CS267 HW3

Stephen Lee, Carmen Matar, Molly McGuire

April 13, 2025

1 Group Members and Contributions

- Stephen Lee - Implemented chaining hash table that was used in our final submission. Wrote documentation of this implementation, data analysis, attempt at batching, and how this might have been implemented if we were using MPI or OpenMP.
- Carmen Matar - Contributed to writing code for possible implementation solutions, wrote linear probing written section, collected data and created tables and plots.
- Molly McGuire - Contributed to writing code for possible one-sided communication implementation, wrote One-Sided Communication Attempt and Attempted One-Sided Communication Optimization sections.

2 Graphs and discussion of the scaling experiments (inter-node using 64 tasks per node and intra-node varying number of tasks per node).

2.1 Inter-Node Experiments

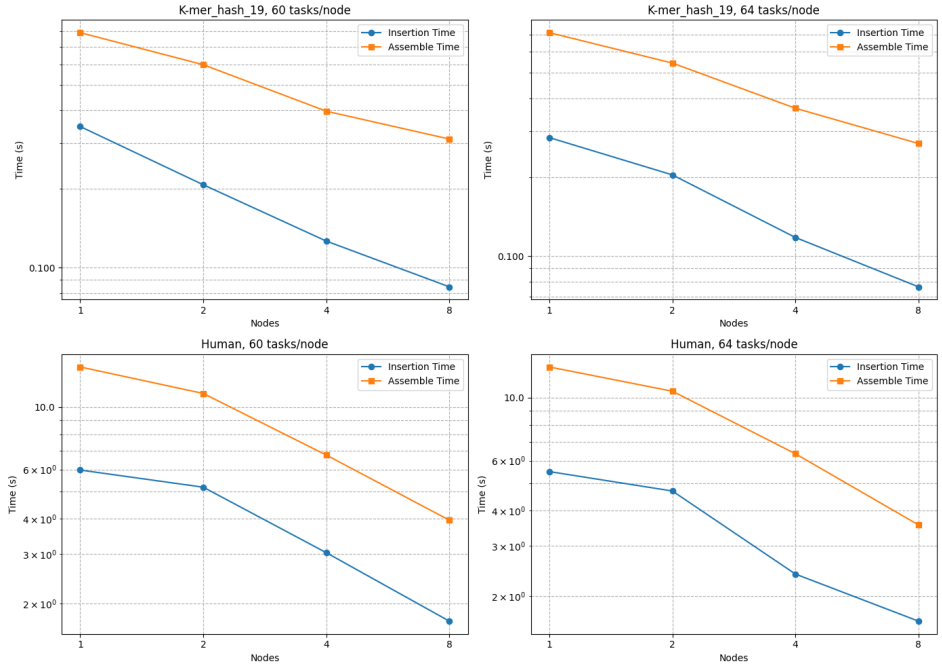

Figure 1: inter-node performance comparison graphs (raw data in Figure 2)

Nodes	K19-60 Insert	K19-60 Assemble	K19-64 Insert	K19-64 Assemble	Human-60 Insert	Human-60 Assembl	Human-64 Insert	Human-64 Assemble
1	0.346671	0.792431	0.283061	0.710664	5.971933	13.922214	5.489272	12.836828
2	0.207572	0.597548	0.20411	0.544322	5.187982	11.200968	4.688368	10.541957
4	0.126405	0.396827	0.117874	0.366155	3.029941	6.754609	2.388687	6.344142
8	0.084483	0.310541	0.076456	0.268818	1.72739	3.958277	1.629962	3.561162

