# Parallel FastAAI

Anh Le

January 2024

## 1 Problem Statement

### 1.1 Definitions

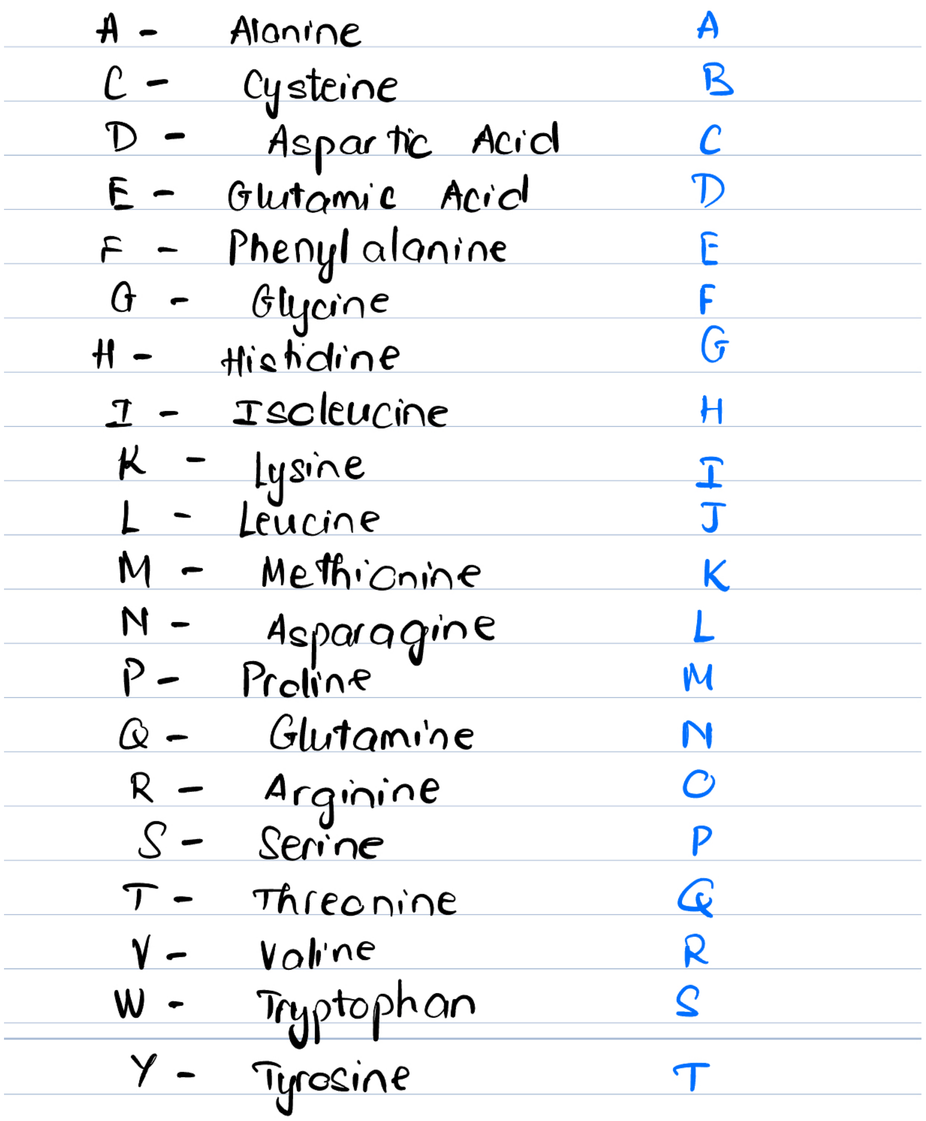
**Genome :** A genome is modeled as a sequence generated from an alphabet of four characters {A, C, G, T}.
**Protein :** A **genome** encodes multiple proteins in its sequence. A protein is modeled as a sequence generated from an alphabet of twenty characters (Figure 1), with each character representing an amino acid that can make up the protein.
**SCP :** A subset of the proteins, that occur only once in the genome, are termed as **single-copy proteins (SCPs)**. We use $\mathcal{P}_A$ to denote the set of SCPs in genome $G_A$.
**Tetramer :** A **tetramer** is a substring of the protein and is of length 4. For example, the first three tetramers of the protein MNRQGKSVEISELS are MNRQ, NRQG and RQGK. For a protein $P_i$ and genome $G_A$, we denote $Q(P_i, G_A)$ as the set of tetramers occurring in $P_i$ of $G_A$.
**Average Jaccard Index : Average Jaccard Index** of a pair of genomes A and genome B is defined as the following ratio :

