CS267-HW3

Yihua Zhou, Johan Fredrik Agerup, Justin Kalloor

April 2023

1 Problem statement

This problem is about using UPC++ to run de novo gene assembly. This process takes randomly divided sections (kmers) of a larger DNA strand to create a brand new DNA strand from scratch. It is a very interesting problem that can be parallelized easily, and we use UPC++ to help with this.

UPC allows the user to treat distributed memory as if it was shared memory. This allows for easier programming with the scalability of a distributed memory. In addition, we can use one-sided communication rather than the two-sided communication used in MPI, which removes a lot of the communication problems.

In particular, for de novo gene assembly, we needed to create a distributed hash table to store the different kmers in order for them to quickly be fetched in assembly. This means that chunks of the hash map are in different pieces of memory. To resolve conflicts, we use linear probing.

2 Hash Map Organization

Overall, our hash map is organized as a C++ vector of UPC global pointer to arrays containing the kmer characters. Each UPC process (rank) has a copy of the base of each global array. Each global array is created by one process (rank) but can be accessed by other ranks using a `rput` or `rget` call.

2.1 In use arrays

In addition to the data arrays, we have an array of booleans of the same size to indicate the occupation of a slot in the hash map. In order to be efficient, we use atomic operations `compare_exchange` and `load` in order to read/write into this array. If the use array is generated atomically, then adding the the data array can be done without any locking.

3 Algorithms

3.1 Initialization

1. Allocate C++ vector with number of ranks elements of global pointers that points to arrays representing kmer pair objects.
2. Allocate vector of number of ranks elements of global pointers to integers representing the occupancy of slots (1 for occupied and 0 for vacant.
3. Allocate futures for the above data structures.
4. Allocate local arrays for kmer pairs and slot occupancy, both of size `my_size = 1 + (size-1)/nranks`.
5. For all ranks: Broadcast local arrays of kmer pairs and slot occupancy integers to the futures.
6. For all ranks: Assign the local array values to the global data structures by applying `.wait()` on the future.

3.2 Insert

For both the insert and the find algorithm the variable `size` is the user defined size of the hash map.

1. Initiate insert position `pos = hash % size`, set initial position `start_pos = pos`
2. While `compare_exchange(pos)!=0` increment `pos` and set `pos = pos % size`, return false if at start position.
3. If true put kmer in global kmair pairs vector at `pos/offset + pos&offset` and wait for future (using `rput` and `.wait()` command).

3.3 Find

1. Initiate insert position `pos = hash % size`, set initial position `start_pos = pos`
2. While `compare_exchange(pos)!=0` increment `pos` and set `pos = pos % size`, return false if at start position.
3. If slot at `pos` is occupied return false.
4. Get kmer pair from global kmer pairs vector at `pos/offset + pos&offset` and wait for future (using `rput` and `.wait()` command).
5. Increment the position and set `pos = pos % size`.

4 Optimization attempts

4.1 Downcasting

Our first optimization attempt was to downcast the global data array if the current rank owned the global array that was attempted to be accessed. When using this array downcasting avoid communication between local and global memory for intermediate operations on the global array. When all the intermediate operations has been completed this is communicated back to the global memory. Due to the communication reduction downcasting should decrease the runtime significantly. However, it made no difference in the performance, showing that `rput` is already optimized to avoid unnecessary communication.

4.2 Removing wait() from Insert

A second optimization we tried was to avoid waiting for inserts to be added to the array. Since the used array was being handled atomically, we knew that the position we found would be available for our kmer, so we only had to barrier at the end of all inserts in order to maintain correctness. However, surprisingly, this led to a massive decrease in performance. We suspect that perhaps without the synchronization of waiting, the atomic operations had to wait longer, but it is unclear exactly what the problem is.

5 Scaling

We ran our scaling tests on two datasets, text.txt and human-chr14-synthetic.txt.

5.1 Scaling the number of nodes

Our first plot shows the scaling of different tasts we change the number of nodes while keeping the nodes-per-task at 60.

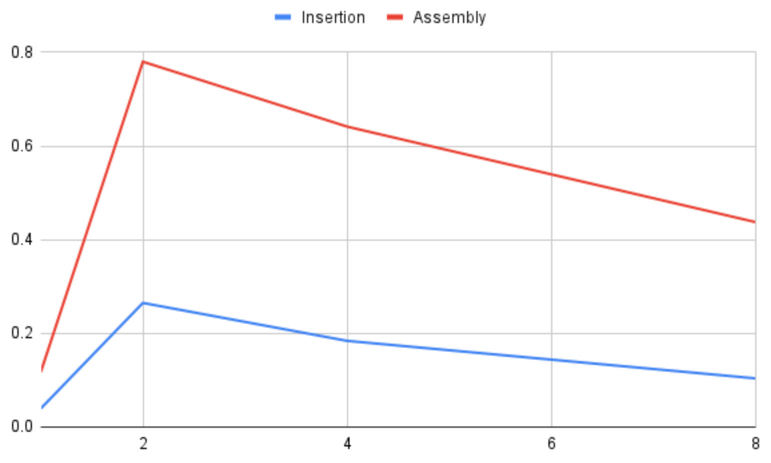

Figure 1: Showing Insertion and Total Assembly time for text.txt while varying the nodes from 1 to 8.

