Parallel Multi-set K-mer Counting on OpenMP for Hierarchical Taxonomy Classification

Shreyas Ramesh
M.S Graduate Student, Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, U.S.A
shreyas2@vt.edu

***Abstract*— Taxonomy classification algorithms today face the "open world problem" of an ever increasing number of genomes to classify, and rely heavily on exact match based approaches to identify organism type. These exact match approaches typically involve building an indexed sequence data structure from k-mers, which are individual overlapping k-length DNA substrings, to discriminate between the many taxonomic divisions. This work reviews and implements the k-mer counting & partition steps of the parallel multi-set k-mer counting algorithm (MKC) on OpenMP, and compares the results to two state-of-the-art tools – SIMKA and Jellyfish.**

Results show that both nested, and perfectly load balanced k-mer counting strategies show good strong scaling characteristics. OpenMP (Compute Only) achieves the highest speedup of 14.25. OpenMP (Nested) achieves a maximum speedup of 6.02 while OpenMP (Load Balanced) achieves a speedup of only 3.56, suggesting the use of 2-level nested parallelism over a perfectly load balanced strategy. SIMKA, the current state-of-the-art multi-set k-mer counting tool achieves a maximum speedup of 9.22 while Jellyfish, the fastest single file k-mer counter achieves an overall speedup of 5.65.

Keywords—comparative metagenomics; k-mer; k-mer counting; pairwise distances; taxonomy classification; exact-match

I. Introduction

The national science foundation has named tree of life genetics as one of the grand challenge problems of the decade. Taxonomy classification algorithms today use exact matching and alignment based approaches to identify organism type. This typically involves building indexed data structures of individual words within DNA strings called k-mers. The size of 'k' is an important parameter in these methods -- using longer k-mers allows for fine-grain classification, but at the cost of ignoring a great deal of information about the evolutionary relatedness provided by other smaller k-mer sizes [1].

Figure 1 and Figure 2 show that varying size of the k-mer can have a large influence on the sensitivity and memory footprint of the resulting exact match database / model. It is thus important to choose a good value of k-mer size that provides high sensitivity with reasonable memory efficiency at every node in the taxonomic tree.

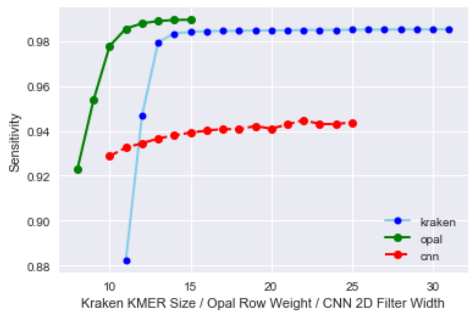


Figure 1. The Sensitivity of three taxonomy classification methods are compared while varying the k-mer size parameter. A saturation point exists for Kraken at a k-mer size of13, beyond which very little improvement in sensitivity is seen up to k-mer size 31. Dataset: Species under Genus TaxID: 1301 "Streptococcus".

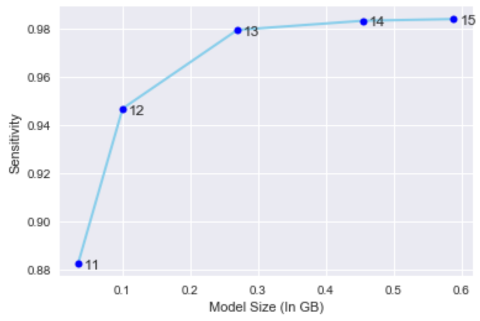


Figure 2. Graph of Sensitivity Vs Model Size reveals diminishing gains in sensitivity while database size increases with increasing k-mer size for all species under TaxID: 1301 "Streptococcus".

II. Objectives

Problem Statement: Given N genomes (also called datasets going forward), denoted as S1, S2, Si,..., SN , compute an N×N distance matrix D where D i,j represents an abundance based Jaccard distance between genome Si and genome Sj.

***Objectives that were realized in this work*:**

Objective 1: Multiset k-mer counting

Each dataset is represented as a set of discriminant features, in our case, k-mer counts. More precisely, a k-mer count matrix KC of size W×N is computed. W is the number of distinct k-mers among all the datasets. KCi,j represents the number of times a k-mer i is present in the dataset Sj. This objective is further divided into two phases, shown concisely in Figure 3.

