Parallel and Memory-efficient Preprocessing for Metagenome Assembly

Vasudevan Rengasamy Paul Medvedev Kamesh Madduri
The Pennsylvania State University
University Park, PA, USA
vxr162@psu.edu, pashadag@cse.psu.edu, madduri@cse.psu.edu

***Abstract*—The analysis of high-throughput metagenomic sequencing data poses significant computational challenges. Most current *de novo* assembly tools use the de Bruijn graph-based methodology. In prior work, a connected components decomposition of the de Bruijn graph and subsequent partitioning of sequence read data was shown to be an effective memory-reducing preprocessing step for *de novo* assembly of large metagenomic datasets. In this paper, we present METAPREP, a new end-to-end parallel implementation of a similar preprocessing step. METAPREP has efficient implementations of several computational subroutines (e.g., k-mer enumeration and counting, parallel sorting, graph connectivity) that occur in other genomic data analysis problems, and we show that our implementations are comparable to the state-of-the-art. METAPREP is primarily designed to execute on large shared-memory multicore servers, but scales gracefully to use multiple compute nodes and clusters with parallel I/O capabilities. With METAPREP, we can process the Iowa Continuous Corn soil metagenomics dataset, comprising 1.13 billion reads totaling 223 billion base pairs, in around 14 minutes, using just 16 nodes of the NERSC Edison supercomputer. We also evaluate the performance impact of METAPREP on MEGAHIT, a parallel metagenome assembler.**

1. Introduction

The field of metagenomics leverages advances in DNA sequencing technology to directly analyze genomes of communities of organisms. Metagenomics can aid applications in medicine, energy, agriculture, and several other areas. Due to the incompleteness of reference genome databases, *de novo* assembly is often performed as a first step when processing metagenomic datasets. Most *de novo* genome and metagenome assembly software tools use the *de Bruijn graph*-based assembly methodology.

Howe et al. [1] present two *preprocessing* strategies, *digital normalization* [2] and *partitioning*, to divide the de Bruijn graph corresponding to large metagenomic datasets into individual components. These preprocessing strategies are designed to separate non-overlapping sequences from different species into distinct components of the de Bruijn graph. The main computational routine in the partitioning strategy of Howe et al. is to perform a weakly connected components (WCC) decomposition of the de Bruijn graph. Flick et al. [3] propose creating a *read graph*, which is an undirected graph whose connected components (CC) correspond to the WCCs of the de Bruijn graph. The CC output on the read graph can be directly used as input to metagenome assembly software. Flick et al. further present a scalable distributed memory CC labeling algorithm tuned for metagenomic read graphs.

In this paper, we present a new tool called METAPREP (*Meta*genome *Prep*rocessing) for performing, in parallel, the read graph partitioning approach described by Flick et al. We use MPI for distributed memory parallelism and OpenMP for shared memory multithreading. A key contribution of our work is the highly memory-efficient implementation of the sequence of steps in this preprocessing strategy. The following are the key new ideas in METAPREP:

- We present a multi-pass strategy that accesses the input files multiple times to reduce aggregate main memory requirements. The main steps are all compatible with multiple passes through the data.
- We use hybrid parallelism (with OpenMP and MPI) in all steps to reduce inter-process communication.
- To ensure load-balanced parallel computation without dynamic scheduling, we store and precompute two index files. These indices can be reused for parallel runs on different compute platforms. They also significantly reduce shared memory synchronization overhead.
- Instead of explicitly constructing the read graph or the de Bruijn graph, we use an implicit graph representation that does not require ordering of vertices or edges.
- Our tool is constructed in a modular manner, and the implementations of each step are competitive in performance with state-of-the-art algorithms and tools.

We evaluate the impact of this preprocessing strategy on the performance of the MEGAHIT [4] metagenome assembler.

2. Background and Related Work

The *de novo* assembly of metagenomes is challenging due to the following reasons: (i) Closely related strains from

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.159

IEEE computer society

the same species might be present in the community sample, and these are difficult to distinguish from repeats in the genomes of individual organisms. (ii) The sequencing depth of coverage and the size of the genome of individual organisms might be different, which leads to creating datasets that are very different from isolate genomes. (iii) Metagenomic sequencing dataset sizes are typically larger than isolate genomic datasets.

Some metagenome-specific assembly tools developed in the past few years include MEGAHIT [4], MetaVelvet [5], Ray Meta [6], and Meta-IDBA [7]. These assembly tools attempt to overcome challenges with metagenomic data processing using a combination of methods.

The input to an assembly program is a set of *reads*. Reads are short sequences from which k-mers (a k-mer is a string of length k) are enumerated. In a de Bruijn graph, each vertex is a k-mer, and edges connect k-mers whose prefixes and suffixes overlap by a string of length $k-1$. Since the genomes in a metagenome can have varying depth of coverage, assemblers such as MEGAHIT use multiple k-mer lengths (i.e., a range of values of k). Small k values help in reconstructing low coverage genomes, and larger k values help in resolving repeats.

Howe et al. [1] present preprocessing strategies that can be used to partition large metagenomic datasets. Their preprocessing strategy filters reads before constructing a de Bruijn graph, and then performs a weakly connected components decomposition of the de Bruijn graph. Howe et al. empirically show that such a k-mer-based data partitioning assigns most reads belonging to a species to the same component (partition), and so, each component can be assembled independently.

However, on applying this partitioning strategy and when constructing the de Bruijn graph, the read information corresponding to k-mers is lost. More recently, Flick et al. [3] presented a similar *read graph* partitioning strategy. Each vertex in the read graph is a read, and two reads are connected by an edge if they share at least one common k-mer. The authors show that (proof sketch in [3]) if two k-mers k_1 and k_2 belong to a WCC of the de Bruijn graph, then the reads containing these k-mers also belong to a CC in the read graph. Further, if k-mers k_3 and k_4 lie in different WCCs in the de Bruijn graph, then the reads containing these k-mers also belong to different CCs. Thus, the problem of WCC decomposition of the de Bruijn graph (without filtering) is closely related to CC decomposition of the read graph. After partitioning, reads in a partition can be assembled independently. Flick et al. did not assess the impact of this strategy on metagenome assembly.

