2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Design and Optimization of a Metagenomics Analysis Workflow for NVRAM

Sasha Ames, Jonathan E. Allen, David A. Hysom, G. Scott Lloyd and Maya B. Gokhale
Lawrence Livermore National Laboratory
Contact: ames4@llnl.gov

***Abstract*—Metagenomic analysis, the study of microbial communities found in environmental samples, presents considerable challenges in quantity of data and computational cost. We present a novel metagenomic analysis pipeline that leverages emerging large address space compute nodes with NVRAM to hold a searchable, memory-mapped "k-mer" database of all known genomes and their taxonomic lineage. We describe challenges to creating the many hundred gigabytes-sized databases and describe database organization optimizations that enable our Livermore Metagenomic Analysis Toolkit (LMAT) software to effectively query the k-mer key-value store, which resides in high performance flash storage, as if fully in memory.**

To make database creation tractable, we have designed, implemented, and evaluated an optimized ingest pipeline. To optimize query performance for the database, we present a two-level index scheme that yields speedups of $8.4 \times -74\times$ over a conventional hash table index. LMAT, including the ingest pipeline, is available as open source at SourceForge.

I. Introduction

Metagenomics is concerned with characterizing and analyzing microbial communities present in diverse natural environments. Metagenomics sequencing has emerged as a powerful genetic survey tool used in research for generating a far more unbiased and detailed description of a biological sample than has ever been possible. It is the nature of a metagenomic input set that the organisms it contains are diverse and largely unknown, and analysis of such data sets represents a data intensive analysis problem that challenges conventional computing approaches. Present approaches to metagenomic analysis rely on sequence alignment tools that match each genetic fragment ("read") to a part of each reference sequence and report a summary of the top reference matches. However, with rapidly growing sequencer throughput, alignment-based approaches are facing severe scaling limitations, even with high CPU core counts. Even within approximately optimal algorithms there is a quadratic complexity of $Q * R$, where Q is the number of query bases and R is the number of searched reference bases. The size of Q has increased by several orders of magnitude in recent years and R, while not increasing at nearly the same rate, is also growing in size.

To enable near real-time analysis of metagenomic datasets at field sites co-located with the sequencer, we have developed a novel alignment-free approach, the Livermore Metagenomic Analysis Toolkit (LMAT) that exploits large (persistent) memory to store a searchable database of k-mers, all k-length sequences from a reference set of genomes [1]. By transferring the computational load to an offline database generation phase, we transform the analysis problem to parallel search of a read-only database. Our approach has been shown to perform favorably in accuracy and speed to competing alignment-based techniques. The source code is available on SourceForge [2].

Our approach leverages emerging large address space compute nodes with NVRAM to hold a searchable k-mer database of all known genomes and their taxonomic lineage. The database resides in a file in NVRAM, which is mapped into the address space of the application, allowing the application to access data structures directly as if in memory. This approach anticipates future memory hierarchies incorporating NVRAM as a high capacity, high latency last-level memory. We show that even with today's NAND flash arrays, we achieve excellent query processing rates: in some cases more than 1 million base pairs processed per second (Mbp/s). We believe this is a practical approach for analysis co-located with sequencing because NVRAM is becoming ubiquitous in emerging architectures and the price of it is falling.

In this paper we describe the workflow pipeline to generate the searchable k-mer database, evaluate database organization optimizations, and analyze speed performance for database query using real metagenomic query sets. The full LMAT database reaches almost 500GB in size. It is organized as a key-value store in which the data stored with each k-mer key records alternative taxonomic lineage for the k-mer. To make database creation tractable for our reference database, we parallelize generating the k-mers of reference genomes. We evaluate alternative key-value store organizations, and have developed a two level index optimized for flash storage. During query, the database is memory-mapped into the address space of the query process, and we use a custom mmap handler optimized for data intensive applications [3]. The optimized index gives a $8.4 \times -74\times$ speedup over a conventional hash table index. The speed performance of our approach is evaluated using 0.5-2.5 GB query sets.

Our contributions are as follows:

- demonstrate a scalable new approach to generating large, memory-mapped searchable k-mer databases, starting from NCBI reference genome sets,
- evaluate alternative approaches to the database ingest

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.200

pipeline, including parallelization on a conventional cluster, on a small ScaleMP cluster, and on a single large memory node,
- design a two-level index data structure uniquely tuned to flash array access,
- demonstrate speed improvement on query of more than an order of magnitude over conventional key-value store organizations.

II. LMAT Search and Classification

LMAT identifies organisms in a metagenomic sample by matching k-mers in the reads in the sample with k-mers in a searchable database. If a set of k-mers from a read match k-mers from a particular reference sequence, then there is some likelihood that the read may come from the species or strain of the reference. On the other hand, a read may contain k-mers that map to a diverse group of organisms that have a common ancestor in the biological taxonomy at the level of genus, family, or higher. The LMAT classification algorithm is rank-flexible, attempting to classify a read at the lowest level possible of the taxonomy tree. LMAT classification uses threshold values for the classifier, so there must be a strong enough signal for a particular match to make a "call" for a particular taxonomy. For rank flexibility, for instance, if there is a match for a particular strain (lowest rank) but it is not enough to pass the threshold, it might be that there is enough for a species (next rank up the NCBI taxonomy hierarchy) level match, and so on. LMAT uses a searchable index of k-mers that map to constituent taxonomic identifiers for the organisms and higher common ranks when appropriate.

Other approaches that map reads to reference sequences using k-mers rely on a small subset of k-mers to represent the database in order to minimize the challenges of searching a large reference database [4], [5]. A unique feature of LMAT is its use of all k-mers in the reference database to improve detection sensitivity. Since the number of 20-mers in the current reference database is approaching 25 billion and will continue to grow as new organisms are sequenced, new approaches are needed to efficiently manage and retrieve information from the reference database. One key challenge is to identify, store and manage the billions of k-mers in a way that avoids duplicates, allows for adding genomes, and can be written out to store for use in subsequent workload stages. The second challenge is choosing an index structure that works well on flash storage. For LMAT to achieve useful performance, it must make use of multicore architectures, in which the application processes many input reads concurrently for classification. Flash has orders of magnitude higher latency than DRAM, requiring the design of latency-tolerant algorithms and data structures for k-mer search. Since our classification algorithm has compute intensive phases in addition to the data-intensive database lookups, the goal is to gain latency-hiding between the two parts of the query application process by running a large workload of query processes in parallel, ideally resulting in higher LMAT application throughput.

