CS267 Homework 3

Parallelizing Genome Assembly

Team members: Tzu-Chuan Lin, Chin-An Chen, Byron Hsu

Abstract

In this homework, we want to assemble a series of contigs from a large collection of k-mers. We first need to find the starting k-mers and then keep merging with the next k-mers until the end.

We parallelize the de novo genome assembly pipeline with UPC++. Each process will maintain a hash table (with certain suffixes) and hold several starting contig. Each process will construct DNA sequences independently.

To further accelerate our program, we package the sending payload if they are sent to the same process. Therefore, we can decrease the overhead of sending payloads.

Implementation

The data structure of the hash table

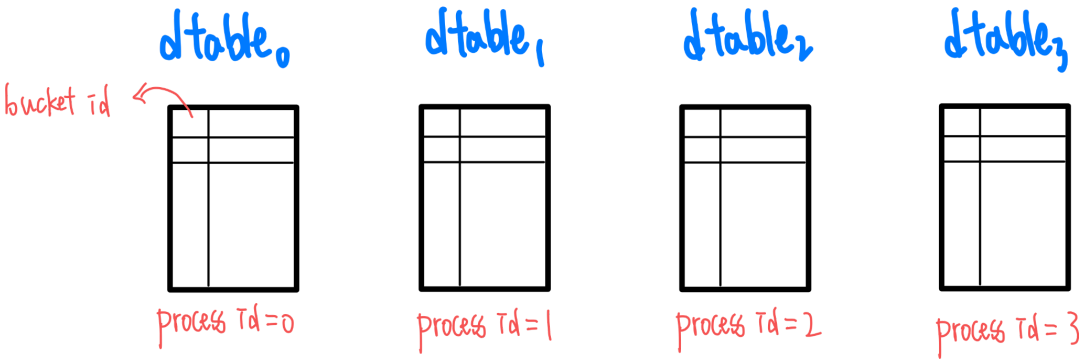


Fig1: The data structure of hash table

Each process will maintain a distributed object, `dtable`. `dtable` is with the type of vector<vector<kmer_pair>>.

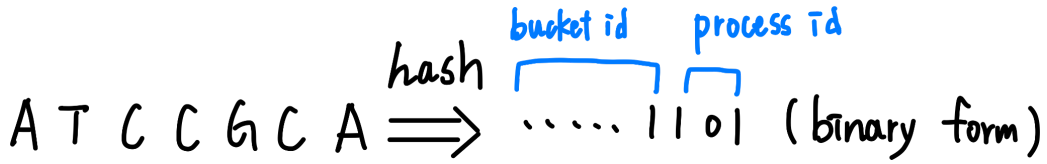


Fig2: How we map hash value to ids

A contig is mapped to a hash value. Viewing from the binary form, the last few bits represent the process id, and the other digits represent the bucket id.
The detailed formula can be written as:

- n: number of processes
- n_bits: log2(n)
- process_id: hash value % n
- bucket_id: hash value >> n_bits

For a key-value pair, we first locate the process by the process id and locate the bucket by bucket index. In each bucket, many pairs are chained together and stored in the format of the vector so that hash collision (i.e. collision of bucket_id for different keys) can be resolved in this way.

Insert

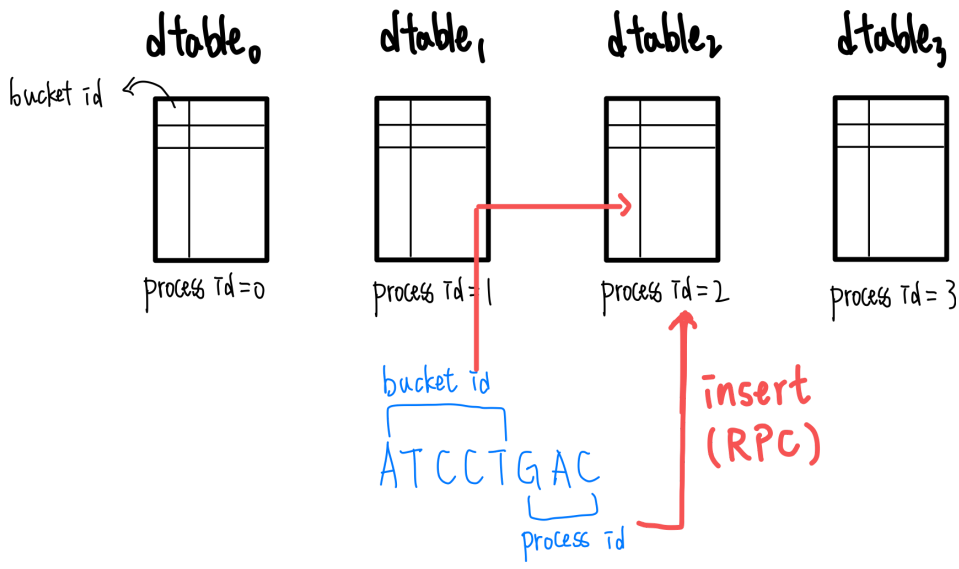


Fig3: How we insert a contig to the hash table

When we want to insert a contig to the hash table, we first locate the process by the last few bits, and locate the bucket index by the other bits. We will call an RPC to the other process and pass the contig as an argument, so we can insert the contig to the `dtable` on that process.

