CS C267 Homework 3 Report

Maya Lemmon-Kishi, Zach Van Hyfte, and Minan Wu

Homework Group 63

1 Overview

In this report, we discuss the strategies we used to design and develop a distributed hash map optimized for constructing de Bruijn graphs of K-mers using the UPC++ partitioned global address space library. In Section 2, we discuss the design choices behind two implementations of the distributed hash map: one that uses blocking remote memory access (RMA) operations (e.g. `rput` and `rget` operations) to read and modify the hash map's internal data structures, and one that uses remote procedure calls (RPCs) to do so. In Section 3, we discuss the performance optimizations we implemented to decrease the RMA implementation's runtime. In Section 4, we discuss how the constraints of the UPC++ library influenced our design, and how we might have implemented the hash map differently using OpenMP or MPI. In Section 5, we analyze the performance of our RMA implementation. When run on one of Cori's Knights Landing nodes, our final RMA implementation completes the assembly pipeline for the `human-chr14-synthetic.txt` dataset in 14.08 seconds; when run across 8 Knights Landing nodes, it completes the assembly pipeline in 6.73 seconds.

2 Implementation

In this section, we describe how the distributed hash map is structured and how access to the data within it is synchronized. We begin by outlining the design of our final RMA implementation (whose performance we analyze in Section 5) and then discuss several variants of an (ultimately unsuccessful) RPC-based approach that we tried.

2.1 RMA Implementation

2.1.1 Hash Map Organization

The hash map in our RMA implementation is organized fairly conventionally, splitting the `data` and `used` arrays provided in the starter code across all of the running UPC++ processes. If the hash map is constructed with size S (e.g. having S slots), and there are P individual UPC++ processes, each process maintains two arrays in the shared/globally accessible portion of their address space, both of which contain $\frac{S}{P}$ elements. At a high level, these two sets of arrays function similar to the `data` and `used` arrays provided in the starter code. The sets of individual `data` arrays store the K-mers, while the sets of individual `used` arrays track whether or not each of the slots of the corresponding `data` arrays are in use. Each of the two sets of arrays can be thought of as one long array whose elements are distributed in large contiguous chunks among multiple processes — an array whose first $\frac{S}{P}$ elements are stored on the process with rank 0, whose next $\frac{S}{P}$ elements are stored on the process with rank 1, and so on.

We added two new members to the `HashMap` struct definition: `rank_data` and `rank_used`, each of which is a vector of exactly P `upcxx::global_ptr` objects. `rank_data[i]` is the global pointer to the portion of the distributed `data` array stored in the address space of the process with rank `i`, while `rank_used[i]` is the global pointer to the portion of the `used` array that is stored in the address space of the process with rank `i`.

In the `HashMap` constructor, each process first uses `upcxx::allocate` to allocate `my_data`, an array of `kmer_pair` objects of length $\frac{S}{P}$, and `my_used`, an array of `int32_t` objects also of length $\frac{S}{P}$. Then, for each rank `r`, calls to `upcxx::broadcast` are used to populate the `rank_data` and `rank_used` arrays, such that

every process ends up with the full set of global pointers to each other process' portions of the distributed `data` and `used` arrays (e.g. the contents of `rank_data` and `rank_used` are identical across all processes).

2.1.2 Probing, `insert`, and `find`

Both our `HashMap::insert` implementation and our `HashMap::find` implementation make use of *blocking* calls to `upcxx::rget` and `upcxx::rput` (i.e. calls to `upcxx::rget(...).wait()` and `upcxx::rput(...).wait()` to read and update the distributed hash table.

At a high level, our hash map uses the same linear probing and open addressing strategies as the starter code. It treats the set of individual `my_data` arrays as one large combined array. It iterates through each of the slots in that large combined array, starting from the slot whose index is given by `h = kmer.hash() % S`, where `kmer` is the K-mer being inserted into the hash map. If the slot `data[h]` (with `data` here representing the implicit global combined array) is not empty (if `used[h]` $!= 0$), our `insert` implementation moves on to check slot `h + 1`, then slot `h + 2`, and so on, all the way up to slot $S - 1$ if necessary. If all of the slots from slot `h` through slot $S - 1$ are full, our implementation "wraps back around" to the start of the array and checks slots 0 through $h - 1$, finally returning `false` if even slot `h - 1` is full.