In this paper, we perform a parallel read graph CC decomposition identical in functionality to the approach of Flick et al. The focus of [3] was on a distributed memory CC labeling algorithm, whereas we also consider the steps before and after the CC labeling step, and assess the impact of read graph partitioning on metagenome assembly quality.

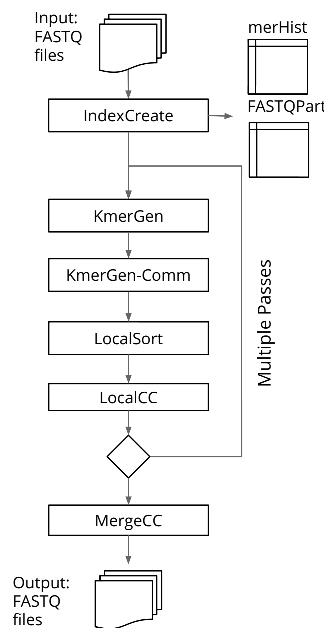

Figure 1. METAPREP overview.

TABLE 1. THE SEQUENCE OF PREPROCESSING STEPS PERFORMED.

	METAPREP step	Function
	IndexCreate	Create index files for parallel runs.
1	KmerGen	On each task, enumerate $\langle k$-mer, read$_i\rangle$ tuples.
2	KmerGen-Comm	Transfer $\langle k$-mer, read$_i\rangle$ tuples to corresponding owner tasks.
3	LocalSort	Local sort of tuples with k-mers as keys.
4	LocalCC	Identify local connected components (CCs).
5	MergeCC	Merge components across tasks, create output FASTQ file with reads from largest CC.

3. METAPREP Algorithms

In this section, we describe METAPREP steps in detail and motivate our implementation choices. Figure 1 and Table 1 summarize the main steps. The input to METAPREP is a collection of FASTQ read files corresponding to the metagenome. The output is another set of FASTQ files containing reads in individual partitions. IndexCreate is a sequential preprocessing step that is performed once for each dataset. This step creates files to ensure static load balancing and easy parallelization of METAPREP steps. Due to the double-stranded nature of DNA, we need to consider both a k-mer and its reverse complement when we process reads. A *canonical k-mer* is the lexicographically smaller string among a k-mer and its reverse-complemented sequence. METAPREP implicitly creates a read graph by enumerating canonical k-mers present in reads (in the KmerGen step), and then sorting the enumerated canonical k-mers to identify reads containing common canonical k-mers (in the LocalSort step). If the output of KmerGen does not fit in

the aggregate main memory of the parallel system, we use multiple passes through the input files to generate disjoint sets of k-mers. The memory required for each pass can be determined using the indices constructed in the IndexCreate step. Once the read graph edges are created, METAPREP partitions reads into weakly connected components by using a distributed union-find approach (LocalCC and MergeCC). Finally, partitioned reads are written in FASTQ format. LocalCC and MergeCC can be considered a distributed-memory implementation of CC decomposition, similar in functionality to the algorithms proposed by Flick et al. [3] and Jain et al. [8]. The KmerGen and MergeCC steps involve some file I/O and the rest of the steps are performed in memory. The KmerGen step is similar to routines in k-mer counting tools (e.g., DSK [9], KMC 2 [10]) and de Bruijn graph preprocessing algorithms.

3.1. Index Creation (IndexCreate)

In this step, we create two tables, merHist and FASTQ-Part. These tables are written to disk in binary format and used as indices for subsequent steps.

3.1.1. m-mer Histogram (merHist). We store counts of all m-mer prefixes of canonical k-mers ($m < k$; we use $m = 10$ in this work) in the merHist table. So there are 4^m histogram bins and the counts are stored as 32-bit integers. The input FASTQ files are read once, canonical k-mers are generated, and the m-mer bin counts are updated. The histogram is used to partition the range of integers spanned by k-mer values (k-mer range) for multipass and parallel execution.

3.1.2. FASTQ partitions table (FASTQPart). Since we want multiple threads to independently enumerate k-mers in a load-balanced manner, we logically partition FASTQ files into C chunks which have approximately the same file size. In the FASTQPart table, each record contains information for one chunk, which includes the location of the chunk within the FASTQ file, global read ID (an integer value) of the first read in the chunk, and the size of the chunk. These parameters are used for reading each chunk in parallel. In addition, each record also stores a m-mer histogram similar to the merHist table, with counts of m-mer prefixes of canonical k-mers present in the corresponding FASTQ chunk. The FASTQPart table structure is shown in Figure 2. The chunk-specific histograms are used for calculating the buffer sizes and offsets for sending/receiving k-mers to/from other processes.

These two tables let us statically determine, for a given task and thread concurrency, the main memory required per thread, the fewest number of passes for the dataset, the k-mer range to enumerate in each pass, the offsets into the FASTQ files that the threads should read from, and the thread offsets for in-memory buffers created in subsequent steps. These tables help simplify the parallel implementation of subsequent steps and also let us preallocate memory for data structures.

Chunk	FASTQ Offset	Size	First Read Id	m-mer Histogram		
				0	...	4^m -1
1						
. .						
C						

Figure 2. FASTQPart table structure.

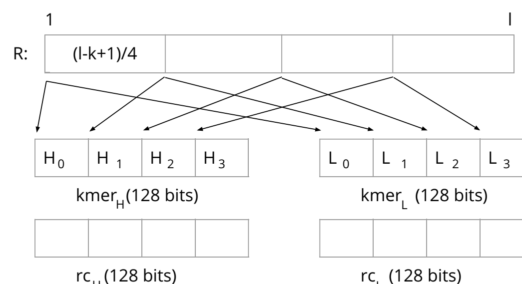

Figure 3. Vectorized k-mer generation.