Figure 2: inter-node performance comparison data table

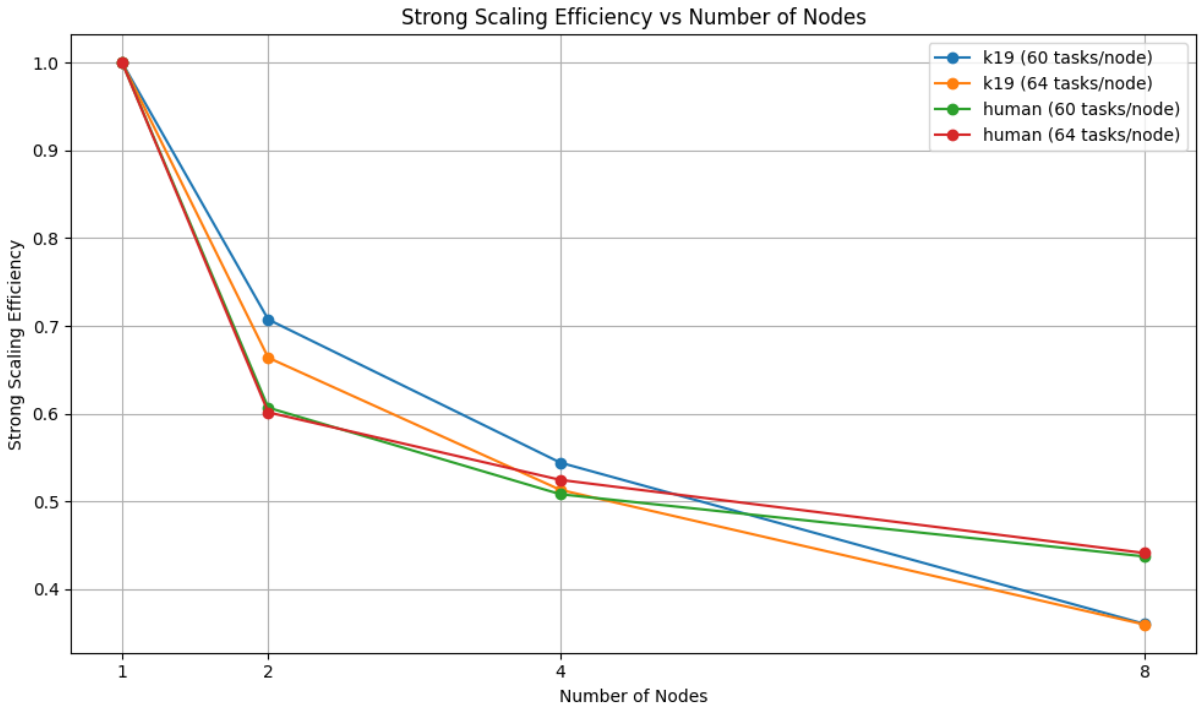

Figure 3: inter-node strong scaling efficiency graph (raw data in Figure 4)

Strong Scaling Efficiency Across Datasets and Node Counts

Nodes	K19-60	K19-64	Human-60	Human-64
1	1.000	1.000	1.000	1.000
2	0.707	0.664	0.607	0.602
4	0.544	0.513	0.508	0.525
8	0.360	0.360	0.437	0.441

Figure 4: inter-node strong scaling efficiency table

From the data presented above in Figures 1-4, we can see that we didn't notice any weird trends or spikes in our raw performance or strong scaling efficiency. The results all seem fairly sensible, as we increase the number of nodes, we see a decrease in the amount of time it takes to complete both insertion and total assembly time for both datasets. We also don't see any drastic performance changes when using 60 tasks/node versus 64 tasks/node, though we do see 64 tasks/node perform slightly better than 60 tasks/node which is expected since we do increase the number of processors by doing this. With regards to strong scaling, the human dataset performed better. We can see in our strong scaling efficiency plot that the strong scaling efficiency as we increase the number of nodes decreases faster for the smaller 19-mer dataset than the human dataset.

