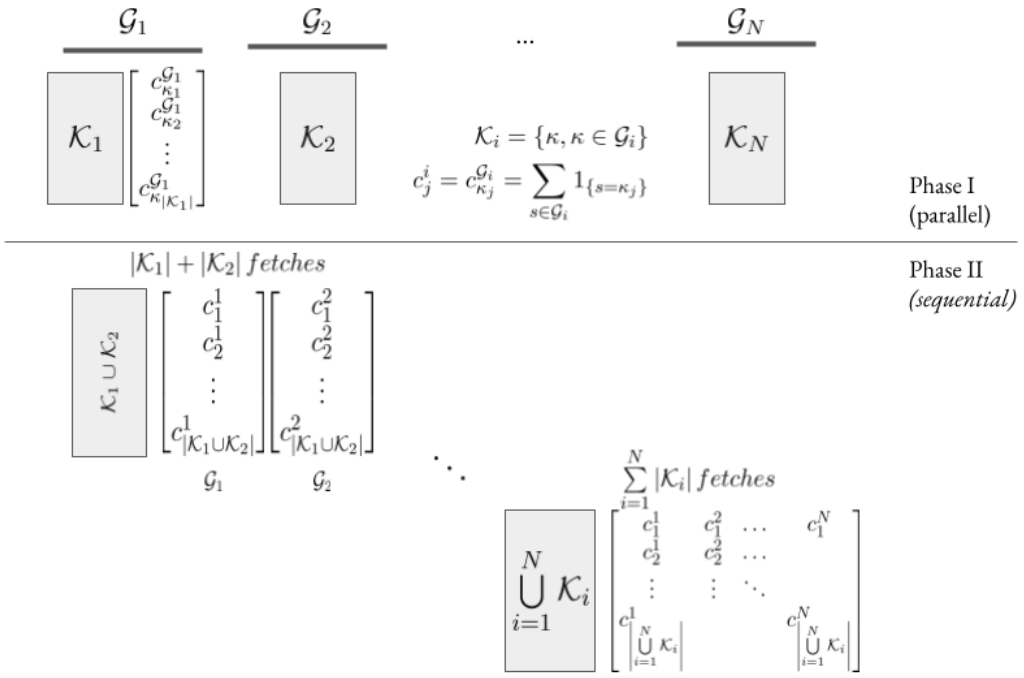

Figure 1: Brute-force computation of *k*-mer frequency matrix across a GWAS panel of N accessions

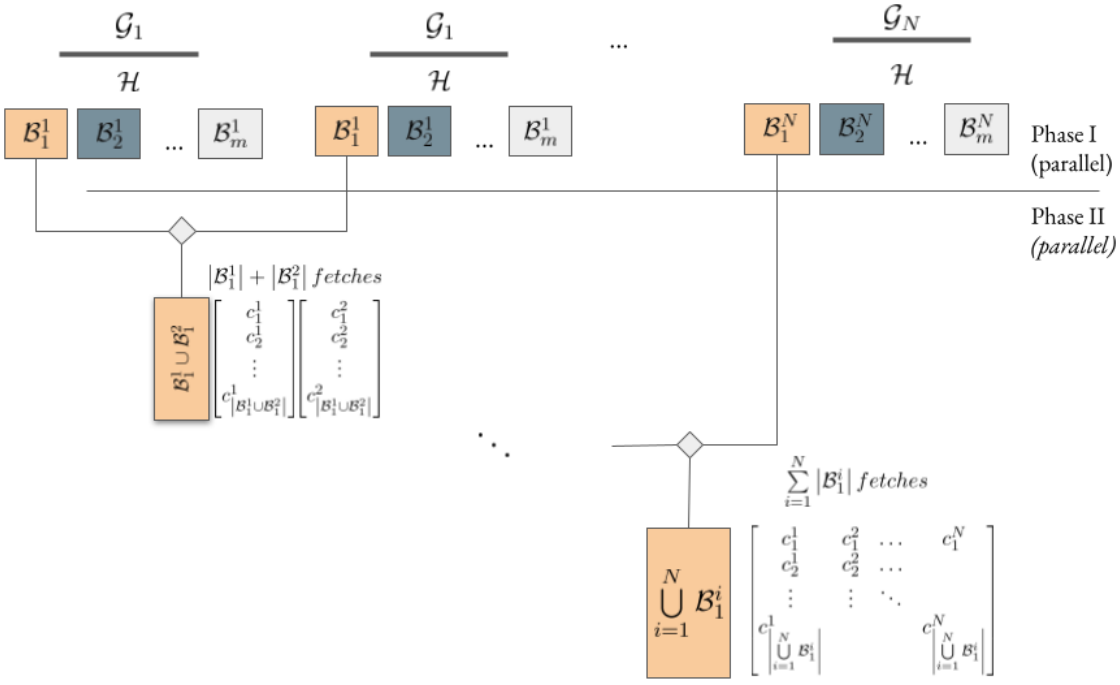

Figure 2: Parallelization of the *k*-mer frequency matrix generation through *k*-mer binning