3.2. Enumerate k-mers (KmerGen)

In this step, on each pass, threads independently read one or more FASTQ file chunks and generate k-mer and global read identifier tuples. We use 64 bits to store k-mer values and 32 bits for global read identifiers, and hence each tuple requires 12 bytes. We do not enumerate k-mers that contain the N symbol in reads. We also use a single read identifier for both ends of a paired-end read, because we want to preserve paired-end read information when generating the final output. The C file chunks are distributed to threads to enable parallel FASTQ file read operations.

3.2.1. k-mer generation. We use SIMD vector extensions to generate four canonical k-mers at a time from a given read sequence. See Figure 3 for an illustration. R is a read of length l containing $l-k+1$ k-mers. For ease of description, we assume $l - k + 1$ is a multiple of 4. In the initial step, four k-mers are generated from four equidistant points (separated by $(l-k+1)/4$ letters) in the read and stored in two 128-bit vector registers. One of these registers (kmer$_H$) stores the MSBs (most significant bits) of the four k-mers and the other (kmer$_L$) stores the corresponding LSBs (least significant bits). Similarly, two more registers (rc$_H$ and rc$_L$) store the reverse complemented k-mers. We then output four canonical k-mers by comparing the original and the reverse complemented k-mers and selecting the lexicographically

smaller of the two. In each subsequent step, we generate the next four k-mers by shifting the four currently-generated k-mers and their reverse complements by one base and adding the bases that appear next to these k-mers.

3.2.2. Storing k-mers. For each MPI task, we allocate a single buffer in memory to store all generated tuples. Since all threads within a MPI task add k-mer tuples to the same buffer, we precompute write offsets for each thread. This lets us update the buffer without synchronization. We use the chunk-specific histograms in the FASTQPart table to compute the offsets as follows: The number of tuples generated by the T threads in a process p that lie in the k-mer range assigned to another process p' can be calculated by adding histogram counts in the FASTQPart table rows corresponding to the chunks assigned to threads in p and columns corresponding to k-mer range of process p'. These counts are stored in an array of size $P \times T$ and a prefix sum of this array gives the offsets for individual threads. We store all k-mers belonging to the k-mer range of a single process p' consecutively so that k-mers can be communicated to their respective processes in a single step.

3.3. Exchange k-mer tuples (KmerGen-Comm)

Once each task enumerates k-mers and read identifier tuples, we need a communication step in the distributed-memory implementation to do a disjoint partitioning of the tuples based on the k-mer value. This communication step can be considered a precursor to sorting the tuples with the k-mer as the key. In a shared-memory implementation, there is no need for this step. We do not use MPI's Alltoallv collective due to the limitation imposed by the sendcounts and recvcounts parameters (that they need to be 32-bit signed integers). Instead, we develop a custom All-to-all approach using multiple point-to-point messages. In addition to calculating the offsets within the send buffer as described in Section 3.2.2, each task also calculates the number of tuples to be received from other tasks and the corresponding receive offsets in advance using the FASTQPart table. The number of tuples received by process p from T threads of process p' can be calculated by adding histogram counts in the FASTQPart table rows corresponding to the FASTQ chunks assigned to individual threads in p' and columns corresponding to k-mer range of process p. Our All-to-all implementation has P stages (P is the number of MPI tasks). In stage i, task p sends tuples to task $(p+i) \mod P$.

3.4. Local sort of k-mer tuples (LocalSort)

Once each task receives tuples corresponding to its k-mer range from all tasks, the tuples are sorted locally with the k-mer as the key. We sort the tuples in order to identify all reads sharing a canonical k-mer and construct the read graph. The local sort in each task has two stages.

1) Parallel Partitioning: This step involves partitioning the received tuples on each task into T disjoint partitions based on k-mer value, so that each partition can be sorted concurrently. Each partition corresponds to one of the T thread-specific k-mer ranges in each process. Since multiple threads write tuples into a common output buffer during range partitioning, we precompute offsets using the FASTQPart table.
2) Serial Radix Sort: After range partitioning, we use T threads to concurrently sort the T disjoint partitions. Each thread sorts its k-mer sub-range using a serial out-of-place radix sort. We reuse the send buffer of KmerGen-Comm step for storing the sorted tuples. We use 8 passes to sort tuples based on the 64-bit k-mers, with each pass sorting 8 bits (using 256 buckets). We find that sorting 8 bits per pass is faster than sorting higher number of bits (say, 16) because accessing bucket counts of 256 buckets repeatedly has better temporal locality than accessing counts of 65536 (2^{16}) buckets randomly, even though the number of passes is high.

3.5. Connected components (LocalCC)

After LocalSort, each thread points to a contiguous list of sorted tuples. These tuples can be used to generate edges of the read graph. Flick et al. implement a distributed-memory parallelization of the Shiloach-Vishkin algorithm [11] for CC decomposition. Their iterative approach uses parallel sorting as a subroutine. We use a different strategy in this work that is compatible with multiple I/O passes over the data. Assuming the total number of reads is known beforehand, we adapt and parallelize the Union-Find approach for CC decomposition. The main advantage of using Union-Find is that the graph need not be explicitly constructed, and components can be dynamically updated. Further, we let each task update a local instance of the disjoint set data structure (whose size is proportional to the number of reads) for the tuples it stores, and the merge step is performed separately and only once even with multiple passes through the data.