This whole process is implemented, as in the starter code, as a `do...while` loop. During each loop iteration, our implementation first computes (a) the rank `r` whose portion of the global array the current slot index falls into and (b) the offset `l` within rank `r`'s portion of the global array of the current slot index. Then, our implementation checks `rank_used[r] + l` (the l-th element of the portion of the pointed to by `rank_used[r]`) to determine whether or not the slot is already occupied and marks it as occupied if it is not already occupied, all in one atomic operation (the details of how this is done are described in the next section). If the slot is not already occupied, a blocking call to `upcxx::rput` is used to remotely insert the K-mer into the empty slot, and the `insert` method returns. Otherwise, the current slot index is incremented modulo S, and a new loop iteration begins.

Our implementation of `HashMap::find` uses exactly the same high-level probing process and blocking RMA calls outlined above. During each loop iteration, after computing `r` and `l`, our implementation reads the value of `rank_used[r] + l` using a blocking call to `upcxx::rget`, to determine whether or not the slot is occupied. If the slot is indeed occupied, another blocking call to `upcxx::rget` is used to retrieve the value of the corresponding array slot (`rank_data[r] + l`). If the K-mer read from the slot matches the requested K-mer, the `find` method returns. Otherwise, the current slot index is incremented modulo S, and a new loop iteration begins.

2.1.3 Synchronization

During each iteration of the loop in `HashMap::insert`, the entry of the global `used` array that corresponds to the target slot in the global `data` array is read, and its value is examined to determine if the target slot in the `data` array is already in use. If the target slot is not in use, our `insert` implementation updates the `used` array to mark the slot as occupied, and then inserts into the slot the K-mer with which it was called. If this process of checking and updating the value of the `used` array entry were to be done by, say, an `rget` operation to read the entry followed by an `rput` operation to update the entry if necessary, it would introduce a potential data race — two separate threads could both read the array entry at nearly the same time, see that the corresponding slot is full, and then both proceed in succession to fill that slot with their respective K-mers, such that one process overwrites the other's K-mer.

In order to avoid this data race, both the read and the optional write should take place atomically — no other thread should be able to read or write to the `used` array entry in between the read and the write. We turned to UPC++'s built-in atomic functions to solve this problem. The sorts of locks and atomic functions that could be used to solve this problem in a shared-memory setting would likely be either incompatible with UPC++ or slower than UPC++'s built-in atomics, as they were not designed to support atomic *remote* memory accesses. Thus UPC++'s built-in atomics were the most straightforward and seemingly the most sensible solution available. Specifically, the UPC++ library includes an atomic function, `upcxx::atomic_op::compare_exchange`, that does exactly what we need: a standard "compare-and-swap" operation. The `compare_exchange` function reads the value pointed to by the passed-in `upcxx::global_ptr`, and if that value is equal to some specified constant $val1$, it atomically sets the value to some other specified constant $val2$.

We modified the `HashMap` struct definition to include an additional member of type `upcxx::atomic_domain<int32_t>`, as all UPC++ atomics functions must be invoked via a `upcxx::atomic_domain` object. As per UPC++ requirements, every process eventually invokes this `atomic_domain` object's `destroy` method (specifically, we added a destructor to the `HashMap` struct, in which the `destroy` method is called).

This is the only atomic operation used in our implementation. The `rput` operation in `HashMap::insert` is only invoked on slots that have been claimed by the current process, and our use of the `compare_exchange` atomic function ensures that any given slot will only ever be claimed by at most one process — thus, the `rput` operation does not need to be made atomic. Neither of the `rget` operations in our `HashMap::find` implementation need to be made atomic, because every single call to the `find` method occurs after all of the calls to the `insert` method have completed, and the `find` method only reads and does not modify the `used` and `data` arrays. Thus, at the time that any given call to `find` is executing, the only operations being performed on both the `used` and `data` arrays are read operations, which cannot result in a data race.