I. Sorting Count

➢ As the number of distinct k-mers is generally huge, the counts of k-mers in all datasets are obtained in parallel. Following the counting process, each dataset with k-mer counts is divided into P partitions. K-mers within each Partition in each dataset are then lexicographically sorted to obtain P X N partitions.

➢ Conceptually, a partition Pi contains a specific subset of k-mers common to all datasets. The counting of k-mers has a high degree of parallelism potential as each dataset can operated on independently. Additionally, the sorting process can also gain from parallelization as there are P X N different partitions that can be sorted independently.

➢ More than 50% of the MKC algorithm time is spent in k-mer counting alone. Hence, all the experiments focus on improving the k-mer counting process.

➢ Performed an in-depth analysis of serial and parallel execution times for the Sorting Count procedure of the MKC Algorithm using the OpenMP API on an Intel Xeon E5-2630 CPU - 2.40 GHz, 16 physical cores, 32 logical cores, and 2 NUMA nodes.

➢ Compared multi-core run-time for OpenMP-MKC implementation with the multi-core run-time of SIMKA-MKC implementation in the software SIMKA[3].

II. Merging Count

➢ Here, the data partitioning introduced in the previous step is used to generate abundance vectors. The N files associated to a partition Pi , are used as input of a merging process. These files contain k-mer counts sorted in lexicographical order. A Merge-Sort algorithm can thus be efficiently applied to directly generate abundance vectors.

➢ During the initial analysis of multi-set k-mer counting, the k-mer merging operation was relatively computationally inexpensive (less than 20%), and thus no parallel algorithms were developed for the merge operation.

Objectives that were not realized in this work:

Objective 2: Parallel Jaccard distance computation

➢ Based on the k-mer count information provided in the count matrix KC, the Abundance Jaccard Distance matrix D is computed in parallel.

➢ The Distance matrix D which represents the Jaccard distance between all pairs of genomes in the dataset is computed for the range k=10 to k=31. These distance matrices will be used later (Sec. 3 Approach B) to evaluate a reasonable sized k-mer to use for classification at a given taxonomic level, and particularly at a taxonomic node.

➢ The parallel version of the distance matrix computation on OpenMP was not done for two reasons:

▪ Computationally Inexpensive - Although computing all distances (Jensen–Shannon, Jaccard, Bray–Curtis etc) is computationally significant, for this experiment only the Jaccard distance was required, and this was assessed to take less than 10% of the total program time.

• Lack of time – Although an OpenMP analysis of distance estimation was possible, it would have not been a thorough analysis.

➢ The k-mer sizes obtained using the Jaccard distances above were not shown in practice by setting the estimated optimal k-mer size in Kraken for each taxonomic node.

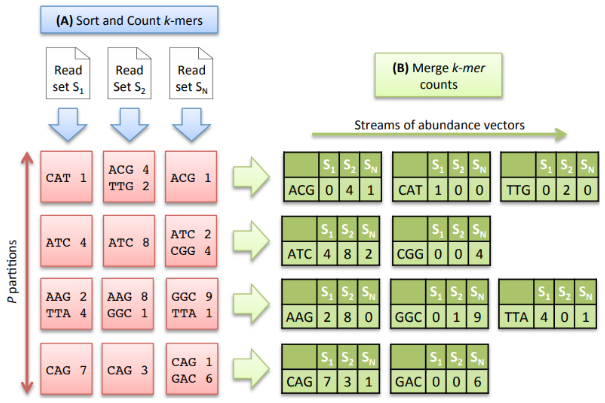


Figure 3: (Taken from [2]) **Multiset k-mer Counting strategy with k = 3**. (A) The sorting counting process, represented by a blue arrow, counts datasets independently. Each process outputs a column of P partitions (red squares) containing sorted k-mer counts. (B) The merging count process, represented by a green arrow, merges a row of N partitions. It outputs abundance vectors, represented in green, to feed the distance computation process.

III. Approach

A. *Objective 1: Estimating Abundance Based Jaccard Distances Between Genome Pairs Using Multiset K-mer Counting*

Inputs:

1) Abundance Vector Stream 'V', of size P representing the P abundance vector streams shown in Figure 3B.