$$AJI(G_A, G_B) = \frac{\displaystyle\sum_{P_i \in \mathcal{P}_A \bigcap \mathcal{P}_B} J(P_i, G_A, G_B)}{\mid \mathcal{P}_A \bigcap \mathcal{P}_B \mid} \tag{1}$$

where $J(P_i, G_A, G_B)$ is defined as:

$$J(P_i, G_A, G_B) = \frac{|Q(P_i, G_A) \bigcap Q(P_i, G_B)|}{|Q(P_i, G_A) \bigcup Q(P_i, G_B)|} \tag{2}$$

Though $AJI(G_A, G_B)$ can be defined in a weighted manner, in this document, we focus only on the **Average Jaccard Index**.



| | | |
|---|---|---|
| A - | Alanine | A |
| C - | Cysteine | B |
| D - | Aspartic Acid | C |
| E - | Glutamic Acid | D |
| F - | Phenylalanine | E |
| G - | Glycine | F |
| H - | Histidine | G |
| I - | Isoleucine | H |
| K - | Lysine | I |
| L - | Leucine | J |
| M - | Methionine | K |
| N - | Asparagine | L |
| P - | Proline | M |
| Q - | Glutamine | N |
| R - | Arginine | O |
| S - | Serine | P |
| T - | Threonine | Q |
| V - | Valine | R |
| W - | Tryptophan | S |
| Y - | Tyrosine | T |

Figure 1: List of Amino acids, the standard IUPAC single letter codes and our condensed character representation.

## 1.2 Problem

Given a database of genomes, calculate the Jaccard similarity between all pairs of genomes as the sum of the **Average Jaccard Index** between each pair of genomes.

## 2 Data Structures

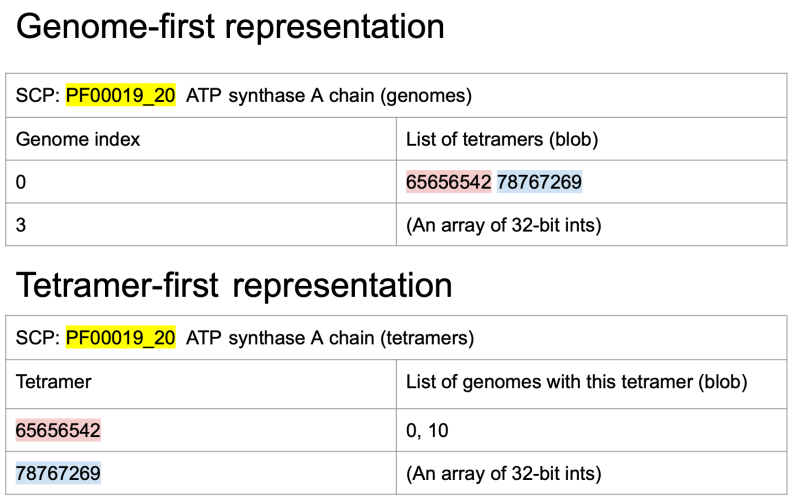
### 2.1 Tetramer Representation

In the current implementation of AAI, the tetramers are represented using 32-bit integers constructed via a charcter-to-ASCII mapping. For example, tetramer "`AAAC`" is represented by the integer 65656567.

Since there are 20 amino acids, there are $20^4 = 160,000$ possible tetramers. We propose to encode our tetramer by mapping tetramer to an integer in the range $[0, 20^4 - 1]$ i.e., $[0, 159, 999]$. Instead of standard IUPAC single letter codes, we use the uppercase English characters in the alphabetical order. We map the amino acids to its "condensed" character form as the table shown in figure 1.

### 2.2 Current Implementation

Current implementation of AAI uses the following two tables for to represent the tetramer data generated from the database of the genomes.



