CS 267: Parallel Computing **(Due: 05/13/20)**

Homework Write-Up 3

Instructor: A.Buluc, J.Dummel, K.Yelick *Name:* Bo Li, Philippe Ferreira De Sousa, Jingran Zhou

1 Members & contribution

- Philippe (non-EECS/CS major): early debugging of the initial version for the midterm checkpoint, worked on scaling simulations with out-of-memory issues and plotted graphs.
- Bo Li: implemented `insert()` function; debugged code; run scaling experiments; wrote Section 2 and 5.
- Jingran Zhou: chose the overall architecture and defined class members; wrote constructor and destructor; implemented `find()` function; debugged code; created helper functions to refactor the code; wrote Section 3, 4, and 5.

2 Scaling experiments

Here are the experiment results we run on both `test.txt` and `human-chr14-synthetic.txt`. The results show the simulation time varies with different number of processes on a single node and different number of nodes.

Figure 1 shows how our code performs with varying numbers of processes on a single node. The specific values in this plot are presented in Table 1.

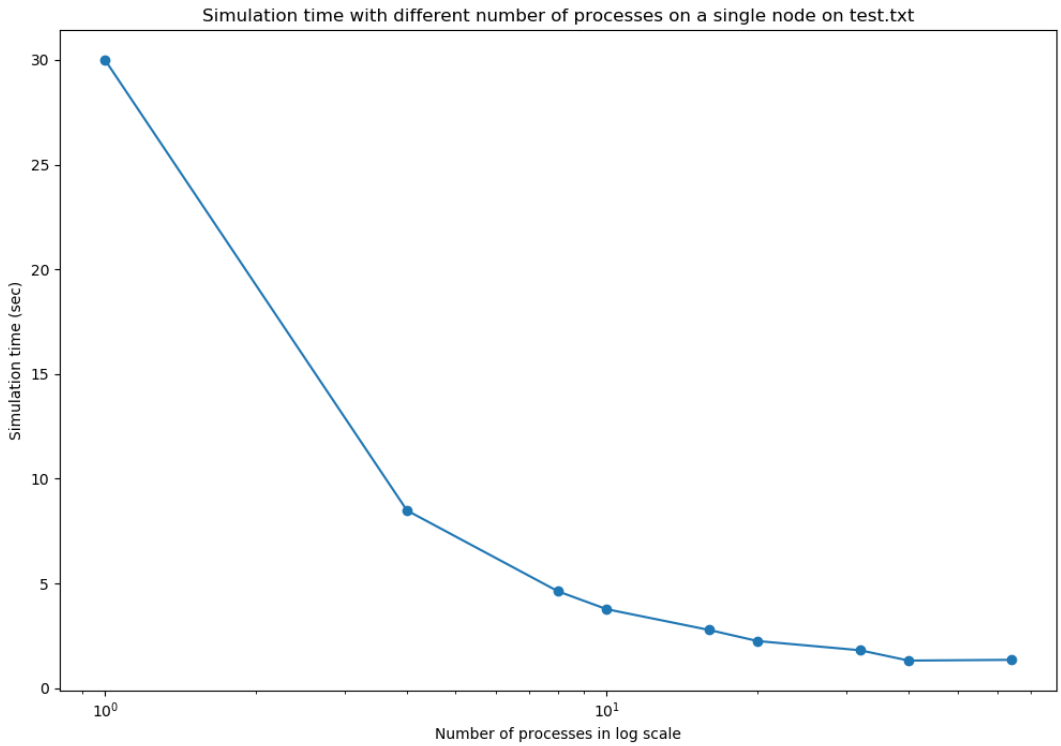


Figure 1: A plot in log-log scale showing our parallel code performance.

Number of Processes	Time (s)
1	29.996162
4	8.480678
8	4.620874
10	3.768400
16	2.777911
20	2.246753
32	1.807536
40	1.314068
64	1.350181

Table 1: Simulation time varies with the number of processes on single-node with `test.txt`.

Figure 1 shows how our code performs with varying numbers of nodes. The specific values in this plot are presented in Table 2.