Our shared memory parallel Union-Find approach combines ideas from prior algorithms by Cybenko et al. [12] and Patwary et al. [13]. In the Find operation, we use the *path splitting* optimization [14]. For Union, we use the *union-by-index* approach since it avoids introducing cycles in the component trees when edges are processed concurrently. In union-by-index, the parent pointer of the root element with lower index is set to the root element with higher index. Initially, the parent of each read (vertex) is set to point to itself. Each thread then processes tuples it stores according to Algorithm 1. The Union and Find operations of different threads proceed concurrently without any synchronization. Similar to [13], we also buffer the edges that result in Union operations to be processed in the next iteration. The shared memory algorithm of Cybenko et al. [12] uses *union-by-size* and Find with path compression. Cybenko et al. show that path compression is safe to perform in a multithreaded setting. To avoid lost updates due to concurrent union operations, Cybenko et al. treat union operations as critical sections. Instead, we keep track of the edges resulting in a

Algorithm 1 LocalCC Union-Find algorithm.

```
1:  E_in ← E                          ▷ Process all edges in the first iteration
2:  E_out ← φ
3:  while E_in ≠ φ do
4:      for Each edge ⟨u, v⟩ ∈ E_in do
5:          root_u ← Find(u)
6:          root_v ← Find(v)
7:          if root_u ≠ root_v then
8:              Union(root_u, root_v)          ▷ Race condition
9:              E_out ← E_out ∪ (u, v)
10:     E_in ← E_out
11:     E_out ← φ
```

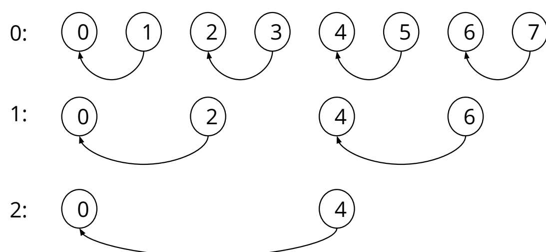

Figure 4. MergeCC: Communication with 8 MPI tasks.

union operation on each thread and verify them after processing all edges. We find that the overall time is dominated by the time for the first iteration.

3.5.1. Multipass Optimization (LocalCC-Opt).

We describe an additional optimization when performing multiple passes through the data. The running time of the Find operation is dominated by random lookups to the component array (p) to identify roots. To improve the data locality while accessing the component array, we enumerate k-mer and component ID instead of the k-mer and global read ID. The component ID can be obtained by performing Find on the corresponding read ID. This change does not affect correctness, but improves locality. Since the number of components is much smaller than the number of reads, the random accesses to the p array are limited to a lower number of locations corresponding to reads which are component roots.

3.6. Merge local components (MergeCC)

After MPI tasks create local components, the results have to be combined to obtain the final components. We combine this information in $\lceil \log P \rceil$ iterations. We show an example with 8 processors in Figure 4. This method was introduced by Cybenko et al. [12]. In each iteration, tasks with a higher MPI rank send their component array (p) to the corresponding lower rank task. In successive iterations, the number of tasks participating in the communication reduces by a factor of 2. The receiving tasks treat the received component arrays (p') as additional edges to be processed and update their local p array. For example, the ith element is treated as an edge from vertex i to vertex $p'(i)$ since vertices i and $p'(i)$ are in the same component in the sending task. A Union operation is performed on the components containing the vertices i and $p'(i)$. The MPI task with rank 0 has the final component information. Since the number of reads is substantially smaller than the total number of graph edges, it is feasible to replicate the component array on each task.

In this step, we also create new FASTQ files for the partitioned read sequences. We currently write the reads corresponding to the largest component to one file, and all other reads to another file, since we observed a giant component being formed for most of the datasets we experimented with. This step is also performed in parallel. The global components list in Rank 0 is broadcast to all other tasks. Each thread extracts reads from its FASTQ chunks and writes them to the corresponding output FASTQ files. Each thread writes to separate FASTQ files.

3.7. Algorithm Analysis

We give analytic estimates for running times and memory used per task in terms of input data parameters. Let M be the cumulative size of the input FASTQ files (in giga base pairs, Gbp), R the total number of paired-end reads, and C the number of file chunks. Let s_c be the size of each FASTQ chunk in bytes. The number of tuples enumerated is upper-bounded by M. Let P be the number of MPI tasks, T be the number of threads per task, and S be the number of passes.

The running time for KmerGen is O($\frac{MS}{PT}$). For local sort, the time is O($\frac{M}{PT}$) (assuming radix sort is linear-time, and in each pass we sort the same number of tuples). In LocalCC, each thread performs O($\frac{M}{PT}$) Union and Find operations on R reads, and the time complexity is O($\frac{M}{PT} \log^* R$) (considering the time for only the first iteration). For the MergeCC step, the time taken by task 0 determines the overall time. The time complexity is O($R \log P \log^* R$). There is additionally inter-node communication in the KmerGen-Comm and MergeCC steps. From this brief analysis, if S is a small constant, the asymptotic running times of the first four steps are essentially the same. The MergeCC step might become a bottleneck if $R \log P \gg \frac{M}{PT}$.

We store the following arrays in main memory: merHist and FastQPart; reads from FASTQ chunks; generated k-mer tuples (kmerOut); received k-mer tuples (kmerIn); component array (p); component array received from another task (p').

Since we use m-mers as histogram bins and 4 bytes to store histogram frequencies, the merHist table requires 4^{m+1} bytes. Since FASTQPart contains the m-mer histogram for each chunk, it requires $4^{m+1}C$ bytes. During KmerGen step, each thread loads a FASTQ chunk into memory. Hence, FASTQBuffer requires Ts_c bytes per task. The number of k-mer tuples generated by each task in each pass is approximately $\frac{M}{SP}$, and so kmerOut and kmerIn each require $\frac{12M}{SP}$ bytes. p and p' are $4R$ bytes each. Thus,

TABLE 2. DESCRIPTION OF DATASETS.

ID	Dataset	Read Count R ($\times 10^6$)	Size M (Gbp)	Source
HG	Human gut	12.7	2.29	NCBI (SRR341725)
LL	Lake Lanier	21.3	4.26	NCBI (SRR947737)
MM	Mock microbial community	54.8	11.07	NCBI (SRX200676)
IS	Iowa, Continuous corn soil	1132.8	223.26	JGI (402461)

the total memory (in bytes) required per task is given by $4^{m+1}(C+1) + Ts_c + \frac{24M}{SP} + 8R$. The dominant term is $\frac{24M}{SP}$, and we can increase the number of passes to reduce the per-task memory footprint.