Find

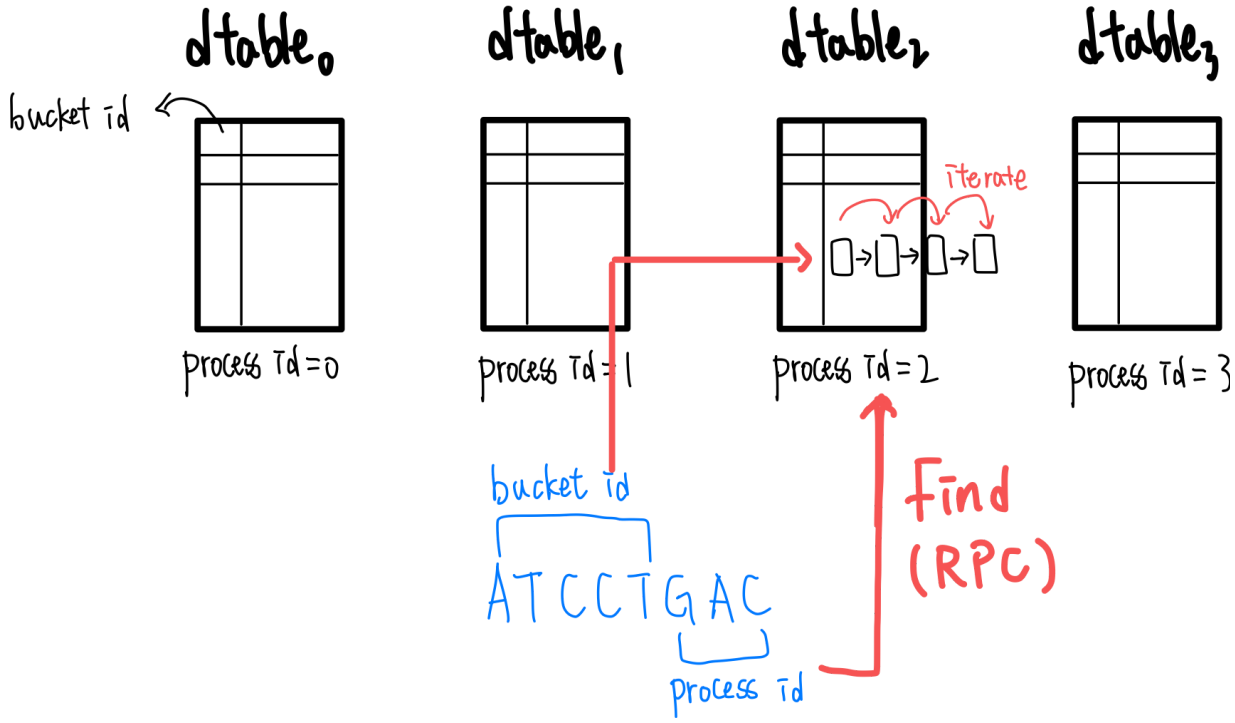


Fig4: How we find the next contig

To find the next contig, we use a similar method as `insert` to locate the process and the bucket. After we locate the bucket, we iterate through the pairs until we find the one with the target kmer.

Insert Many (Optimization)