2.2 RPC Implementations

2.2.1 Plain RPC Implementation

In the PRC implementation, we used a distributed object `upcxx::dist_object` to implement the distributed hash map. For all P processes, each of them keeps a local `HashMap` object. The local `HashMap` is accessible by all other processes via Remote Procedure Calls. Similar to the RAM implementation, the local `HashMap` object is also split into two parts, `data` and `used`. The `used` keeps track of whether a slot is occupied by hashed and inserted `kmer` data, and the `data` just stores the `kmer` at each slot.

During the `kmer` insertion phase, we first compute the hash value of the `kmer` then mod the hash value by `upcxx::rank_n()` to get the destination process of this `kmer`. If the destination process is same to the parsing process, we just do a local insertion to the `data` and flip the `used` bit to 1 identifying the slot is occupied. Else, we initiate a Remote Procedure Call to the destination process to insert the `kmer` to its local hash map. The `upcxx::future<>` object returned from the RPC at the caller side joins the `upcxx::future<> fut_all`. Upon finish parsing the `kmer`, the `upcxx::future<> fut_all` is waited for all PRCs running to completion. At this point we finish the insertion phase.

During the assembling phase, each process continuously finds the next `kmer` based on the previous `kmer`. Similar to insertion phase, we first compute the hash value of the the previous `kmer` to get the destination process of the `kmer_pair`. If the destination process is same to the caller process, we just do a local search to get the `kmer_pair`. Else, we do a Remote Procedure Call to the destination process to retrieve the `kmer_pair`. Since the assembling phase is serial, so we have to finish retrieving the next `kmer_pair` before finding the further next ones. In the Plain RPC Implementation, we make all the finding RPCs blocking calls.

Using 1 node and 64 processes per node on `human-chr14-synthetic` test set, this implementation finishes insertion in 6.650474 seconds and the entire program in 25.415038 seconds. Comparing to the RMA implementation, this finishes insertion slightly faster but finishes the entire program much slower. We assume the reason is that the RPC implementation designates a fixed destination process upon insertion based on the hash value of a `kmer`, while the RMA continuously does `upcxx::atomic_op::compare_exchange` on remote `HashMap` objects and the synchronization takes longer time. In the assembling phase, the RPCs incur much larger overhead due to the context switch of the callee processes, while RMA just needs to read remote memory without synchronization.

2.2.2 Asynchronous RPC Implementation

To utilize the asynchronous computation, we modified the assembling phase to an asynchronous and non-blocking version. Instead of constructing each `contig` to completion, we iterate through all the unfinished `contigs` and append one next `kmer` for each `contig` in one iteration. We also divided the appending into two steps. At step one we wait the `upcxx::future<kmer_pair>` to get the next `kmer_pair` returned by the PRC initiated at the last round of iteration. At step two, we initiate a PRC without waiting its `upcxx::future<kmer_pair>` object. A little tweak is we first wrap the `start_node` to be `upcxx::future<kmer_pair>` then we can unwrap them at the step one of the first round of iterations.

Using 1 node and 64 processes per node on `human-chr14-synthetic` test set, this implementation finishes insertion in 6.690642 seconds and the entire program in 435.235986 seconds. Comparing to the plain RPC implementation, the insertion time is the same while the finding time is much longer. We assume the reason is that the asynchronous implementation incurs too much unnecessary data structures and operations.

2.2.3 Client & Server RPC Implementation

We also tested a novel idea: splitting the processes into two groups, where one group initiates the RPCs and the other group executes the received RPCs.

During the `kmer` insertion phase, the client group reads in and parses the `kmer` then send them to destination processes in the server group by initiating a PRC. The destination processes in the server group receive these PRCs then insert the `kmer` to their local `HashMap`. If a `kmer_pair` parsed at client group is a starter `kmer_pair`, the corresponding client process keeps it in an list and build the full congit out of it in later steps.

During the `kmer` assembling phase, the client processes initiate RPCs to server processes to get their next `kmer` for construction. The server processes local search their `HashMap` and return a `kmer_pair` to the RPC caller process.

Using 1 node and 64 processes per node on `human-chr14-synthetic` test set, with 50% of the processes set to be client processes and the rest 50% set to be servers, this implementation finishes insertion in 10.076661 seconds and the entire program in 35.447062 seconds. Comparing to the plain RPC implementation, insertion and finding are both slower. We assume the reason is that the insertion phase is intensive for server side while the finding phase is intensive for client side. And the rate between client processes and server processes can not be subverted in the execution. So setting any fix rate can not outperform the plain RPC implementation.

