# CS 267 Homework 3

Xingyou Song [*], Yao Yuan [†], Jingbo Wu[‡]

March 23, 2018

## 1 Algorithm

For each rank, it calls upc++ *newarray* function to allocate a shared global hashmap. After each process allocates their own global hashmaps, they broadcast the global address pointers to other ranks so that each rank is aware of others' global memory addresses. Besides, each rank creates a local LockerTable to make sure that no two ranks would try to write the same slot at the same time (race condition).

### 1.1 Data Structures

- Hash Map consists of n (number of ranks) global kmer arrays. Each rank is responsible for its own global arrays.
- Locker Table is a local char array for each rank. It is used to ensure that each corresponding slot in the global memory would only be written once (no duplicate write operations by different ranks). Whenever a rank try to insert its kmer into the global memory, it would first issue an rpc call the corresponding rank to require a valid slot. Only when the caller receives a success signal back from the other rank together with an assigned position, it would then do an rput operation. The locker table is implemented via C++ atomic or operation.

### 1.2 insert_kmer function

In insert_kmer, we do the following locally:

- Compute a hash by calling kmer.hash(), and mod this by the size of the hash table to compute a position in the memory.

[*]xsong@berkeley.edu - Xingyou wrote the report, presented techniques and strategies for optimization, and coded some parts

[†]yao_yuan@berkeley.edu - Yao coded a significant part of the code in C and tested using various compiler settings

[‡]wu622@berkeley.edu - Jingbo mainly contributed to the report and coding.

- Compute which rank this position belongs to, and the relative address of the rank's global memory (respectively, variables whichProc and whichSlot). Issue a rpc command to the corresponding rank to check if the corresponding slot is open.
- Use C++ atomic function to check whether the corresponding slot is open or not. If not, use linear probing to probe the next slot until an empty slot is found. If all the rank's slots are full, a "Not Found" message is returned to the rpc caller and the caller would issue another rpc insert kmer command to the next rank. This process is repeated until an open slot is found. One thing that is worth mentioning is that there are at most up to the number of ranks RPC calls for one insertion operation.
- If we succeed in finding an open slot, insert this globally.

### 1.3 find_kmer function

We will also do the following for find_kmer:

- Compute the hash and position, like we did with insert_kmer.
- Retrieve a kmer using rget() function. If it is not equal to the key, increment the probe variable by 1 (which moves the position by 1), and continue checking until success or we go over the hash_map limit.

Note that when we initialize, we collect all of the meta data (i.e. size of each processor's collection, their ranks, their global hash map pointers etc.) by a master call, with this passed as a global pointer to each processor.

### 1.4 Explanation

There are multiple ways to do this, but because of the hash collision nature, it was only sensible to split the memory to be allocated for nodes, and take the communication penalty for when a hash belongs to another node's memory, which is perfect for the UPC++ case. Besides, we do a greedy probing algorithm within one rank's locker table to minimize the number of rpc functions.

## 2 Results

### 2.1 Single Node, Varying Processes

1 From our results, we see that the strong scaling for all datasets are very similar. The strong scaling has a steep drop for few processes on a single node, most likely because the overhead of communication is too significant to be neglected. However, we see the slope is rather flat when we have more than 4 ranks. One possible explanation is that the extra communication overhead is offset by the distributed workload. (the number of rpc calls per insertion is always around 1-2. So the more evenly distributed the workload is, the more efficient the code is). On average, we have a scaling efficiency of 0.48.

### 2.2 Multi Node

2 For the multi-node case, the scaling remains similar for all datasets. Again, we see a very sharp drop in scaling due to both collision overheads and node-communication (based on the hash assigned to different nodes).

Note that there is a strange peak at nodes = 2 for all test cases; this is the beginning of multi-node parallelization, and it suggests that communication complexity between nodes (through the ethernet) dominates when we start parallelization.

Also, note that for this graph, we don't see significant speed-up for extra nodes, which is possibly explained by two reasons.

Firstly, as the whole hash map is further divided into small chunks when the number of ranks increases and the linear probing method tends to suffer from clustering. It is highly possible that some rank's hash maps might be full really early in the insertion procedure, thus suffering from long probing time (kind of similar to a naive for loop).

Secondly, the access of other nodes' global memory is not cheap at all. Compared to rget / rput on a single node across different ranks, doing rget / rput remotely from another single node suffers from large communication overhead. To mitigate the issue, one possible solution is to create some minibatch or the requests so that we can better make use of the TCP packets padding, thus reducing communication overhead. On average, we have a scaling efficiency of 0.24.

## 3 Other frameworks

In other frameworks, we run into the same architecture as before - we must assign each memory "worker" (either a thread or process) to its own unique set of memory locations; this is the cleanest way to do so, or else there will be very large memory conflict overheads requiring locks (i.e. who writes first, and who reads first on the same location?)

### 3.1 MPI

In the MPI setting for synchronized and parallelized hashing, the clearest way to do so is to allow each processor hold a section of memory for the hash table, just like we did with UPC++. However, there is a problem when it comes to communication: i.e. when one processor detects that the hash position value does not correspond to its own memory location, it must pass this signal to the correct processor (which is not known ahead of time), which forces all of the processors to wait for a message (about whether they do nothing, or need to perform a memory insert). A similar problem exists for the find_kmer function.

Perhaps a better method exists, where we allocate a few processors (called the masters) to process the parallelized hashing requests, and send the memory requests to the "worker" processors for memory operations. The problem with this occurs when two master processors send in 2 work requests to a worker at the same time, making synchronization difficult.