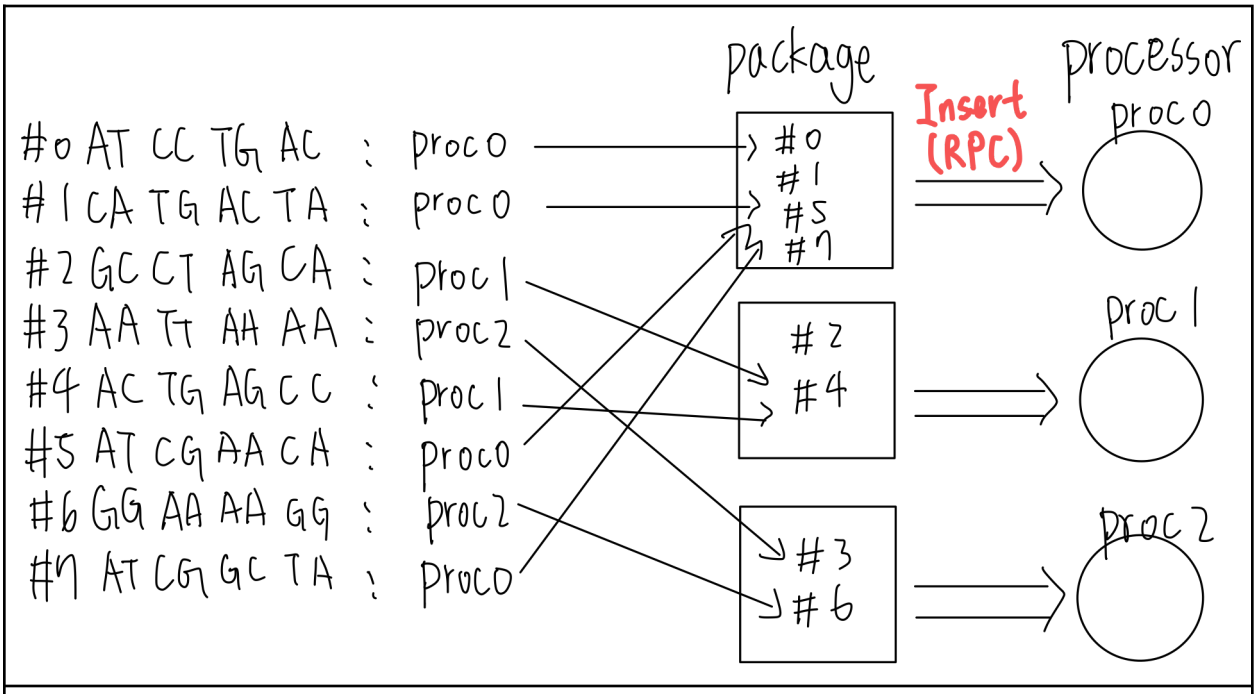


Fig5: How we insert many contigs

To reduce the time we call RPC, we can package the contigs which are sent to the same processor together, and send them once. We first iterate through the contigs and store them in `vector<vector<kmer_pair>>`. The index indicates the process id, and the value is the collections of kmer_pair. Later, we call RPC on the processors and send the collection of contigs to them.

Find Many (Optimization)

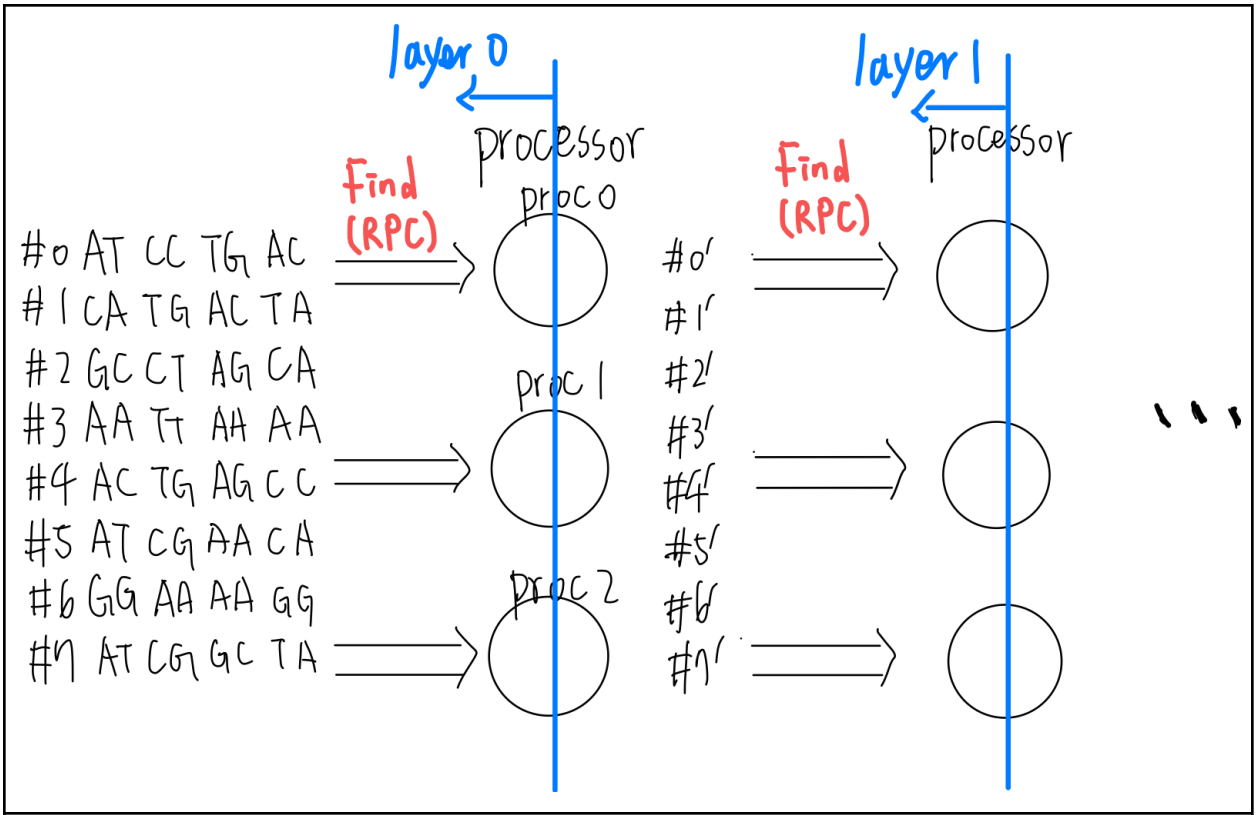


Fig6: How we find many contigs

The spirit of `find many` is similar to `insert many`: We package the contigs which are sent to the same process together and send them once. The process is similar to BFS. In the first layer, we package the contigs located on the same processor and find them, and we use them as the input of the second layer, so on and so forth.