This paper focuses on speed performance and scalability aspects of the LMAT workflow. Discussion of accuracy and in-depth comparison with other techniques for metagenomic sample analysis can be found in [1]. We include a new speed performance comparison with two alignment tools using an updated reference database in Section IV-E. The choice of k was selected to store k-length sequences that uniquely identify a small number of reference genomes to improve runtime speed while maintaining accuracy with inexact matching between the query reads and the reference genomes. Longer values of k generate more k-length sequences that map to fewer reference sequences and limit the scope of the search. With longer values of k there can be fewer differences between the read and the matched reference genome. As k decreases, each k-length sequence is associated with more genomes increasing the difficulty to efficiently identify the best reference match. In developing LMAT, we experimented with several values of k in the range $17 \ldots 21$ to maximize classification accuracy and performance. In this paper we report performance primarily with 20-mers, which were previously found [1] to support accurate classification.

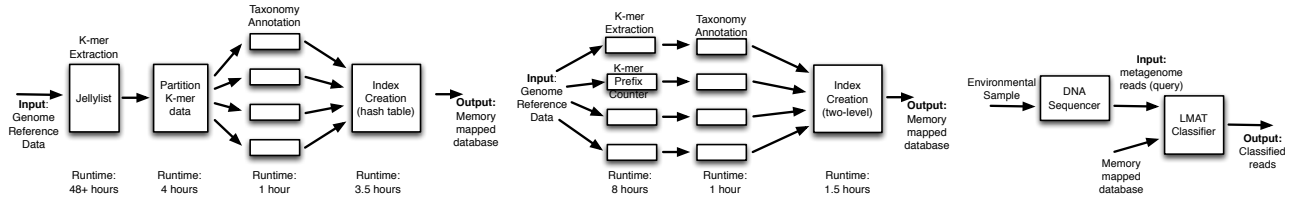
The following sections discuss the workflow to create the index, the optimizations to several workflow components, and the evaluation of those optimizations.

III. Database preparation

The LMAT classifier depends on a searchable k-mer database. In our implementation, the k-mer database is a key-value store in which the key is a 64-bit binary encoded form of the k-mer and the value is the list of genomes and/or higher level taxonomy classification (strain, species, genus, family, ...) that contain the k-mer, called the taxonomy identifiers (tax IDs). Figures 1(a) and 1(b) contrast two workflows we have used for generating the database. The first phase is k-mer extraction, performed by the program Jellylist (see Section III-A) in Figure 1(a) and K-mer Prefix Counter in Figure 1(b). The k-mer extraction phase takes a preprocessed genome reference database and enumerates all the canonical k-mers. In canonical form, a k-mer and its reverse complement[1] are considered equivalent, and the k-mer with a lower value internal representation (64-bit ID) is used.

All LMAT applications that convert Fasta ascii to integer-encoded k-mers use the following procedure. Each base is encoded as the following two bits: A=00, C=01, G=10,T=11. We read the first base, encode using the above formulae to our placeholder. For the subsequent base, we shift left

[1]The reverse complement is the reverse of the k-length sequence with each nucleic acid base replaced by its complement: A $\leftrightarrow$ T and C $\leftrightarrow$ G.



(a) Original Ingest (b) Optimized Ingest (c) Query

Figure 1. LMAT ingest and query workflows. The original workflow has four stages. The optimized LMAT workflow improves the runtimes of the k-mer extraction (K-mer Prefix Counter vs. Jellylist), database ingest stages, and removes the partitioning stage. Example times assume 19 GB reference sequence input.

the placeholder and bitwise OR with the encoded base, performed k times. For subsequent k-mers within a sequence or read, we can shift and mask by $2 * k$ bits rather than re-encode the entire k-mer.

Then, each extracted k-mer is annotated with a list of tax IDs that contain the k-mer. Tax IDs are stored as 32-bit integers. The two workflows accomplish this task in different ways. In the first workflow (Figure 1(a)), the output from Jellylist is so large that it must be partitioned for the Taxonomy Annotation phase. In the optimized workflow of Figure 1(b), the output of k-mer extraction is already partitioned. The partitioned k-mer sets are annotated with the taxonomy lists. Finally, the annotated k-mers are indexed into a searchable form. The final phase of both workflows reduces the many partitions into one single output database.

The workflow makes use of large and fast scratch storage for the intermediate storage of all the files generated by the first two phases. These files can be stored on any conventional file system, though use of a parallel file system is suitable for the workflow stages running parallel tasks. The k-mer extraction phase produces output file(s) roughly 10 times larger than the original input files. An equivalent amount of temporary storage is needed for the partitions, given the workflow in Figure 1(a) where we have a single output from the k-mer extraction phase. Output from the taxonomy annotation phase takes 2.15 times more storage than its input (for $k = 20$). For example, a 19 GB input set, produces 190 GB of files with the extracted k-mers and genome identifiers. Partitioning requires an additional 190 GB. The data size after taxonomy annotation is 400 GB.

A. K-mer extraction

The software program Jellylist was initially selected to do the k-mer extraction phase since it was an existing k-mer analysis tool with an early software variant, Jellyfish [6], which was previously shown to enable fast multithreaded k-mer counting. (The Jellylist code is not published but freely available from the Jellyfish authors.) Jellylist uses lock-free hashing techniques and creates lists of k-mers and genome identifiers using an in-memory data structure. Thus, the application requires a large shared-memory to process a set of genome reference sequences. Jellylist includes the feature of storing positions for each k-mer within the containing genomes. Although we chose not to use this feature, as the positions are not required for LMAT, the feature remains a key part of the application and may impact its performance even when not in use. Unfortunately, Jellylist processing resulted in excessively long runtimes (including the final writing of k-mers to storage). For instance, to extract 20-mers from a 19 GB input reference set took close to 48 hours on a single server with 1 TB DRAM. The addition of the human genome to that input set, 3 additional GiB, added unforeseen complexity, and the run could not complete within our seven day allocation on the server.

