-systems and have presented our analysis in Section 4.2.
- Our wiring algorithm relies on a greedy heuristic that takes the visit count of a k-mer into account while determining neighbors. As a result, we have observed instances wherein a false connection may lead to a mismatch, thereby, contributing to unaligned regions in the contigs. The quality of the wiring decisions made can be potentially improved with the incorporation of auxiliary information (such as paired-end) as available.

4 Experimental Evaluation

We evaluated *PaKman* using the datasets shown in Table 1. All read datasets contain single-end reads generated from the real genome sequences, using either the ART Illumina read simulator [12] or BBMap *randomreads* [13]. This approach is consistent with practice, as the simulators record the originating location for each read which can be later used as groundtruth for validation.

Our distributed memory experiments were conducted on the NERSC Cori machine (Cray XC40), where each node has 128 GB DDR4 memory and is equipped with dual 16-core 2.3

TABLE 1
Input Datasets Used in Our Experiments

#	Organism	Genome size (bp)	Coverage	No. of reads	Species group	File size
i	SARS-CoV-2 (coronavirus 2 isolate Wuhan-Hu-1; March 17,2020)	29,903	200x	59,800	Coronavirus	6.3MB
ii	*Streptomyces* (GCF_000242715.1_ASM24271v2)	11,005,945	150x	16,508,918	Bacteria	1.8GB
iii	*C.elegans*	100,286,401	100x	100,286,100	Nematoda	11GB
iv	*Betta splendens* (GCA_003650155.1 ASM365015v1)	456,232,186	100x	456,232,200	Fishes	49GB
v	*Phocoena sinus* vaquita (GCA_008692045.1 mPhoSin1)	995,426,950	100x	995,426,880	Mammals	106GB
vi	full human (hg19, GRCh37 (GCA_000001405.1))	3,095,677,412	100x	2,861,320,858	Mammals	383GB
vii	bread wheat (GCA_900000045.1_Synthetic_W7984)	9,134,017,866	90x	6,778,801,997	Plants	949GB
viii	*Ambystoma mexicanum* Salamander (GCA_002915635.2 ASM291563v2)	32,396,370,977	80x	22,676,432,025	Amphibians	3.1TB

The read length was set to 100. bp *refers to base pair.*

GHz Intel Haswell processors. The nodes are interconnected with the Cray Aries network using a Dragonfly topology. Cori supports different file systems including: a) Lustre file system (with 30 PB of disk and $>$ 700 GB/second I/O bandwidth), b) Burst Buffer, and c) GPFS. Our shared memory experiments were conducted on Koothan, a single-node machine with 6TB memory and dual 224-core 2.20 GHz Intel Xeon(R) Platinum 8276M CPU.

All genomes presented in this study were obtained from NCBI (www.ncbi.nlm.nih.gov). Table 1 provides greater detail on the assembly, size and species group for each of the reference genomes.

In Section 4.1, we compare the performance (both runtime and output quality) of *PaKman* with other state-of-the-art implementations utilizing datasets *i-vi* (of Table 1) on both distributed and shared-memory systems. Section 4.2, conducts an extensive evaluation and presents a phase-wise breakdown of *PaKman*'s performance pipeline, analyzing the time spent in computation versus communication. We also include a discussion on an alternative approach to k-mer counting utilizing non-blocking MPI collective operations in Section 4.2.2. Section 4.3 analyzes the effect of tweaking certain input parameters on the performance of *PaKman*. Lastly in Sections 4.4 and 4.5 we describe the results of evaluating *PaKman* on two larger, more complex (highly repetitive) genomes namely the bread wheat and axolotl genomes (datasets *vii-viii*) respectively.

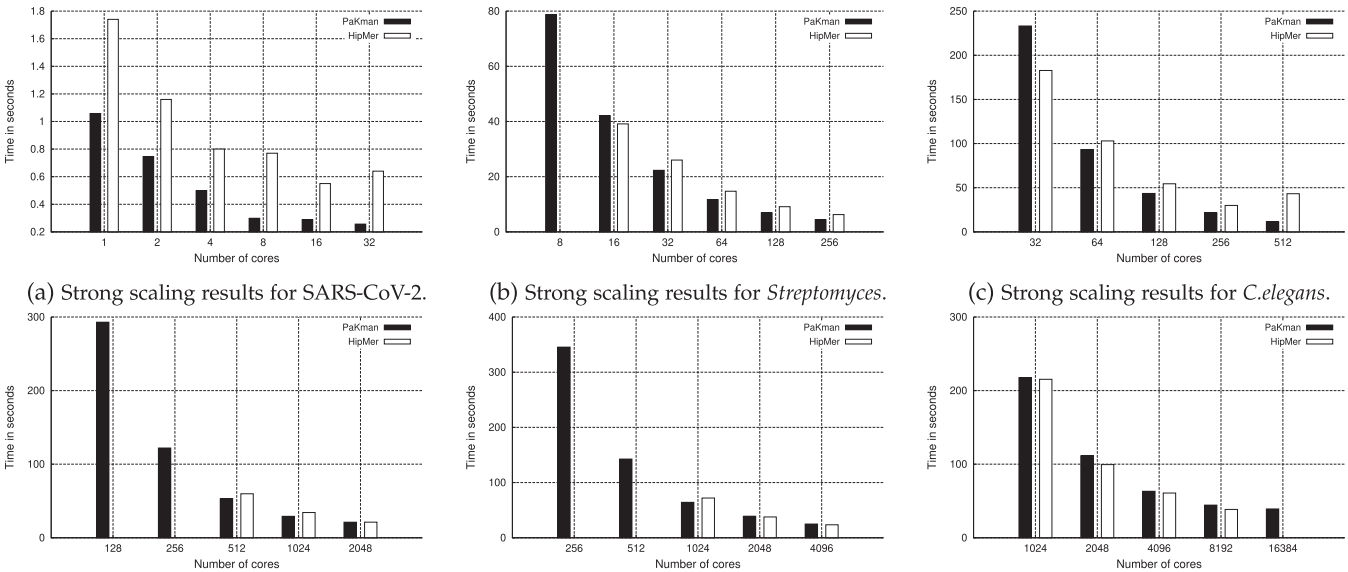
We have used the default setting of *PaKman* (unless explicitly specified) for generating all the results i.e., k = 32 and l = 8, with blocking MPI collectives.

4.1 Comparative Evaluation

4.1.1 Evaluation on Distributed Memory System

We compared the performance of *PaKman* with the latest version of *HipMer* (v0.1.2.1) available on Cori, a state-of-the-art distributed memory genome assembly tool [14], [15]. We used the installation by the authors on the NERSC Cori system. Parameters for *HipMer* were left as default, with k=31. For *HipMer* we combine the execution times obtained from their log files corresponding to the tags 'loadfq', 'kcount-31', 'meraculous-31' and 'contigMerDepth-31' and report that as the total time. Results were obtained with ppn=16 for both assemblers. Table 5 shows Lustre striping details.