Genome-first representation

| SCP: PF00019_20 ATP synthase A chain (genomes) | |
|---|---|
| Genome index | List of tetramers (blob) |
| 0 | 65656542 78767269 |
| 3 | (An array of 32-bit ints) |

Tetramer-first representation

| SCP: PF00019_20 ATP synthase A chain (tetramers) | |
|---|---|
| Tetramer | List of genomes with this tetramer (blob) |
| 65656542 | 0, 10 |
| 78767269 | (An array of 32-bit ints) |

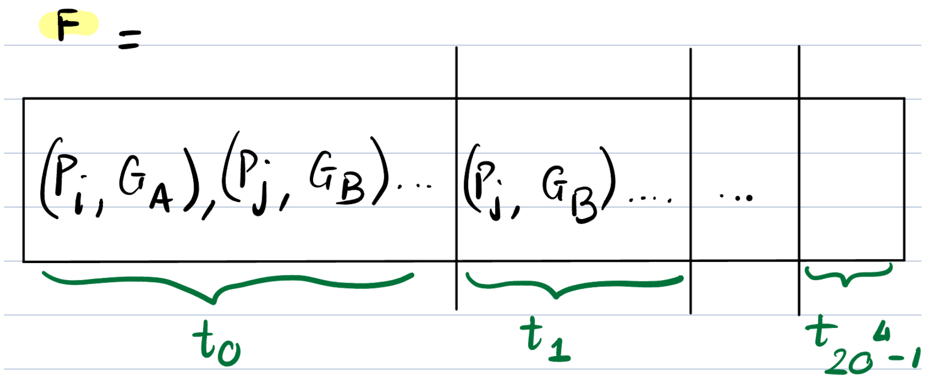
For each single copy protein $P_i$, two tables are used : (i) $DG_{P_i}$ Genome – Tetramer table (which maps each genome to the list of tetramers stored in it), and (ii) $DT_{P_i}$ Tetramer – Genome table (which maps each tetramer to a list of genomes where this tetramer exists in).

## 2.3 Proposed Data Structures

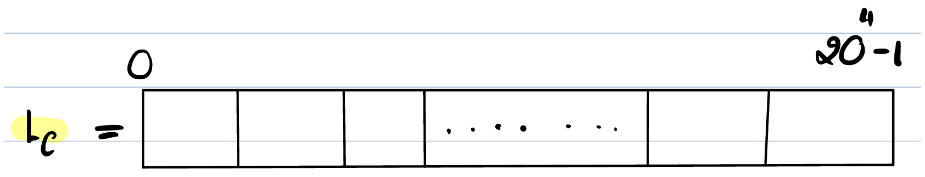
Let $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{T}$ be the set of all genomes, SCPs and tetramers present in the database respectively. Correspondingly genomes, SCPs and tetramers are given identifiers in the ranges $[0, |\mathcal{G}| - 1]$, $[0, |\mathcal{P}| - 1]$ and $[0, |\mathcal{T}| - 1]$ respectively. Let $p$ be the number of processors available in the system. The processors are assigned identifiers in the range $[0, p - 1]$.

Our parallel algorithm uses the following four data structures.

1. $F$, a distributed array of protein, genome pairs, $(P_i, G_A)$. For every tetramer $t_j, j \in \{0, \ldots, 20^4 - 1\}$, the array $F$ stores all pairs of $(P_i, G_A)$, if and only if the tetramer $t_j$ exists in SCP $P_i$ of Genome $G_A$. $F$ stores this data as a blocks of sub-arrays for every $t_j$, in the ascending order of tetramers. Within each block of information for a tetramer, the pairs are first sorted by the protein, and then by the genome.



2. $L_c$, an array of counts of length $20^4 = 160,000$, where $L_c[t_j]$ is size of the block corresponding to $t_j$. In other words $L_c[t_j]$ contains the number of entries corresponding to $t_j$ in $F$.