The first approach we considered to addressing the problem of intractable Jellylist runtimes was to partition the input reference set of sequences. This approach is problematic as well. Ultimately the partitions need to be merged. The challenge in the merge is accounting for the k-mers extracted from several genomes in multiple partitions. For these repeated k-mers, the merge step must combine the lists of constituent genome identifiers. Moreover, all partitions require sorting in order to facilitate efficient merging. We went through this process to merge the set of k-mers extracted from a human genome with those extracted from our microbial genome data set. The process was problematic due to operator error and machine failure and required days of machine time due to out-of-core sort and merge steps. These issues made the merge process not realistic for a batch scheduling environment, and therefore, it was eventually eliminated from the pipeline.

We instead designed an alternative approach to partitioning. Rather than partitioning the input sequences, we partition the k-mers as they are extracted. We call our application kmerPrefixCounter. Given that nucleic acids are comprised of four bases, every k-mer has a "prefix" of n bases, so there are 4^n possible prefixes. We can pick the desired number of partitions. Based on picking a prefix length n we can determine the number of partitions that fits our resources. For instance if we want to have four partitions, $n = 1$, each partition extracts k-mers of a particular initial base (A, C, G, T). Each process works independently and no inter-process communication is required. For example, with sixteen partitions, the prefixes are AA, AC ... TT. The prefix identifiers are assigned to processes via a command-line

argument. All partitions' processes see the same sequence information, but based on the prefix, each process chooses whether or not to include a particular extracted k-mer within its own k-mer set. The output files do not require additional sorting and merging because each partition has a non-overlapping set of k-mers.

Our implementation of kmerPrefixCounter uses C++ STL data structures, the std::map for recording the k-mers (one per partition) and the gnu hash_set for managing each list of genome identifiers per k-mer. The execution of kmerPrefixCounter creates a set of output files with each file being already sorted by k-mer key, as enabled by the use of the std::map data structure. The sorted output becomes important because it is needed to create the two-level index, which is optimized for sorted input. Sorted output is also useful because it facilitates merges of sorted sequences if the user chooses to add extracted k-mers from an additional set of organisms at a later date.

Each partition is started as its own single-threaded process with the input reference sequence file, the prefix value n, and the partition ID to identify which prefix the process handles. A drawback of this method is that all partitions must read the entire reference sequence file and parse individual k-mers, but the creation of many independent partitions allows for flexibility in using job schedulers within cluster computing systems.

B. Taxonomic Annotation

The output from k-mer extractions contains k-mers, each with a list of genome identifiers. The genome identifiers refer to individual sequences rather than particular organisms. In the taxonomic annotation process, we map the genomes to an NCBI taxonomy identifier for the organism. At the level of the genome, this is typically a species or strain identification. K-mers that map to more than one organism have a common taxonomic ancestor within the NCBI taxonomy tree. The second part of the taxonomy annotation computes the lowest common ancestor (LCA). The procedure is embarrassingly parallel, as it is performed on a per-k-mer basis. We process each partition of extracted k-mers and their corresponding genome lists as separate processes. Each writes its own output file. Additionally, this stage allows us to collect counts of total k-mers and taxonomy identifiers for use in sizing the final indexed database.

C. Index Creation

A primary requirement to running the LMAT classification application is efficient k-mer lookup. The k-mer database index must be organized for very fast lookup as opposed to insert or delete. For the purpose of classification only, the indices will not be updated, as they serve in a write-once, read-many usage model. Thus, use of an off-the-shelf RDBMS database package for the index is not appropriate,

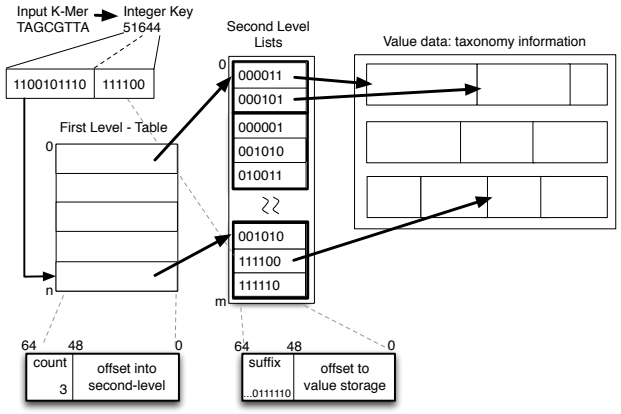


Figure 2. Design of the two-level index. Each k-mer is split. The prefix is the index of the first level. Lists of the suffixes with offsets into the Value Data store are in the second level.

and other available key-value stores incur unnecessary transactional overhead.

Given that our lookup requirement for LMAT does not include range queries, we determined that a hash table would be an appropriate fit. The keys specific to our application are 64-bit integer values: the encoded k-mer. To begin rapid prototyping for the LMAT classifier, we selected the gnu hash_map data structure. We performed some simple performance tests of the data structure in comparison with Google's SparseHash and C++ std::map (a "baseline" for a tree-based map for a point of comparison) and found that gnu hash had the best performance. We manage the storage of the taxonomy identifier lists in a byte array, written to in an append-only "log-like" manner; thus, their storage is compact without additional compression. Offsets into this byte array are stored as the values of the hash_map.

The hash table is built in a persistent heap as a memory-mapped file. The use of memory map presents a convenient programming abstraction for performing out-of core data access without the overhead of application buffer management and mix of standard I/O and in-core data structure access. To enable the index to be persistent after ingest and be used afterward for classification, we configure gnu hash with a custom memory allocator that is backed by storage. Previous work at LLNL, "perm-je" has modified the Jemalloc memory management library [7] to enable memory allocation from an address range memory-mapped to a file residing on a storage device. Jemalloc is a drop-in replacement for regular malloc routines for allocating memory. Our modification to jemalloc allows for an additional step to specify the database filename (jemalloc memory-maps to temporary files). Our "perm-je" library is available as open source and distributed with LMAT at SourceForge.