2.2.4 Batched RPC Implementation

We also tried implementing the aggregated store insertion as described in HipMer and MetaHipMer. This built on top of the RPC implementations with a distributed object and included a local buffer within each processor for each other processor. In the insertion function, if the destination rank was of that processor, the kmer pair was inserted as normal. However when the destination rank was not equal to the processor rank, that kmer pair was inserted into the local buffer for the destination rank. Once the buffer for a processor was full, a rpc call was made to that destination processor and inserted the kmer pairs as a batch.

3 Optimizations

In this section, we detail some of the optimizations we made to improve the performance of the first working version of our RMA implementation.

3.1 Eliminating Function Call Overhead

In the provided starter code, the tasks of checking if a slot is used, attempting to claim a slot, reading the K-mer in a slot, and inserting a K-mer into a slot are abstracted away into smaller helper methods of the `HashMap` class. This does make the code more elegant, as it decouples the details of accessing the underlying data structures logic from the probing logic. However, eliminating those separate member functions and performing all `data` and `used` array manipulations directly in the `find` and `insert` methods seems to yield non-insignificant performance gains — presumably by eliminating the overhead of creating and destroying new function call frames. In our tests, this optimization was able to decrease the runtime of our on the `test.txt` dataset by $\sim 13\%$.

3.2 Global Pointer Downcasting for Shared Memory

Slides 32 and 58 (or 38 and 72 if going by the on-slide page numbers) from the UPC++ lecture slide deck note that downcasting `upcxx::global_ptr` objects to standard C++ pointers when possible — and exploiting the resulting ability to use standard assignment statements rather than `rget` and `rput` calls to manipulate

that memory — is an "important performance optimization" that can eliminate the "extra data copies and communication overheads" required by `rput` and `rget` function calls. Slide 59 (or 73) also mentions that, for two processes sharing the same physical memory (e.g. any two processes running on the same Cori node) it is possible for one process to downcast global pointers to the other process' objects, and thus access that other process' memory using standard assignment statements. We implemented this optimization (in an efficient manner that minimizes the overhead of determining whether, for a given loop iteration, the global pointer or the standard C++ pointer will need to be used), and confirmed that this cross-process global pointer downcasting does indeed work, but curiously, we did not observe any performance increase.

3.3 Loop Restructuring

In the process of implementing the global pointer downcasting optimization, we restructured the `do...while` loops in the `HashMap::insert` and `HashMap::find` methods. With the original versions of those loops, during each loop iteration, three indices were each recalculated:

- `slot_index`: The index of the next slot to be examined in the implicit global `data` array (takes two additions and a modulus operation to calculate).
- `destination rank`: The rank of the process whose local portion of the `data` array contains the next slot to be examined (takes one division operation to calculate).
- `local_index`: The offset into process `destination_rank`'s local portion of the `data` array at which the next slot to be examined is located (takes a modulus operation to calculate).

Our refactoring allowed us to get away with recalculating only one of those variables during every loop iteration (in essence, replacing two additions, a divisions, and two modulus operations with just an addition and a modulus operation). We ran some quick tests to see if this optimization might have on its own improved performance by some small amount, but it did not yield any noticeable performance improvements.

4 UPC++ Design Choices

Given the fact that the UPC++ can handle irregular and unpredictable communication pattern better than MPI and/or OpenMP, we exploit this feature in this project. In any of the UPC++ implementations, we can initiate a RMA or RPC at any time when a communication is needed. The callee of the RMA or RPC can handle the injected calls and its local functions concurrently. We can also choose either to wait for the communication running to completion or not.

If we are using MPI and/or OpenMP to solve the same problem, the communication should be scheduled in batches. We may need to accumulate enough remote communications before we issue them, as each communication needs manually-issued synchronizations at both send and receive sides. Under such communication pattern, the synchronization heavy communication incurs significantly higher overhead.

Also, the API designs of UPC++ allows easier communication managements as a `rput` or a `rget` at only one side of the communication effectively equals combining a `send` and a `recv` at both sides.

