Parallel acceleration of UPGMA algorithm

Suraj Sathya Prakash†, Ashwin Agesh Somanathan †, Kumar Divij †
†University of California San Diego

***Index Terms*—UPGMA, CUDA, Parallel Algorithms**

***Abstract*—UPGMA algorithm is used to create phylogenetic trees from raw base pair data of genetic material. The UPGMA algorithm has a complexity of $O(n^3)$ complexity in time and space, where n is the sequence length. A typical genetic material sequence runs into billions of base pairs. Hence, it is not feasible to deploy the algorithm serially for practical purposes. To accelerate the algorithm, we propose and implement several parallelization techniques using GPUs.**

I. Introduction

Phylogenetic trees represent evolutionary relationships among different species, organisms, or genes. They are often depicted as a branching pattern from a common ancestor. Nodes within a tree are called taxonomic units. Phylogenetic trees are a powerful tool in bio-informatics for deciphering the complexity of biological systems and addressing fundamental questions in biology and medicine. They provide insights into the spread of diseases, the evolution of drug resistance in pathogens, and the design of effective treatments and vaccines. They also aid in taxonomic classification, biodiversity assessment, and conservation efforts. A recent example is that of the COVID-19 virus. Phylogeny of the SARS-CoV-2 virus. Fig 1 has been used to estimate the spread rates across regions.

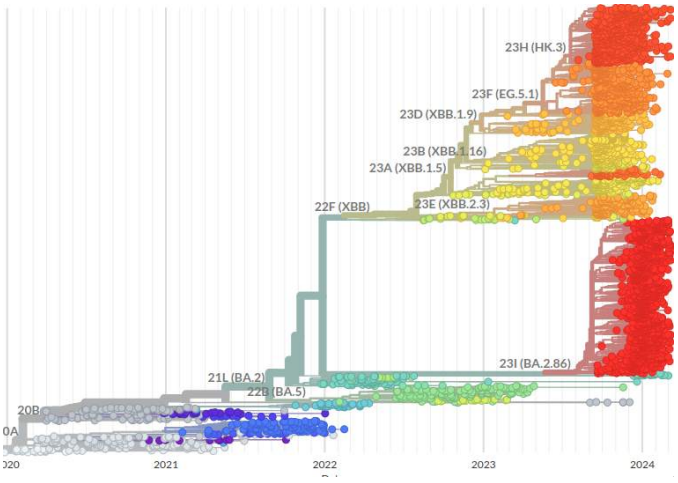


Fig. 1: The graph describes the Genomic epidemiology of SARS-CoV-2, adapated from Nextstrain

Given the massive size of genetic information, the construction of phylogenetic trees is computationally expensive. There has been extensive research on developing algorithms to deal with these complexities efficiently. Fig 2 shows the broad classification of approaches and specific implementations:

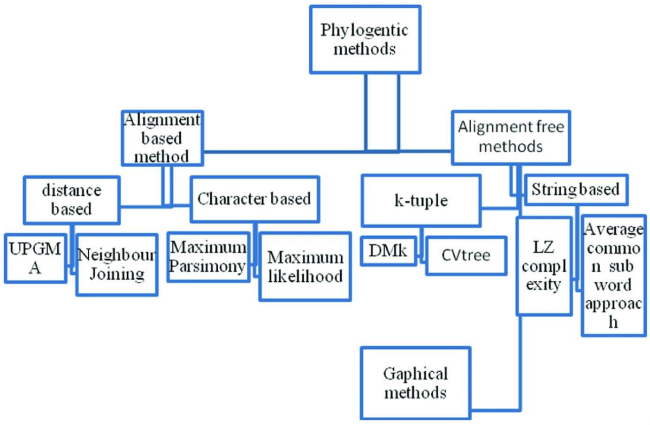


Fig. 2: Describes various phylogenetic methods. The above figure is borrowed from [1]

Of these many solutions, we opted to study and parallelize UPGMA, which stands for Unweighted Pair Group Method with Arithmetic mean. We chose to study this algorithm because it is an alignment-based method that uses distance matrices. Matrix-based algorithms are highly efficient on GPUs because they can handle large numbers of computations in parallel, specialized hardware blocks like Streaming Processors, and large memory bandwidth.