2.2 Intra-Node Experiments

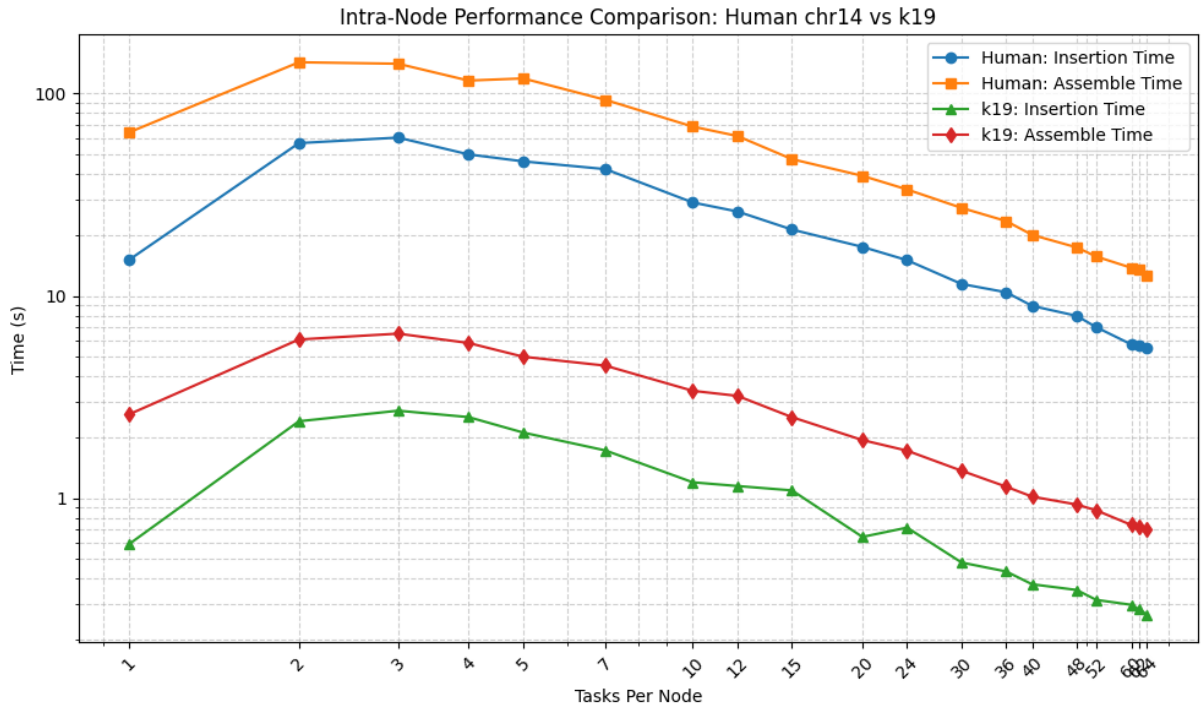

Figure 5: intra-node performance graph (raw data in Figures 6 and 7)

Intra-Node Timing Table (Human chr14 Dataset)

Tasks Per Node	Insertion Time (s)	Assemble Time (s)
1.0	15.098499	64.3233
2.0	56.887402	142.738711
3.0	60.585674	140.321086
4.0	50.028394	115.888315
5.0	46.228881	118.757084
7.0	42.310095	93.045111
10.0	28.942881	68.632284
12.0	26.131094	61.689539
15.0	21.271394	47.382499
20.0	17.471227	39.20834
24.0	15.01578	33.576477
30.0	11.453157	27.212203
36.0	10.419933	23.424815
40.0	8.929906	20.033098
48.0	7.970136	17.365579
52.0	7.007453	15.648801
60.0	5.760318	13.771168
62.0	5.653045	13.419778
64.0	5.514032	12.679995

Figure 6: intra-node performance table for human dataset

Intra-Node Timing Table (k19 Dataset)

Tasks Per Node	Insertion Time	Assemblance Time
1.0	0.59485	2.602029
2.0	2.402942	6.092548
3.0	2.709933	6.510511
4.0	2.519356	5.853814
5.0	2.113245	5.00882
7.0	1.723583	4.522637
10.0	1.199577	3.396919
12.0	1.150109	3.207689
15.0	1.094053	2.517018
20.0	0.64463	1.939687
24.0	0.71657	1.719789
30.0	0.48155	1.367788
36.0	0.434917	1.141036
40.0	0.376135	1.019464
48.0	0.3525	0.931834
52.0	0.314827	0.872346
60.0	0.297767	0.738736
62.0	0.282338	0.725794
64.0	0.265636	0.702886