The UPC++ is a CPU based library and relies heavily on global memory, while the GPU based CUDA also takes the advantage this idea. In theory they should exhibit similar behaviors in this task. But the genome assembly task has substantial serial parts while CUDA is more advantageous on highly-parallelizable tasks, more experiments are needed to compare the performances of the two solutions on this task.

5 Performance Analysis

In this section, we analyze the intra-node strong scaling and inter-node scaling properties exhibited by our final optimized RMA implementation. Our RMA implementation exhibited better performance and higher overall efficiency (across both the `insert` and `find` stages of the assembly pipeline) than the RPC implementations.

5.1 Intra-node Strong Scaling

5.1.1 Test

When run on one of Cori's Knights Landing nodes with a single task, the total running time of our UPC implementation on the `test.txt` dataset is 11.31 seconds. When repeated with 64 tasks on a single node, the total running time was 0.54 seconds. This represents a total runtime speedup of 20.91x and a strong scaling efficiency of 32.67%.

When broken down to individual parts, the insertion time with a single task was 1.81 seconds and 0.27 with 64 tasks while the read time with a single task was 9.51 seconds and 0.19 seconds with 64 tasks. This is a 6.64x speed up with 10.38% efficiency and 50.71x speed up with 79.23% efficiency respectively. This indicates that the strong scaling of our implementation varied. The algorithm's strong scaling efficiency is visualized in **??**, which shoes a rather linear relationship between the number of tasks per node used and the algorithm's runtime.

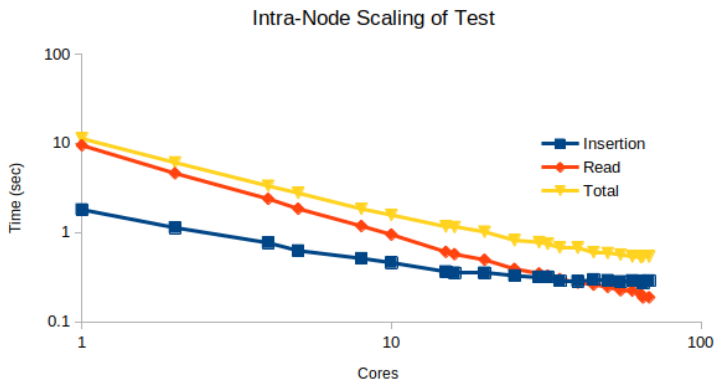

Figure 1: Strong scaling of the UPC algorithm on the test.txt dataset, broken down to insertion time, read time, and total time. Axis are in log scale.

5.1.2 Human Chromosome 14

When run on one of Cori's Knights Landing nodes with a single task, the total running time of our UPC implementation on the `human-chr14-synthetic.txt` dataset is 263.92 seconds. When repeated with 64 tasks on a single node, the total running time was 19.11 seconds. This represents a total runtime speedup of 13.81x and a strong scaling efficiency of 21.6

When broken down to individual parts, the insertion time with a single task was 40.68 seconds and 13.5 with 64 tasks. This is a 3.01x speed up with 4.7% efficiency. This indicates that the strong scaling of our implementation varied, similar to the behavior seen with the test set. The algorithm's strong scaling efficiency is visualized in **??**, however there are many discrepancies in the behavior. This could potentially be due to non-ideal sizes for UPCXX shared heap, gasnet max size, or upcxx segment sizes.

5.2 Inter-node Scaling

In both the test and `human-chr14-synthetic.txt` datasets, we found that the runtime with 64 tasks per node and 68 tasks per node over various numbers of node (1, 2, 4, and 8) similar. This means that despite the number of tasks increasing, the runtime does not decrease substantially indicating that there is runtime overhead.