Fig. 8 presents strong scaling results for both assemblers for the first six datasets. The plots show that both tools exhibit almost linear speedup under strong scaling. Although it is interesting to note that *PaKman* exhibits higher scalability



(a) Strong scaling results for SARS-CoV-2. (b) Strong scaling results for *Streptomyces*. (c) Strong scaling results for *C.elegans*.

(d) Strong scaling results for *Betta splendens*. (e) Strong scaling for vaquita. (f) Strong scaling results for full human.

Fig. 8. Strong scaling results for *PaKman* versus *HipMer* across multiple genomes. The readings not reported (in few cases for *HipMer*) indicate either a 'out of shared memory' or job time out errors.

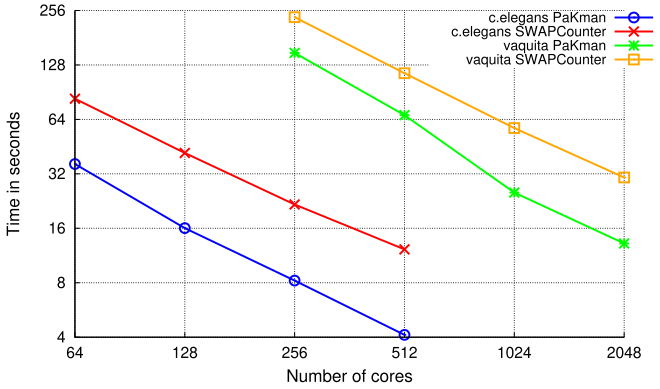


Fig. 9. Comparing k-mer counting results obtained for *PaKman* versus SWAPCounter for the *C.elegans* and vaquita datasets.

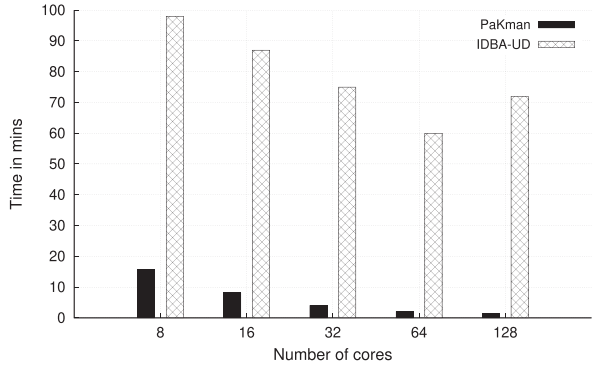


Fig. 10. Single node shared memory scaling results for *C.elegans* across assemblers IDBA-UD and *PaKman*.

across all the datasets, even for the smaller sized genomes at larger core counts.

Using PaKman, we are able to assemble a complete set of contigs for the full human genome in 39.1 seconds on 16K cores, with the k-mercounting phase taking under 5 seconds.[2]

We also compared the performance of our k-mer counting phase with a scalable distributed state-of-the-art k-mer counting tool namely SWAPCounter [16]. Fig. 9 presents the results of this comparison across multiple cores on both the *C.elegans* and vaquita datasets. We observe that *PaKman* shows a speedup over SWAPCounter by at least 1.5× (on 256 cores for vaquita) and at most 2.9× (on 512 cores for *C.elegans*).

4.1.2 Evaluation on Shared Memory System

Even though *PaKman* is designed for distributed memory machines, it can also be used on shared memory systems that support MPI. Consequently, we also performed a head-to-head comparison of *PaKman* running p MPI processes versus the shared memory tools running p threads on the same machine (Koothan). We compared against the state-of-the-art shared memory tool namely IDBA-UD [1].

Fig. 10 presents the results of the comparison for the *C.elegans* dataset. As can be seen, *PaKman* shows near-linear scaling, while IDBA-UD shows some performance improvement only up to 64 cores. More importantly, for all core counts tested, we observe that *PaKman* is considerably faster; for instance, *PaKman* shows a speedup over IDBA-UD by 6.2× (8 cores) and 50× (224 cores). These improvements are significant as these were observed despite the overheads associated with MPI.

Fig. 11 presents the single node strong scaling results of *PaKman* on the vaquita dataset. It is to be noted that using *PaKman* we can produce an output set of contigs for the vaquita dataset in just under 20 minutes on 128 cores, whereas it takes IDBA-UD over 13 hours; i.e., a speedup of 41× (128 cores).

4.1.3 Quality Evaluation

We compared the output quality of the assemblies produced by the various tools we tested. For this purpose, we compare the output contigs generated against the ground-truth (known genome from which reads were generated). The QUAST [17] tool was used for this comparison. The quality metrics reported are as follows: total number of contigs and the largest contig length; N50 contig length (larger the better); % of genome covered (larger the better); and largest alignment length (larger the better).

Tables 2 and 3, summarizes our qualitative evaluation. As can be observed, *PaKman* generally outperforms or performs comparably to the second best tool by almost all metrics (and is able to retrieve over 90 percent of the smaller sized genomes). We note here that, to enable a comparison, we did not test inputs with that paired-end read information, where an estimate of genomic distance is provided between pairs of reads (at input) alongside sequence information. This is because our tool does not yet include this feature; whereas tools such as *HipMer* and IDBA-UD do. We expect that with paired-end information these quality comparison results could change. Yet, without paired-end information, we observed *PaKman* to be competitive. It is also to be noted that *HipMer* proceeds more conservatively during contig generation and is known to produce longer contigs at the end of the scaffolding step.

4.2 PaKman: Detailed Performance Evaluation

4.2.1 Complete Pipeline Performance

To better understand the behavior of each phase at scale we further break down the execution time. Fig. 12 presents the strong scaling results of *PaKman* for the six genomes (datasets *i-vi* of Table 1), on up to 16K processes, broken down by