To meet the goal of running LMAT using flash storage for indexing, it became necessary to consider an alternative index to gnu hash. Our preliminary work with the index showed poor query performance when reading the data

structure from flash. In designing such an index, we focus on the goals of improving page reuse and locality of access.

We have collected some statistics on our use of the gnu hash table. The maximum bucket count hard coded into gnu hash is 4,294,967,291. Thus, on average, regardless of hash function, the chain length for each lookup is 2.15 for the 9.21 billion items of a 19 GB reference sequence set. The chaining becomes problematic when the hash table resides on flash storage: each pointer dereference to traverse the linked lists potentially forces additional page faults. For this reason, we focus on improving locality when handling collisions.

Two-level index Our alternative data structure to the gnu hash_map is a "two-level" index. For this approach, we split the 64-bit k-mer into a prefix and suffix. (Note that this "prefix" is different from the kmerPrefixCounter prefix used to partition the reference database.) The first level maps every prefix into an array of the suffixes stored in the second level. This structure resembles a hash table in that the first level is similar to the hash table structure; in our case the "hash function" returns the prefix of the key, which is the index into the first-level array. The second level manages the collisions that occur — as we expect many matching prefixes — by maintaining array-based lists containing the suffix. These lists are maintained in sorted order so they can be quickly binary searched during lookup operations. It is unnecessary to store the entire key (the encoded k-mer) because once the prefix is used within the first level lookup, it is no longer needed. In contrast, chaining within a hash table stores the entire key.

Figure 2 illustrates the structure of the index, where n is the length of the first-level array (corresponding to selected k-mer prefix size), and m is the length of the second-level array of lists. Note that m is equal to the total number of k-mers in the database. In our implementation of the index, both the first and second levels are 64-bit integer arrays. The boxes at the bottom of each table show the fields within each 64-bit value from each of the tables. For each 64-bit value in the first-level array, the 16 most-significant bits store the count of items in the corresponding second-level list of suffixes for k-mers that share that common prefix (the index into the first-level array). The remaining least-significant 48 bits store the offset into the second level, pointing to the sorted list of k-mer suffixes. In the second-level array, the k-mer suffix is stored in the most-significant 16 bits of the 64-bit value. The remaining 48 bits of the second-level array value are used to store the offset to find the taxonomy identifier lists in the value data array, equivalent to the gnu hash_map value field.

Note that this data structure is specific for integer keys. While we apply it in this work to integer-encoded k-mers, it can potentially be applied to other integer-specific workloads. It is not suitable for general purpose key-value storage that typically performs a hash-function calculation on variable-length strings of ascii characters.

An important determination needed to use the index is how to split the input k-mer between the first and second level. The split parameter determines both the size of the first level table and the maximum length of each second level list. The average length of each list and distribution of such lengths depends on the particular integer key data indexed. Our goals in considering specific split parameter are to: (1) try to keep the second level lists on a single page of memory so as to incur fewer page faults; (2) keep the first level table size small enough to remain cached in DRAM. An additional consideration (implementation specific) is that the 16-bit count field limits second level list lengths to integers of that size. Thus, we must balance the tradeoff of optimizing for each of these goals.

We select split factors where the prefix sizes range from 24 to 31 bits with suffixes ranging from 16 down to 9 bits respectively, each pair a total of 40 bits for the k-mer length of 20 base pairs. At one extreme of these parameters, we have a 128 MB table for 2^{24} possible k-mer prefixes with maximum second level list lengths of $2^{16} - 1$ spanning up to 128 4K pages for the longest lists. Note that this configuration fits the constraint of the largest possible list length based on a 16-bit integer count value. At the other extreme of the range, a 9-bit suffix guarantees that the maximum list fits on a 4K page, given 512 64-bit values. However, the first level is 16 GB (2^{31} prefixes) using this configuration, which may not remain fully in the mmap handler's buffer cache when memory is limited. Our evaluation considers LMAT classification performance given this range of settings.

The ingest procedure for the two-level index assumes that input data is sorted by the k-mer key. The output of the kmerPrefixCounter code produces sorted k-mers, which facilitates this process in our workflow. This enables the second level list values to be added in sorted order, so no additional sorting is needed. The ingest procedure is very straightforward. The ingest code initializes the first-level array to zero values. The second-level is written in a log-like fashion, so the procedure must maintain and increment the offset pointing to the last value written. For each k-mer in the input set, the procedure first splits the k-mer into the prefix and suffix. For the first-level, it checks if the value in the array for that prefix has been written. If not, it writes the current second-level last value offset to the lower 48 bits of the value and sets the count field to an initial value of 1 for that prefix. If the location in the array has been written, the count is incremented, but the offset into the second level for that particular prefix has already been set. For each input k-mer, the suffix and offset to taxonomy storage are written to the second-level array at that last-value offset, maintaining the sorted order of suffixes for common prefixes as they are written. To additionally contrast the two-level index with gnu hash (considering ingest), despite the disadvantages of gnu

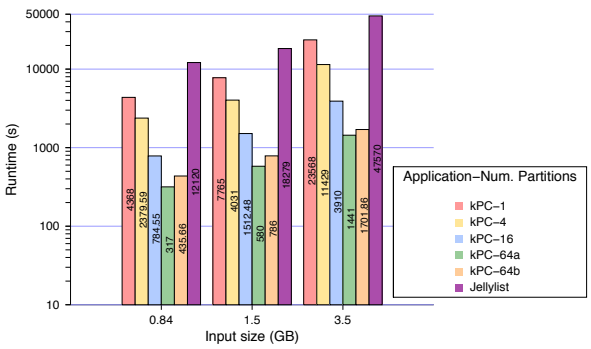


Figure 3. Performance of methods for k-mer extraction using 1 TB node(s). kpC-64a (kmerPrefixCounter, 64 concurrent partitions) ran on two nodes (32 processes on each) and kPC-64b ran on a single node with hyper-threading enabled (80 hardware threads). All other series ran on single node with hyper-threading disabled. Jellylist set to use 32 threads. The sizes on the x-axis represent collections with 500, 1000 and 2000 individual genome sequences respectively.