We give an example for the memory requirement per MPI task for processing IS, the largest dataset used in our experiments. When using 8 passes, 16 processes, and 24 threads per process, we can calculate the memory per task as follows. The merHist data structure requires 4 MB. We divide the reads into 1536 chunks and hence the FASTQPart table requires approximately 6 GB memory. Each FASTQ chunk is approximately 0.3 GB. Since we use 24 threads per process, the size of FASTQBuffer per process is approximately 7 GB (24×0.3). The number of tuples enumerated by each MPI process in a pass is approximately 1.3 billion, and so kmerIn and kmerOut each require approximately 14 GB. The number of reads is 1.13 Billion, and thus the p and p' arrays require approximately 8 GB memory. Thus, the total memory per MPI task is 49 GB ($6 + 7 + 2 \times 14 + 8$).

4. Performance Results

We now present results from an empirical evaluation of METAPREP on a collection of large-scale metagenome datasets. The datasets used in our experiments are listed in Table 2, and we chose these datasets as they were used in prior studies [1], [3], [4], [8]. We set k to 27 in most experiments.

We perform experiments primarily on the NERSC Edison Cray XC30 supercomputer. Each compute node of Edison has two 12-core Intel Xeon E5-2695 v2 processors and 64 GB memory. We use the Lustre-based scratch file system for I/O on Edison. We compile all programs on Edison with the default Cray compiler settings and use the scatter policy for pinning threads to cores. The memory bandwidth obtained using the STREAM Triad benchmark is 99 GB/s. The point-to-point link bandwidth of large messages is 8 GB/s. For all multi-node experiments on Edison, we use 1 MPI task per node and 24 threads on each task. We also present results on a single node of the Penn State Ganga cluster. Each node has two 6-core Intel Xeon E5-2620 processors and 64 GB memory. On Ganga, we compiled code using the GNU C compiler v4.9.2.

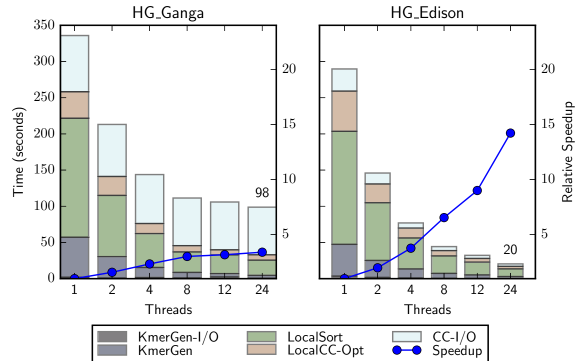

Figure 5. Single node execution times and relative speedup. The number on top of bar gives execution time in seconds.

4.1. Parallel Scaling

In this section, we evaluate scalability of METAPREP preprocessing steps using single and multi-node experiments.

4.1.1. Single-node scaling. Figure 5 shows multithreaded execution times and speedup for processing the HG dataset on a single node of Ganga and Edison. KmerGen-I/O indicates the time to read FASTQ chunks in the KmerGen step and CC-I/O denotes the I/O time in the CC step. A single Edison node is nearly 5 times faster than a Ganga node for this dataset. Parallel file writes do not scale well on the shared file system of Ganga, resulting in poor overall scalability ($3.4\times$ relative speedup on 24 threads). On Edison, both I/O and compute steps scale well, resulting in a $14.5\times$ speedup on 24 cores. This experiment indicates the need for efficient parallel I/O to achieve good overall performance. On Edison, LocalSort is the most time-consuming step for all thread concurrencies.

4.1.2. Multi-node scaling. Figure 6 gives multi-node execution times and relative speedup (compared to single node execution time) for three datasets. In this figure, MergeCC-Comm indicates the step that merges local components. The k-mer tuples for the HG dataset fit in a single node's memory, and so we use 1 I/O pass for all runs. The other two datasets are bigger. We use 2 passes for LL and 4 passes for MM in all experiments. The relative speedup on 16 nodes varies from $3.23\times$ (HG) to $7.5\times$ (MM). The speedup values are less than the ideal $16\times$ speedup primarily because of the additional inter-node communication and merge steps. Also, the KmerGen-I/O step does not scale to high process and thread counts, leading to a reduction in overall speedup. However, it is notable that we can process a 11.1 Gbp data set (MM) in just 22 seconds on 16 nodes.

Figure 7 shows 16-node, 8-pass and 64-node, 2-pass results for the large IS dataset. The KmerGen step is the dominant time-consuming stage in both runs. We achieve

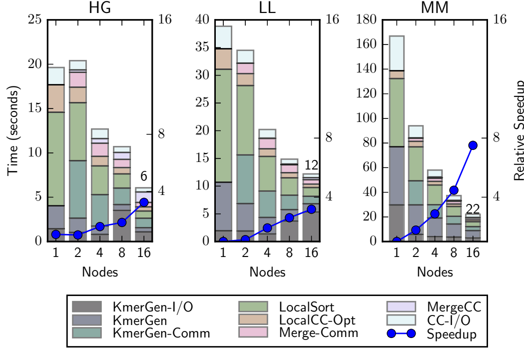

Figure 6. Multi-node execution times and relative speedup. The number on top of bar gives execution time in seconds.

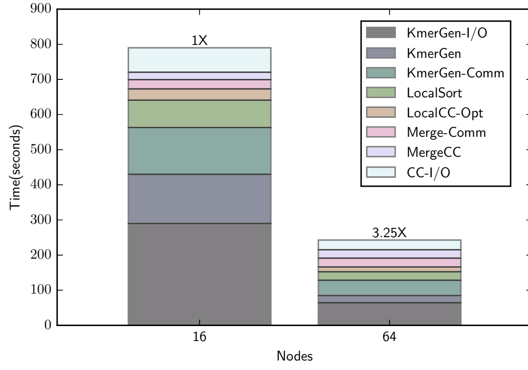

Figure 7. Multi Node execution time for Iowa Continuous corn soil dataset (8 passes for 16 nodes, 2 passes for 64 nodes).

a 3.25× speedup going from 16 to 64 nodes, due to the reduction in the number of passes and an increased 4× parallelism. Local sort is not the dominant step, unlike the single-node case for the other datasets.