2) Vector of size N containing the number of unique k-mers in each dataset.

Output: A Jaccard Distance Matrix Dist

MKC Algorithm:

Step 1: A matrix, denoted M∩, of dimension N×N is initialized to record the final shared k-mer counts between the crossed terms of each (genome) pair in the datasets.

Step 2: P independent processes are run to compute P partial crossed term matrices, denoted M∩part, in parallel.

Step 3: Each process iterates over its abundance vector stream.

Step 4: For each abundance vector in the vector stream, loop over each possible pair of datasets.

Step 5: The matrix M∩part is updated if the k-mer is shared, meaning that it has positive abundance in both datasets Si and Sj.

Step 6: Since the number of shared k-mers in the matrix is symmetric, limit the computation to the upper triangular part of the matrix M∩part. The current abundance vector is then released.

Step 7: When all streams are done, the algorithm reads each written M∩part and accumulates it to M∩.

Step 8: The last loop computes the Jaccard distance for each pair of datasets and fills the distance matrix.

The amount of abundance vectors streamed by the MKC Algorithm is equal to Ws, which is the total number of distinct k-mers in the N datasets. This algorithm has thus a time complexity of O(Ws×NxN).

B. Objective 2: Computing K-mer Size That Best Separates Sets of Genomes

Searching For Optimal 'K-mer' Size:

Step 1: Compute the Jaccard Distance matrix between all genomes (N) for k-mer size ranging from 10 to k=31.

Step 2: For every taxonomy node (Blue label in Figure 4) within every level in the taxonomy tree (Red label in Figure 4), identify all the genomes that lie under that node.

Step 3: For a given parent taxonomy node, identify all the children taxonomy nodes under it. For e.g. Parent Taxonomy node "Bacilli" has two children Taxonomy nodes "Bacillales" and "Lactobacillales" under it.

Step 4: Construct a list of all child taxon node pairs for the given parent node. In the previous example, there exists only one such child taxon node pair.

Step 5: For each child taxon node pair, construct a non-symmetric matrix where the number of rows of the matrix is the number of genomes within child_taxon_node_1 and the number of columns is the number of genomes within the child_taxon_node_2.

Step 6: For each value of k between 10 and 31 do the following:

Step 6.1: Fill the cells of the matrix with the appropriate Jaccard distances between the genomes pairs.

Step 6.2: Compute the overall "Meta-Jaccard" of the matrix as the mean of all the Jaccard distances in the matrix.

Step 6.3: STOP when the Meta-Jaccard value for the matrix goes below a user-defined threshold (*5%*). The value of k at which search stops can then be taken as a good k-mer size that can distinguish between child_taxon_node_1 and child_taxon_node_2.

Step 7: Since there can exist many values of k-mer size for every parent taxon node (because of one k-mer value per child taxon node pair), selecting the maximum value of k-mer size is the best option, as a larger k-mer size is guaranteed to better distinguish genome pairs than a smaller k-mer size.