Experiments

Perform multinode experiments using 1, 2, 4, and 8 nodes with 64 tasks per node (-S 4 should be set in your script, see job-cori-starter in the HW3 repository).

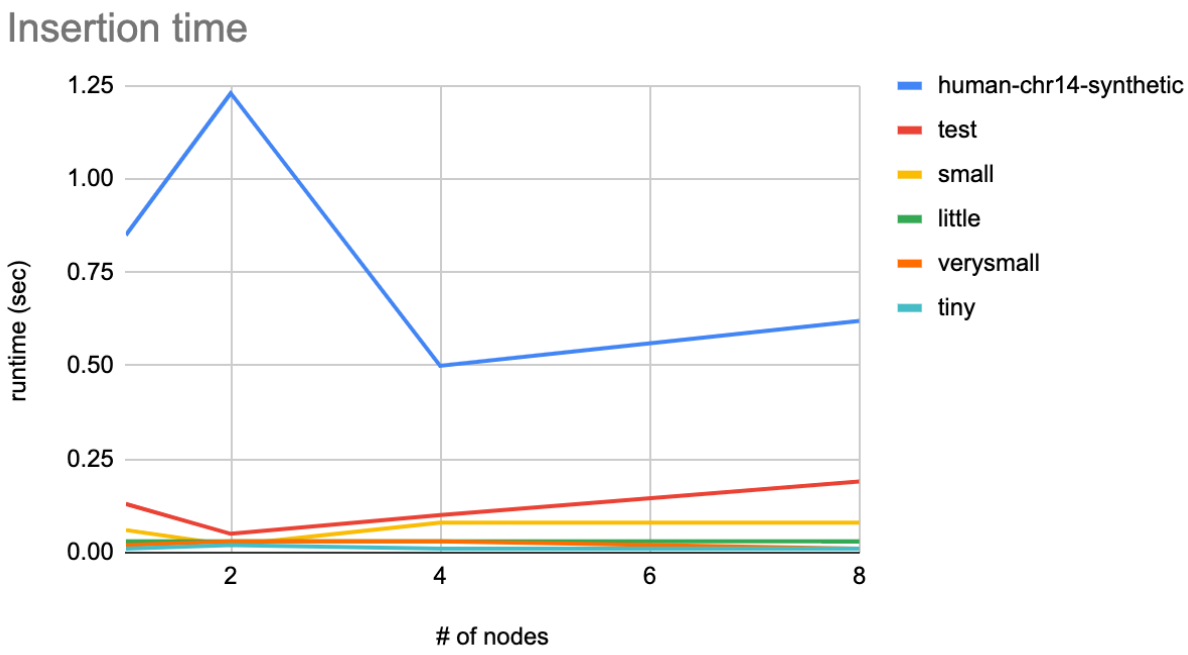
- Graph the runtime
- Graph the strong scaling efficiency