4.1.3. Load balance. To show load balance among MPI tasks, in Figure 8, we give a box plot of execution times for processing the MM dataset on 16 nodes. We note that the KmerGen, LocalSort and LocalCC-Opt steps have good load balance due to the use of the indexes. The MergeCC-Comm and MergeCC stages have $\log P$ sub-steps (4 in this case) of communicating local component information and merging components. The difference in the time spent by different tasks in these steps is due to fewer tasks participating in successive iterations of the distributed merge step.

4.1.4. Multi-pass execution statistics. Table 3 evaluates the impact of multi-pass execution on METAPREP's time and memory consumption. For MM, we vary the number of passes from 1 to 8 and use 4 compute nodes for each run. The KmerGen time increases with an increasing number

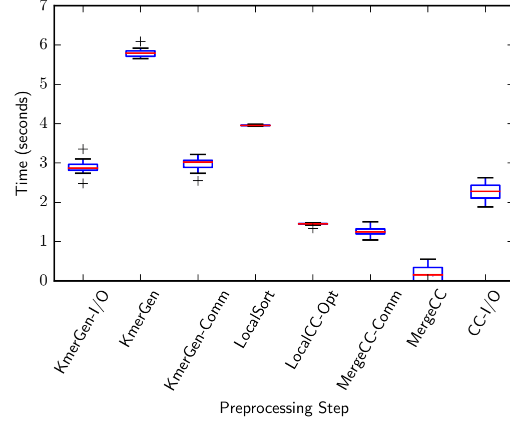

Figure 8. Load balance among 16 MPI tasks - MM dataset.

of passes because FASTQ files are redundantly read on each pass. The k-mer tuple communication time decreases with an increasing number of passes. This is because the communication time for the first pass is higher (due to communication setup time), and communication in subsequent passes takes less time. The LocalSort time does not change much, as the total number of tuples to sort remains the same. LocalCC time decreases with increasing number of passes due to the multi-pass optimization described in Section 3.5.1. By enumerating component identifiers instead of read identifiers during k-mer enumeration, cache locality improves considerably during LocalCC step. MergeCC time decreases with increasing number of passes as the communication time decreases similar to KmerGen-Comm step. Finally, CC-I/O time does not change because same number of reads are written for all passes. As expected, the per-node memory footprint decreases with increasing number of passes.

4.2. Comparisons to State-of-the-art

4.2.1. KmerGen performance comparison. We compare the efficiency of our KmerGen step to the KMC 2 k-mer counting tool [10]. KMC 2 is a shared-memory parallel approach using the idea of minimizers (super k-mers), and has a significantly smaller memory footprint compared to other k-mer counters. Figure 9 compares the k-mer enumeration time of KMC 2 and our method for the datasets (HG, LL, and MM), and also shows the relative speedups. The Stage1 time in KMC 2 involves reading FASTQ input files, enumerating, and binning super k-mers. In Stage2, the bins are sorted, compacted, and written to disk. Stage1 time for METAPREP corresponds to the KmerGen and KmerGen-Comm steps, and Stage2 corresponds to LocalSort. For HG, METAPREP performs better in Stage1, but worse in Stage2 compared to KMC 2. This is because in Stage1, KMC 2 incurs an additional overhead of finding super k-mers and in Stage2, the number of tuples to be sorted by METAPREP is higher than that of KMC 2. For the other two

TABLE 3. METAPREP EXECUTION TIME AND MEMORY USE FOR MM DATASET WHEN VARYING NUMBER OF I/O PASSES. ALL RUNS USE 4 NODES.

Number of	Time (seconds)							Memory/Node
Passes	KmerGen	KmerGen-Comm	LocalSort	LocalCC-Opt	MergeCC	CC-I/O	Total	(GB)
1	10.95	20.91	12.48	6.51	5.00	5.4	61.32	49.72
2	12.04	12.35	15.23	4.9	3.13	5.35	53	27.02
4	20.21	9.93	15.23	3.9	3.57	5.4	58.24	15.64
8	33.42	8.56	15.16	2.52	1.70	5.34	66.70	9.96

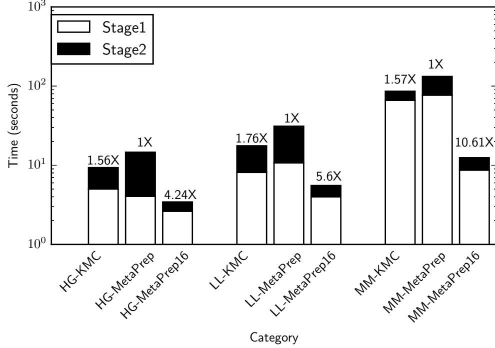

Figure 9. KmerGen time: Comparison with KMC 2.

TABLE 4. EXECUTION TIME COMPARISON WITH METAGENOME PARTITIONING WORK (AP_LB).

Dataset	Time (seconds)		METAPREP
	METAPREP	AP_LB	Speedup
HG	5.59	23.6	4.22×
LL	11.53	25.97	2.25×
MM	19.6	56.1	2.86×

larger datasets, Stage1 for METAPREP becomes slower than KMC 2 because KmerGen becomes slower, due to more passes for these datasets. Finally, we see that METAPREP on 16 nodes performs better as the dataset size increases.

4.2.2. LocalSort performance comparison. We evaluate the efficiency of our parallel radix sort implementation by comparing with the NUMA-aware out-of-place stable LSB radix sort implementation of [15]. We benchmarked our LocalSort step and the NUMA-aware radix sort on a single node of Edison using 24 threads. The NUMA-aware sort processes up to 196 million tuples per second, whereas our LocalSort implementation processes up to 154 million tuples per second, thereby achieving 78% performance of this state-of-the-art sort implementation. We could not directly use the NUMA-aware implementation in our work, because this code requires that both the key and payload be 64 bits, and also assumes that the input data is stored in disjoint partitions across NUMA domains.