### 3.2 OpenMP

For OpenMP, each thread possesses its own memory locations in a global memory. A similar problem exists as seen above - even if we had "master" threads to control the flow of memory requests, there would be multiple synchronization issues, requiring many barriers, which decreases performance.

## 4 Bonus Questions

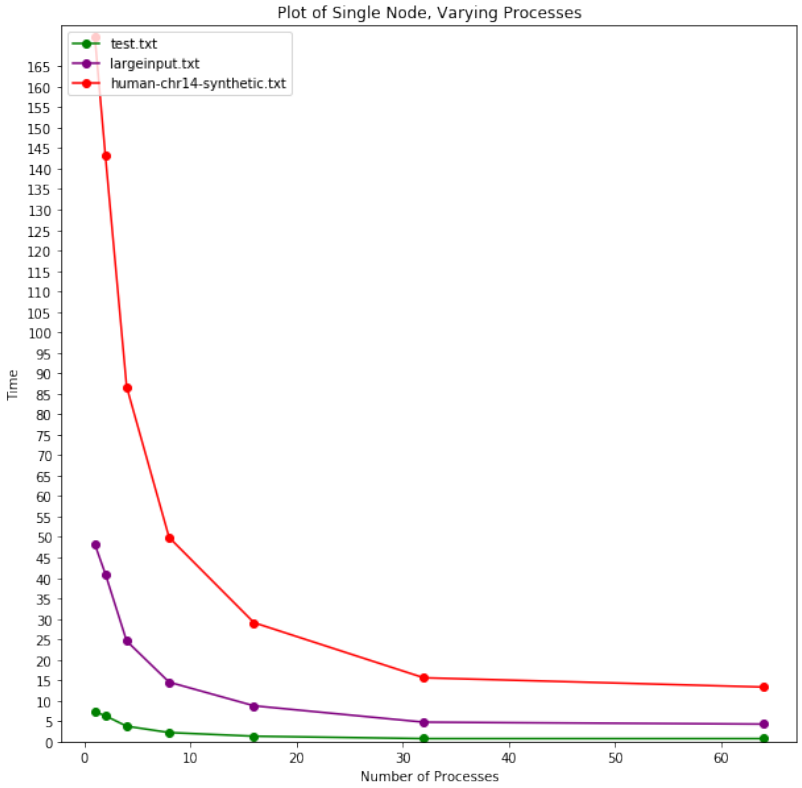
### 4.1 Short Answer - Contig F Extensions

Then it will be impossible to know where to start and where to end. In other words, we need to have a separate list telling us which kmer is the starting point and which kmer is the end point. If there is a random base whose kmer is not within our dataset, we have two options. We can either pretend there is one (as we know what that kmer should be) or we just abort the generation of that kmer. In other words, we discard the half-complete kmer and move to the next one. It will not affect my code a lot as the hash map find function will return false if the kmer is not within the dataset and we can simple continue the iteration over the start_kmers while aborting the current contig.

### 4.2 Short Answer - Load Balancing

The idea behind it is to distribute the workload evenly. One simple approach is to let assign a single process as the coordinator and collect all the start_nodes from other peers (including its own). Then, it can evenly split the start_nodes and distribute them to other processes. It is easy to implement but might not be time efficient as a process has to receive and distribute all the start_nodes.

Figure 1: Single Node, Varying Processes



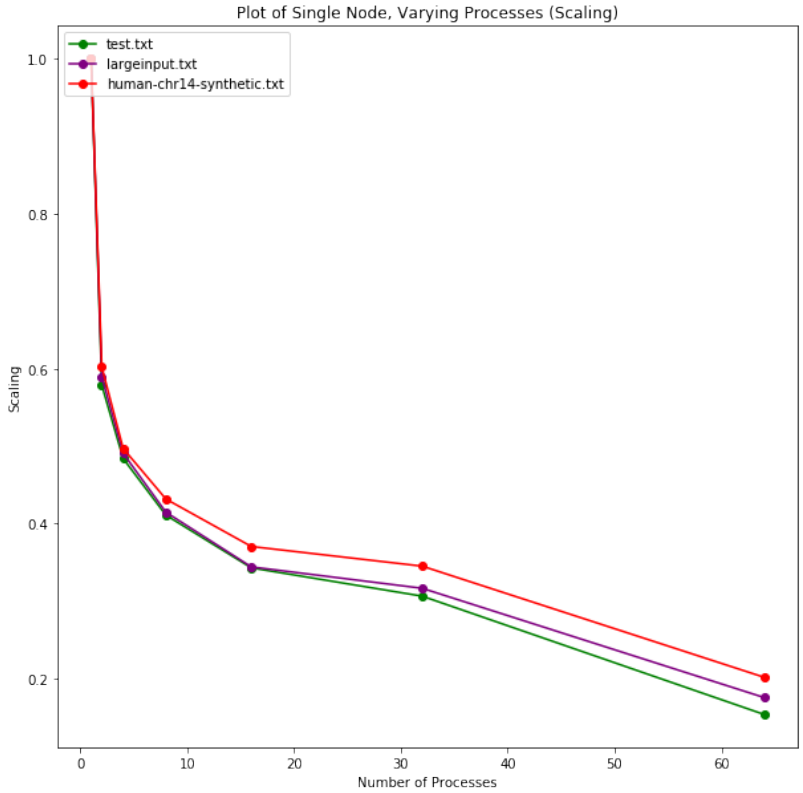



Figure 2: Multi-Node, 32 Ranks Per Node