CS 267 HW 3: Parallelizing Genome Assembly

John Yang & Yulin Zhu (Project Group 27)*

I. Group Members and Contributions

John Yang - Implemented the aggregating stores on top of 2D Arrays, which is the submitted version. Generated performance and profiling metrics for the final submission. Created the profiling graph. In the report, he discusses the aggregating stores implementation, profiling behavior, and comparisons with OpenMP and MPI.

Yulin Zhu - Wrote the initial version of the parallelized implementation with 2D Arrays. Generated the performance values for the first version of the code. Created the scaling and efficiency graphs. In the report, she described the 2D Array Algorithm we employed, and included some explanation regarding the scaling graphs' results.

II. General Overview and Experimental Setting

In this homework, we explore the parallelization of the de Novo Genome Assembly algorithm using the Unified Parallel C (UPC) framework. Given the starter code which provides a serial implementation for traversing de Bruijn graphs, we explored different parallel approaches to the code. We then run experiments on the test and human datasets, comparing performance to the number of processes and nodes used per run. Finally, we discuss how our optimizations affected our performance, and how our approaches would have varied had we used an alternative framework like OpenMP or MPI.

When testing the large dataset like the human-chr14-sythetic, there are some important adjustments we need to make. As we are using Cori-KNL system to test our code, we need to specify the segment size for our nodes and processors. For our algorithms, it proved to be feasible to set the UPCXX_SEGMENT_MB=4000 and GASNET_MAX_SEGSIZE=4G when testing the single node (N=1) with fewer than eight processors, and UPCXX_SEGMENT_MB=512 and GASNET_MAX_SEGSIZE=4G for a greater number of processors.

III. UPC++ Implementation

1. **Atomic Operations**

   Atomic operations are effective tools that we can use to avoid race conditions that could potentially arise when making modifications to the distributed hash table. The atomic

*The authors are arranged in lexicographical order.

operations provided by UPC++ is a bit different from those made available in C++. Therefore, we decided to use an atomic domain that encapsulated different atomic methods and types. The main methods we are using in the experiment are the atomic_fetchAdd() and atomic_compareExchange() methods. Both methods allows us to make updates to remote data atomically and avoid data hazards.

2. **2D-Array Algorithm**

The first algorithm that we implemented is the "2D-Array" algorithm. It's actually not a traditional 2D-Array, but we'll call it this way in the following paper and graphs. To parallelize the serial implementation of the hash table, we'll use multiple processors with UPC++ tools to help us.

The entry of the distributed hash table is the k-mers of the De Brujin Graph that have store the information of itself and its neighboring k-mers. The 3-lengths long k-mers can be represented as [AGCT][AGCT][AGCT] with two other letter codes [AGCTX][AGCTX] representing the forward extension and the backward extension. Thus, we will store the three-letter long k-mer as the key and the two-letter code as the value of the hash table.

In the serial implementation, the hash table used a single array to store the data. For each k-mer, it calculates the hash value and store it to the corresponding address. If there are any conflicts in that location, the k-mer will be moved to the next empty address. In this way, the 1D-Array will be divided into several sections with each section storing one related hash value and its conflicted values. This algorithm will be extremely slow as the dataset grows larger, so we need to implement parallelization over it.

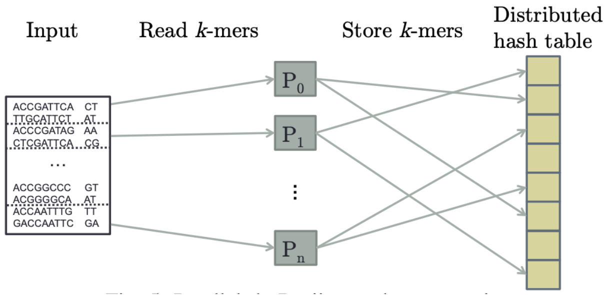


Figure 1: Hash Table Framework 1(referenced from Evangelos Georganas' paper)

Data Structures

In the parallel system, we are allowed to use multiple processors. We first create an global vector *data* to store the information of each processor. The length of this vector is the same as the number of ranks(processors) we specify. For each of the $data(i)$, we create a new array to store the values allocated to the ith processor. This is similar to how we separated and designated different sections of the single array in the serial algorithm to storage corresponding to different processors, allowing each processor to store its own data.

Another data structure we'll need is the *used* data array that stores the 0/1 values to represent whether this location is used. This structure will help us to identify whether the k-mer is stored in the hash table or not. It follows a similar 2-D structure as the *data* buffer.