The process of generating *k*-mer frequency matrices in the context of GWAS analyses is computationally intensive both in terms of memory requirements and compute time. This computational bottleneck becomes more pronounced for larger panels with sizable genomes sequenced at high depth; a tendency that is recently enabled by the increasing affordability of high throughput sequencing. Typically, *k*-mer counts are obtained for each accession independently using a *k*-mer counting software such as Jellyfish [1]. For this first phase, the counts can be computed in parallel, and saved in separate look-up hash tables (dictionaries) which can be subsequently queried as part of the matrix generation phase. Traditionally, merging the hash tables is performed in a brute-force fashion by iterating over the panel sequentially and incrementally inserting the *k*-mer entries and their counts for the corresponding accession, or assigning the count for the *k*-mer in question and the accession in process if the entry already exists. The number of hash table look-ups required for filling the matrix is equivalent to the size sum of all hash tables, and the size of the matrix grows with the panel size and the cardinality of the *k*-mer set. Nowadays, this can amount to hundreds of accessions with billions of *k*-mer entries, making this naive approach impractical to say the least. This computational cost is irreducible, but without parallel computation it becomes restricting. As part of this work, we developed a simple yet efficient scheme for parallelizing this workload, which we include here for the benefit of all.

Let us assume we have a panel of N accessions $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_N\}$, and let $\mathcal{K}_i = \{\kappa, \kappa \in \mathcal{G}_i\}$ be the set of *k*-mers κ pertaining to an accession $\mathcal{G}_i$. Also, let c_j^i denote the frequency of *k*-mer κ_j in accession $\mathcal{G}_i$:

$$c_j^i = c_{\kappa_j}^{\mathcal{G}_i} = \sum_{s \in \mathcal{G}_i} 1_{\{s=\kappa_j\}}$$

Note that the *k*-mers do not assume any particular order and are indexed merely according to their arbitrary insertion order. Merging the first two accessions takes $|\mathcal{K}_1| + |\mathcal{K}_2|$ look-ups/fetches, and results in the $|\mathcal{K}_1 \cup K_2| \times 2$ partial matrix:

$$\begin{bmatrix} c_1^1 \\ c_2^1 \\ \vdots \\ c_{|\mathcal{K}_1 \cup \mathcal{K}_2|}^1 \end{bmatrix} \begin{bmatrix} c_1^2 \\ c_2^2 \\ \vdots \\ c_{|\mathcal{K}_1 \cup \mathcal{K}_2|}^2 \end{bmatrix}$$

Following an incremental process of incorporating one accession into the matrix at a time, the final matrix obtained through this process is:

$$\begin{bmatrix} c_1^1 & c_1^2 & \ldots & c_1^N \\ c_2^1 & c_2^2 & \ldots & \\ \vdots & \vdots & \ddots & \\ c^1_{\left|\bigcup_{i=1}^N \mathcal{K}_i\right|} & & & c^N_{\left|\bigcup_{i=1}^N \mathcal{K}_i\right|} \end{bmatrix}$$

This matrix has a size:

$$\left|\bigcup_{i=1}^{N} \mathcal{K}_i\right| \times N$$

and the number of hash table look-ups required for its construction is:

$$\sum_{i=1}^{N} |\mathcal{K}_i|$$

(see Figure 1)

In order to parallelize this process we introduce a hashing function h which bins the binary encoding of a *k*-mer $\mathcal{E}(\kappa)$ to one of m buckets:

$$h : \mathcal{K} \to \{1, 2, ..., m\}$$
$$\kappa \mapsto h(\kappa) = \mathcal{E}(\kappa) \mod m$$

The hash function introduces a partitioning $\mathcal{H}$ of the *k*-mer space:

$$\mathcal{H} : \mathcal{K} \to \{\mathcal{B}_1, \mathcal{B}_2, , \mathcal{B}_m\}$$
$$\kappa \mapsto \mathcal{H}(\kappa) = \mathcal{B}_{h(\kappa)}$$

This way an accession $\mathcal{G}_i$ is sharded into m bins during the *k*-mer counting phase:

$$\mathcal{G}_i \xrightarrow{\mathcal{H}} \{\mathcal{B}_1^i, \mathcal{B}_2^i, ..., \mathcal{B}_m^i\}$$
$$\mathcal{B}_j^i = \{\kappa, \kappa \in \mathcal{K}_i \wedge h(\kappa) = j\}$$