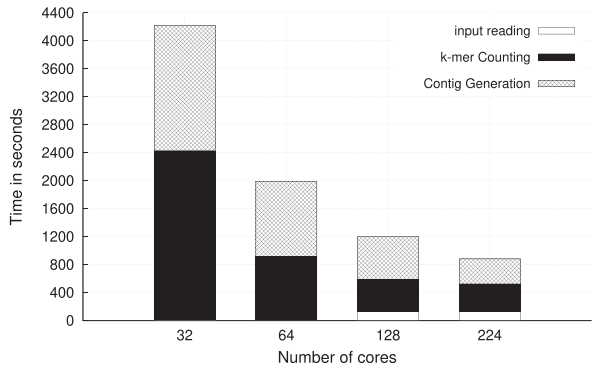


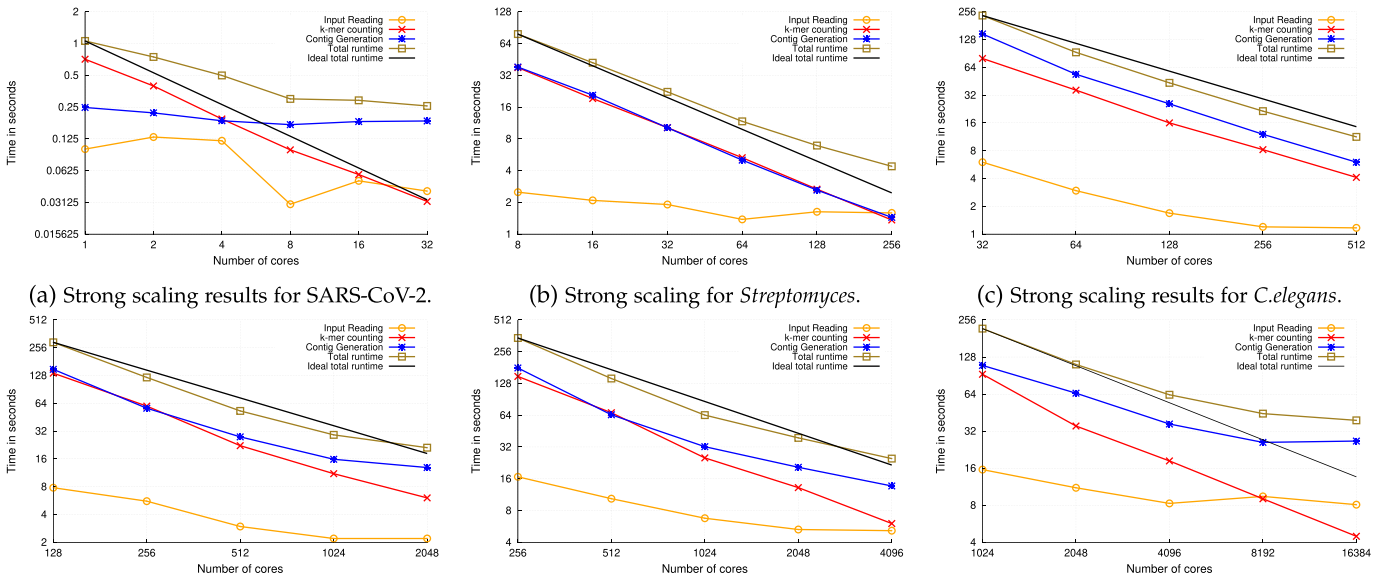
Fig. 11. *PaKman* single node strong scaling results for vaquita genome.

2. The performance timings obtained for *PaKman* across the genomes listed in Fig. 8 have been provided in the supplementary material, available online.

TABLE 2
Quality Statistics Across Assemblers

Assembler	*Betta splendens*					*vaquita*					*full human*				
	#contigs	N50	% genome coverage	Largest alignment	Largest contig len	#contigs	N50	% genome coverage	Largest alignment	Largest contig len	#contigs	N50	% genome coverage	Largest alignment	Largest contig len
PaKman	192,126	6,685	87.4	70,316	72,819	641,415	3,503	80.01	38,149	38,160	2,252,808	2,683	76.82	36,848	41,156
HipMer	102,197	5,580	81.3	59,093	59,093	322,205	3,086	70.92	38,145	38,145	1,059,809	2,405	65.66	33,414	33,414
IDBA-UD	101,776	5,650	81.8	60,520	60,520	319,382	3,135	71.55	38,147	38,147	-	-	-	-	-

'-' denotes a failed run (due to tool errors or exceeding memory capacity).



(a) Strong scaling results for SARS-CoV-2. (b) Strong scaling for *Streptomyces*. (c) Strong scaling results for *C.elegans*.

(d) Strong scaling results for *Betta splendens*. (e) Strong scaling for vaquita. (f) Strong scaling results for full human.

Fig. 12. Strong scaling results for *PaKman* across multiple datasets. Total runtime corresponds to sum of all the phases.

its phases. The *input reading* step is I/O bound, dominated by calls to MPI-IO. For this step, while an improvement in time can be seen with increase in the number of cores, speedup is hardly linear. However, the runtime contribution to the total time is almost negligible (<10 percent) in most cases. The *kmer counting* phase shows near perfect liner speedup with the number of cores across all the datasets tested. The *contig generation* step, which includes most of the communication-intensive steps such as iterative graph compaction, also shows near linear speedup, especially for the smaller genomes; although for larger core sizes (>8K cores) the speedup shows some deterioration (as can be expected).

Fig. 13 shows *PaKman*'s runtime broken down by computation, communication, and I/O. We observe that our algorithm scales efficiently and is noticeably compute bound, with the contribution from communication remaining under 20 percent even for large core counts, except at 16K cores wherein the cost of communication and computation is almost equal. Thus the algorithm is well-suited for massively parallel systems that offer greater support for compute operations than communication bandwidth. The current implementation uses blocking collectives. We have implemented a version that uses non-blocking collectives, during the *k-mer counting* phase. We discuss the findings for this non-blocking version in the next section.

Because a significant fraction of contig generation phase is spent in communication, we take a deeper look at the performance breakdown of the individual steps within the phase. Shown in Fig. 14, the steps *Initial setup wiring* and *Generate Independent set* scale linearly and are almost negligible on 8K and 16K core runs. In spite of being communication-intensive,

TABLE 3
Quality Statistics Across *PaKman* and *HipMer* for Genome Data Sets i-iii

Genome	*PaKman*				*HipMer*			
	#contigs	N50	% genome coverage	Largest align -ment	#contigs	N50	% genome coverage	Largest align -ment
SARS-CoV-2	2	29,886	99.9	29,886	1	29,894	99.9	29,894
Streptomyces	1,817	22,474	98	97,025	1,269	15,643	96.3	79,323
C.elegans	50,166	7,324	90.6	108,546	27,046	6,313	83.3	108,537

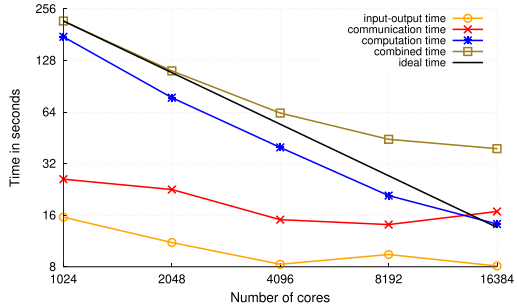


Fig. 13. *PaKman* breakup of total time spent in I/O, communication and computation for full human genome.

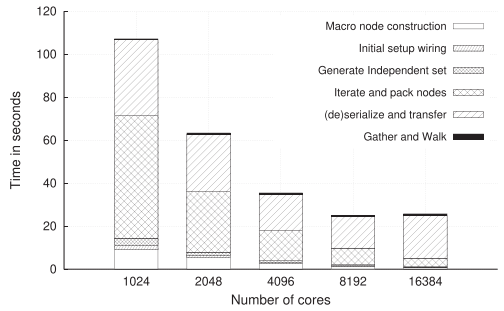


Fig. 14. *PaKman* performance breakup of the several stages in the *contig generation* step for the full human genome.