In the initialization process, we will first define the total size of our hash table, the number of processors we will use and the sub size of the data buffers in each processor. After initializing the *data* array and *used* array in the processor, we will broadcast them to all the processors.

In the inserting process, we will calculate the hash value of the k-mer, and request a location for the calculated address. Using the calculated address, we can figure out which processor and location it is in and see whether it is empty. Here, to make sure it's always empty, we will use a pointer *probe* to automatically point to the next empty location after one insertion is finished. After we request a slot, the corresponding location in the *used* array will be set to 1 and the value will be stored in the *data* array. However, setting the *used* array's value from 0 to 1 introduces the possibility of race conditions occur and could result in coherency issues. To avoid this situation, we introduce the atomic_fetchAdd method as described above. To write in the location in a parallel way, we will use upcxx::rput().wait().

In the finding process, we will also need to first calculate the hash value and find the location of the k-mer value. We then detect the *used* data to see whether the k-mer exists in the hash-table. After identifying the k-mer, we extract its value from the *data* array using upcxx::rget().wait() method, and output the found value.

The overall algorithm is simple and clear, but it shows that the runtime results for the large dataset takes a long time to run. To further optimize it, we introduced the aggregating stores algorithm in the following section.

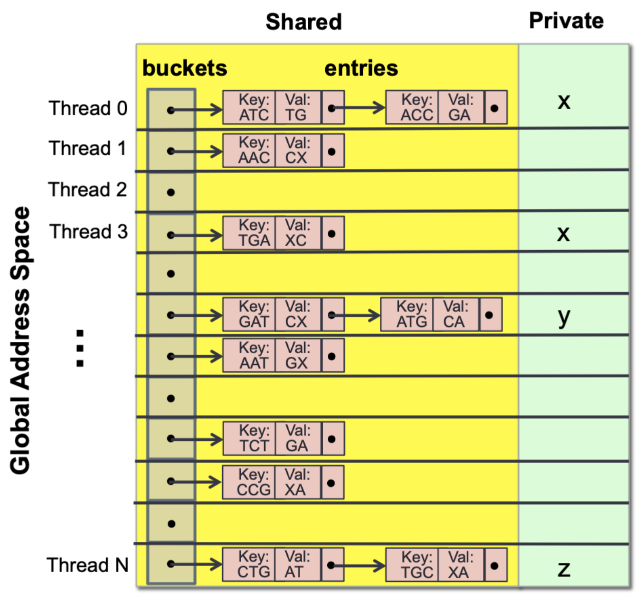


Figure 2: Hash Table Framework 2(referenced from Evangelos Georganas' paper)

3. **Aggregating Stores**

The 2D-Array Algorithm provided a good foundation for parallelization. However, it suffers from a lot of fine-grained communication and synchronization that is needed to store k-mers into a distributed hash table. When multiple processes are reading and writing from the same shared body of memory, the need to maintain atomicity and prevent race conditions causes inevitable slowdowns.

The main improvement that builds upon the algorithm focuses on the idea of reducing the amount of communication needed by introducing a hierarchical set of data structures and memory. As described in the lecture, the idea that we implement is essentially the "aggregating stores" approach.

In the 2D parallel algorithm, there are three main places where communication is necessary: 1. Performing a hash table lookup for a k-mer 2. Adding k-mers to the hash table and 3. Adding k-mers to the start list. The code is mainly separated into two phases of constructing the graph by adding k-mers to the hash table, along with traversing the graph. In this section, I will explain how we reduce communication with 1. An overview of the new data structures we introduce and 2. A walkthrough of how the main methods (constructor, insertion, and clean up) of the submitted implementation works. In a nutshell, we keep the 2D algorithm mainly intact while creating multiple tiers of buffer data structures that reduce total communication required.

Data Structures

The most prominent advancement of the second version of our implementation is the introduction of eight new arrays and pointers. These data structures are organized and used in a hierarchical fashion.

The data_l and used_l variables are individual upcxx::global_ptr objects that serve as local buffers for storing kmers. When full, they are then flushed to the data and used vectors. Using local copies reduces the amount of communication required. Before, each insertion call required communication when directly inserted into the data array. With regards to a distributed hash table, this means incurring a costly remote access. Now, the processor will store the entry to the appropriate local buffer, which when full, flushes n k-mers to be inserted in a single call instead of n separate calls.

The stack variables (four in total - the stack itself, a stack pointer + local copies of each) serve as local buffers that correspond to other processes. These stacks are meant to handle kmers that are "not assigned" to the current running process. When the "slot" of a kmer, calculated from the kmer hash, probe value, and HashMap size, is not equal to the current process's ID, the process will save it to the stack a.k.a. local buffer corresponding to the process that should be handling it.