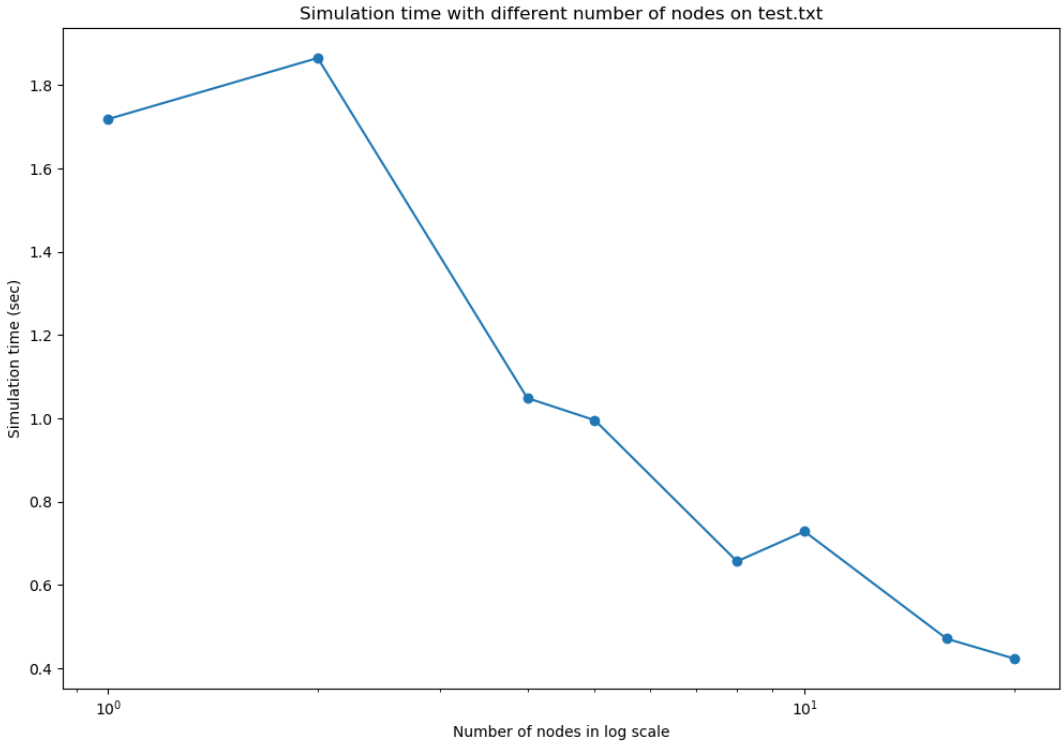
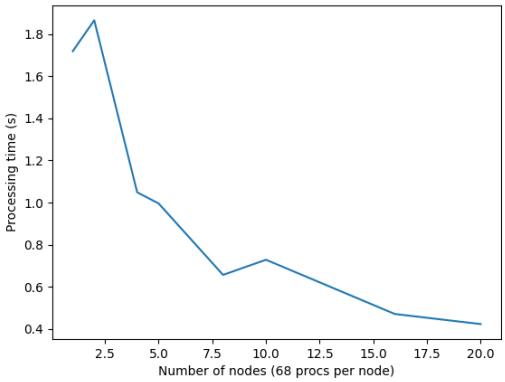


Figure 2: A plot in log-log scale showing our parallel code performance.

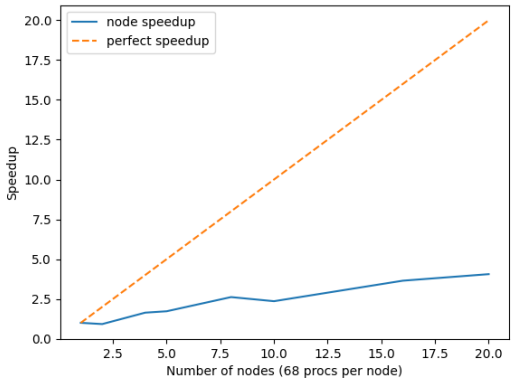
With the test.txt, we observe that the simulation time decreases as the number of processes increases on a single node. And it also decreases as we increase the number of nodes, with 68 tasks per each node. We can see that both cases, the relationship follows a good strong scaling.

Number of Nodes	Number of Processes	Time (s)
1	68	1.718625
2	136	1.865435
4	272	1.049049
5	340	0.995854
8	544	0.657025
10	680	0.728897
16	1088	0.471424
20	1360	0.423637

Table 2: Simulation time varies with the number of nodes with `test.txt`.