4.2.3. Comparison to read graph connectivity [3]. Table 4 compares the execution time of some of the steps in METAPREP to the corresponding steps in the *AP_LB* (Active Partition with Load Balancing) metagenome partitioning work [3]. AP_LB filters reads/k-mers based on k-mer frequency before partitioning. We do not report the time for this step. We use 16 Edison nodes for all runs. The speedup of METAPREP over AP_LB varies from $2.25\times$ for LL to $4.22\times$ for HG. The improvement is primarily because our method requires fewer communication rounds ($\log P$) in comparison to the $O(\log M)$ iterations for the Shiloach-Vishkin algorithm. AP_LB requires 19, 20, and 21 iterations for the HG, LL, and MM datasets, respectively.

TABLE 5. INDEX CREATION TIME (SEQUENTIAL).

Dataset	# Chunks	Time (seconds)	
		FASTQPart	merHist
HG	384	32	109
LL	384	32	154
MM	384	33	343
IS	1536	180	5160

4.3. Index Creation

The FASTQPart and merHist table creation steps are currently sequential, since they are executed just once for each dataset, and hence are not in the critical path. Table 5 lists the time to create these indices. Creating logical chunks of FASTQ files involves finding the start offset in the FASTQ files for every chunk. This step incurs additional overhead in case of paired-end FASTQ files containing trimmed reads because after finding the chunk offset in one FASTQ file, the same read has to be located in the other FASTQ file. Creating k-mer frequency histograms is similar to the KmerGen preprocessing step and can be parallelized in the same manner. Currently, this step takes 5160 seconds for the largest dataset (Iowa continuous corn soil) on Edison.

4.4. Use with MEGAHIT

We next evaluate the impact of preprocessing on metagenome assembly time and quality. After preprocessing, reads in different components can be assembled in parallel. However, we find that read-based preprocessing results in a single giant component and numerous extremely small components. For instance, for the MM dataset, when using $k = 27$, 99.5% of the reads belong to the giant component. We instead desire a balanced decomposition. So, we explore two strategies to reduce the size of the largest component: we use a k-mer frequency-based filter

TABLE 6. IMPACT OF k ON SINGLE-NODE METAPREP EXECUTION TIME (MM DATASET).

k	Time (seconds)				
	KmerGen	LocalSort	LocalCC-Opt	CC-I/O	Total
27	77.02	55.33	6.41	5.40	144.16
63	59.73	67.60	5.16	5.35	137.84

TABLE 7. LARGEST COMPONENT SIZE WITH DIFFERENT VALUES OF k AND k-MER FREQUENCY FILTER (KF) SETTINGS.

k	Filter	LC size (% Reads)		
		HG	LL	MM
27	None	95.5	76.3	99.5
63	None	87.1	58.9	97.8
27	$KF < 30$	73.5	67.6	45.0
27	$10 \leq KF < 30$	55.2	45.2	40.0
63	$10 \leq KF < 30$	51.6	30.6	59.0

and try larger values of k. The k-mer frequency-based filter only considers read graph edges that correspond to a user-specified k-mer frequency. High frequency k-mers may occur due to repeated sequences in the metagenome. Low frequency k-mers may occur due to sequencing errors. Most de Bruijn graph-based assemblers include such filters in the graph construction step, and this results in improved assembly quality. Additionally, using a larger value of k could result in fewer edges between reads. However, note that very aggressive filtering or an unreasonably large value of k could result in very small components.

We modify the METAPREP k-mer enumeration code to support k-mer sizes up to 63. With this change, the size of a k-mer tuple is 20 bytes (16-byte k-mer and 4-byte read ID). Table 6 shows the impact of a higher k on single-node execution time for the MM dataset. We compare performance with k set to 27 and 63. Although using 63-mers requires 20 bytes per k-mer tuple, the number of 63-mers is less than the number of 27-mers (4.12 billion tuples vs 8.4 billion tuples), and hence the KmerIn and KmerOut buffers require less storage (91 GB for 27-mer run vs. 78.65 GB for 63-mer run). Due to the lower number of tuples, the execution time of all steps except LocalSort step is lower for the 63-mer run, resulting in better overall time. LocalSort requires 8 radix passes for sorting 27-mers and 16 radix passes for 63-mers, and so the sort is slower in the 63-mer run.

Table 7 shows the percentage of reads in the largest component, when running METAPREP with different filter settings. Even without using the k-mer frequency filter, the size of the largest component decreases when k is increased from from 27 to 63. However, even with $k = 63$, the largest component size is still considerable for HG and MM datasets. We find that applying the k-mer frequency filter reduces the largest component size in general. For the MM dataset, there are many 27-mers occurring more than 30 times when compared to 63-mers and hence, more 27-mers are filtered out. This is the reason for the larger component size for the $k = 63, 10 \leq KF < 30$ filter in comparison to $k = 27, 10 \leq KF < 30$. We chose the values 10, 30, and 63 arbitrarily. An extensive evaluation of filtering strategies and k settings is left for future work.

TABLE 8. MEGAHIT ASSEMBLY TIME (IN SECONDS) WITH AND WITHOUT PREPROCESSING. KF DENOTES k-MER FREQUENCY FILTER.

Dataset	MEGAHIT time (seconds)					METAPREP time (seconds)	Speedup
	No Preproc	LC No Filter	Other No Filter	LC $KF < 30$	Other $KF < 30$		
HG	1082	1009	51	848	141	39	1.22×
LL	2857	2228	456	2100	553	75	1.31×
MM	2211	2252	25	1464	962	168	1.36×

Table 8 compares the time taken by the MEGAHIT metagenome assembler to assemble the whole sequence data (No Preproc) with the time to assemble reads partitioned by METAPREP, with and without using filters. The table also shows METAPREP preprocessing time for each dataset. METAPREP time includes the time for reading and writing FASTQ files. All experiments are run on a single node of Edison supercomputer, since MEGAHIT does not support multi-node parallelism. It can be seen that METAPREP's preprocessing time is very low compared to the actual assembly time even on a single node. METAPREP time can be further reduced using multiple compute nodes. We report MEGAHIT time for assembling reads belonging to the largest component (indicated as LC) and the time for assembling the remaining reads (indicated as Other). These two runs can be performed in parallel using 2 compute nodes. We define speedup to be the time for MEGAHIT assembly on the full data set divided by the sum of METAPREP time and the time to assemble the largest component reads (with filtering). We obtain speedups up to 36% with this approach.