5.2.1 Test

Along with testing the scaling within a single node, we tested our algorithm over multiple nodes using 64 tasks per node and 68 tasks per node. With a single node with 64 tasks per node the total runtime was 0.55

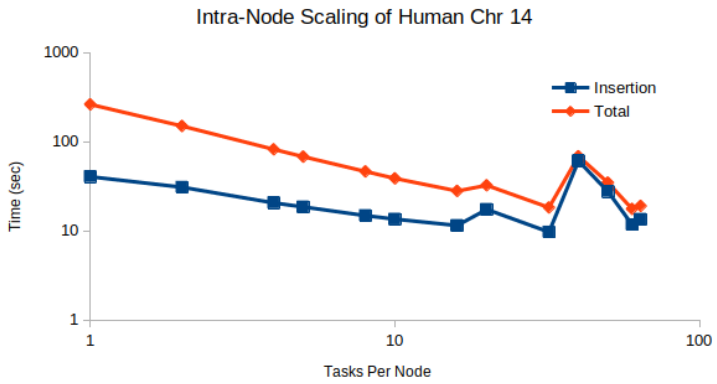

Figure 2: Strong scaling of the UPC algorithm on the human-chr14-synthetic.txt dataset, broken down to insertion time, read time, and total time. Axis are in log scale.

seconds and 0.53 seconds using 8 nodes. When the tasks per node were increased to 68 tasks per node, the single node runtime was 0.59 and 0.53 seconds.

We also broke down the runtime to insertion time and time spent reading from the hash table. Insertion time with 64 tasks per node on a single node was 0.26 seconds and 0.11 seconds with 8 nodes. This was similar to 68 tasks per node where a single node's time was 0.28 and 0.13 seconds with 8 nodes. Read time with 64 tasks per node was 0.20 seconds and 0.21 seconds respectively, with time at 68 tasks per node at 0.19 and 0.23 seconds.

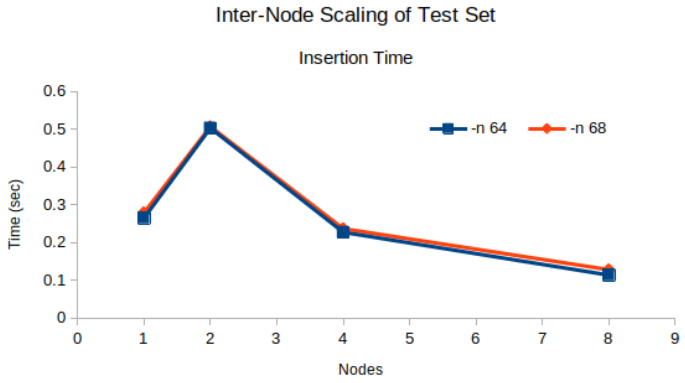

Figure 3:

5.2.2 Human Chromosome 14

Along with testing the scaling within a single node, we tested our algorithm over multiple nodes using 64 tasks per node and 68 tasks per node on the `human-chr14-synthetic.txt` dataset. With a single node with 64 tasks per node the total runtime was 14.08 seconds and 6.73 seconds using 8 nodes. When the tasks per node were increased to 68 tasks per node, the single node runtime was 14.79 and 6.55 seconds.

We also broke down the runtime to insertion time and time spent reading from the hash table. Insertion time with 64 tasks per node on a single node was 8.72 seconds and 2.46 seconds with 8 nodes. This was similar to 68 tasks per node where a single node's time was 9.32 and 2.43 seconds with 8 nodes. Read time with 64 tasks per node was 4.94 seconds and 3.31 seconds respectively, with time at 68 tasks per node at 5.04 and 3.24 seconds.

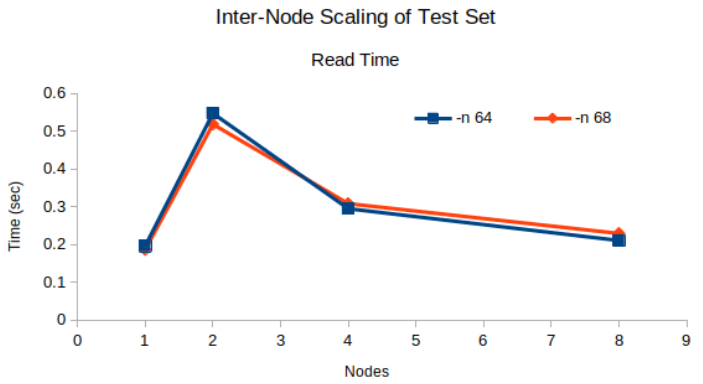

Figure 4:

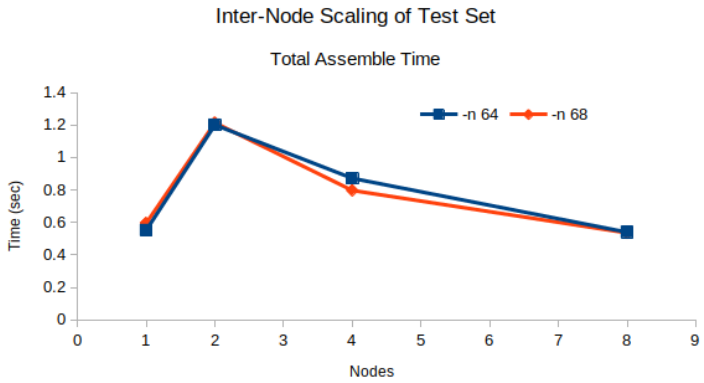

Figure 5:

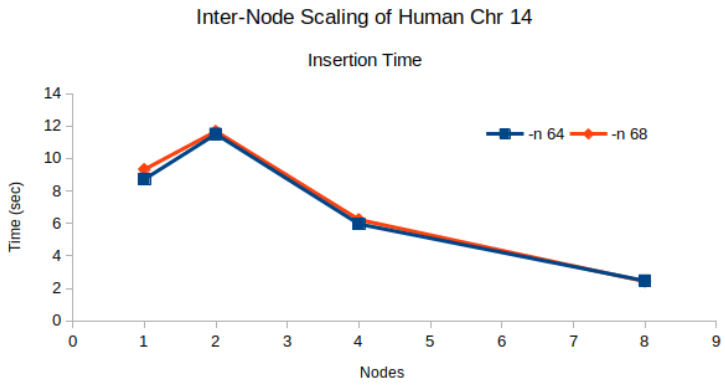

Figure 6:

6 Team Member Contributions

The members of our team contributed equally to this report, to discussions, and debugging of our UPC implementation of a distributed hash table for genome assembly. The individual contributions of each team member are as follows:

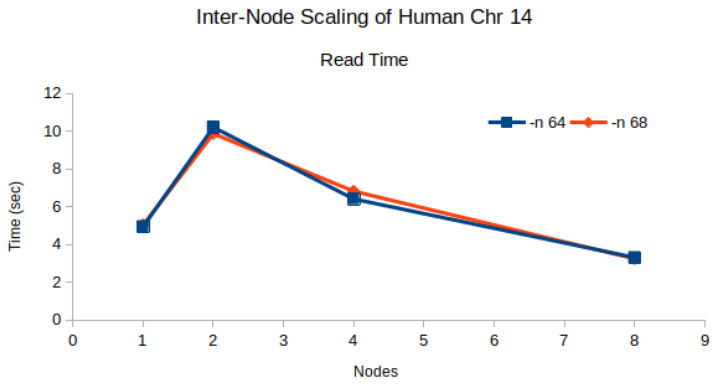

Figure 7:

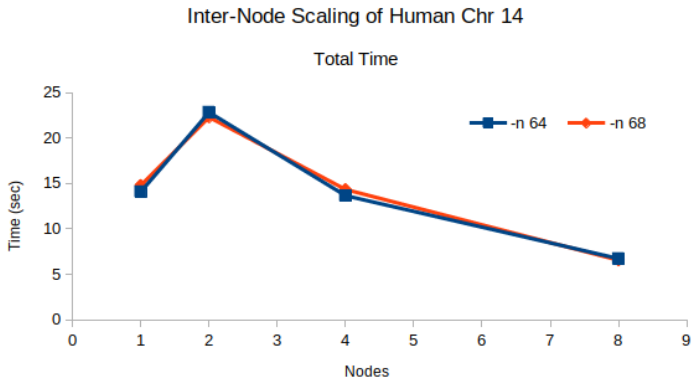

Figure 8:

- **Maya Lemmon-Kishi** developed an alternate RMA implementation and helped optimize both the RMA and RPC implementations.
- **Zach Van Hyfte** developed and helped optimize the final RMA implementation.
- **Minan Wu** developed and helped optimize the RPC implementations.