Each bin $\mathcal{B}_j^i$ is a hash table indexing the counts of *k*-mers mapped to bin j against accession $\mathcal{G}_i$. The obtained bins are mutually exclusive and their union represents a full *k*-mer count index of $\mathcal{G}_i$:

$$\bigcup_{j=1}^{m} \mathcal{B}_j^i = \mathcal{K}_i$$
$$\mathcal{B}_j^i \cap \mathcal{B}_{j'}^i = \phi, \forall j \neq j'$$

Consequently, it follows that:

$$\sum_{j=1}^{m} \left|\mathcal{B}_j^i\right| = |\mathcal{K}_i|$$

More importantly, a *k*-mer is guaranteed to be mapped to the same bin index for all the accessions it belongs to, and the *k*-mer counts can now be looked-up from a pre-defined subset of bins, rendering the merger phase data parallel (see Figure 2):

$$\kappa \in \mathcal{G}_i \cap \mathcal{G}_j \wedge \kappa \in \mathcal{B}_x^i \implies \kappa \in \mathcal{B}_x^j$$

For each bin index we assign a merger job which acts on the corresponding bins in a similar fashion as the brute-force approach described above. However, each job can run independently on much smaller data, requiring much less computation and memory. In particular, the number of hash table look-ups required per job is:

$$\sum_{i=1}^{N} \left|\mathcal{B}_j^i\right|$$

With a matrix size of:

$$\left|\bigcup_{i=1}^{N} \mathcal{B}_j^i\right| \times N$$

But how does that compare with the original approach?
Let $\overline{\mathcal{K}}$ denote the mean *k*-mer index size per accession:

$$\bar{\mathcal{K}} = \frac{1}{N}\sum_{i=1}^{N} |\mathcal{K}_i|$$

In GWAS, accessions usually pertain to closely related strains and are therefore similar in genome size. It follows from this that *k*-mer index size has low variance across accessions (regardless of whether the sequencing depth varies across the panel, which only affects the recorded frequencies):

$$|\mathcal{K}_i| \approx \overline{\mathcal{K}}, \forall i$$

Let $\overline{\mathcal{B}}^i$ denote the mean bin size obtained through $\mathcal{H}$ for accession $\mathcal{G}_i$:

$$\overline{\mathcal{B}}^i = \frac{1}{m}\sum_{j=1}^{m} \left|\mathcal{B}_j^i\right|$$

Because of the random nature of *k*-mer occurrence within a genome, which is for all practical considerations akin to a sampling from a uniform distribution of *k*-mer patterns, the hash function h produces almost equally sized *k*-mer buckets for each accession, which, combined with the previous observations, gives:

$$\left|\mathcal{B}_j^i\right| \approx \bar{\mathcal{B}}^i = \frac{1}{m}\left|\mathcal{K}_i\right| \approx \frac{1}{m}\bar{\mathcal{K}}, \forall (i,j)$$

This is desirable, because it means the partitioning not only allows for the independent merger of same- index bins, but it also ensures that the computational load across the merger jobs has low variance. For example, in a fully parallel setting (where all jobs have immediate access to computational resources without queuing), the time it takes for the merger to finish depends on the worst performing job, and as we have demonstrated, this is not far from the average case, which is m fold faster:

$$\sum_{i=1}^{N} \left|\mathcal{B}_j^i\right| = N\overline{\mathcal{B}}^i \approx \frac{N}{m}\overline{\mathcal{K}} = \frac{1}{m}\sum_{i=1}^{N} |\mathcal{K}_i|$$

Note that m can theoretically be made arbitrarily large bar a few practical considerations, and we have had runs with $m > 1000$, enabling the computation of the matrix within mere minutes as opposed to more than a month using older code. The main consideration when deciding on the level of parallelism is whether the HPC resource can support the required IO throughput without introducing too much overhead. Note that the indexing phase would have to produce mN files with N parallel read/write tasks, while the merger phase would be merging N files per job, with m jobs running in parallel and performing read/write operations. With the advent of HPC storage and the increasing trend of incorporating cost-effective high performance solid-state storage solutions and supporting middleware to cater for the emerging data intensive applications, it is becoming more customary to leverage high throughput IO for achieving high data parallelism and reducing memory requirements per job.

We implemented this procedure as a C++ application (https://github.com/githubcbrc/KGWASMatrix) which comes with two binaries, one for producing a sharded *k*-mer count index, and a second for performing the merger step in order to produce the final matrix. For this study, we elected to perform our analysis based on the presence/absence binary matrix, however, the codes also allow for producing frequency matrices. The code is dockerized and the jobs can be run as singularity instances on HPC resources. The resource includes a brief documentation with details of how to run the codes, the various exposed parameters, and example template scripts for how to deploy this on an HPC cluster or a supercomputer.

References

[1]Guillaume Marçais , Carl Kingsford, A fast, lock-free approach for efficient parallel counting of occurrences of k-mers, Bioinformatics, Volume 27, Issue 6, March 2011, Pages 764–770, https://doi.org/10.1093/bioinformatics/btr011