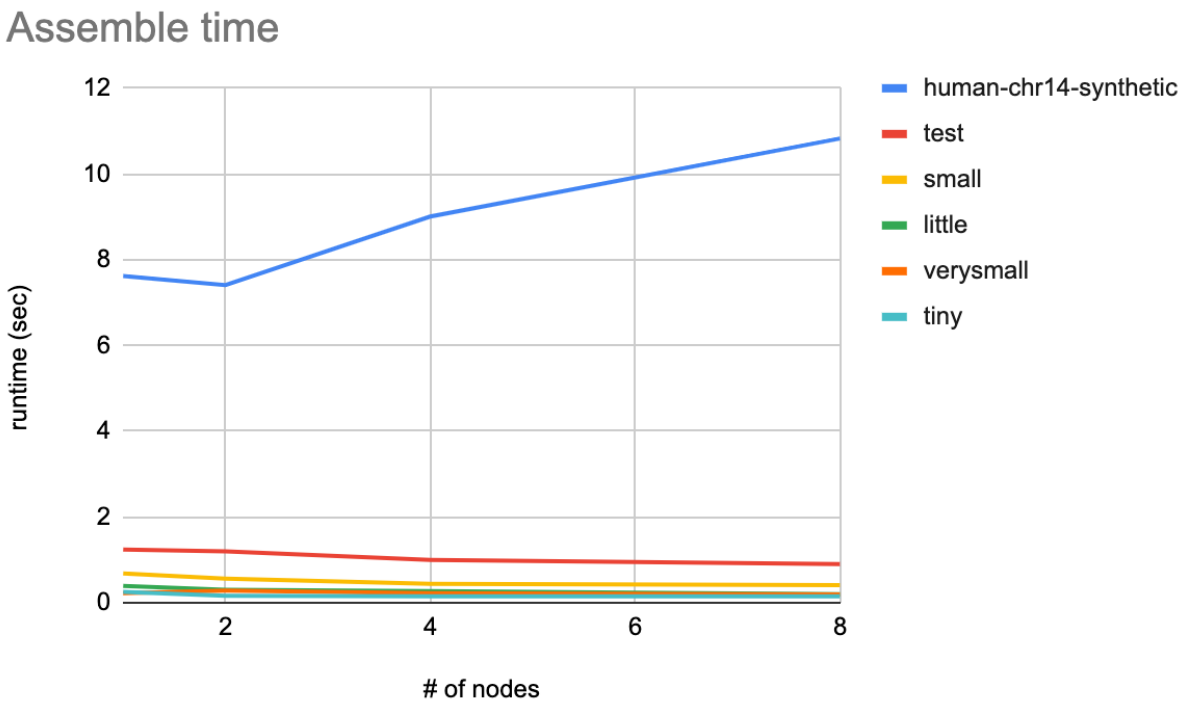
-N 1,2,4,8
-n 64 128 256 512

srun -N {1,2,4,8} -n {64,128,256,512} ./kmer_hash $SCRATCH/my_datasets/human-chr14-synthetic.txt

(inserting time (I)/ assembled time (A))

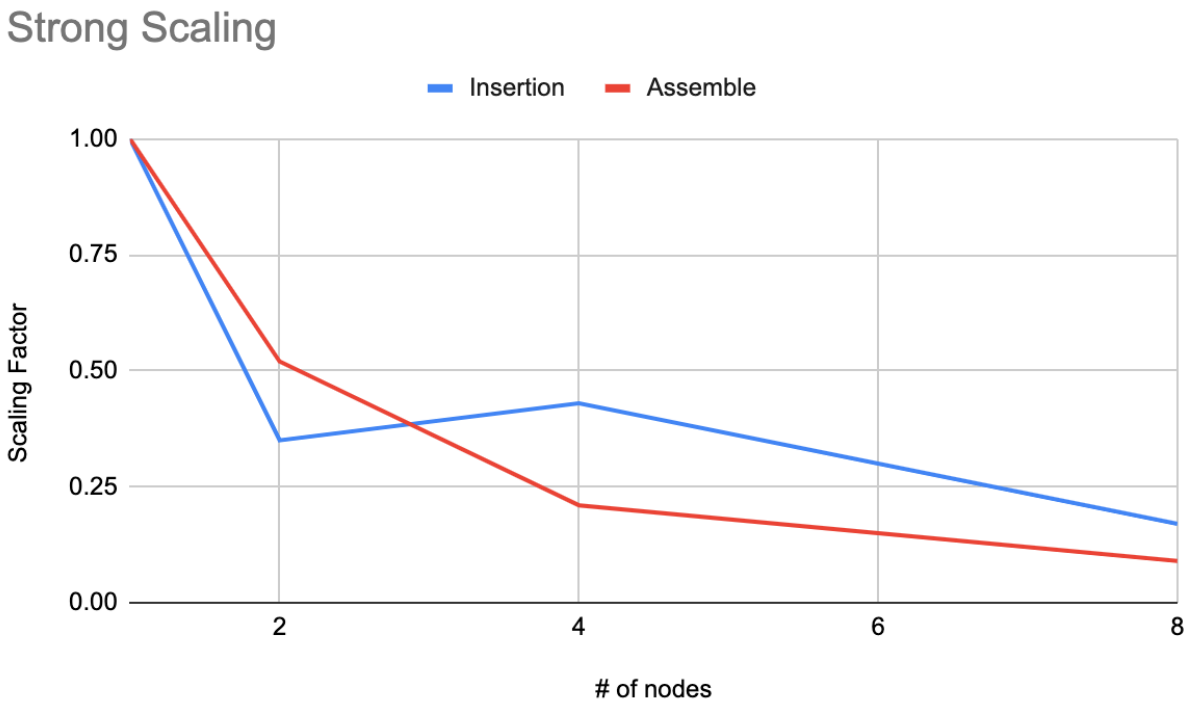
Dataset/ Nodes	**1 (I)**	**(A)**	**2 (I)**	**(A)**	**4 (I)**	**(A)**	**8 (I)**	**(A)**
human-chr14-synthetic	**0.85**	**7.62**	**1.23**	**7.41**	**0.50**	**9.01**	**0.62**	**10.83**
test	**0.13**	**1.24**	**0.05**	**1.20**	**0.10**	**1.00**	**0.19**	**0.90**
small	**0.06**	**0.68**	**0.02**	**0.56**	**0.08**	**0.44**	**0.08**	**0.41**
little	**0.03**	**0.39**	**0.03**	**0.30**	**0.03**	**0.27**	**0.03**	**0.20**
verysmall	**0.02**	**0.22**	**0.03**	**0.28**	**0.03**	**0.22**	**0.01**	**0.19**
tiny	**0.01**	**0.25**	**0.02**	**0.16**	**0.01**	**0.15**	**0.01**	**0.15**