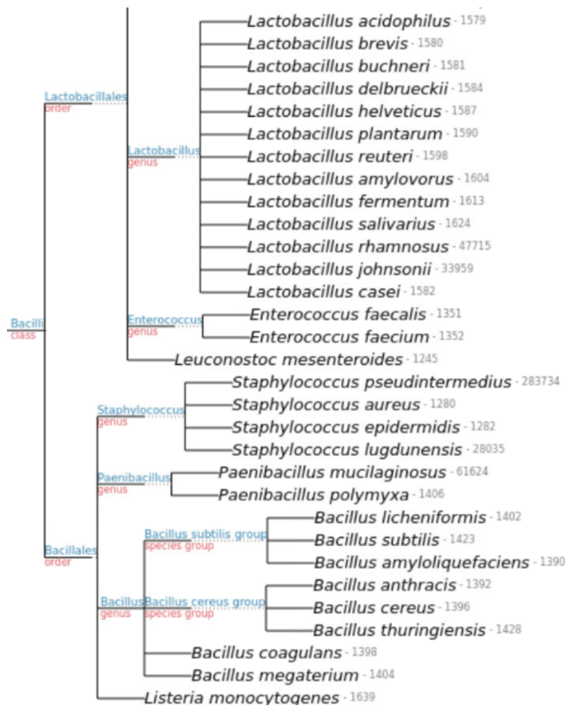

Bacilli
class
Lactobacillales
order
Lactobacillus
genus
Lactobacillus acidophilus - 1579
Lactobacillus brevis - 1580
Lactobacillus buchneri - 1581
Lactobacillus delbrueckii - 1584
Lactobacillus helveticus - 1587
Lactobacillus plantarum - 1590
Lactobacillus reuteri - 1598
Lactobacillus amylovorus - 1604
Lactobacillus fermentum - 1613
Lactobacillus salivarius - 1624
Lactobacillus rhamnosus - 47715
Lactobacillus johnsonii - 33959
Lactobacillus casei - 1582
Enterococcus
genus
Enterococcus faecalis - 1351
Enterococcus faecium - 1352
Leuconostoc mesenteroides - 1245
Bacillales
order
Staphylococcus
genus
Staphylococcus pseudintermedius - 283734
Staphylococcus aureus - 1280
Staphylococcus epidermidis - 1282
Staphylococcus lugdunensis - 28035
Paenibacillus
genus
Paenibacillus mucilaginosus - 61624
Paenibacillus polymyxa - 1406
Bacillus
genus
Bacillus subtilis group
species group
Bacillus licheniformis - 1402
Bacillus subtilis - 1423
Bacillus amyloliquefaciens - 1390
Bacillus cereus group
species group
Bacillus anthracis - 1392
Bacillus cereus - 1396
Bacillus thuringiensis - 1428
Bacillus coagulans - 1398
Bacillus megaterium - 1404
Listeria monocytogenes - 1639

Figure 4. Shows the Taxonomy Nodes (Blue Labels) and Taxonomy Level (Red Label) for all species under Class "Bacilli". Each species (leaf nodes) is further composed of a set of genomes. These genomes are the source for k-mer count computation in the MKC algorithm.

IV. Importance

➢ It is important to set a serial and parallel run-time benchmark on the time taken to compute Jaccard distances between all genome pairs at different values of k-mer size.

➢ Using longer k-mers allows for fine-grain classification, but at the cost of ignoring a great deal of information about the evolutionary relatedness provided by other smaller k-mer sizes. Determining a good k-mer size at every level in the taxonomy hierarchy can directly be put to use as a parameter in tools such as Kraken[4] to enhance sensitivity and provide memory efficient taxonomic classification.

➢ As shown in Sec.3 Approach A and B, the algorithms have a high degree of coarse-grain parallelism potential. With API's such as OpenMP and OpenACC, it is now possible for software developers to implement faster versions of their software with simple informed changes to their code.

V. Related Work

Jellyfish[5], a k-mer counting algorithm implements the lock-free hash table using CAS atomic operations that permit the design of truly concurrent data structures that are fast in serial mode and scale almost linearly with the number of processors used.

As per my knowledge, MetaPalette[1], a k-mer count based metagenomic taxonomy profiling tool is the only other research work that implements the idea of creating a non-symmetric matrix to count the percentage of common k-mers between genome pairs. The authors construct a common k-mer matrix between all genomes in the dataset at only k=30 and k=50, and use the two matrices as palettes (features) to determine if a novel organism queried has common k-mer characteristics as the genomes within the two matrices.

The authors create bloom count filters (using Jellyfish [5]) for each of the "training" genomes, followed by the counting of common k-mers using a C++ program based on the heap data structure. Additional information about the algorithm's implementation was not mentioned in their paper.

SIMKA[3], is the only software that implements the MKC algorithm on k-mers. However, the datasets for the SIMKA software are metagenomic sample reads from the sequencing of ecological samples, and not complete sequenced genomes, which is my use case.

Computing a separating value of 'k' at every taxonomy node within every level of the taxonomy hierarchy has not been looked into previously because it is an expensive operation. It is important to characterize the benefits that parallelism can provide in this problem, as new genomes are being sequenced rapidly today.