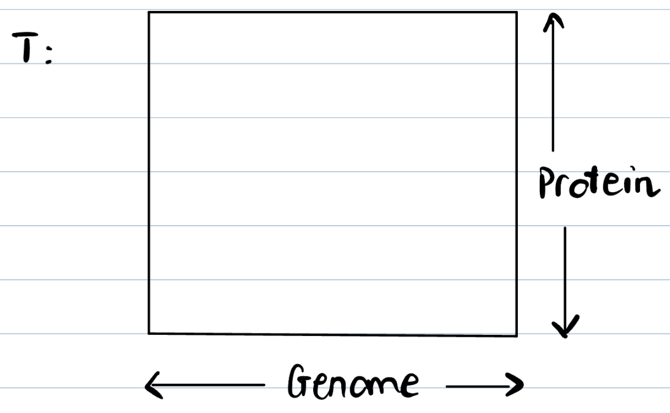


where $L_c[t_i]$ = # occurrences of tetramer $t_i$ in the database

i.e. $L_c[t_i]$ = # pairs of $(P_i, G_A)$ for $t_i$ that denotes $t_i \in P_i$ in $G_A$

3. $L_p$, a pointer array of length $20^4$. We construct the pointer array $L_p$ by performing a prefix sum on $L_c$ i.e., $L_p[t_j] = \sum_{k<j} L_c[t_k]$. The array $L_p$ can be used to quickly identify the location of the block in $F$ corresponding to $t_j$ i.e., $L_p[t_j]$ locates the index of starting position corresponding to a the tetramer $t_j$ in of $F$ .

4. $T$, a 2-D Array of size $|\mathcal{P}| \times |\mathcal{G}|$ that stores the number of tetramers in each of the SCP of every genome. $T$ is required for computing the denominator of equation 2.



$T[P_i, G_A]$ = # tetramers in $P_i$ of $G_A$

**Note**: $|X \bigcup Y| = |X| + |Y| + |X \bigcap Y|$ for any 2 sets X and Y. The $T$ table stores the $|X|, |Y|$ values for our calculations.

## 2.4 Construction of the proposed Data Structures

We assume that the input datasets are available in memory in a format as shown in the section 2.2. The assumption implies that Tetramer–Genome tables ($DT$ tables) and Genome-Tetramer tables ($DG$ tables) are already constructed for the set of all SCPs, $\mathcal{P}$, as tables of pointers to arrays in memory. We also assume that enough memory is available to construct $F$, $L_c, L_p$ and $T$. If enough memory is not available to load all the proteins, the proteins can be processed in batches, with enough memory available to accommodate each batch. We further discuss this in section 2.5.

We first construct the array $L_c$ array in parallel via algroithm 1 by counting the number of entries in $F$ corresponding to each tetramer across all proteins and genomes. Each processor is assigned approximately $|\mathcal{T}|/p$ number of consecutive tetramers. Algorithm 2 enumerates the steps undertaken by processor with processor id $j$, to construct the part of the array corresponding to processor $j$, $L_{c,j}$. Since obtaining the total number of entries per protein is a constant operation, each processor executes $O((|\mathcal{T}| \times |\mathcal{P}|)/p)$ operations and hence, the algorithm takes $O((|\mathcal{T}| \times |\mathcal{P}|)/p)$ time.

**Algorithm 1** Construct $L_c$ and $L_p$ in parallel

```
1:  function CONSTRUCT $L_c$ AND $L_p$($\mathcal{P}$, $\mathcal{T}$, $DT$, processor id $j$)
2:      $[t_{start}, t_{end}] \leftarrow$ Tetramer range assigned to proc. $j$.    $\triangleright \approx \frac{|\mathcal{T}|}{p}$ per proc.
3:      Initialize $L_{c,j}$ to zeros and $|L_{c,j}| : t_{end} - t_{start} + 1$
        Note: $L_{c,j}$ the sub-array of $L_c$ corresponding to processor $j$
4:      for each tetramer $t_k \in [t_{start}, t_{end}]$ do
5:          for each protein $P_i \in \mathcal{P}$ do
6:              $w \leftarrow$ Total no. of entries in $DT_{P_i}$ for tetramer $t_k$.
7:              Increment $L_{c,j}[t_k - t_{start}]$ by $w$.
8:          end for
9:      end for
10:     Construct $L_p$ as prefix sum $L_c$, i.e., $L_p[t_j] = \sum_{k<j} L_c[t_k]$
11:     return $L_c, L_p$
12: end function
```

Algorithm 2 presents our proposed parallel algorithm to construct $F$ in parallel. Similar to Algorithm 1, algorithm 2 lists only the steps undertaken by processor $j$, to construct the block of the array $F$ corresponding to processor $j$. We denote this block as $F_j$.