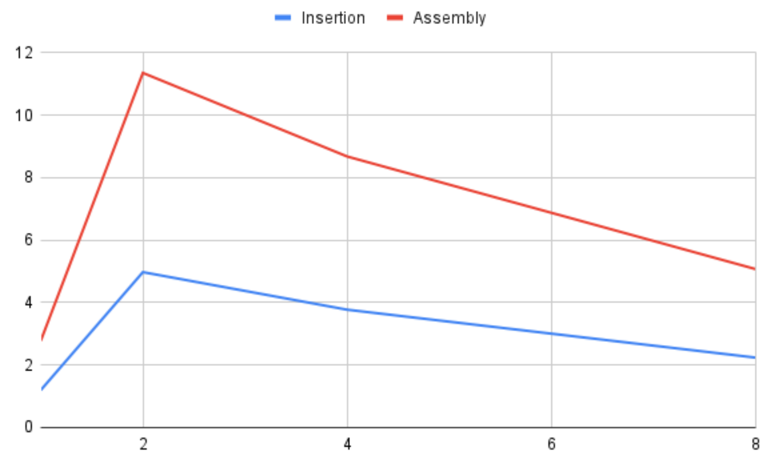

Figure 2: Showing Insertion and Total Assembly time for human-chr14-synthetic.txt while varying the nodes from 1 to 8.

Both of these showcase that there is a big jump from 1 to 2 nodes as we increase the number of nodes. The distributed memory causes a massive jump in latency, which leads to poor scaling. However, after that, the scaling tends to follow the number of nodes.

5.2 Switching from 60 nodes per task to 64

There is almost no difference in performance when we change from 60 tasks to 64:

5.3 Intra node scaling

Additionally we showcase scaling on one node when we increase the number of tasks per node. As expected, we see ideal scaling since the memory does not become distributed and we have more workers.

Nodes	Num Tasks	Insertion	Assembly
1	60	1.197332	2.778371
1	64	1.157058	2.68612
2	60	4.976139	11.356589
2	64	4.703375	10.7405
4	60	3.773192	8.679843
4	64	3.838972	8.526432
8	60	2.238362	5.072118
8	64	2.098693	4.812499

Table 1: Varying nodes per task from 60 to 64 fat each number of nodes

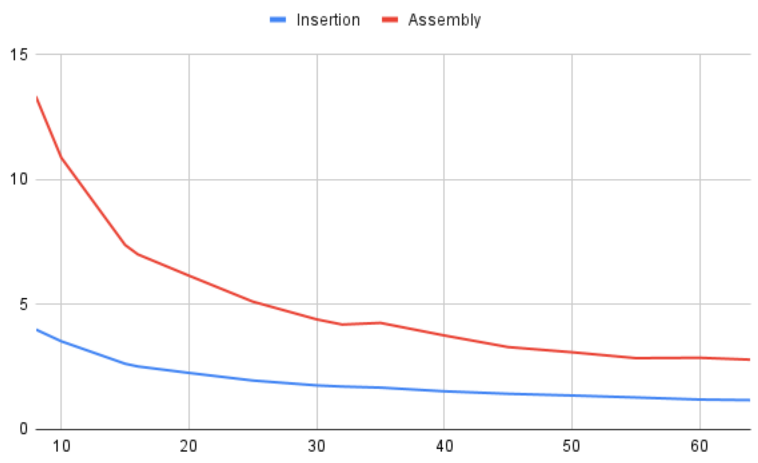

Figure 3: Showing Insertion and Total Assembly time while varying the nodes per tasks from 8 to 64

6 Discussion

We will try to map this same problem to OpenMP. OpenMP uses a shared memory programming model, so we can insert all the kmers into one hash map. For OpenMP, we just need to do the local get and put. In UPC++, we need to do the remote get and put. However, considering the size of one shared memory space, OpenMP is limited by the size of the memory. UPC++ uses distributed memory system, so the size of the problem can be larger than OpenMP.

On the other hand, let us consider using MPI to implement this problem. Firstly, we would divide the kmers evenly and insert different kmers into different processors. Then, we start to build the contig. We always need to find the next kmer. If the next kmer is inside other processors, we need to request for it. But since we would be using two sided communication, it would be difficult to send and receive properly, because the information that needs to be communicated can not be known before hand. Additionally, having to wait for each process to receive would be very slow. Something such as linear probing with atomic operations would be extremely expensive.

In conclusion, for distributed hash map problem, UPC++ has a lot of advantages over the usage of Openmp and MPI. UPC++ can improve performance by reducing communication overhead and eliminate some of the synchronization between processes. However, the speedup depends on the problem size. Thus, one should always compare the benefit of using distributed to non-distributed memory.

7 Contributions

1. Justin - Tried to set up a distributed object implementation, but offered no speedup. Also wrote the report and generated the plots.
2. Yihua - major code implementation
3. Johan - report and further optimizing by aggregating inserts