Figure 7: intra-node performance table for 19-mer test dataset

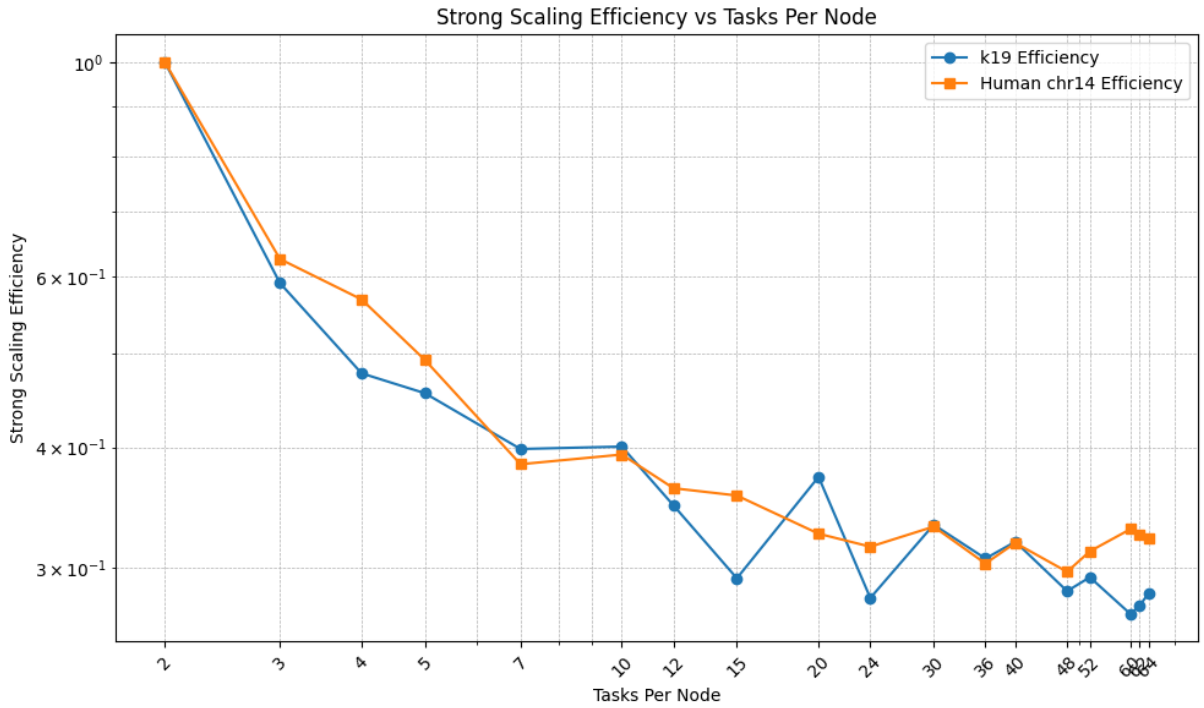

Figure 8: intra-node strong scaling efficiency graph

From the data presented above in Figures 5-8, we can see that going from 1 task per node to 2 tasks per node increased the computation time, likely due to some communication overhead that's introduced by going from serial on one processor to distributed across multiple processors. The data following that point follows some reasonable trends, decreasing the amount of time that it takes to both insert and assemble as we increase the number of ranks per node. There are some weird spikes in this data still that wasn't the case in our inter-node experiments, particularly with strong scaling efficiency. This is likely due to the non-powers of 2 for the numbers of nodes that we used in

our testing hitting or missing cache size breakpoints. It doesn't appear that either dataset had a clear advantage with regards to strong scaling efficiency, though with regards to absolute time performance, the human dataset took longer, which makes sense since it's much larger.

3 Describe your implementation–how is your hash table organized? What data structures do you use?

3.1 Linear Probing Attempt

One implementation attempt involved using linear probing to insert each k-mer into the hash map. Linear probing was used on the globally distributed hash table, and each probe calculated the owner rank and offset of the rank and used RPC (remote procedure call) to update memory. The main issue we ran into was that every slot was probed, but the hash table was consistently full due to an overflow of k-mers from poor management. After this approach was tested, we decided to test a chaining approach further.

3.2 Chaining

We implemented our finalized distributed hash table by using a chaining approach. The hash table consisted of a `std::vector<std::vector<kmer_pair>>`, which we wrapped in a `upcxx::dist_object`. Each rank was responsible for a `size / upcxx::rank_n() + 1` portion of the hash table, where `size` used the default parameter from the starter code of being double the number of kmers we are trying to insert into the hash map. To implement insert and find, we first compute the hash of the kmer using the provided hashing function. We then simply use modulo with the overall size of the hash table to find the global index of where the kmer should be, and modulo with `upcxx::rank_n()` to find the target rank for where this operation should happen. We can then take the global index and compute the local index (i.e. where on the target rank) by taking the modulo with `size / upcxx::rank_n() + 1`, the portion of the hash table that each rank is responsible for. Now we simply check if the target rank that we computed is the current process or not. If it is, that means that we can execute our insert or find logic immediately without having to communicate with other processors. For insert we simply `push_back(kmer)` at the local index we computed, and for find we linearly walk through kmers at local index until what we find matches the key and set our value equal to that element. Now finally, in the cases where the target rank is not the rank of the current processor, we do a `upcxx::rpc` at the target rank with the relevant insert or find logic that we described previously.