II. Prior Work

Authors in [2] presented GPU-UPGMA, an implementation of the UPGMA algorithm on an NVIDIA Tesla $C2050$ GPU using CUDA for software development, achieving a $95\times$ improvement in speed compared to a 21.3 GHz CPU.

The extension to multiple GPU's was proposed in [3]. They used 4 NVIDIA GTX 980s. Authors in [4] took this even further, using a DGX-1 server with 8 NVIDIA P100 graphic cards in tandem using NVIDIA Collective Communications Library. They achieved a linear increase in speed compared to a single GPU and CPU. While matching and surpassing the results mentioned above has been beyond the scope of this project given the time and resource constraints, we took inspiration from these works to implement and optimize the UPGMA algorithm on a single NVIDIA 2080-Ti GPU, taking the work done in [2] as an indicator to set expectations for performance improvements. Another interesting work, [5] discusses various matrix operations that be performed efficiently on GPUs.

III. UPGMA Algorithm

In this section, we will discuss the basic UPGMA algorithm.

Consider a 5×5 distance matrix. Let each species be named A, B, C, D, and E. A distance matrix can be prepared using various methods such as,

- Similarity or Levenshtein Distance
- Dissimilarity or Hamming Distance
- Jukes-Cantor Distance

The UPGMA algorithm broadly involves the following steps:

1) Find and merge the pair with the shortest separation (Fig 3b)
2) Update the distance matrix, considering the merged objects as a single entity (Fig 3c)
3) Repeat step 2 until all objects have been grouped or have been reduced to a predefined threshold. (Fig 3d, 3e and 3f)

While each of these steps has to be carried out serially in order, a lot of parallelism can be found within a step.

- Import Distance Matrix: each element needs to be initialized independently, allowing for **map parallelism**.
- Find minimum: finding the minimum of a set of elements is an associative operation. Thus, finding the minimum in the matrix is a **reduction operation**.
- Group and update elements: every impacted element can be updated independently, allowing **map parallelism**. Added to this will be overheads for identifying and grouping elements.

IV. UPGMA Methodology

Before diving into parallel programming aspects of UPGMA, we prototype the algorithm in Python, verify the serial implementation in C++, and finally parallelize the codebase in CUDA. In the following section, after the Prototype section, we discuss the serial implementation in detail. In the parallel implementation section, we discuss the specific algorithmic modification used to parallelize the implementation.

A. Prototype

We first implemented a prototype of the UPGMA algorithm in Python. This was tested against examples found on the internet, such as [6]. This model then became our benchmark to verify functional correctness. Python was chosen as the language of choice for the following reasons:

1) Easy to setup
2) Convenient constructs such as a list of lists.
3) Libraries to easily handle large amounts of data

B. Serial Implementation

Once the prototype was ready and tested, we transcribed the code into C++. This was slightly challenging since C++ does not implicitly provide some of the functionalities that a high-level language like Python does. After a few iterations, we were able to implement the UPGMA in the following manner:

1) Read distance matrix from file:

	A	B	C	D	E
A	0	0	0	0	0
B	**6**	0	0	0	0
C	**8**	**8**	0	0	0
D	**8**	**8**	**4**	0	0
E	**8**	**8**	**4**	**2**	0

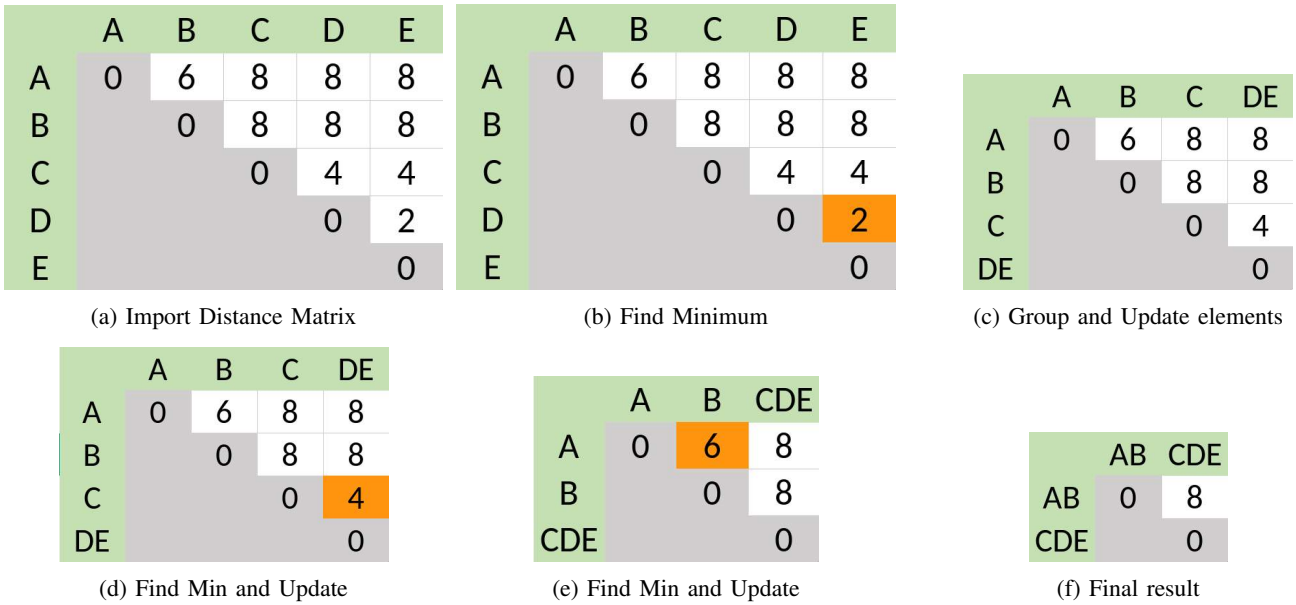
Fig. 4: Reading Distance matrix

2) Mirror the distance matrix:
This is an optimization step for extracting parallelism and reducing thread divergence. In the parallel implementation described in the next section, we use reduction tree architecture. Hence, we mirror the distance matrix to reduce thread divergence as we have to constrict ourselves to the upper or lower half of the matrix.

2	A	B	C	D	E
A	0	6	8	8	8
B	**6**	0	8	8	4
C	**8**	**8**	0	4	4
D	**8**	**8**	**4**	0	2
E	**8**	**8**	**4**	**2**	0

Fig. 5: Mirrored Distance matrix

3) Initialize operations matrix:
An operation matrix, a copy of the distance matrix, was initialized as continually updated, whereas the distance matrix was used as the golden matrix to calculate the scores.
4) Initialize the cluster matrix:
The cluster matrix was a $N \times N$ matrix whose first column was initialized to encode the species names. The second column is marked by -1's, and the rest of the entries are -2's. The -1's serve as a boundary condition during search operation, while the -2's serve as an invalid bit location. In the context of cluster matrix, an invalid bit corresponds to the non-occurrence of species in that particular cluster. The cluster matrix looks similar to Fig. 6



(a) Import Distance Matrix (b) Find Minimum (c) Group and Update elements

(d) Find Min and Update (e) Find Min and Update (f) Final result

Fig. 3: UPGMA algorithm

CL0	A	-1	-2	-2	-2
CL1	B	-1	-2	-2	-2
CL2	C	-1	-2	-2	-2
CL3	D	-1	-2	-2	-2
CL4	E	-1	-2	-2	-2

Fig. 6: Cluster matrix initialization

5) Find minimum:
Find the minimum among the lower triangle matrix as shown in Fig. 11. We use a simple greedy search to find the minimum in the serial implementation.

