# CS267 - Homework 3: Parallelizing Genome Assembly

Quentin Nicolas - Ankur Mahesh

March 2023

# I Description of Implementation

## I.1 The "data" and "used" array structures

Our hash table is organized in a very similar way to the serial code: conceptually, we are still using two arrays, one of `kmer_pair` objects (call it the "data" array) and one of `int` to keep track of the used/unused slots (call it the "used" array). The difference is that these arrays are distributed across all processes, in the shared memory space. Each of the two arrays is divided evenly across processes (each rank hold arrays of size `slots_per_rank` = `hash_table_size/upcxx::rank_n()+1`). We will use quotes to refer to the "data" and "used" arrays because they are not proper arrays in the code - they are a conceptual concatenation of sub-arrays held by each rank.

Two arrays of UPC++ global pointers, `data_pt` and `used_pt` (of size `rank_n()`), point to the heads of the sub-parts of "data" and "used" held by each rank. Every rank hold a copy of `data_pt` and `used_pt`. Some arithmetic is then used to find which global pointer holds a given index of the "data" or "used" arrays.

Let's take an example: the `i`th element of "used" is held on rank `i/slots_per_rank`, at the position `i` mod `slots_per_rank`; hence, the global pointer to this element is obtained by adding `i` mod `slots_per_rank` to `used_pt`[`i/slots_per_rank`].

In order to initialize the global pointers, we used UPC++ distributed objects. A distributed object of `upcxx::global_ptr<int>` is created to initialize one global pointer per rank; then we loop over each rank to fetch the global pointer it holds and add it to the `used_pt` array. The same procedure is used to populate the `data_pt` array.

## I.2 Inserting in the hash table

The procedure for inserting is the same as the starter code: we use open addressing with linear probing. Because kmers are now added in parallel, we want to avoid data races. Data races are only expected to happen on the "used" array: once it has been determined that a slot is free, only one process is going to try and access that slot in the "data" array.

In order to avoid data races when inserting, we use a UPC++ atomic domain with the `fetch_add` operation. The insertion procedure is as such: we start with one potential slot (defined from the hash of the current kmer, as in the starter code) and atomically add 1 to "used"[slot] (really, `used_pt`[`slot/slots_per_rank`] + `slot` mod `slots_per_rank`). If this call returns 0, it means that the slot was unused before we added 1 to it – hence, we can go ahead and emplace the kmer in the "data" array. If not, the slot was already used - hence, we go ahead and check the next slot, etc. until we find a free slot. Each check is performed by the `request_slot` method using one atomic `fetch_add` call.

In order to emplace a kmer once a free slot has been found, we use a RPUT call. Because there is no need to wait for the result of this call to proceed, we do not explicitly wait on it – the final barrier at the end of the insertion process ensures that it will complete.

### I.3 Reading from the hash table

The process of finding an element in the hash table is quite similar to the insertion: start with the first potential slot, (check if it is used), check whether the kmer contained there is the desired one, and if not, go to the next slot until it is.

Checking whether a slot is used is actually not needed, because the insert process checks slots in the same order as the read process. Hence, our submitted version commented that part out. However, the graphs shown on this report did have this part built in; in order to check whether a slot was used, we were using an atomic `load` to read from the "used" array. **Getting rid of this unnecessary check sped up the code by 20-30% compared to the graphs presented here**. It did not change the bad scaling we obtain when increasing the number of nodes for a fixed number of tasks per node.

In order to read from the "data" array, we simply use a RGET. Because we need to check right away whether the retrieved kmer is the one we are looking for, we immediately wait on this RGET.

### I.4 Final notes on the implementation

Most of the changes we made were in the `hash_map.hpp` file. The `kmer_hash.hpp` is unchanged except for two print statements and a call to `hashmap.ad.destroy()` to destroy the atomic domain. This reflects the fact that speedup is obtained from changes in the implementation of the hash table, not in the contig assembly algorithm itself. We also note that we did not implement a version where inserts are performed in batches.

## II UPC++ design choices

UPC++ allows for relatively cheap communication, hence we were able to obtain decent results even when adding elements one by one to the hash table. Because of the use of global pointers, the HashMap read and write leveraged the ability to put to / read from remote locations.

In MPI, it would have been necessary to insert elements in batches to obtain good performance. During the insertion step, each rank would hold $N_{\text{ranks}}$ buffers holding kmers (and their respective slots) to be inserted into the part of the "data" array held by the other ranks. Those buffers would be communicated once, in an all-to-all fashion, at the end of the insert step, minimizing the number of messages, hence latency. Since MPI does not allow for global pointers, sections of the hashmap would have to be sent and received by processors as necessary. When accessing elements in the hash table to build the contigs, there would be no obvious way to reduce the number of messages. One-sided communication could be used to reduce costs.

In OpenMP, the implementation would have been very similar as in UPC++: the "data" and "used" arrays would be shared between threads, and "used" would be accessed using atomics during the insertion step. There would be no need for pointer arithmetic. The main issue would be the lack of scalability, as it would be limited to one node.