The algorithm assigns a range of tetramers to each processor based on the number entries in $F$ corresponding the those tetrarmers. The tetramers are distributed such that each processor constructs approximately $|F|/p$ number of entries in $F$. In order to identify the size of the $F$ block that corresponds to a range of tetramers, the algorithm makes use of the $L_c$ array.

After the assignment, the relevant data is queried from the relevant $DT$

tables, and then, the algorithm proceeds to append entries to $F_j$ portion of $F$ and update $L_{c,j}$ whenever $F_j$ is extended.

**Algorithm 2** Construct $F$ in parallel

```
 1: function CONSTRUCT $F(\mathcal{P}, \mathcal{T}, DT$, processor id $j)$
 2:     $[t_{start}, t_{end}] \leftarrow$ Tetramer range w.r.t proc. $j$. from $L_p$
        Note: Tetramers are distributed such that each processor constructs approximately $|F|/p$ number of entries in F
 3:     $F \leftarrow$ Distributed array of (protein, genome) pairs        $\triangleright \approx \frac{|F|}{p}$ per proc.
        Note: $F_j$ the sub-array of $F$ corresponding to processor $j$
 4:     for each tetramer $t_k \in [t_{start}, t_{end}]$ do
 5:         for each protein $P_l \in \mathcal{P}$ do
 6:             $SDT \leftarrow$ Query in $DT_p$ for tetramer $t_k$.
 7:             Sort $SDT[t_k]$ by genome ids, if needed.
 8:             for each genome $G_A$ in $SDT[t_l]$ do
 9:                 Append $(P_l, G_A)$ to $F_j[t_k]$.
10:             end for
11:         end for
12:     end for
13: end function
```

Since the entries of $F$ are distributed approximately equally across the $p$ processors, this algorithm takes $O(|F|/p)$ time.

Algorithm 3 depicts our proposed algorithm to build the two dimensional array $T$ of size $|\mathcal{P}| \times |\mathcal{G}|$. In parallel construction of $T$, the rows are distributed equally across each processor. Each processor constructs a slice of $T$, as submatrix of size $\frac{|\mathcal{P}|}{p} \times |\mathcal{G}|$. The locally compute $T$ is then broadcast to all the other processors, after which, each processor gets access to $T$ in its entirety.

**Algorithm 3** Construct $T$ in parallel

```
 1: function CONSTRUCT $T(\mathcal{P}, \mathcal{T}, DG$, processor id $j$ ):
 2:     $P_{start}, P_{end} \leftarrow$ Proteins assigned to proc. j        $\triangleright \approx \frac{|\mathcal{P}|}{p}$ per proc.
 3:     $T \leftarrow$ Initialize 2D array of size $|\mathcal{P}| \times |\mathcal{G}|$
 4:     for protein $P_k \in [P_{start}, P_{end}]$ do
 5:         $SDG \leftarrow$ Query $DG_{P_k}$.
 6:         for genome $G_l \in SDG$ do
 7:             $T[P_k, G_l] \leftarrow |SDG[G_l]| - 3$
 8:         end for
 9:     end for
10:     Broadcast locally-computed $T$ to all the processors.
11: end function
```

Since size of the $T$ is relatively small compared to $F$, this is expected to consume much less time compare to the construction of $F$ and the algorithm to compute Average Jaccard Index.

### 2.5 Handling Memory Constraints

In case of running the algorithm in a old desktop or laptop with limited main memory, all the protein and tetramer data may not be completely loaded in the main memory. In such cases, the above algorithms can be run in batches of protein. In each batch, only a subset of SCPs are processed at a time. Here, $F$ and $L_c$ will be rebuilt while processing each batch, where as $T$ will be constructed only once and retained in the main memory.