Strong Scaling Efficiency: on human-chr14-synthetic **Dataset**

Dataset/ Nodes	**1 (I)**	**(A)**	**2 (I)**	**(A)**	**4 (I)**	**(A)**	**8 (I)**	**(A)**
Runtime (sec)	**0.85**	**7.62**	**1.23**	**7.41**	**0.50**	**9.01**	**0.62**	**10.83**
Speedup	**1**	**1**	**0.69**	**1.03**	**1.7**	**0.85**	**1.37**	**0.70**
Scaling	**1**	**1**	**0.35**	**0.52**	**0.43**	**0.21**	**0.17**	**0.09**

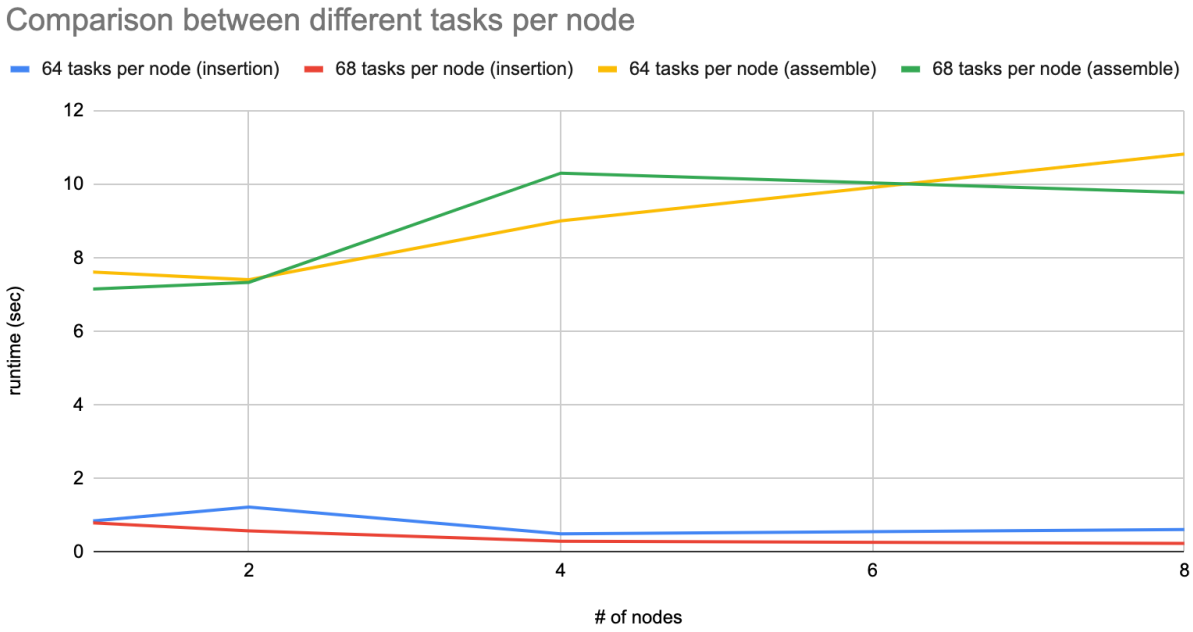


Changing from 64 tasks per node to 68 tasks per node.
Using dataset: human-chr14-synthetic
68/136/272/544

# of tasks per node/ Nodes	**1 (I)**	**(A)**	**2 (I)**	**(A)**	**4 (I)**	**(A)**	**8 (I)**	**(A)**
64	**0.85**	**7.62**	**1.23**	**7.41**	**0.50**	**9.01**	**0.62**	**10.83**
68	**0.80**	**7.16**	**0.58**	**7.34**	**0.30**	**10.31**	**0.24**	**9.78**