Method	Partitions	Node-hours
1 TB 16 jobs	64	23.5
ScaleMP 4 TB 40 jobs	64	130
Cluster 32 GB 512 nodes 2 jobs	1024	950

Table I
KMERPREFIXCOUNTER RUNTIME ON THREE COMPUTE PLATFORMS. THE CLUSTER PLATFORM REQUIRED 1024 PARTITIONS, WHICH WERE RUN ON A MAXIMUM OF 512 NODES (AS ALLOCATED BY CLUSTER JOB SCHEDULER). THE FIRST COLUMN INDICATES THE MEMORY PER NODE AND NUMBER OF JOBS PER NODE.

hash that we observe, an advantage of the gnu hash index is that it accepts unsorted k-mer input to build the index.

IV. EVALUATION

Our evaluation of the LMAT workflow considers three aspects. First, we examine the performance of our cluster k-mer extraction application in comparison to the Jellylist application. Second we profile the build of the index, comparing the performance of gnu hash_map with the two-level approach. Third, we compare the two index techniques when used for query in LMAT classification and examine the performance of the two-level index under varying amounts of memory and configurations.

Single node measurements are run on a 4 socket 2 GHz Intel E7 4850 CPU with 1TB of memory. Index creation and database query experiments use this hardware. K-mer extraction is evaluated on large memory nodes and on conventional cluster nodes. The latter are Infiniband-connected 2.6 GHz Intel E5-2670 CPUs with 32 GB of DRAM each. All query experiments use PCIe attached flash memory. We use a software RAID with 2 FusionIO 1.2 TB ioDrive cards. Our experiments run in a standard supercomputing center environment in which input and output files are read from or written to a 1.5 PB Lustre file system.

A. K-mer extraction

Figure IV plots the runtime for k-mer extraction comparing Jellylist with the kmerPrefixCounter (labeled kPC). We consider three input sizes (X-axis) and for kmerPrefixCounter, five configurations from 1 to 64 partitions (powers of 4). The three input sizes represent collections with 500, 1000 and 2000 individual genome sequences respectively. The number following the label indicates the number of partitions run concurrently. For 64 partitions, we consider two variations: (kPC-64a) use of two nodes with all processes running on physical cores or (kPc-64b) use of a single node with hyper-threading enabled; with hyper-threading the 64 processes run concurrently on the 80 available hardware threads (with 40 underlying physical cores).

The results show that using the kPc-64b configuration (single node with hyper-threading) incurs an 18% performance penalty (for the largest workload shown) over kPc-64a, which uses 64 CPUs (on two nodes) without hyper-threading. In other words, using two CPUs without hyper-threading reduces runtime by only 18% compared to a single hyper-threading-enabled CPU. To run separate kmerPrefiCounter processes in parallel we use the "gnu parallel" tool. In all, we observe a range of speedups of 4.17× for kmerPrefixCounter over Jellylist when using four processes and 27× when using 64 processes and hyper-threading enabled. Experience with Jellylist has shown that hyper-threading does not improve the application's performance.

We compare the use of a cluster to extract k-mers from a larger (55 GiB) set of sequence data with (1) a ScaleMP (version 5.1) 4 1TB node configuration (3.8TB in vSMP) and (2) running the same workload split into groups of partitions on a single 1 TB node. For the single node, we had to limit the number of partitions to 16 running concurrently in order to not exceed the 1 TB memory limit on the node. For the cluster run, we require 1024 partitions so the largest does not exceed the 32 GiB available DRAM per node group. However, we pair each job with a complementary job so we may run on 512 nodes, and each pair of jobs will not exceed the limit. The higher number partitions use much less memory than lower numbers due to the canonicalization of k-mers, which allows us to find the pairings. For this group, we report total node-hours (wall clock-time for each job). For the 1 TB node run each group is 16 partitions out of 64, run in four batches, and we report total wall-clock time, as we also do for the single-job ScaleMP run. Additionally for these runs, we attempt load balancing by interleaving partitions we expect to produce larger, average and smaller quantities of system DRAM for the 40 out 64 or 16 out of 64 concurrent processes on the ScaleMP and 1 TB runs respectively.

If 512 nodes were simultaneously available, the cluster job could complete in slightly under two hours but as shown in Table I we require 950 node-hours to complete. We also

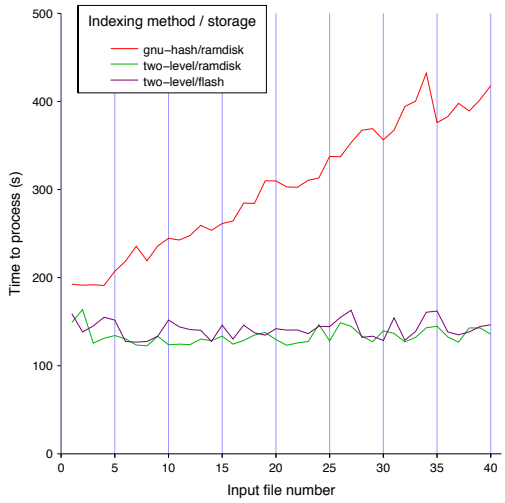


Figure 4. Comparison of indexing approaches measuring ingest timings. For the two-level index we consider performance of both ramdisk and flash. Gnu hash performance on flash was too poor for consideration. Each point on the x-axis represents a roughly equal number of k-mers along the x-axis.

observe that it takes 5.5× longer to complete the entire batch of 64 partitions on the 40-core 4 TB scaleMP system over the 1 TB system where the task must run as four batches of 16 concurrent processes. On the ScaleMP system, the processes use close to 3 TB of the available DRAM.

B. Index Creation

We discuss the performance of the ingest process that builds the index from files containing k-mers and taxonomy information, comparing use of gnu hash with the two-level index. Figure 4 shows timings of the processing of input files on a 1 TB node. For this experiment we use input data derived from 22 GB data set, which includes microbial organisms and a human genome. Each partition (x-axis) is a single file, containing roughly the same number of k-mers, and they are processed sequentially. The ingest application measures the time to process each input file.