## III Strong scaling Experiments

### III.1 Scaling on Multiple Nodes

Strong scaling on multiple nodes is evidently bad (see Figures 1 and 2). We were not able to obtain decreasing wall times when going from one to two nodes (ideal scaling would be a 2x speedup - we get a 5x slowdown). This seems to indicate that the inter-node network communication overhead is very important. In fact, the benefits of increased processors for dividing up the task is not as large as the overhead with multiple nodes. On the smaller test dataset, which involves less computation, the performance is flat when increasing for 2 to 8 nodes - indicating that communication costs are dominating. On the chromosome 14 dataset,

we get increasing performance beyond 2 nodes (1.4x speedup per doubling nodes), indicating a decrease in computation time as more ranks are added.

Using 60 tasks per node rather than 64 overall slightly slows down the performance. This indicates that inter-node communication is the bottleneck: less tasks per node reduces the number of available ranks, increasing computation costs, and the reduction in communication that comes with it does not compensate.

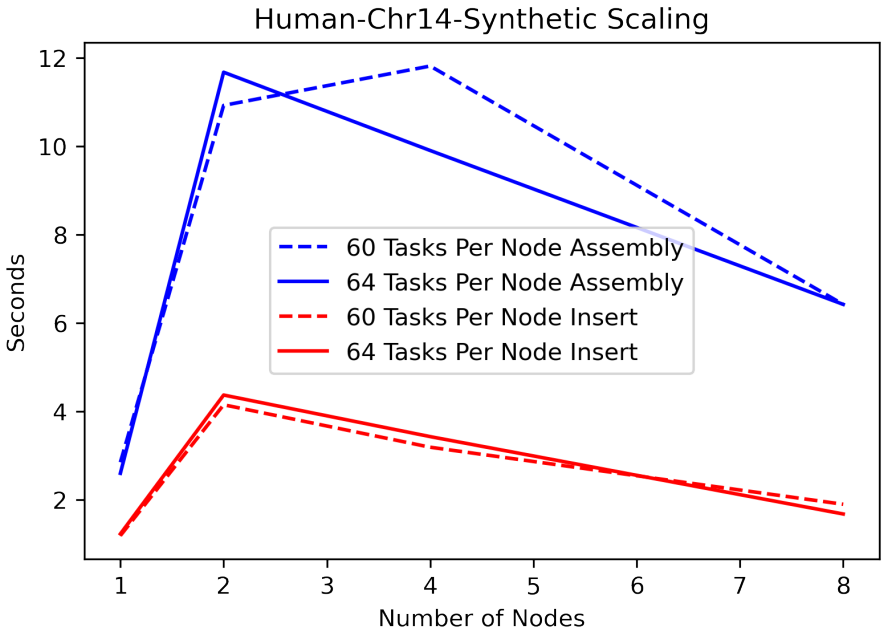


Figure 1: Strong scaling on Human Dataset with fixed numbers of tasks per node, varying node count.

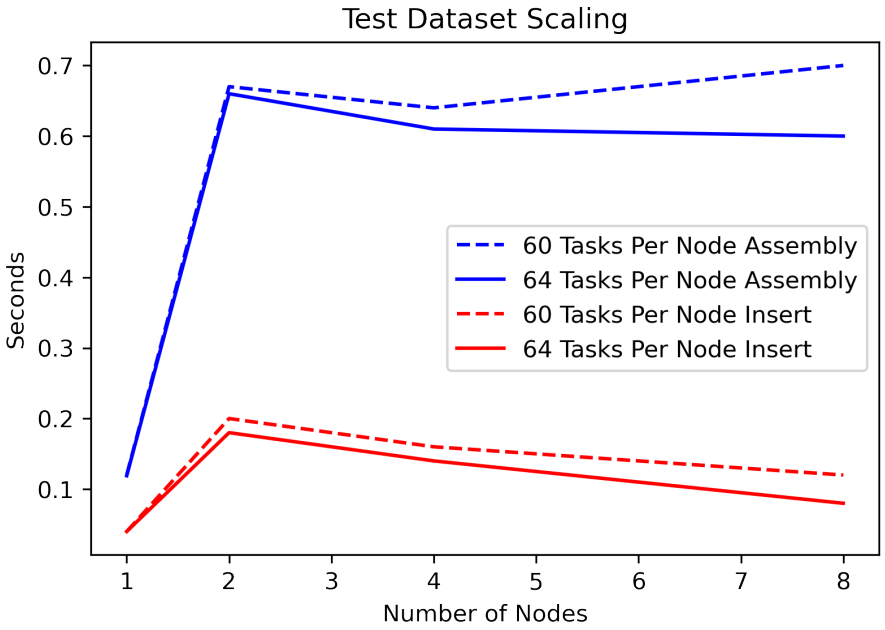


Figure 2: Strong scaling on Test Dataset with fixed numbers of tasks per node, varying node count.