***Not Using Batching technique 64 tasks per node**

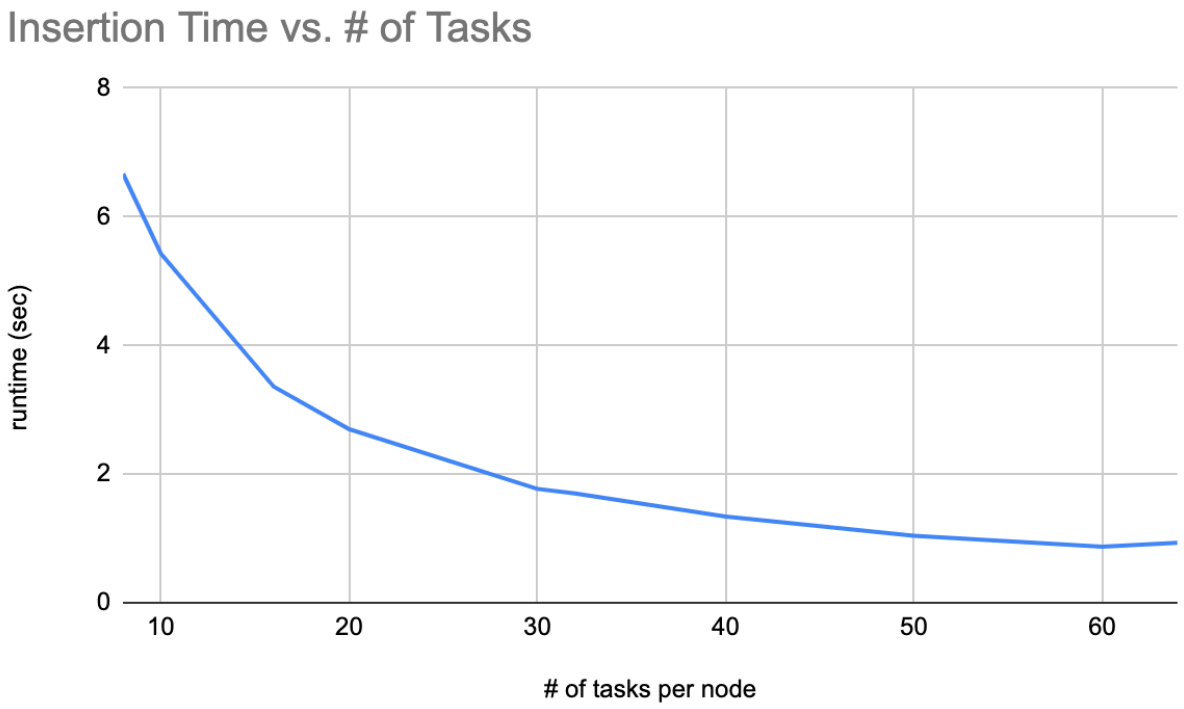
Dataset/ Nodes	**1 (I)**	**(A)**	**2 (I)**	**(A)**	**4 (I)**	**(A)**	**8 (I)**	**(A)**
human-chr14-synthetic	**11.39**	**28.11**	**21.82**	**44.97**	**16.69**	**33.65**	**9.84**	**19.81**

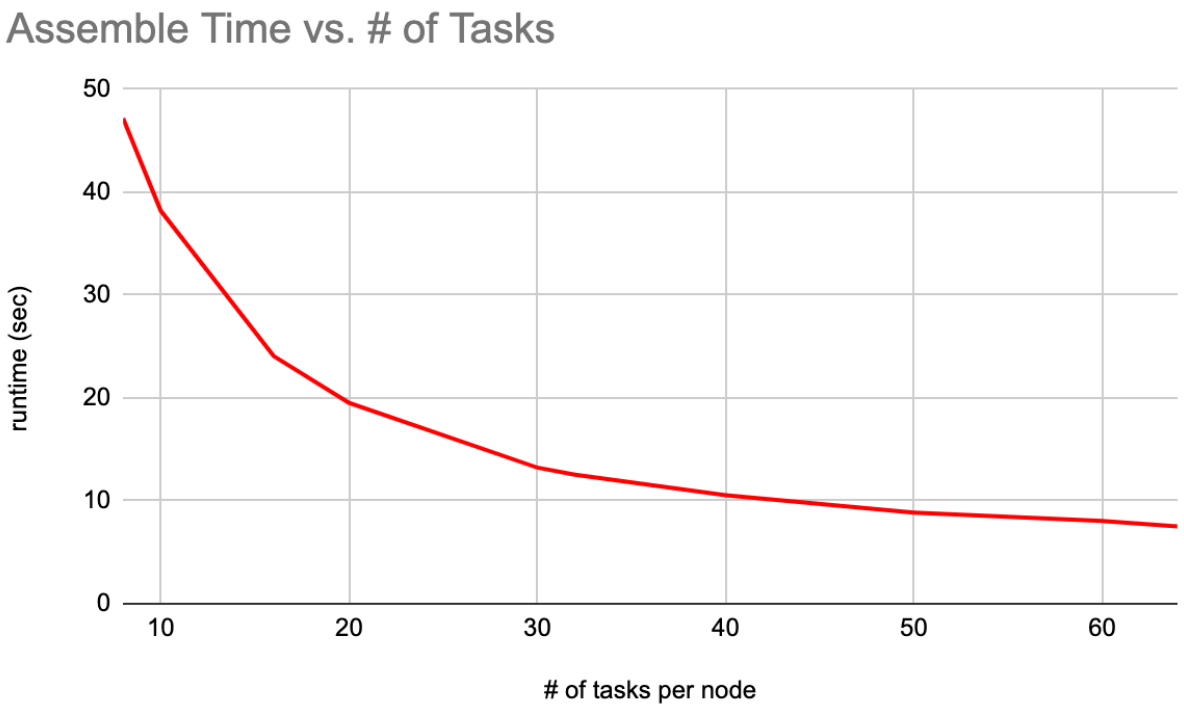


Run intra-node experiments using 1 node and varying the number of ranks per node (don't just use powers of two tasks per node). Also, see the note right below about memory segment size.