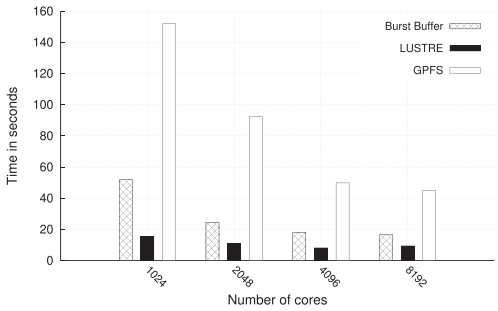


Fig. 15. I/O scaling of the *Input Reading* phase of *PaKman* for the full human dataset across various file systems.

TABLE 4
List of All MPI Calls and Their Respective Counts in Each *PaKman* Phase

PaKman Phase	MPI directive	#calls
Input reading	MPI_File_open	1
	MPI_File_get_size	1
	MPI_File_read_at_all	1
	MPI_File_close	1
k-mer counting	MPI_Allreduce	2
	MPI_Alltoall	b*1
	MPI_Alltoallv	b*1
Contig generation: MN-node construction	MPI_Alltoall	1
	MPI_Alltoallv	1
	MPI_Allreduce	1
Contig generation: (de)serialize and transfer	MPI_Allreduce	i*1
	MPI_Alltoall	i*2
	MPI_Alltoallv	i*1
Contig generation: final gather	MPI_Allgather	2
	MPI_Allgatherv	1

Term 'b' in k-mer Counting phase denotes the number of batch rounds of communication; term 'i' in Contig Generation *phase denotes the number of iterations of the* (de) serialize and transfer *phase in a given run.*

the *Iterate and pack nodes* step also scales linearly. Scalability is limited for the *(de)serialize and transfer*, as it is highly communication-bound during iterative graph compaction. The amount of communication involved in *Macro node construction* and *Gather and Walk* is minimal and does not impact the overall performance.

We note here that our algorithm uses only five distinct collectives (excluding the calls to MPI-IO), as shown in Table 4. This is a useful property to have in MPI implementations, greatly simplifying performance portable installations on new parallel systems.

As for the I/O time, Fig. 15 shows I/O scaling of the input reading step, for different file system configurations (Burst Buffer, Lustre, and GPFS). In the case of Burst Buffers, we need to scale up the number of BB nodes with compute nodes, to keep the BB nodes busy but not over-subscribed. The performance was comparable to our Lustre runs. However, we needed to tune the BB settings for each run. While GPFS served well for the smaller datasets, for the full human genome, it did not scale beyond 2,048 processes.

Table 5 shows the Lustre settings used for our *PaKman* runs. We observed that, unlike *HipMer*, the total time for *PaKman* responded to changes made to the striping configuration; this configuration can be varied and tested quickly for improving *PaKman*'s performance. Furthermore, we observe that Lustre I/O times (for the settings presented) remain constant for a given dataset across all our experiments. Table 5

TABLE 5
PaKman Runtime Statistics

PaKman	*C.elegans* (p=512)	vaquita (p=4,096)	full human (p=16,384)
# distinct k-mers	1,585,416,564	18,355,318,481	42,182,152,288
#total Macro nodes	188,188,574	1,686,172,684	4,987,035,369
#total MPI calls	155	1610	2910
Total time (secs)	11.31	24.86	39.14
Lustre I/O parameters	sc=25, ss=4M, cb=16	sc=50, ss=8M, cb=64	sc=50, ss=8M, cb=256

Lustre parameters: sc denotes stripe count, ss denotes stripe size; cb denotes the number of MPI aggregators (cb_nodes).

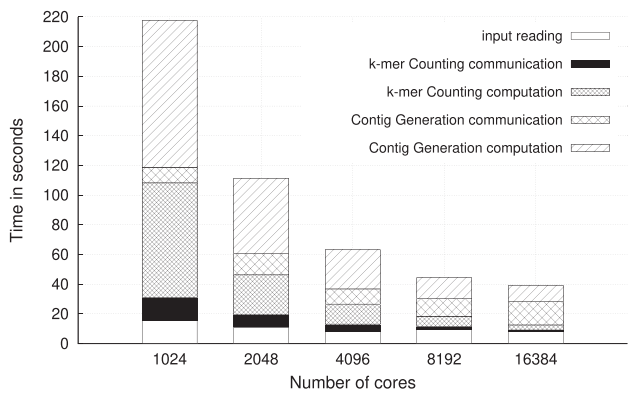
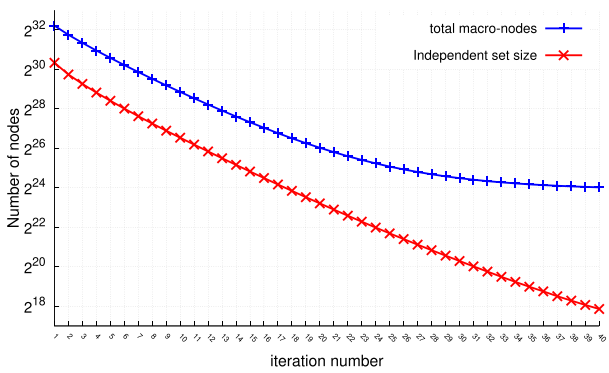


Fig. 16. *PaKman* computation and communication timings for various stages for full human genome.

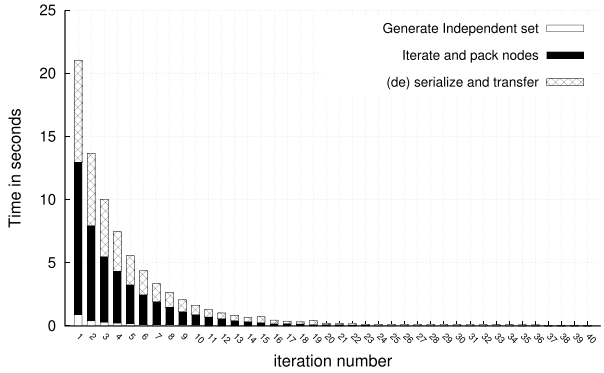
also lists various other statistics for *PaKman*. We observed that the number of macro-nodes in the initial PaK-Graph is roughly an order of magnitude smaller than the number of distinct *k*-mers. This result shows the initial degree of compression that *PaKman* achieves (even before graph compaction) compared to standard de Bruijn graph implementations.

Fig. 16 captures the computation versus communication breakdown for the individual steps of *PaKman*. We observe that the communication time for the *contig generation* step does not scale as well as the time taken for communication in the *k-mer counting* phase. This is attributed to the *contig generation* step being far more communication-intensive and thus more challenging to scale, owing to a much greater volume of communication involved as compared to *k-mer counting*.