(a) Strong scaling for `test.txt`



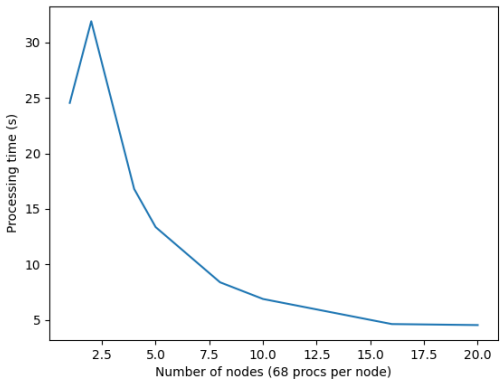
(b) Speedup for `test.txt`

Number of Processes	Time (s)
1	MLE
4	MLE
8	MLE
10	MLE
16	MLE
20	MLE
32	MLE
40	MLE
64	22.281353

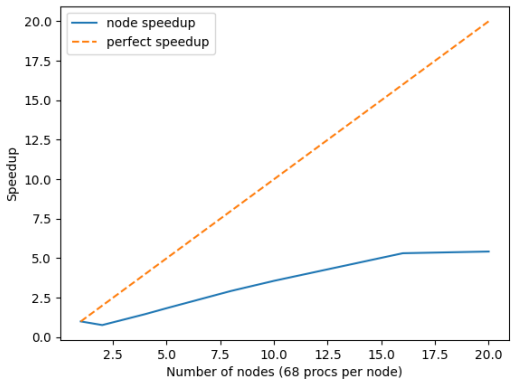
Table 3: Simulation time varies with the number of processes on single-node with `human-chr14-synthetic.txt`. Only with enough processes can the program run without memory limit exceeded (MLE) errors.

Number of Nodes	Number of Processes	Time (s)
1	68	24.558463
2	136	31.918422
4	272	16.796600
5	340	13.360135
8	544	8.384138
10	680	6.88025
16	1088	4.618405
20	1360	4.527592

Table 4: Simulation time varies with the number of nodes with `human-chr14-synthetic.txt`.



(a) Strong scaling for `human-chr14-synthetic.txt`



(b) Speedup for `human-chr14-synthetic.txt`

3 Implementation

Our hash table is a collection of arrays residing in the global address space provided by UPC++ and is accessible by multiple processes simultaneously. The architecture of our implementation is shown in Figure 5, featuring a hash table of capacity 10, distributed among three processes. Each process holds a different instance of the `HashMap` class (grey rectangles) that gives the process a "view" to the global hash table (green rectangle). Our hash table consists of two parts: occupancy statuses (blue array) and k-mer objects (orange array). The former indicates if a slot is empty (0) or occupied (1), and the latter is the actual value stored in the table.