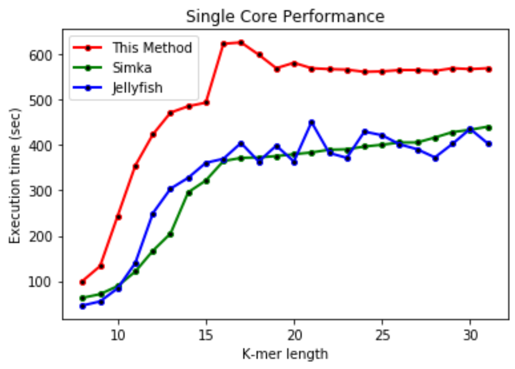
The idea to compute separating value of 'k' is based on the following intuition: "The percentage of common k-mers, decays more slowly as a function of k for closely related organisms than for distantly related ones.".

VI. Dataset, Results and Computing Resources

To test the run-time performance of Serial-MKC, OpenMP-Compute Only, OpenMP-Nested, OpenMP-Load Balanced and SIMKA-MKC, the dataset of all genomes under Taxonomy level "Class" with TaxID: 91061 (Bacilli) is used. This dataset consists of 210 genomes, and four taxonomy levels: Class, Order, Genus, Species.

The five implementations of the MKC algorithm were evaluated on an Intel Xeon E5-2630 CPU with 16 physical cores, 32 logical cores clocked at 2.40 GHz, and 32 GB of main memory.

Result 1: The single core performance of this work's in-memory unstructured map based k-mer counting procedure is about 40% less efficient than SIMKA's k-mer counting procedure and Jellyfish's lock-free hash table based counting.



Three main configurations within OpenMP were experimented with in this work:

- ➢ 2-level nested parallelism
- ➢ Perfect load balancing - Equal work for each thread
- ➢ Combine – Nested + load balanced parallelism

There are three levels of parallelism that can be extracted from the dataset:

- ➢ **Level 1 (FASTA)** – Every FASTA file, of which there are 210 is an independent unit for the k-mer counting procedure.

- ➢ **Level 2 (Contigs)** – A *contig* within a FASTA file represents a small fraction of an organism that was sequenced. The number of contigs within a FASTA file ranges from 1 to around 1000. When only one contig exists within a FASTA file, it means an entire organism was sequenced completely and not in parts.

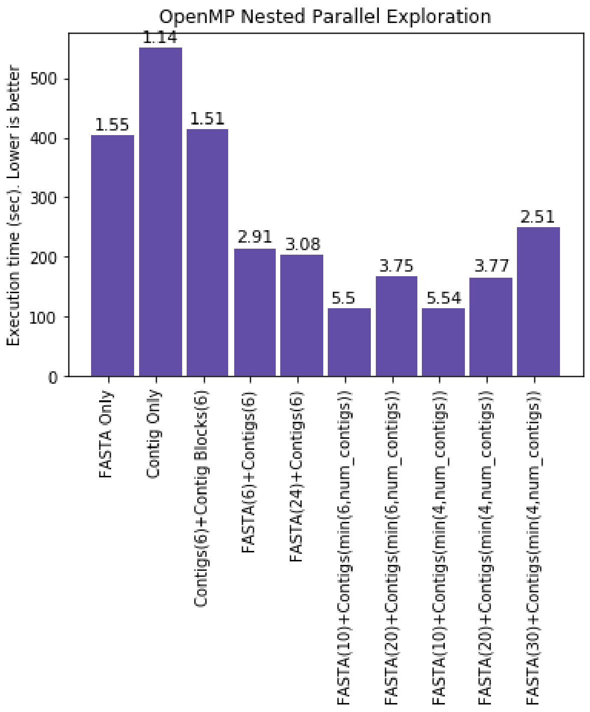
- ➢ **Level 3 (Contig Blocking)** – Since there often exists one contig within a FASTA file, allocating 'n' threads to work on one contig will lead 'n-1' threads to be idle. To solve this problem, every contig is divided into independent contig blocks that can be combined at the end to retrieve the final solution. This process is called contig blocking.

The three configurations of OpenMP were explored and the results are shown below.

Result 2: *OpenMP 2-Level Nested Parallel Exploration*
In the figure shown below, the name before '+' symbol represents the first level of parallelism, and the name after the symbol represents the second level of parallelism. The integer within round braces is the number of threads used at that level. All experiments were performed on k-mer size of 17. Speedups are represented on top of each bar, and are based on the serial execution time of OpenMP (627 sec).