3.3 One-Sided Communication Attempt

In addition to the chaining-based implementation described in Section 3.2, we explored an alternative approach based on one-sided communication. The idea behind this approach was to construct the hash table using UPC++ distributed arrays and to perform insertions and lookups with direct remote memory access (RMA) operations (i.e. using `upcxx::rget` and `upcxx::rput`).

In our one-sided implementation, each rank allocated a contiguous segment of the global hash table consisting of an array of `kmer_pair` objects and an accompanying array for usage flags. These arrays were exposed via global pointers (obtained through UPC++ allocation routines) and then broadcast so that every process could access the complete hash table without requiring explicit two-sided message matching.

The insertion operation was implemented as follows. After computing the hash value of a given k-mer, we used modulo arithmetic to obtain a global index and then determined the owner rank by dividing the global index by the local segment size. Using UPC++ RPCs, we attempted to claim the target slot using a one-sided test and set the

corresponding usage flag atomically using remote put operations. For lookup, the remote memory was accessed via one-sided rget calls to retrieve the occupant of the target slot. Linear probing was employed to resolve collisions.

During testing, however, we encountered synchronization challenges. In particular, ensuring proper memory ordering with the raw `rget` and `rput` operations proved complex, and race conditions sometimes occurred during contig assembly. Although this one-sided strategy had the theoretical appeal of reducing synchronization overhead, its complexity and debugging difficulties ultimately led us to reject it in favor of our chaining approach.

4 Any optimizations you tried and how they impacted performance.

4.1 Batching

We attempted to implement batching to reduce the number of expensive RPC calls that we would have to make through the life time of the program. The idea that we tried to implement was for each rank to store an additional `std::vector<std::vector<kmer_pair>>` as its local cache. The size of this cache would be `rank_n()` instead, where we store a vector of cached kmers for each respective rank to send later in a batch. When the target rank was not the rank of the processor it was on already, we add it to the cache instead of instantly calling the RPC. We track how many elements have been added to the cache, and after some tunable threshold parameter we flush out the cache and make the RPC calls to each of the appropriate other ranks.

4.2 Attempted One-Sided Communication Optimizations

Building on our early one-sided communication implementation, we experimented with additional optimizations aimed at reducing the number of expensive RPC calls and overlapping communication with computation. One such optimization involved batching multiple insertion and lookup requests. Specifically, each rank maintained a local cache (implemented as a `std::vector<std::vector<kmer_pair>>` indexed by target rank) to accumulate pending operations. When the cache reached a preset threshold, all pending operations were flushed in a single batch RPC call per target rank, reducing overall communication overhead.

Another early safeguard we employed during contig assembly was a maximum chain length check (set to 1000) to prevent infinite loops. However, this maximum chain length limit proved too inflexible – it often terminated contigs prematurely, before they naturally ended. As a result, we replaced it with a helper function (`contig_ended()`) that examines the forward extension marker(s) of each `kmer_pair` to more accurately detect when a contig has reached its end.

Despite these efforts, the further optimizations applied to the one-sided communication implementation did not translate into a clear performance benefit. In fact, this approach exhibited unpredictable behavior and increased complexity, particularly when compared with the straightforward chaining implementation described in Section 3.2.

5 Discussion: UPC++ vs. MPI/OpenMP

We used distributed objects for our data structures, so every process had the same global name for the object, but holds its own local value. Each process would initialize its own local `std::vector<std::vector<kmer_pair>>` of the size that we specified. The communication between ranks was handled by making RPC calls. These RPC calls can be broken down into a send and receive step. It begins by sending relevant data to the target rank, which then does some computation that we define. When this completes, we receive some return value from the target rank that we can access by calling `wait()`, which would be some synchronization barrier. If we were to have implemented this using MPI, we would probably have something very similar to what we described above, where each rank has its own

local section of the hash table. We can then do send and receive calls to handle times where we need to communicate between ranks. If we were to have implemented this using OpenMP, this would look a little different since we no longer have distributed memory. We would probably still divide the hash table up into sections, though this time distributed between threads. We don't need to worry about send and receive calls, but would need to atomically insert elements into the hash table.