Table 9 gives assembly quality results with and without preprocessing. We set k to 27 in METAPREP for all the experiments. For all the datasets, 'No Preproc' and 'No Filter' (i.e., using METAPREP and splitting the dataset into large component and Other reads) result in similar qualitative results. For example, the length of the largest contig recovered (indicated Max in the table) and N50 values are very similar in these two cases.

When $KF < 30$ filter is used, the size of the largest component decreases, and this results in better assembly time, as shown in Table 8. The corresponding assembly quality is shown in the rows labeled $KF < 30$. The total number of bases assembled improves with filtering. The largest contig is recovered in the LC assembly for two of the three datasets: HG and LL. For these two datasets, we find that N50 value is higher than unpartitioned read assembly. For the MM dataset, this filtering criterion becomes very aggressive and reduces assembly quality. Hence, filter settings need to be chosen carefully, with the overall goal of reducing assembly time while not reducing quality.

TABLE 9. ASSEMBLY QUALITY COMPARISON.

Dataset	Type	MEGAHIT assembly output statistics			
		Contigs	Total (Mbp)	Max (bp)	N50 (bp)
HG	No Preproc	63 519	116.19	217 183	5071
	No Filter	63 483	116.18	217 183	5098
	LC	58 770	113.83	217 183	5510
	Other	4713	2.35	2860	513
	$KF < 30$	64 571	119.01	217 183	5123
	LC	56 732	110.13	217 183	5687
	Other	7839	8.87	43 863	2271
LL	No Preproc	179 828	165.63	225 770	1273
	No Filter	181 751	166.67	225 805	1263
	LC	141 136	148.75	225 805	1593
	Other	40 615	17.9	4028	432
	$KF < 30$	182 717	168.42	225 770	1275
	LC	140 081	147.51	225 770	1587
	Other	42 636	20.90	43 718	465
MM	No Preproc	24 931	203.65	1 067 762	50 607
	No Filter	25 002	203.65	1 067 762	50 550
	LC	23 959	202.99	1 067 762	50 781
	Other	1043	0.66	5788	695
	$KF < 30$	40 632	208.24	611 608	23 126
	LC	26 233	156.04	611 608	28 135
	Other	14 399	52.19	591 560	12 285

5. Conclusions and Future Work

In this work, we developed a new bioinformatics tool called METAPREP for partitioning metagenomic reads into disjoint components, such that each component can be independently assembled by existing assemblers. Empirical results show that METAPREP exhibits good strong scaling on a single compute node with 24 cores (14.5× speedup), while also exhibiting reasonable scaling on multiple compute nodes. We also evaluated the performance of individual steps in METAPREP by comparing with corresponding state-of-the-art implementations. Using just 16 compute nodes of the NERSC Edison system, METAPREP partitioned the Iowa Continuous Corn soil dataset in around 14 minutes.

The scalability of METAPREP is partially limited by the MergeCC step, the complexity of which increases with increasing number of MPI tasks. This step could be improved by adopting the component graph contraction methods described in [16]. Also, we find that the metagenome datasets we processed result in creation of a giant component (i.e., a single large component). We explored two simple strategies to reduce the size of the giant component in this work. In future work, we will explore alternate component-splitting strategies, and additional settings in which we can use existing genome and metagenome assembly software.

Acknowledgment

This research is supported in part by NSF awards #1453527, #1356529, and #1439057. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank Chita Das for providing access to the Ganga cluster and the reviewers for their helpful comments.

References

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

[2] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, no. 33, pp. 13 272–13 277, 2012.

[3] P. Flick, C. Jain, T. Pan, and S. Aluru, "A parallel connectivity algorithm for de Bruijn graphs in metagenomic applications," in *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[4] D. Li, R. Luo, C.-M. Liu, C.-M. Leung, H.-F. Ting, K. Sadakane, H. Yamashita, and T.-W. Lam, "MEGAHIT v1.0: A fast and scalable metagenome assembler driven by advanced methodologies and community practices," *Methods*, vol. 102, pp. 3–11, 2016.

[5] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara, "MetaVelvet: an extension of velvet assembler to de novo metagenome assembly from short sequence reads," *Nucleic acids research*, vol. 40, no. 20, pp. e155–e155, 2012.

[6] S. Boisvert, F. Raymond, É. Godzaridis, F. Laviolette, and J. Corbeil, "Ray Meta: scalable de novo metagenome assembly and profiling," *Genome biology*, vol. 13, no. 12, p. R122, 2012.

[7] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Meta-IDBA: a de Novo assembler for metagenomic data," *Bioinformatics*, vol. 27, no. 13, pp. i94–i101, 2011.

[8] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, "An adaptive parallel algorithm for computing connectivity," *arXiv.org e-Print archive*, 2016. [Online]. Available: http://arxiv.org/abs/1607.06156

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

[10] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: Fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

[11] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[12] G. Cybenko, T. G. Allen, and J. E. Polito, "Practical parallel Union-Find algorithms for transitive closure and clustering," *International Journal of Parallel Programming*, vol. 17, no. 5, pp. 403–423, 1988.

[13] M. M. A. Patwary, P. Refsnes, and F. Manne, "Multi-core spanning forest algorithms using the disjoint-set data structure," in *Proc. IEEE Int'l. Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[14] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[15] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison and radix-sort," in *Proc. ACM SIGMOD Int'l. Conf. on Management of Data (SIGMOD)*, 2014.

[16] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of connected-component labeling algorithms for distributed-memory systems," *Parallel Computing*, vol. 44, pp. 53 – 68, 2015.