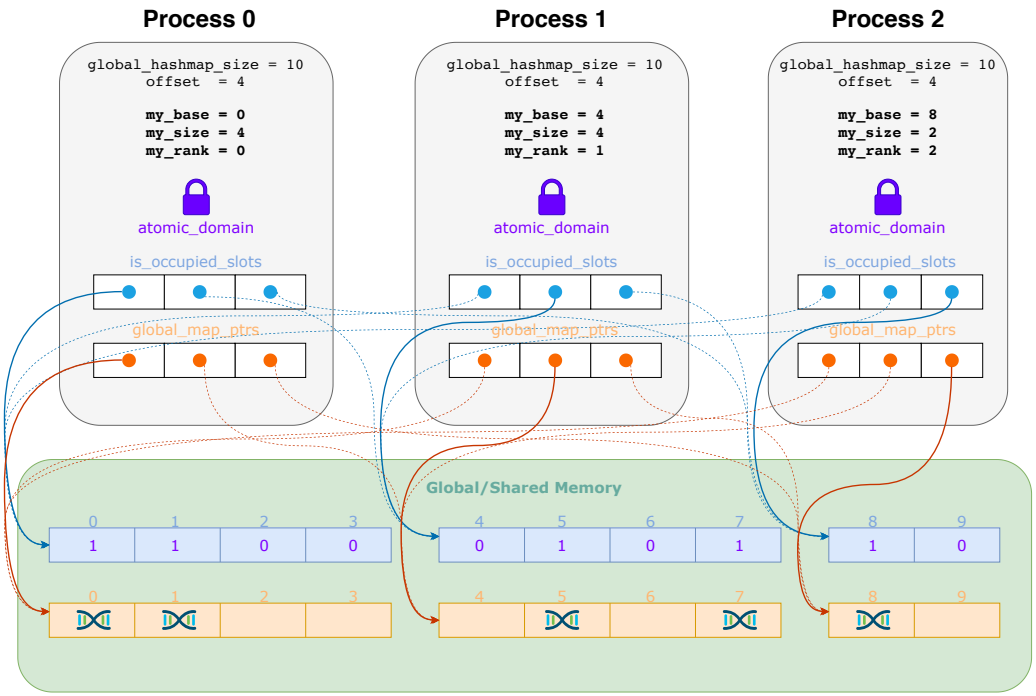


Figure 5: The architecture of our distributed hash table. The bottom green region is a global hash table of capacity 10 shared by three processes via global pointers (depicted as arrows); the blue array indicates if a slot if empty (0) or taken (1); the orange array contains the objects inserted into the hash table (i.e., k-mers). The top grey rectangles correspond to three `HashMap` instances, each held by a different process; a `HashMap` instance allows a process to keep track of the distributed hash table: `offset` denotes the number of entries between every two processes, `my_base` denotes the number of entries before this process, `my_size` denotes the number of entries created by this process, `atomic_domain` is used for atomic operations to prevent race conditions, and `is_occupied_slots` and `global_map_ptrs` contain global pointers to different parts of the distributed hash table.

The task of creating the hash table is distributed to each process evenly. Each process is responsible for creating a portion of the global hash table. For example, in Figure 5, a table of 10 entries is constructed collaboratively by three processes in chunks of size 4, 4, and 2, respectively. Naturally, each process retains a global pointer to the part of the hash table that it has created (shown as solid arrows). Then, the processes exchange their global pointers with each other via `upcxx::broadcast()`. With the global pointers shared by *other* processes (shown as dashed arrows), a process can also access any *other* parts of the global hash table created by *other* processes. Hence, each process has a global view of the entire hash map.

Our `HashMap` supports two key functions: `insert()` for inserting a k-mer pair into the hash table, and `find()` for finding a k-mer that matches a given key. To insert a k-mer pair, we obtain its hash code and convert that with modular arithmetic to an index for the global hash map. We can then insert the k-mer into the slot at that index using `upcxx::rput()`, should the slot be empty based on `is_occupied_slots`. However, if that slot has already been taken, we use linear probing to resolve the hash collision. To prevent multiple processes from inserting into the same slot concurrently, we use compare-and-swap (CAS), an atomic operation, to check and modify `is_occupied_slots`. If we cannot find any available slot, insertion fails. The `find()` function works in a similar way – we first convert the hash code of the query key to an index, then try to find a non-empty slot that matches the input key using linear probing starting from the index. To check if a slot if empty, we use the atomic load operation on `is_occupied_slots`.

Lastly, UPC++ has one noteworthy, bug-inducing (and moderately annoying) rule regarding atomic operations: "*User code is responsible for ensuring that data accessed via a given atomic domain is only accessed via that domain, never via a different domain or without use of a domain.*" Therefore, each process creates and owns a unique `atomic_domain` object (shown as purple locks in Figure 5) to enable the aforementioned atomic CAS and load operations. When the program ends, these `atomic_domain` objects will be destroyed in the destructor of `HashMap`.

4 Optimization

The initial version of our `hash_map.hpp` contained several `upcxx::barrier()`'s as a synchronization mechanism. These barriers ensured that our initial code could run correctly without concurrency issues. However, we also realized that barriers entailed significant performance overhead, limiting our execution speed by waiting for the slowest process. Hence, we set out to find a way to reduce the amount of barriers needed. After theoretical analysis and trial-and-error, we were surprised to discover that no barrier is needed in our `hash_map.hpp` at all. The removal of these unnecessary barriers yielded some performance gains: to process `human-chr14-synthetic.txt` on 20 nodes with 68 processes each, our initial code took 3.997 seconds, while the optimized barrier-less version only took 3.960 seconds.

5 Design choices

UPC++ is appropriate for this project because it supports Partitioned Global Address Space which provides a writing efficient way on distributed-memory. Because each process has a private memory as well as the access to the global address space. We can compute the hash value locally and then finish the insert and find task on the global memory with `rput` and `rget` operation. In MPI, we do not have this global address space. To communicate with other processes, each processor has its own copy of the whole hash table. When the results related to other processors are involved, we have to explicitly send and receive the message to other processors. Finally we need to combine the information from each processor to get the final hash table.

OpenMP, while also being a shared-memory programming model like UPC++, is arguably a simpler but less powerful alternative. Similar to UPC++, in OpenMP, the distributed hash table can reside in the global/shared address space, leading to less communication overhead compared with MPI. However, due to the nature of shared-programming, we need to address concurrency issues. If we use the popular critical section constructs in OpenMP, we would not have been able to achieve the fine-grained, element-level synchronization we have in UPC++ using atomic operations.