### III.2 Intra-Node Scaling

One node with 64 ranks yields the best performance we got, around 2.5 seconds of total assembly time. The strong scaling obtained when varying the number of ranks from 1 to 64 is close to optimal with the large chromosome 14 dataset, that is more compute-bound. On the smaller test dataset however, communication costs are quite significant and the scaling isn't perfect (especially from 1 to 2 ranks where it is flat).

On a single node, we needed to increase the memory. For 30,32,40,50,60, and 64 tasks, we were able to use the default values for UPCXX_SEGMENT_MB and GASNET_MAX_SEGSIZE. However, for 20 tasks, we had to increase the values to 256MB and 4GB. For 10 tasks, we had to increase to 512 and 6G. For 4 tasks, we increased to 1024 and 32G. For 2 tasks, we increased to 2048 and 32G. With fewer tasks, each task needed more memory in order to perform the computation, and the amount of communication between tasks increases (forcing to increase the Gasnet bandwidth limit). With more tasks, it was tractable to use remote operations, and there were enough UPC segments to support the calculation on the whole dataset.

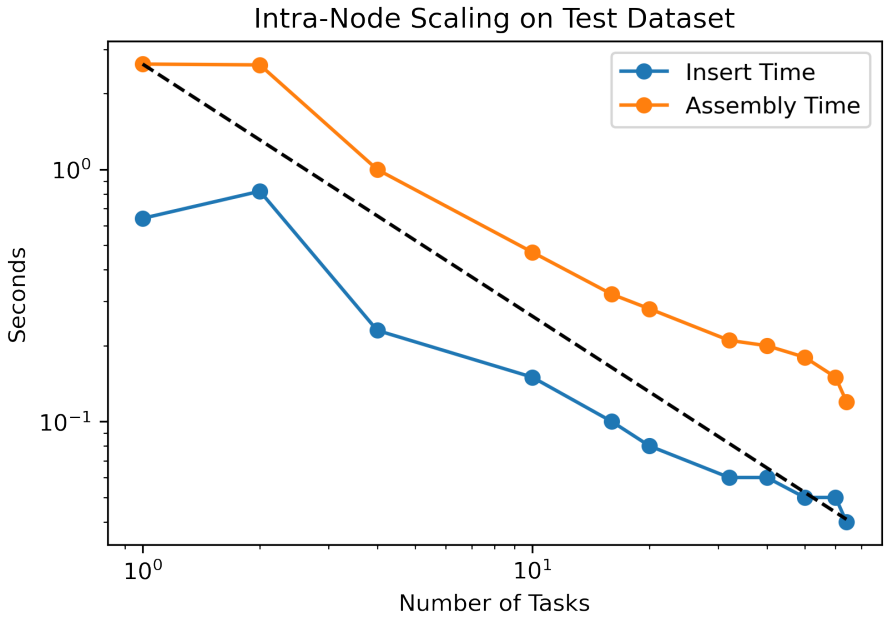


Figure 3: Strong scaling on one node with the Test Dataset.

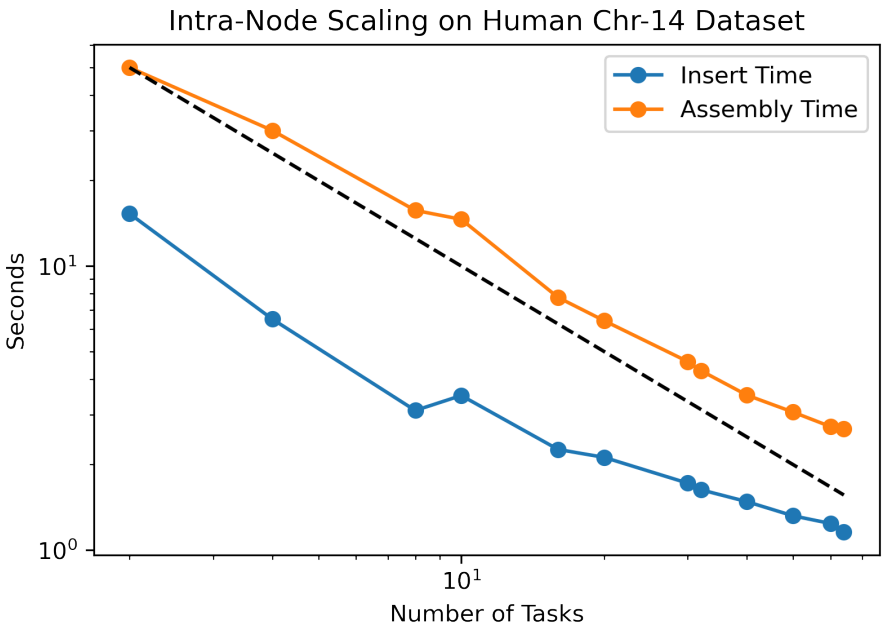


Figure 4: Strong scaling on one node with the Test Dataset.