Using dataset: human-chr14-synthetic

# of tasks per node	**10 (I)**	**(A)**	**20 (I)**	**(A)**	**30 (I)**	**(A)**	**40 (I)**	**(A)**
	5.43	**38.20**	**2.70**	**19.52**	**1.77**	**13.24**	**1.34**	**10.54**
	50 (I)	**(A)**	**60 (I)**	**(A)**	**8 (I)**	**(A)**	**16 (I)**	**(A)**
	1.04	**8.86**	**0.87**	**8.04**	**6.67**	**47.18**	**3.36**	**24.07**
	32 (I)	**(A)**	**64 (I)**	**(A)**				
	1.70	**12.54**	**0.93**	**7.52**				

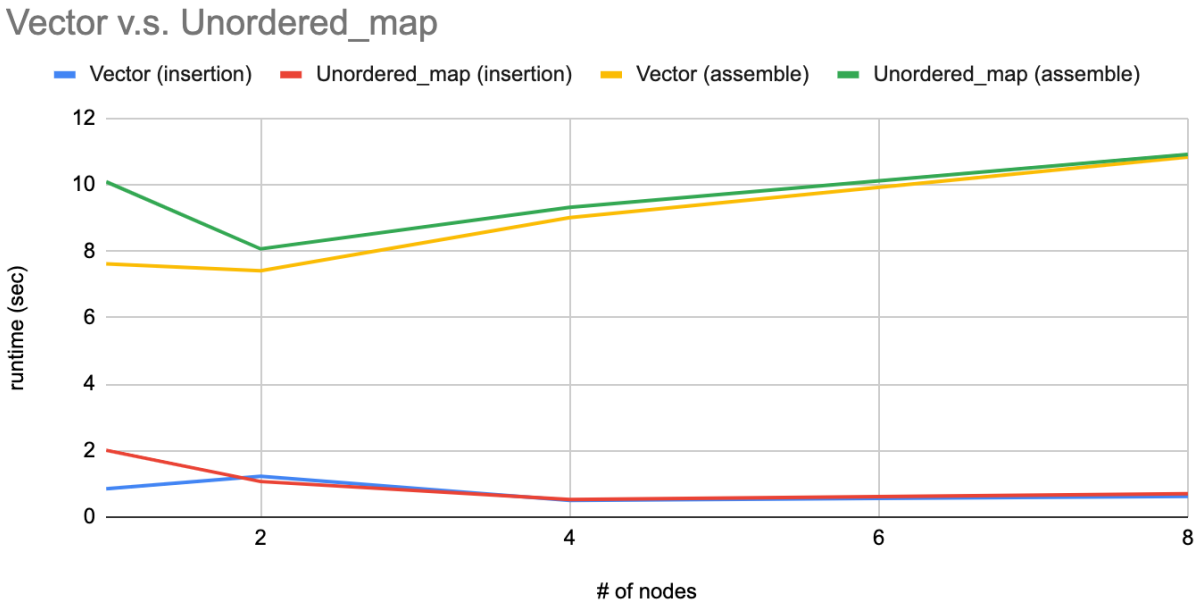




***Umap (64 tasks per node)**

Dataset/	**1**		**2**		**4**		**8**	

Nodes	(I)	(A)	(I)	(A)	(I)	(A)	(I)	(A)
human-chr14-synthetic	**2.014620**	**10.089655**	**1.068563**	**8.067453**	**0.533904**	**9.323776**	**0.701469**	**10.912894**



Discussion

How using UPC++ implemented your design choices. How might you have implemented this if you were using MPI? If you were using OpenMP?

By UPC++ design, the distributed object comes in handy. As each process needs to own a separate hash table while in the logical view these tables must be logically one big table. Therefore, naturally, we choose to use the UPC++ built-in distributed object and RPC to achieve this goal.

- If we were using MPI:
  In this scenario, we will still let each process own a `vector<vector<...>>` hash table. However, because there is no RPC concept (one-sided) in MPI, we will need to turn the RPC concept into `**MPI_Alltoall**` when we use MPI. Note that we use **Alltoall** because each process might receive (key, value) for any other process. For example, say, process 0 wants to insert a (key, value) pair into process 1's hash table. Each process will all use `**MPI_Alltoall**` to send the (key, value) while simultaneously expecting to

receive (key, value) from all other processes. This can also apply to finding (key, value), where we will need `MPI_Alltoall` in replacement of RPC's concept.

- If we were using OpenMP:
  In this case, because each thread will be able to access a shared memory space in the process, we can only store a big hash table in this process and distribute works among threads. The way we can do this in OpenMP is that assume we can **# of threads = num_threads** and we have **# of (key, value) to insert = num_insertions**, **we can let each thread insert (num_insertions / num_threads) (key, value) pairs so that the work is perfectly balanced among the threads (we might need some locking mechanism to avoid race condition)**. This idea can also be applied to finding (key, value).

Contribution

Tzu-Chuan: Implementation
Chin-An: Text of the report
Byron: Illustration of the report