Fig. 17 shows the performance breakdown of the first 40 (out of 700+) iterations on the iterative phase of the contig generation step on 1K cores for the full human genome. We observe a superlinear decrease in total time, which plateaus after the first 20 iterations. Fig. 17a also shows the number of macro-nodes and the independent set size, at each iteration.

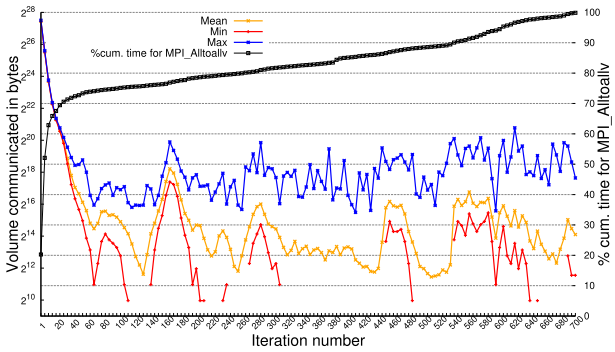


(a) Macro-node count and independent set size. y-axis is in log scale.

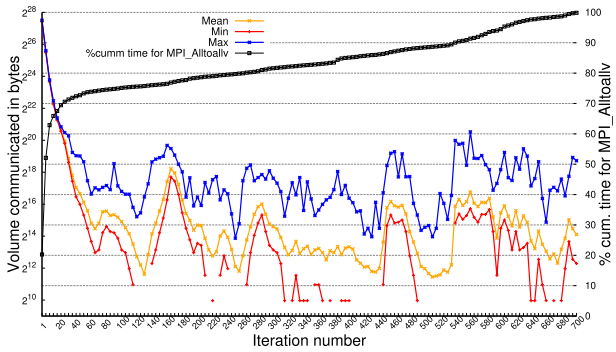


(b) Execution time of the three phases across iterations

Fig. 17. *PaKman* behavior of the first 40 iterations in iterative phase of contig generation for full human genome (p = 1024).

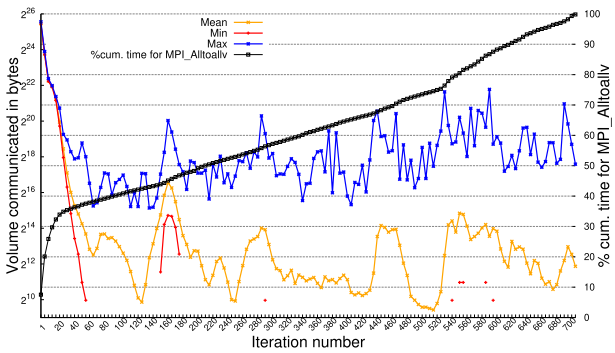


(a) Volume of data communicated in send buffer per iteration.

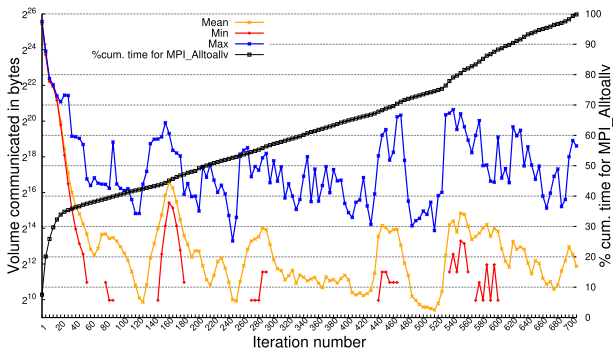


(b) Volume of data communicated in receive buffer per iteration.

Fig. 18. *PaKman* data volume communicated with corresponding cumulative percentage runtime for `MPI_Alltoallv` across all iterations, during the iterative compaction phase of contig generation for full human genome with 1024 cores.



(a) Volume of data communicated in send buffer per iteration.



(b) Volume of data communicated in receive buffer per iteration.

Fig. 19. *PaKman* data volume communicated with corresponding cumulative percentage runtime for `MPI_Alltoallv` across all iterations, during the iterative compaction phase of contig generation for full human genome with 4096 cores.

We observe that the fraction of macro-nodes included in the independent set size at each iteration shrinks gradually from 28 to 13 percent at the end of 40 iterations, to eventually 7 percent in the final iteration. Fig. 17b shows that the contributions to runtime vastly diminish after the first 10-12 iterations.

We conducted further analysis on the volume of data communicated (per iteration) during the compaction phase. Figs. 18 and 19 present our findings for all 700+ iterations executed for the full human dataset on 1,024 and 4,096 cores respectively, wherein we report the mean, minimum, and maximum bytes transferred in the corresponding send and receive buffers during the calls made to `MPI_Alltoallv`. In addition we also include the percentage cumulative runtime taken to perform `MPI_Alltoallv` across all the iterations (as represented by the y-axis on the right). For both instances of executing *PaKman* on 1K and 4K cores, we observe that the mean, minimum and maximum readings for the message volume tend to overlap for the first 30-40 iterations, which constitutes bulk of the total time (over 70 percent of the runtime) on 1,024 cores. However, at 4,096 cores, more than half the number of iterations have been consumed to reach over 70 percent of the aggregated runtime.

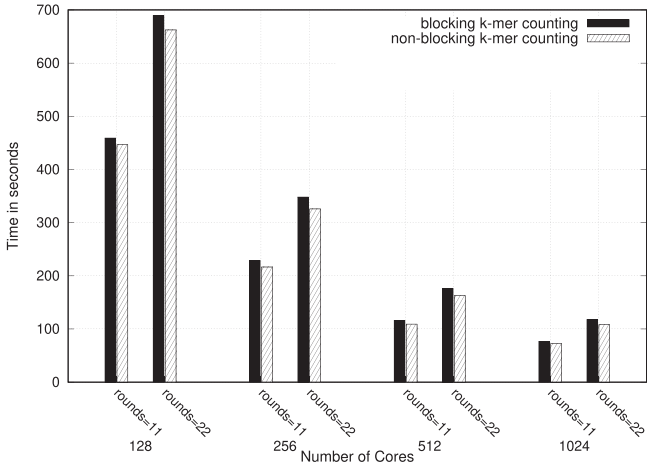


Fig. 20. Performance results of *PaKman's* blocking and non-blocking implementations for the k-mer counting phase for vaquita genome. *rounds* implies the number of rounds of communication executed in the k-mer counting phase.

This can be attributed to a wider gap observed between the minimum and maximum readings for message volume.

Taking into account the observations noted above, we would like to state that the scaling of *PaKman* in the iterative compaction phase is intricately tied to the scaling of `MPI_Alltoallv` itself for the underlying platform; and we have found that our scaling for *PaKman* is consistent with what we observed for a standard benchmark written solely to perform `MPI_Alltoallv`, and is not specific to our implementation.