The best configuration found was:

- Level-1: 10 threads reading FASTA files in parallel
- Level-2: Number of threads used is either the number of contigs within the current FASTA file or 4, whichever is smaller.
- Speedup achieved: 5.54



Result 3: *OpenMP Contig Blocking for load balancing*
As mentioned previously, sometimes, a FASTA file may contain just one contig. In these cases, spawning multiple threads (one per contig) will lead to CPU cycle wastage. To combat this, a contig blocking strategy is applied.

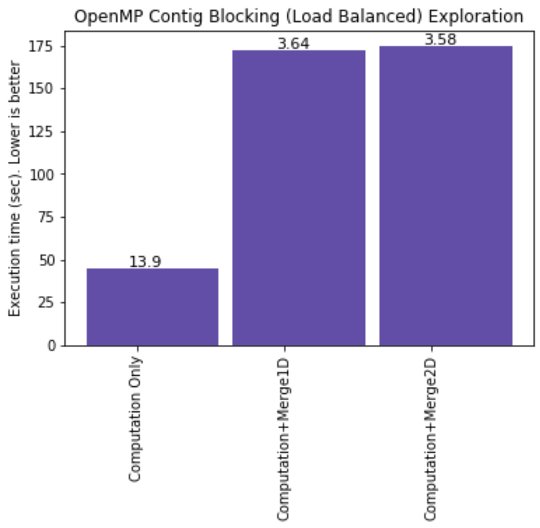
The strategy takes as input the number of threads available to process the contig. Each contig is broken into contig-blocks of at least 1000 nucleotides, and ever contig block overlaps the previous contig block by 'k-1' characters, where 'k' is the k-mer size.

There are two stages within the OpenMP Contig blocking based k-mer counting procedure.

- Computation Stage – This stage is applied on every Contig block. The results from each contig block are stored in a lock unordered hash map.
- Merge Stage – Each local unordered hash map is combined (Reduce) to create a global k-mer count hash map.

To avoid false sharing, the merge stage can store each local unordered hash map in a 2D array. Each unordered hash map is stored in the first element of a cache line.

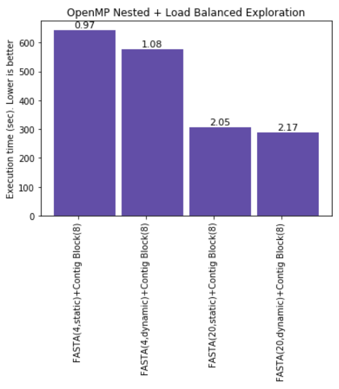
A comparison of the above steps is shown in the image below.



Result 4: *OpenMP Nested + Load Balanced*

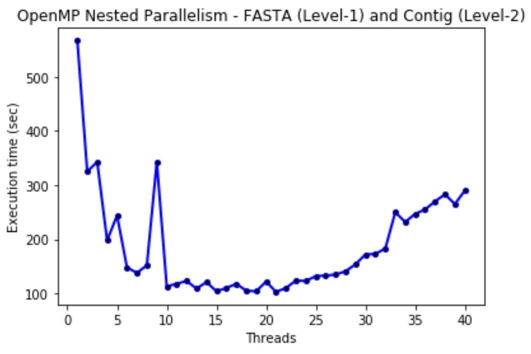
A combination of Level-1 parallelism on FASTA files, and Level-2 parallelism on Contig Blocks was performed to try and get the best of both worlds, however the results were counterintuitive. A maximum speedup of only 2.17 was obtained with the combination.

Both static and dynamic schedules were experimented on, and neither performed well enough.



Result 5: *Strong Scaling of OpenMP Nested Parallelism – FASTA (Level-1) and Contigs (Level-2)*

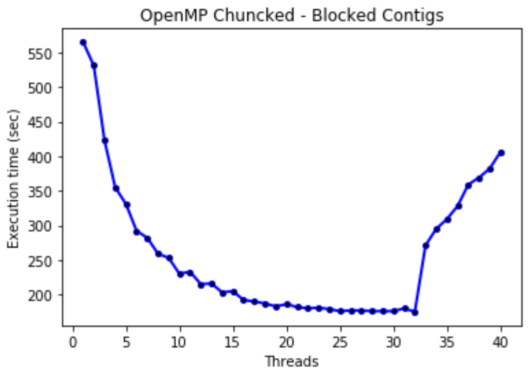
The total number of 17-mers in the 210 genome dataset were counted and the computation times were noted with varying number of Level-1 threads (FASTA files processed concurrently). The number of Level-2 threads was maintained at 4 as this configuration gave the highest speedup in previous experiments.