0	A	B	C	D	E
A	0	6	8	8	8
B	**6**	0	8	8	4
C	**8**	**8**	0	4	4
D	**8**	**8**	**4**	0	2
E	**8**	**8**	**4**	**2**	0

Fig. 7: 2 is the minimum score among the lower triangle in the matrix

6) Cluster list updation:
The two species of indices corresponding to the minimum value are sorted to find the lexicographically least species encoding. We then append the lexicographically greater species to the smaller cluster and update the cluster list. An example of the process is shown in Fig 8. We also log the species names and their location in a look-up table to increase the speed of random access to the Cluster matrix. Let's call the data structure as $clusterLookup$. The access time for finding the location of the species in the cluster matrix now reduces to $O(1)$. In other words, a simple call to $clusterLookup(species)$ would give us the single-dimensional location of the species in the cluster matrix.

CL0	A	-1	-2	-2	-2
CL1	B	-1	-2	-2	-2
CL2	C	-1	-2	-2	-2
CL3	D	E	-1	-2	-2
CL4	-2	-2	-2	-2	-2

Fig. 8: The species names corresponding to the indices of minimum score were D and E. Hence, CL4 values are appended to CL3.

7) Operations matrix updation:
The latest row update to the cluster matrix also serves as an index for the operation matrix updation. Let's call the row where we update the cluster matrix as $updateIndex$. We iterate through the cluster matrix with row index as $updateIndex$ and then track which column index we find -1 and store it in a shared memory location. The location of -1 gives us the number of

cluster elements or species present in the clusters. This information is then used to find the clustering score as:

$$score = \frac{\sum Cluster\ element\ score}{\sum Cluster\ element} \quad (1)$$

The score usually tends to be a float value. However, to save some memory space and increase the execution speed, we upscale the score values by multiplying them by 32 or left-shifting the score by 5. We then update the $updateIndex$ row with the above formula and assign 0 to the other row index, contributing to the operation matrix's minimum location.

The above operations are shown in Fig. 9

0	A	B	C	DE	-
A	0	6	8	8	0
B	**6**	0	8	8	0
C	**8**	**8**	0	4	0
DE	8	8	4	0	0
-	0	0	0	0	0

Fig. 9: The row highlighted in blue corresponds to the $updateIndex$ row, and the other row corresponds to the row that previously had a cluster with E. As we observe, the updateIndex row has a newly updated score, and the other row is assigned as invalid or 0

C. Parallel implementation

The serial implementation of UPGMA provides a good starting point for parallelizing the algorithm. We identified the code sections that can exploit different parallelization techniques by analyzing the repeating and diverging conditions. We also run each operation in separate kernels. This not only gives us the flexibility to analyze the performance of each step but also gives it finer control over parallelization.

1) Mirror the distance matrix:
Map Parallelism can be exploited to mirror the matrix as it for $\frac{N \times N}{2}$ number of elements.
2) Initialize operations matrix:
The operations can be parallelized using **Map Parallelism** as we have to copy only a fixed number $N \times N$ elements.
3) Initialize the cluster matrix:
Since this operation requires the updation of only 2 columns after initializing the cluster matrix, it can be again done exploiting **Map Parallelism**
4) Find minimum:
   a) We first go through the Operation matrix and assign each element with value 0 to $\_\_INT32\_MAX\_\_$ to make it easier for us during minimization. The elements that are 0 are the ones whose scores have been modified to invalid as they were already part of a cluster or the diagonal matrix, which is always 0. We exploit **Map parallelism** for the $\_\_INT32\_MAX\_\_$ assignment operation.
   b) The next step is to run a **Parallel reduction** operation for minimization. We are required to calculate the minimum for the entire $N \times N$. Our strategy assumes we have N grid size and N block size on the GPU (assuming that $N \times N < 65535 \times 1024$). Each block would be assigned N sequences of the matrix to reduce parallelly. This would thus ensure that by the end of all the reduction, we would need to reduce only a handful of sequences of elements, which are the output of each parallelly reduced block. We ensure the synchronization of all the blocks as we instantiate the above operation with a kernel and use $cudaDeviceSynchronize()$ to wait for all blocks to finish.
5) Cluster List updation:
The cluster updation simply appends the row indexed by $updateIndex$ with the row corresponding to the other index obtained during the minimization of the matrix. It is pictorially depicted in Fig. 10