4.2.2 k-mer Counting Optimization

The k-mer counting phase involves a number of batch rounds of communication and therefore is a good candidate for overlapping the respective compute and communication bound regions. In an effort to further improve the performance of this phase, we implemented a version that executes the work performed in this phase across a three stage pipeline and uses non-blocking MPI calls to overlap the communication and computation. Fig. 20 presents the performance comparison of both the blocking (default) and non-blocking versions of the k-mer counting phase across multiple processes for the vaquita genome. As expected, even though increasing the number of rounds of communication (by limiting the batch size) results in a longer runtime, it allows *PaKman* to execute on machines with insufficient memory. The performance improvement when using the non-blocking version does not exceed 10 percent owing to some overhead incurred in load imbalance. It is to be noted that for each data point, when doubling the number of cores, we halved the batch size in order to ensure that the number rounds of communication remained constant; for instance at 128 cores, a batch size of 50M k-mers was reduced to 6.25M at 1,024 cores.

4.3 Parametric Evaluation

We evaluated the effect of varying two input-based parameters—viz., coverage and read length—on the performance of *PaKman*. Table 6 presents the statistics across all four full human datasets, wherein we compare the baseline dataset (i.e., dataset i of Table 1) with three datasets of the same genome with varying read length and coverage.

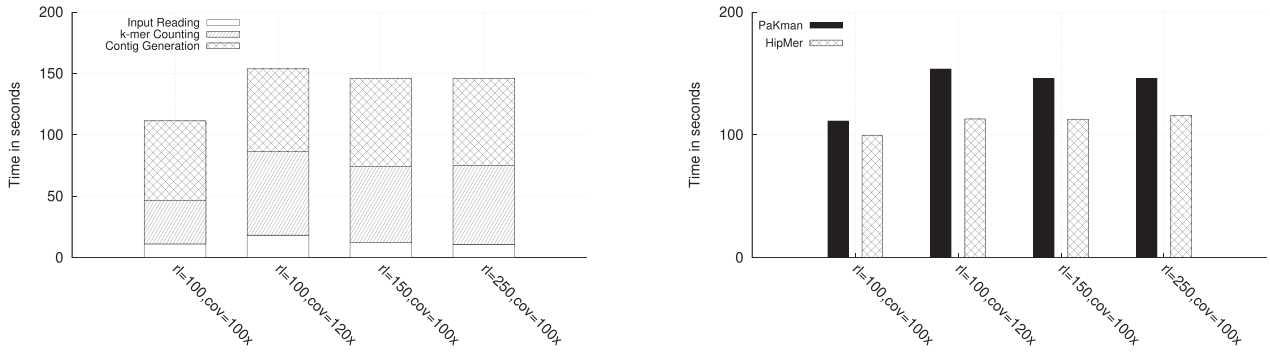
As seen in Table 6, increasing the coverage or the read length causes an increase in total runtime. Specifically, an increase in coverage contributes to a larger number of reads, which subsequently increases the work during the *k-mer counting* phase. We notice a similar effect for increasing the read length, wherein despite the presence of fewer reads, more work is needed to parse each read given its longer length, to generate the k-mers.

While these results are to be expected, we also observed that despite the increase in number of distinct k-mers, the number of macro-nodes resulting from our threshold-based pruning step (post-*k-mer counting*) remains relatively uniform across the different input settings. This is a desirable property owing to two reasons: a) Not increasing the number of macro-nodes implies negligible impact on the *contig generation* time as seen in Fig. 21a; b) Second, note that the macro-nodes capture the key *information* encoded within the input reads that contribute toward the output contigs; therefore these results show the ability of our pruning step to capture that information in a stable manner even as the input read length or coverage changes. Eventually, after contig generation, we see the positive impact of increasing coverage and read length on the output quality metrics (N50, coverage). We also note that the quality improvement is better with increased read length than with increased coverage. This is to be expected as longer reads (with a lower error rate) are a more valuable source of information than increased coverage (which simply increases the redundancy in information beyond a certain value).

The effects of these two parameters on *HipMer*'s performance as shown in the Fig. 21b in comparison to *PaKman*

TABLE 6
PaKman Performance Evaluation Across Four Full Human Datasets With Varying Parameters on 2048 Processes

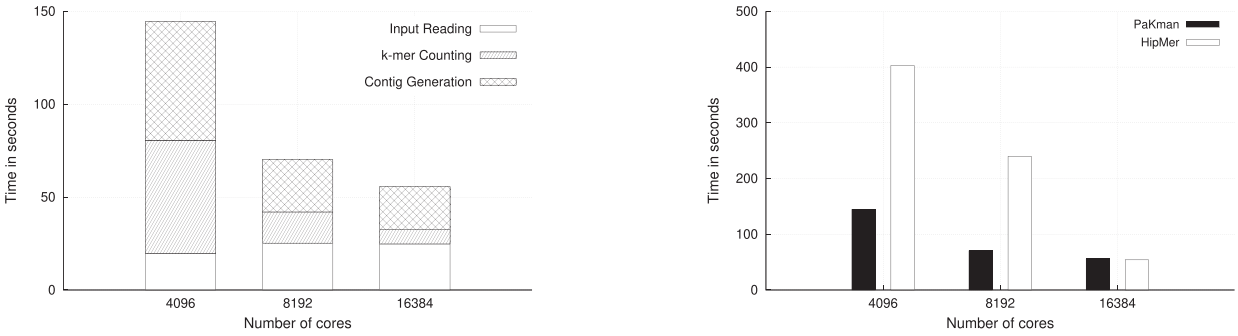
	read_len=100, cov=100x	read_len=100, cov=120x	read_len=150, cov=100x	read_len=250, cov=100x
	baseline dataset	*human_cov120*	*human_rl150*	*human_rl250*
File size (GB)	383	460	344	313
#reads:	2,861,320,858	3,433,578,880	1,907,540,469	1,144,527,774
#distinct k-mers:	42,182,152,288	49,242,849,348	16,394,494,176	30,368,099,909
#macro_nodes:	4,987,035,369	4,985,198,573	4,976,415,942	4,977,761,715
#iterations in compaction	728	730	733	803
Total time in secs	111.35	153.02	145.81	146.14
N50 (bp)	2683	2787	2962	2983
% genome coverage	76.82	78.13	79.79	80.29
largest contig	41,156	37,784	39,520	53,584
largest alignment	36,848	33,459	37,062	36,853



(a) *PaKman* performance breakdown of different phases for each dataset.

(b) Performance comparison of *PaKman* with *HipMer*.

Fig. 21. *PaKman* Performance across all four full human datasets with varying parameters for p=2048. *rl* corresponds to read length and *cov* corresponds to coverage.



(a) *PaKman* performance by phases for the bread wheat genome.

(b) Performance of *PaKman* and *HipMer* for the bread wheat genome.