For the gnu hash index, the time to ingest each successive partition increases with elapsed runtime. We attribute the increase to the growth of chaining as the hash table fills up. In contrast, the processing time for each partition file for the two-level varies slightly between successive files, but does not appear to progressively increase. The total times (approximate) for gnu hash and two level hash are respectively 3.5 hours and 1.5 hours. For the two-level index, we have included timings from both a memory mapped ramdisk and memory-mapped flash array with memory available for the application and buffer cache limited to 16 GiB. The overhead for using flash over ramdisk in this case is 6.2%. This small overhead suggests that the working set size for the two-level index ingest is smaller than 16 GiB.

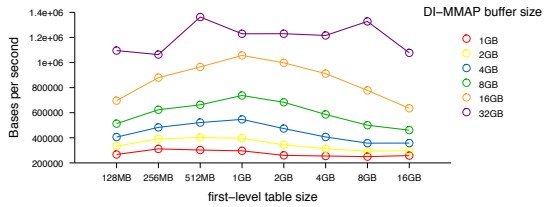
Previous experiments with the gnu hash_map ingest have shown that using flash storage and a limited amount of memory for buffer cache produces considerable increases in time. Therefore, we choose not to show those as well, as they are not comparable and would not complete in a reasonable time for this input data set. For this data set, the total storage used by the gnu hash_map index was 522 GB, including the taxonomy information. In contrast, the two-level index used 320 GB. Based on the the size of the hash table, we have measured 27.3 bytes per k-mer. In contrast, the two-level index uses 8.012 to 9.53 bytes per k-mer for this workload. The two-level index is fixed to 8 bytes per k-mer from the second level plus the memory overhead of the first level, whose size varies depending on the split parameter. Thus, as the number of k-mers increases, that overhead becomes a less significant part of the total storage. Given that the two-level index is more efficient in its use of storage, we expect it to have better page reuse than gnu hash when queried from a flash device.

C. Query Performance

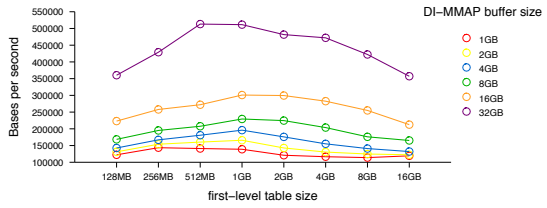
We examine the performance of LMAT classification using flash memory as a store, in which we evaluate the two-level index in two ways. First, we compare the performance of eight index configurations. Each configuration has a different split between prefix and suffix in the k-mer key, resulting in varying first-level table sizes and second-level list lengths. Second, we compare the query performance of the two-level table with gnu hash.

These experiments make use of the DI-MMAP custom memory map handler [3] to access the memory-mapped database files. The goal of DI-MMAP was to address the performance gap in standard linux memory-map for large memory mapped files. The key features of the runtime are a fixed size page buffer, whose size is a configuration parameter; minimal dynamic memory allocation; a simple FIFO buffer replacement policy; and preferential caching for frequently accessed pages. DI-MMAP superior performance to regular system mmap is measured up to 4.88× as shown in [3]. Our experiments consider 6 configurations of the DI-MMAP buffer size from 1 GiB to 32 GiB.

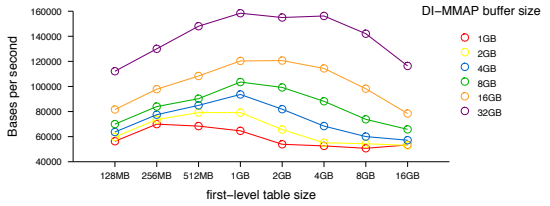
For these experiments we consider three non-synthetic metagenomic samples representing a human microbiome with viruses (SRX022172) (0.5 GiB), a human bacterial community metagenome (ERR011121) (2.5 GiB) and a single species raw read 'metagenome' (DRR000184) (0.6 GB) for which many genomes are represented in our reference database, taken from the NCBI sequence read archive for our input sets (each are abbreviated to the initial three letters). These were selected to measure three different practical experimental conditions and not meant to be an exhaustive list of examples. These experiments run the LMAT software V1.1 adapted to run with the two-level index using a 11.2 billion k-mer index from five kingdoms or domains of microorganisms and the human genome. The results of these experiments measure the input base-pairs per



(a) SRX



(b) DRR



(c) ERR

Figure 5. Index configuration performance selecting various first-level table sizes with varying DI-MMAP buffer sizes. The table sizes are selected based on the range of k-mer split parameters considered in section 3.C. Each subfigure is a different input set taken from a non-synthetic metagenome, each with different redundancy and taxonomic diversity characteristics.

second (correlates with input bytes per second) processed by the LMAT classification application, shown on the y-axis of Figures 5 and 6. All the experimental runs for LMAT classification use a single 40-core node running 160 concurrent threads.

Figure 5 show the performance with the eight different index configurations based on the k-mer prefix-suffix split range we identified in Section 3.3. We use the first-level table sizes based on those splits for the x-axis because those sizes can be compared with the various DI-MMAP buffer sizes that each series represents. Each subfigure uses input data from the SRX, DRR, and ERR data sets, respectively. Note that the performance differences among the three sets varies considerably. This is due to different amounts of redundancy affecting hit rate and taxonomic diversity affecting classification performance. The pattern we observe with most of the curves plotted within the figure is that performance improves from the 128 MB first-level table size to 1 GB size, then declines to the 16 GB size. Under the conditions of the smaller size, the entire table fits comfortably into the DI-MMAP buffer, yet the second level lists are longer on average.

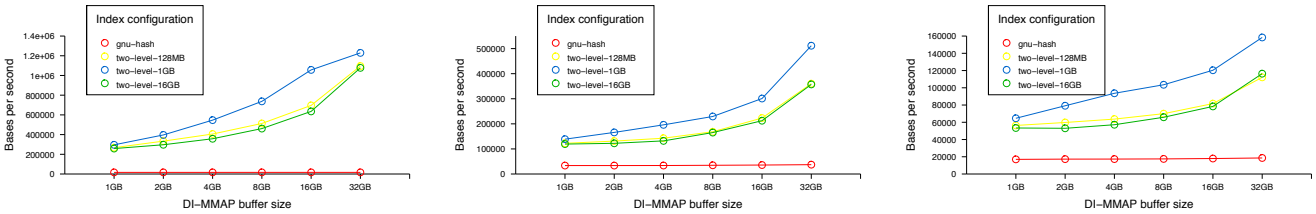
We note several exceptions to the pattern observed above. One difference among the three input sets is that the SRX data set with 32 GB buffer does not have the same single peak pattern. We have observed that the SRX data set has a relatively high number of redundant k-mers. In contrast to the other data sets, the hit rate becomes extremely high for this data set with the 32GB buffer. Additionally, with a 1 GiB buffer, the 1 GiB table size is not the peak performer. This result we expect because the 1 GB table cannot fit entirely in the buffer without its pages being evicted when requests are made for pages from the other components of the index.