CL0	A	-1	-2	-2	-2
CL1	B	-1	-2	-2	-2
CL2	C	-1	-2	-2	-2
CL3	D	-1	-2	-2	-2
CL4	E	-1	-2	-2	-2

Fig. 10: The arrow marks indicate the location of the CL4 elements after copying

Since it is a direct assignment, we can exploit **Map parallelism** for the operation. We update the $clusterLookup$ with the new location values when appending the cluster matrix rows. We also need to note that the $clusterLookup$ data structure cannot be on the shared memory, but it would be in the device memory as it has to be synchronized across blocks and not threads.
6) Update operations matrix:
Score calculation has to be done for every element in the operation matrix indexed by $updateIndex$. This process has to happen serially as we need to pair all the species with the cluster elements and calculate the score. However, we can calculate the score across elements in parallel. Thus, the score calculation and subsequent assignment operation can be parallelized using **Map parallelism**.

V. Results

A. Functional verification

We first modeled a Python implementation based on [6]. We then verified our implementation from sources such as [7] and [8] . After we were confident that our code was working, we generated random test vectors and tested them on Python and the CUDA implementation. We could visually verify the cluster list for a few test cases with $N < 10$, and they mostly matched. The small number of mismatches arise from randomness associated with choosing a minimum score from a set of same score elements in UPGMA.

The following figure shows the python's final score vs. the CUDA implementation's final score.

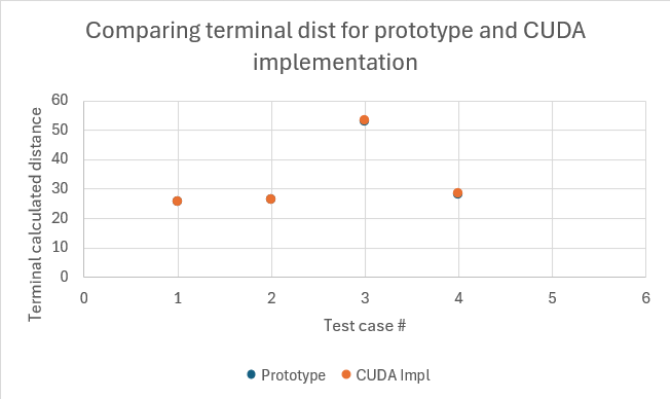


Fig. 11: python vs CUDA implemenatation

1) Execution time vs matrix dimension:

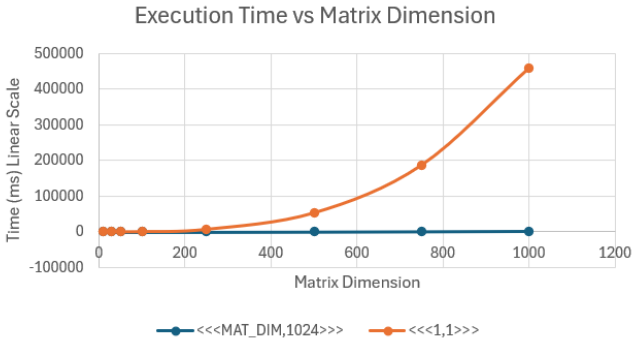


Fig. 12: The graph shows us the execution time vs the matrix dimension in linear scale

As we increase the matrix dimension, our implementation is much faster than the baseline single-thread implementation.