## 3 Parallel FastAAI Algorithm

Algorithm 4 enumerates the key steps our parallel algorithm for FastAAI as executed by processor $j$. After constructing the data structures in lines 2 – 4, the algorithm proceeds to construct a distributed array $E$, containing the tuples $(P_i, G_A, G_B)$ for each tetramer matched in protein $P_i$ between $G_A$ and $G_B$ (lines 6- 14).

Sorting the array $E$ by genome pairs brings the entries corresponding to the same genome pairs can be processed together to compute the Jaccard coefficients. We accomplish this in lines 15 – 25 using the distributed array $JAC$ of tuples $((G_A, G_B), S, N)$, where $S$ and $N$ maintain the sum of Jaccard coefficients and number of common proteins respectively.

After eliminating the duplicates in $JAC$ array, the AJI values can be computed with a scan across the $JAC$ array (lines 26–29).

Each step in the algorithm distributes the work equally among all the processors, the algorithm is efficiently parallel.

**Algorithm 4** FastAAI Parallel Algorithm

```
1: function PARALLEL FASTAAI($\mathcal{P}$, $\mathcal{T}$, $DT$, $DG$, processor $j$):
   Phase 1 : Construction of the data structures
2:     $L_c, L_p \leftarrow$ CONSTRUCT $L_c$($\mathcal{P}$, $\mathcal{T}$, $DT$, processor id $j$)
3:     $F \leftarrow$ CONSTRUCT $F$($\mathcal{P}$, $\mathcal{T}$, $DT$, processor id $j$)
4:     $T \leftarrow$ CONSTRUCT $T$($\mathcal{P}$, $\mathcal{T}$, $DG$, processor id $j$ )
   Phase 2 : Generate tetramer tuples
5:     $[t_{start}, t_{end}] \leftarrow$ Tetramers assigned to proc. $j$ using $L_p$.
       Note: Tetramers are distributed s.t. a processor is assigned $\approx |F|/p$ entries.
6:     $E$ : Distributed array of tuples $(P_i, G_A, G_B)$.
       Note: $E_j$ is the sub-array corresponding to processor $j$.
7:     for each tetramer $t_k \in [t_{start}, t_{end}]$ do
8:         for each block $B_l$ in $F_j$ w.r.t. protein $P_i$, and tetramer $t_k$ do
9:             for each pair $(G_A, G_B)$ in block $B_l$ do
10:                Append the tuple $(P_i, G_A, G_B)$ to $E_j$
11:            end for
12:        end for
13:    end for
14:    Parallel sort $E$ by $(G_A, G_B)$, and then by $P_i$.
   Phase 3 : Compute the Jaccard Coefficient values
15:    $JAC$ : Distributed array of tuples $((G_A, G_B), S, N)$, where
         - $(G_A, G_B)$ is a genome pair,
         - $S$ is $\sum_i J(P_i, G_A, G_B)$ w.r.t. the tuples in $F$ processed so far, and
         - $N$ is $|\mathcal{P}_A \cap \mathcal{P}_B|$ w.r.t the tuples in $F$ processed so far.
         Note: $JAC_j$ is the sub-array corresponding to processor $j$.
16:    for each block $B_l$ of $E_j$ with the same genome pair $(G_A, G_B)$ do
17:        $S \leftarrow 0$; $N \leftarrow 0$
18:        for each sub-block $B_k$ of $B_l$ with the same protein $P_i$ do
19:            $J_{P_i, G_A, G_B} \leftarrow \frac{|B_k|}{T[P_i, G_A] + T[P_i, G_B] - |B_k|}$
20:            Increment $S$ by $J_{P_i, G_A, G_B}$.
21:            Increment $N$ by 1.
22:        end for
23:        Append $((G_A, G_B), S, N)$ to the table $JAC_j$
24:    end for
25:    Remove the duplicate entries in $JAC$ for a given $(G_A, G_B)$ pair by summing up the corresponding $S$ and $N$ values.
26:    $AJI$ : Map of AJI values.
27:    for each entry of $((G_A, G_B), S, N) \in JAC_j$ do
28:        $AJI[(G_A, G_B)] = \frac{S}{N}$
29:    end for
30: end function
```