Fig. 22. Performance evaluation for the bread wheat genome.

were close, although *HipMer*'s contig generation step showed greater scalability than that of *PaKman* especially for the datasets with larger coverage and read lengths.

4.4 *PaKman* Performance Evaluation for Plant Genome

We also conducted experiments on a larger more complex genome namely the bread wheat genome, characterized as highly repetitive and much larger in size (more than three times the size of the full human genome). The bread wheat genome utilized in our experiments spans 9.1 Gbp (over 56 percent) of the 16Gbp genome of hexaploid wheat, *Triticum aestivum*.

Fig. 22 presents strong scaling results for both *PaKman* and *HipMer* for the bread wheat dataset. Fig. 22a presents the breakdown in time for all the distinct phases of *PaKman*. Fig. 22b shows the total runtime wherein we observe that both tools exhibit almost linear speedup. *PaKman* completes execution of the bread wheat dataset in 55 seconds on 16K cores (with the k-mer counting phase completing in under 8 seconds) and is reported to be at-most 3.4× faster than *HipMer* at 8K cores.

4.5 *PaKman* Performance Evaluation of Salamander (Axolotl) Genome

We were able to execute runs for one of the largest genomes to be ever sequenced- Mexican axolotl (*Ambystoma mexicanum*) [18]; a key representative salamander genome used widely for molecular investigations. Axolotl provides a powerful tool in studying molecular basis for limb and other forms of regeneration, including studies pertaining to severed spinal cord and other retinal tissue. So far the complete *de novo* assembly of this genome has been a challenge owing to its sheer size: 32 billion base pairs (ten times larger than the full human genome), as well as its inherent complexity due to the presence of a significant number of large repetitive regions.

Table 7 presents the performance breakdown numbers for executing the 3.1TB dataset of the axolotl genome using *PaKman* on 16,384 cores (∼1024 nodes) on the NERSC Cori machine. *PaKman* was able to process the dataset with over 294 billion distinct k-mers and produce a full set of contigs in just over 200 seconds with the k-mer counting phase taking merely 41.8 secs. Fig. 23 presents a pie chart illustrating the time spent in each phase of the *PaKman* pipeline. We

TABLE 7
PaKman Performance Breakdown for the Salamander (axolotl) Dataset on 16 384 Cores

PaKman phase		Time in secs
input reading		35.42
k-mer counting breakdown:	communication	12.41
	computation	29.39
k-mer counting total		**41.8**
contig generation breakdown:	communication	69.19
	computation	57.37
contig generation total		**126.56**
Total assembly runtime:		**203.78**

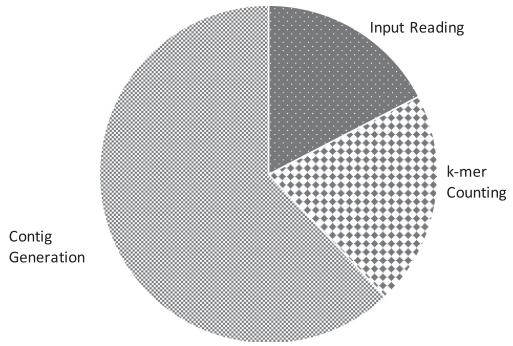


Fig. 23. Pie chart representing the time spent in each phase of *PaKman* when executing the Salamander (axolotl) genome on 16K cores.

observe that for genomes of this magnitude, *PaKman*'s contig generation phase produced a higher communication footprint, consuming nearly 34 percent of the total runtime. This is primarily attributed to the iterative compaction phase that took 1,336 iterations in order to condense a 37 billion macro-node PaK-Graph (to under 100,000 nodes). The performance results for HipMer for this dataset have been omitted owing to lack of storage available on the Cori scratch filesystem to undergo further tests.

5 Related Work

De novo genome assembly is a widely researched topic with a number of assembly tools and algorithms developed over the last two decades. Therefore in this section, we focus primarily on parallel short-read assemblers. Short read assemblers correspond to that subset which target reads generated from NGS technologies (e.g., Illumina, 454 pyrosequencing, SOLiD). Most modern day short read assemblers have widely adopted the de Bruijn Graph-based (DBG) method of assembly, originally introduced by Pevzner *et al.* [19]. Popular shared memory DBG based assemblers include (but not limited to): Velvet [2], ALLPATHS-LG [20], and SOAP-denovo [3], all of which utilize OpenMP/Pthreads for parallelization. More recent implementations of the method include IDBA-UD [1], an iterative DBG assembler that generates assemblies by sequentially iterating from small to large k-values used in graph construction. Although OpenMP parallelized (for a single k), this method can be time intensive since graphs for multiple k values proceed sequentially. SPAdes [5] has support for multithreading and produces assemblies of high quality owing to its detailed error correction step. However, it is costly with respect to the amount of time and memory it consumes. We were unable to run SPAdes for our medium to large datasets owing to its significant memory footprint. In an attempt to reduce the memory footprint of assembly, Chapman *et al.* proposed the Meraculous algorithm [21]. The algorithm uses the Bloom filter [22], which is a probabilistic data structure for answering membership queries, to generate its version of the de Bruijn graph.

Notable examples of distributed memory DBG based assemblers include Ray [23], PASHA [24] and YAGA [25]. Ray and YAGA have shown to be scalable except for the I/O that proves to be a bottleneck when reading and writing to files. ABySS [4] is a full end-to-end assembler and is one the first to be parallelized using MPI and was the first software to assemble a human genome from short reads. However their input reading step presents a bottleneck to the overall performance. ABySS 2.0 [26] departs from using MPI and instead employs Bloom filters to represent a de Bruijn graph and reduce memory requirements. HipMer [14], [15] as discussed in the previous section, also uses Bloom filters to generate its version of the de Bruijn graph. For parallelization, HipMer uses both MPI and the Unified Parallel C language (UPC) [27], [28]. In our evaluation, we observed HipMer to scale well for all sizes of input data at high core counts. SWAP [29] and SWAP-2 [30] are also among the newer set of assemblers that have been MPI parallelized for executing at large scale for large genomes. Although the assembly output from SWAP-2 was high in quality, we observed it failed to execute on NERSC Cori beyond small to medium sized datasets.

There have been notable contributions in the field of distributed k-mer counting, with several standalone implementations that can perform the process at scale. Bloomfish [31] is a memory-efficient, scalable k-mer counting tool that leverages a single-node k-mer counting framework Jellyfish [32] coupled with a MapReduce over MPI framework. Kmerind [33] is another example that presents a generic distributed-memory library for both k-mer counting and indexing. SWAPCounter [16] on the other hand utilizes a MPI streaming I/O module coupled with MPI non-blocking collectives, and a bloom filter implementation for discarding low-abundance k-mers. We compare the performance of SWAPCounter with *PaKman*'s k-mer counting phase in Section 4.1.1.