Figure 6 shows the performance of LMAT using the two-level index compared with the gnu hash_map index. We include three index configurations for comparison. The x-axis in each plot is the DI-MMAP buffer size. As correlating with the previous data, the buffer size has considerable impact on the performance of the two-level index. However, due to the mostly random-access pattern of the gnu hash_map, increasing the buffer has little impact. Looking at each subfigure for the three input data set and the optimum configuration of the two-level, we observe speedups of $74\times$ for SRX, $13.7\times$ for DRR and $8.4\times$ for ERR over the gnu hash_map index. The varying levels of k-mer redundancy accounts for the large differences in speedups.

D. LMAT vs. Other Methods

We compare the runtime query performance of LMAT with two alignment methods that can cover the full 22 GB of reference sequences: BLASTn and Bowtie2. Other approaches that reduce the size of the reference database leave much of the query input unclassified; thus, those methods are outside the scope of this comparison. While neither program is a metagenomic classifier, their output is the primary input to PhymmBL (BLASTn) and Genometa (Bowtie2) and represents the dominant computational cost. Since alignment tools are used by all competing metagenomic classification tools currently available, the LMAT speed performance goal is to meet or beat the competing alignment runtimes. Bowtie2 and BLASTn use all 40 available cores on the single 1 TB node for processing. Because Bowtie2 limits database sizes to 4 GiB, we partition the DB into 8 groups and run each with 5 threads in a parallel task. BLASTn was run with default settings; Bowtie2 with the "sensitive-local" setting. Since Bowtie2 and BLASTn run their computation in main memory, we include LMAT runtime performance using ramdisk for index storage (see Figure 7). LMAT measurements use the two-level index.

LMAT on ramdisk outperforms Bowtie2 by close to 50% to $3.7\times$. For the SRX workload only, the LMAT running on flash is 75% faster than Bowtie2 (in memory), but more than $15\times$ slower on the ERR data set. The ERR has lower taxonomic diversity but less redundancy; thus, when running with the index stored on the ramdisk, it achieves much higher performance with the lower latency. The higher diversity in



(a) SRX (b) DRR (c) ERR

Figure 6. LMAT classification performance comparing the two-level index with gnu hash-map. For the index configurations, the size indicated after two-level gives the first-level table size for the index: the two extremes of the range of values considered and one median value. Each subfigure is a different input set taken from a non-synthetic metagenome, each with different redundancy and taxonomic diversity characteristics.

SRX makes classification (aside from index lookup) take longer, so moving to lower-latency storage does not make as great a difference. BLASTn performs at several orders of magnitude slower than either LMAT configuration. We have previously compared LMAT on species classification accuracy with Genometa and PhymmBL [1]. We observed that LMAT produces no false positive calls and 531 correct out of 541 in the test set, while the other produce up to 526 correct calls with 265 false positive calls (phymmBL) or 504 correct calls with 95 false positive calls (Genometa).

E. Discussion

Ingest Although a prefix-oriented partitioning strategy significantly reduces runtime for concurrent processing, memory requirements appear to increase with the input size. Thus, either larger memory nodes or larger cluster allocations are needed, as we expect to see larger inputs as more organism are sequenced and incorporated into genome reference sets. ScaleMP provides large memory to ameliorate the need for batching on large-memory nodes, but comes at a significant cost in runtime.

Our results suggest that database ingest directly onto flash storage should be a viable alternative to ingest using a large-memory node when using a two-level index for k-mers; it is not viable with hash-based indexing.

Query While there is not one clear choice for how to configure the two-level index for query, we observe that configuring the index within the range of 26 - 28 bits for k-mer prefix is favorable for best query performance, and the extremes of the range are not. within the input set affects the performance of both conventional hash and two-level index; thus, we incur a wide range of speedups when comparing the two strutures with various input sets. LMAT configured with a two-level index using 1 GiB for the first-level is faster than the Bowtie2 read mapping application when using ramdisk storage and competitive in two cases using flash storage. It is orders of magnitude faster than BLASTn in both flash and ramdisk configurations on all tested data sets.

V. Related Work

For metagenomic analysis, two approaches applied toward making search more scalable include reference database size

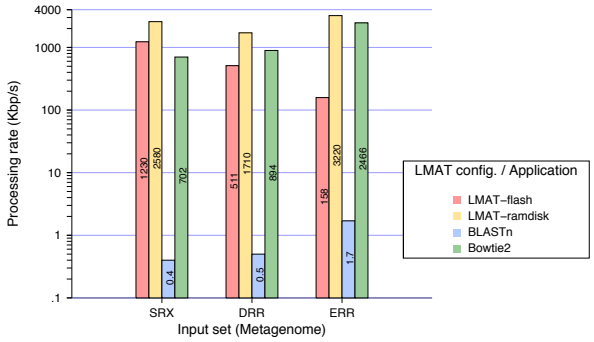


Figure 7. Comparison of LMAT using flash or ramdisk for index storage with BLASTn and Bowtie2 to process three metagenomes.

reduction and faster database search methods. Reference database size reduction is achieved through the use of genetic markers storing only the more informative sequences [8]. Genometa features a user interface to enable simple best match classification using the output from Bowtie2 [9]. MetaPhlAn uses a reduced database set and can use either Bowtie2 or BLAST to map reads to its input database prior to taxonomic classification [4]. Marker based approaches to reduce database size offer efficient summarization of metagenomic contents, but only cover a portion of the query set, which could lead to missing informative reads and prevent recovering complete genomes for more accurate classification [5]. A recent alternative approach reduced sequence redundancy by storing only the genetic differences among reference genomes. This approach was shown to speed up BLAST and BLAT genome database searches [10].