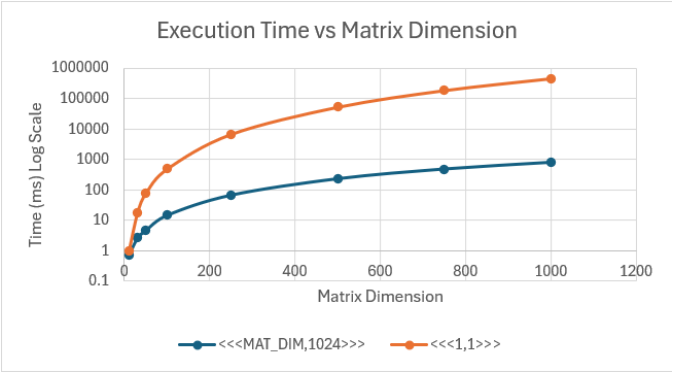


Fig. 13: The graph shows us the execution time vs the matrix dimension in log scale

2) Speedup vs matrix dimension:

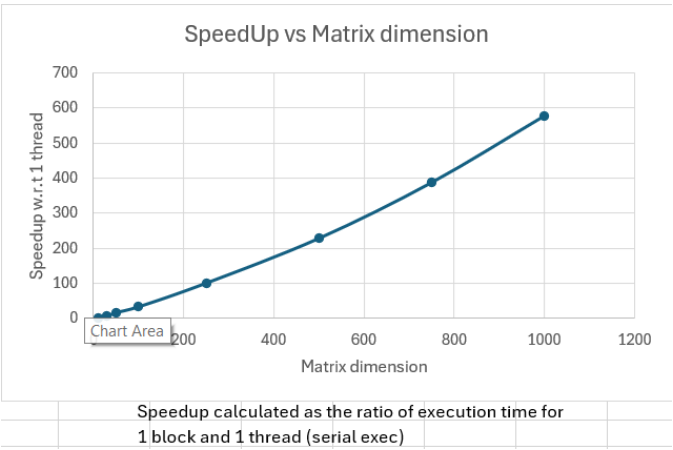


Fig. 14: The graph shows the speedup vs matrix dimensions

References

[1] Geetika Munjal, Madasu Hanmandlu, and Sangeet Srivastava. "Phylogenetics Algorithms and Applications". In: *Ambient Communications and Computer Systems*. Ed. by Yu-Chen Hu et al. Singapore: Springer Singapore, 2019, pp. 187–194.

[2] Yu-Rong Chen et al. "Parallel UPGMA Algorithm on Graphics Processing Units Using CUDA". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012, pp. 849–854.

[3] Guan-Jie Hua et al. "MGUPGMA: A Fast UPGMA Algorithm With Multiple Graphics Processing Units Using NCCL". In: *Evolutionary Bioinformatics* 13 (2017). PMID: 29051701, p. 1176934317734220. eprint: https://doi.org/10.1177/1176934317734220.

[4] Che-Lun Hung et al. "Efficient parallel UPGMA algorithm based on multiple GPUs". In: *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2016, pp. 870–873.

[5] Lezhou Wu. "Accelerating Matrix Power Operations with GPUs". In: *Proceedings of the 4th International Conference on Electronics, Communications and Control Engineering*. ICECC '21. Seoul, Republic of Korea: Association for Computing Machinery, 2021, 20–25.

[6] Dr. Richard Edwards. *UNSW Australia*. 2016. URL: http://www.slimsuite.unsw.edu.au/teaching/upgma/.

[7] Michael Goldwasser. *Saint Louis University*. 2019. URL: https://cs.slu.edu/~goldwasser/courses/slu/csci1020/2019_Spring/lectures/UPGMA/.

[8] Fred Opperdoes. *de Duve Institute*. 1997. URL: http://www.icp.ucl.ac.be/~opperd/private/upgma.html.