6 Conclusion and Future Work

We introduced *PaKman*, a new algorithm for efficient scaling of two crucial phases of the genome assembly pipeline. We presented a new data structure—*PaK-Graph*—for contig generation with simplified communication requirements. Our method demonstrated a speedup of up to $2\times$ on average in comparison with state-of-the-art distributed and up to $41\times$ compared to shared memory implementations, respectively.

Our goals for future work include: a) Incorporation of paired end information toward improving quality of the final set of contigs: although not trivial the wiring function has the capacity to be extended for this purpose. But a space-efficient representation will be needed to make it work as the graph compacts. We speculate that this shall minimize the occurrence of false connections in the wiring phase, leading to longer contigs. b) Provide an end-to-end solution, by incorporating the scaffolding phase to fully complete the assembly pipeline. c) Extension to support new generations of sequencing technologies, particularly long reads. d) Extension to use heterogeneous architectures– GPU's and future many-core machines.

Acknowledgments

This research used resources of the NERSC Office of Science User Facility supported by U.S. DOE under Contract No. DE-AC02-05CH11231. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award No. 63823. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract No. DE-AC05-76RL01830. The research was in parts supported by U.S. NSF Grants CCF 1815467, OAC 1910213, and CCF 1919122 to WSU.

REFERENCES

[1] 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.

[2] D. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Res.*, vol. 18, pp. 821–829, 2008.

[3] R. Li *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Res.*, vol. 20, no. 2, pp. 265–272, 2010.

[4] 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 Res.*, vol. 19, pp. 1117–1123, 2009.

[5] A. Bankevich *et al.*, "SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing," *J. Comput. Biol.*, vol. 19, no. 5, pp. 455–477, 2012.

[6] P. Ghosh, S. Krishnamoorthy, and A. Kalyanaraman, "PaKman: Scalable assembly of large genomes on distributed memory machines," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 578–589.

[7] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[8] A. Kalyanaraman, S. J. Emrich, P. S. Schnable, and S. Aluru, "Assembling genomes on large-scale parallel computers," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, pp. 10–pp.

[9] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de Bruijn graphs," in *Proc. Int. Conf. Res. Comput. Mol. Biol.*, 2014, pp. 35–55.

[10] E. Cohen, "Min-hash sketches," in *Encyclopedia of Algorithms*. Berlin, Germany: Springer, 2008, pp. 1–7.

[11] W. S. Grant and R. Voorhies, "Cereal–A C++ 11 library for serialization," 2013. [Online]. Available: https://github. com/USCiLab/cereal

[12] 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, 2011.

[13] BBMap guide. Accessed: Dec. 14, 2020. [Online]. Available: https://jgi.doe.gov/data-and-tools/bbtools/bb-tools-user-guide/bbmap-guide/

[14] 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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 437–448.

[15] E. Georganas *et al.*, "HipMer: An extreme-scale de novo genome assembler," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 14.

[16] J. Ge, J. Meng, N. Guo, Y. Wei, P. Balaji, and S. Feng, "Counting kmers for biological sequences at large scale," *Interdisciplinary Sci., Comput. Life Sci.*, vol. 12, no. 1, pp. 99–108, 2020.

[17] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, "QUAST: Quality assessment tool for genome assemblies," *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, 2013.

[18] S. Nowoshilow *et al.*, "The axolotl genome and the evolution of key tissue formation regulators," *Nature*, vol. 554, no. 7690, pp. 50–55, 2018.

[19] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proc. Nat. Acad. Sci. United States America*, vol. 98, no. 17, pp. 9748–9753, 2001.

[20] S. Gnerre *et al.*, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proc. Nat. Acad. Sci. USA*, vol. 108, no. 4, pp. 1513–1518, 2011.

[21] J. A. Chapman *et al.*, "Meraculous: De novo genome assembly with short paired-end reads," *PloS One*, vol. 6, no. 8, 2011, Art. no. e23501.

[22] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. London, U.K.: Pearson Education, 1998.

[23] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *J. Comput. Biol.*, vol. 17, no. 11, pp. 1519–1533, 2010.

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

[25] 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 *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–10.

[26] S. D. Jackman *et al.*, "ABySS 2.0: Resource-efficient assembly of large genomes using a bloom filter," *Genome Res.*, vol. 27, pp. 768–777, 2017.

[27] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center Comput. Sci., Tec. Rep. CCS-TR-99-157, 1999.

[28] K. Yelick *et al.*, "Productivity and performance using partitioned global address space languages," in *Proc. Int. Workshop Parallel Symbolic Comput.*, 2007, pp. 24–32.

[29] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "Swap-assembler: Scalable and efficient genome assembly towards thousands of cores," *BMC Bioinf.*, vol. 15, no. 9, 2014, Art. no. S2.

[30] 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 *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 195–204.

[31] T. Gao *et al.*, "Bloomfish: A highly scalable distributed k-mer counting framework," in *Proc. IEEE 23rd Int. Conf. Parallel Distrib. Syst.*, 2017, pp. 170–179.

[32] G. Marcais 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.

[33] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A flexible parallel library for K-mer indexing of biological sequences on distributed memory systems," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 16, no. 4, pp. 1117–1131, Jul./Aug. 2019.

Priyanka Ghosh received the MS degree in computer science from the University of Houston, Houston, Texas, and the PhD degree in computer science from Washington State University, Pullman, Washington, in 2019. She is currently a postdoctoral research associate at Pacific Northwest National Laboratory, Richland, Washington. Her research interests include the field of parallel algorithms, high performance computing, and computational biology. She is the recipient of a Best Student Paper Award (ACM BCB'16) and a Best Paper Award (IEEE/ACM NOCS'19).

Sriram Krishnamoorthy (Senior Member, IEEE) is a computer scientist and laboratory fellow at PNNL's High Performance Computing Group, and a research professor at the School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington. His research focuses on parallel programming models, quantum computing, fault tolerance, and compile-time/runtime optimizations for high-performance computing. He has more than 100 peer-reviewed conference and journal publications, receiving three Best Paper and two best student paper awards. In 2013, he received a U.S. Department of Energy Early Career Award. That year, he also earned PNNL's Ronald L. Brodzinski Award for Early Career Exceptional Achievement. In 2008, he received The Ohio State University's Outstanding Researcher Award. He is a senior member of ACM.

Ananth Kalyanaraman (Senior Member, IEEE) received the bachelor's degree from the Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the MS and PhD degrees from Iowa State University, Ames, Iowa, in 2002 and 2006, respectively. Currently, he is a professor and boeing centennial chair in computer science, at the School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington. He also holds a joint appointment with Pacific Northwest National Laboratory, Richland, Washington. His research focuses on developing parallel algorithms and software for data-intensive problems originating in the areas of computational biology and graph-theoretic applications. He is a recipient of a DOE Early Career Award, Early Career Impact Award, and two best paper awards. He serves on editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* and *Journal of Parallel and Distributed Computing*. He is a member of ACM, ISCB, and SIAM.