Faster database search methods apply large search seeds and examples include BLAT [11], BWA [12] and other read mapping tools [13], but the analysis of search results can lead to inaccurate findings with some approaches taking a less informative but conservative approach by selecting the lowest common ancestor of multiple matches and others using variants of a best match selection procedure to improve rank specificity of the reported taxonomic label with the increased risk of overly specific calls. Moreover, parameter settings of the search tools can dramatically alter the outcome of the reported label and must be considered carefully

[14].

Other projects have taken approaches to read mapping and scaled them to cluster computing. Note that the cluster approaches alone do not attempt metagenomic classification. mpiBLAST supports the mapping of reads utilizing cluster computing resources. It partitions its reference genomes database by creating what the authors refer to as database fragments and also partitions the input query read into multiple segments. mpiBLAST-PIO features several parallel I/O optimizations, namely, it offloads the formatting and writing of results from a master process to the workers. This feature increases the scalability of mpiBLAST to hundreds of thousands of processors [15].

CloudBurst makes use of the Hadoop framework to scale read mapping to a cluster. CloudBurst uses a k-mer based approach — well-known seed-and-extend algorithms — but unlike LMAT, it does not index every k-mer in the reference database [16]. Crossbow is another Hadoop-based cluster read mapper, which is based on using bowtie as the kernel that maps the input reads against the reference sequences [17].

VI. Conclusion

This paper evaluates scalable techniques to index reference sequence data sets, generating a searchable metagenomic database. We have evaluated the alternative approaches to the database ingest pipeline. This evaluation includes a comparison of k-mer extraction using a conventional cluster, a single large memory node, and a vSMP configuration. To facilitate query, we designed a two-level index data structure uniquely tuned for flash storage, and we demonstrate its speed performance: an improvement of 8-74 times versus the use of a conventional hash table. We additionally compare query speed performance to state-of-the-art read mapping software. The LMAT open source software is in use both at LLNL and in the international bioinformatics community.

VII. Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded in part by Laboratory Directed Research and Development grants 33-ER-2012 and 25-ER-2013, and DOE Office of Science grant KJ0402000-SCW1076. We thank Roger Pearce, Brian van Essen and Shea Gardner for their valuable feedback.

References

[1] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. LLoyd, M. B. Gokhale, and J. E. Allen, "Scalable metagenomic taxonomy classification using a reference genome database," *Bioinformatics*, vol. 29, no. 18, pp. 2253–2260, July 2013.

[2] J. E. Allen, S. Ames, D. Hysom, S. Garnder, and G. S. Lloyd, "Lmat: Efficient taxonomic labeling of very large metagenomic datasets," http://http://sourceforge.net/projects/lmat/, 2013.

[3] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale, "DI-MMAP: A high performance memory-map runtime for data-intensive applications," in *International Workshop on Data-Intensive Scalable Computing Systems (DISCS-2012)*, Nov. 2012.

[4] N. Segata, L. Waldron, A. Ballarini, V. Narasimhan, O. Jousson, and C. Huttenhower, "Metagenomic microbial community profiling using unique clade-specific marker genes," *Nat Meth*, vol. 9, no. 8, pp. 811–814, 2012.

[5] M. H. Mohammed, T. S. Ghosh, N. K. Singh, and S. S. Mande, "SPHINX–an algorithm for taxonomic binning of metagenomic sequences," *Bioinformatics*, vol. 27, no. 1, pp. 22–30, 2011.

[6] G. Marais 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.

[7] J. Evans, "A scalable concurrent malloc(3) implementation for freebsd," in *BSDCan - The Technical BSD Conference*, 2006.

[8] B. Liu, T. Gibbons, M. Ghodsi, T. Treangen, and M. Pop, "Accurate and fast estimation of taxonomic profiles from metagenomic shotgun sequences," *BMC Genomics*, vol. 12, no. Suppl 2, p. S4, 2011.

[9] C. F. Davenport, J. Neugebauer, N. Beckmann, B. Friedrich, B. Kameri, S. Kokott, M. Paetow, B. Siekmann, M. Wieding-Drewes, M. Wienhfer, S. Wolf, B. Tmmler, V. Ahlers, and F. Sprengel, "Genometa - a fast and accurate classifier for short metagenomic shotgun reads," *PLoS ONE*, vol. 7, no. 8, p. e41224, 08 2012.

[10] P.-R. Loh, M. Baym, and B. Berger, "Compressive genomics," *Nat Biotech*, vol. 30, no. 7, pp. 627–630, 2012.

[11] V. K. Sharma, N. Kumar, T. Prakash, and T. D. Taylor, "Fast and accurate taxonomic assignments of metagenomic sequences using metabin," *PLoS ONE*, vol. 7, no. 4, p. e34030, 04 2012.

[12] C. F. Davenport, J. Neugebauer, N. Beckmann, B. Friedrich, B. Kameri, S. Kokott, M. Paetow, B. Siekmann, M. Wieding-Drewes, M. Wienhfer, S. Wolf, B. Tmmler, V. Ahlers, and F. Sprengel, "Genometa - a fast and accurate classifier for short metagenomic shotgun reads," *PLoS ONE*, vol. 7, no. 8, p. e41224, 08 2012.

[13] J. Martin, S. Sykes, S. Young, K. Kota, R. Sanka, N. Sheth, J. Orvis, E. Sodergren, Z. Wang, G. M. Weinstock, and M. Mitreva, "Optimizing read mapping to reference genomes to determine composition and species prevalence in microbial communities," *PLoS ONE*, vol. 7, no. 6, p. e36427, 06 2012.

[14] S. S. Mande, M. H. Mohammed, and T. S. Ghosh, "Classification of metagenomic sequences: methods and challenges," *Briefings in Bioinformatics*, 2012.

[15] A. E. Darling, L. Carey, and W. chun Feng, "The design, implementation, and evaluation of mpiblast," in *In Proceedings of ClusterWorld 2003*, 2003.

[16] M. C. Schatz, "Cloudburst: highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[17] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, "Searching for SNPs with cloud computing." *Genome biology*, vol. 10, no. 11, pp. R134+, Nov. 2009.