Result 6: *Strong Scaling of OpenMP Contig Blocking (Load Balancing)*

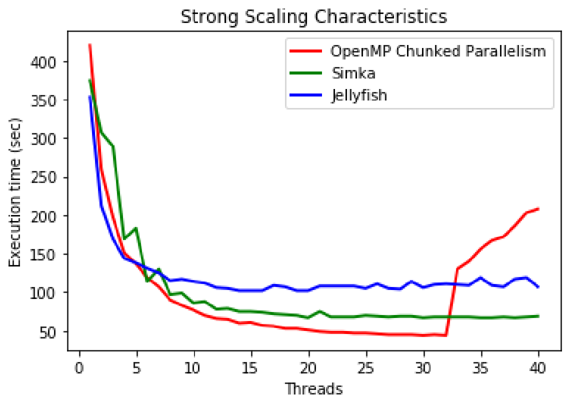
As it can be seen below, OpenMP with the Contig Blocking strategy strong scales with the number of threads of execution. A point of interest in both graphs is the sudden rise in execution time when using more than 32 threads. This is seen because the number of logical cores in the system is 32 and so, using more than 32 threads will lead to frequent context switching between threads in the CPU. The context switching time can be significant – More than 50 seconds added to overall execution time.

Thus, it is required to limit the maximum number of threads of execution using *omp_get_thread_limit*.



Result 7: *Strong Scaling characteristics of OpenMP (Computation Only), SIMKA and Jellyfish*

All three methods achieve an exponential drop in execution time with increasing thread count initially, but the gains quickly saturate at 15 threads of execution, and the gain in performance thereafter is negligible.



Result 8: *K-mer values obtained from Jaccard distances*

As shown in the table above, the constant k-mer value of 31 is not accurate most of the time, and the algorithm proposed in this work can be used to find a better value of k-mer size at a particular node in the taxonomy tree.

As a future work, I propose to determine the sensitivity and memory footprint properties when using k-mer sizes suggested here. This will help validate the algorithm in practice.

TaxID	1578	1350	1385	186826	1301	44249	1279	1386	91061
K-mer value **before** search	31	31	31	31	31	31	31	31	31
K-mer value **after** search	**17**	31	**15**	**24**	31	**8**	31	**22**	**8**

CONTRIBUTIONS AND ACKNOWLEDGEMENT

This was an individual project by Shreyas Ramesh. The author would like to thank the course instructor, Dr. Wu Feng for his advice on prioritizing contents that went into this work. Dr. Andrew Warren for his feedback on the initial proposal and project idea. Abhinav Kumar for verifying the OpenMP Nested and Load Balancing strategies used in this work.

Some sections within this report have been taken directly from [3] to ensure the highest degree of clarity. Every one of these sections has been referenced and contain a notice explicitly mentioning the extracted source.

References

[1] D. Koslicki and D. Falush, “MetaPalette: a k -mer Painting Approach for Metagenomic Taxonomic Profiling and Quantification of Novel Strain Variation,” mSystems, vol. 1, no. 3, pp. e00020-16, Jun. 2016.

[2] G. Benoit et al., “Multiple comparative metagenomics using multiset k -mer counting,” PeerJ Computer Science, vol. 2, p. e94, Nov. 2016.

[3] Benoit G, Peterlongo P, Lavenier D, Lemaitre C. (2015) Simka: fast kmer-based method for estimating the similarity between numerous metagenomic datasets. Hal-Inria

[4] D. E. Wood and S. L. Salzberg, “Kraken: ultrafast metagenomic sequence classification using exact alignments,” Genome Biology, vol. 15, no. 3, p. R46, 2014.

[5] G. Marçais and C. Kingsford, “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers,” Bioinformatics, vol. 27, no. 6, pp. 764–770, Mar. 2011.