Last but not least are the cache and cache pointer. In a similar vein to the stack, the cache is a 2D array for storing k-mers, the two indices being the associated slot process ID along with an offset denoting the position of the k-mer within the bucket. When a particular bucket is full, the cache is flushed to the stack, then rest to be empty. The actual size of the cache and stack are specified by a parameter CACHE_SIZE that we define at the top of the function.

This approach no doubt uses a lot more memory. Since we're creating an additional set of buffers of CACHE_SIZE length for every process, this is an overall increase of CACHE_SIZE x (rank - 1) extra memory. However, it reduces the amount of communication, quantified in units of messages, by a factor of CACHE_SIZE, since a cache's content is communicated only when it becomes full. Last but not least, we circumvent the problem of data races because the owner process is now responsible for hash table insertions.

Optimization Discussion

The constructor records the desired size of the hashmap and the number of processes allocated for the task. We set the 'data' and 'used' vectors to a length equal to the number of processes. Each individual 'global_ptr' object of the vectors is approximately the total size of the hashmap divided by the rank. These objects are broadcast to all processes, then added to the 'data' and 'used' vectors. If there is more than one process available, we employ the aggregated stores strategy and initialize a stack so processes can perform insertions of k-mers assigned to other processes. Like the 'data' vector, the stack stores a set of global pointers. However, in this case, these pointers will correspond to k-mers that are assigned to other processes. Similar to the purpose of 'data_l', extraneous k-mers are first placed into the "cache" variable, then flushed only when the cache is full.

The insertion method goes through the greatest number of changes. The 2D array algorithm inserts k-mers naively. The insertion method is separated into three distinct cases, distinguished by whether the k-mer's hash evaluates to the current process's ID, and whether or not the result is being "cleaned up". Here, clean up is meant to indicate the final pass of the algorithm, where any remaining results in the cache and stack are flushed and written to the main distributed hash table. Up until the cleanup, the 'data' and 'used' vectors are not interacted with at all. This is in lieu with the idea of having each process work on its own, local copy of data to minimize communication. First, the insertion method checks whether the slot value, derived from the k-mer's hash, is equal to the current running process ID ('rank_me' method). If true, the k-mer is stored in the local copy of the data array, and the used flag is set to one. If false, the k-mer is stored, in a similar fashion, to the cache, indexed by both the slot value and the 'cacheptr' variable (tracks if a row of the cache, corresponding to a single process's cached k-mers, becomes full). If the cache storing k-mers at the given slot ID is full, a series of atomic domain operations and "rput' calls flush the data to the stack and resets the cache for the given slot ID to be empty.

One of the additional, minor optimizations we added involved setting the atomic domain, which was declared as a separate variable outside of the hash table, to be a field of the HashMap class. The atomic domain object was used in a similar fashion before, but we replaced the 'fetch_add' call in the 'request_slot' function with 'compare_exchange' instead, which only writes the value specified by the pointer if that slot is currently available.

For completeness, we'd also like to mention that the helper methods basically stayed the same as the initial implementation. The write, read, and slot checking functions were written as wrappers to "upcxx::rput" and atomic domain calls, just with varying sets of parameters. In the context of aggregating stores, there weren't any particularly necessary changes.

IV. Results and Analysis

1. **Single Node Runtime Results**
   From the figure, we can see both the algorithms we implemented show a good scaling on the single node case for the large human dataset. When the number of processors is small, the runtime tend to be pretty long, usually up to hundreds of seconds. This means that we don't have much parallelization property in our hash table. The basic idea of the hash table is to separate the data into multiple processors to increase the runtime performance. However, when the number of processors is small, the performance is much like that in the serial setting - all the data is stored in a single(few) processor(s) so that the local data buffer will be long and relatively hard to insert. When the number of processors becomes large, the runtime decreases greatly. Even after using 68 processors per node, we can speed up the runtime to even faster than 30 seconds in both implementations.

   The aggregating optimization over the first 2D-Array method shows a great improvement on the runtime performance. we can observe from the figure that the aggregating store is almost 2 times faster than the 2D-Array in the single node setting. It also matches better to the ideal scaling. There is a slight increase when we change the number of processors from 64 to 68, with the time of 28.8s and 27.5s for the Array and 14.3s and 15.5s for the aggregating store. This might because some parallel system performs better for the number of nodes that are the power of 2.

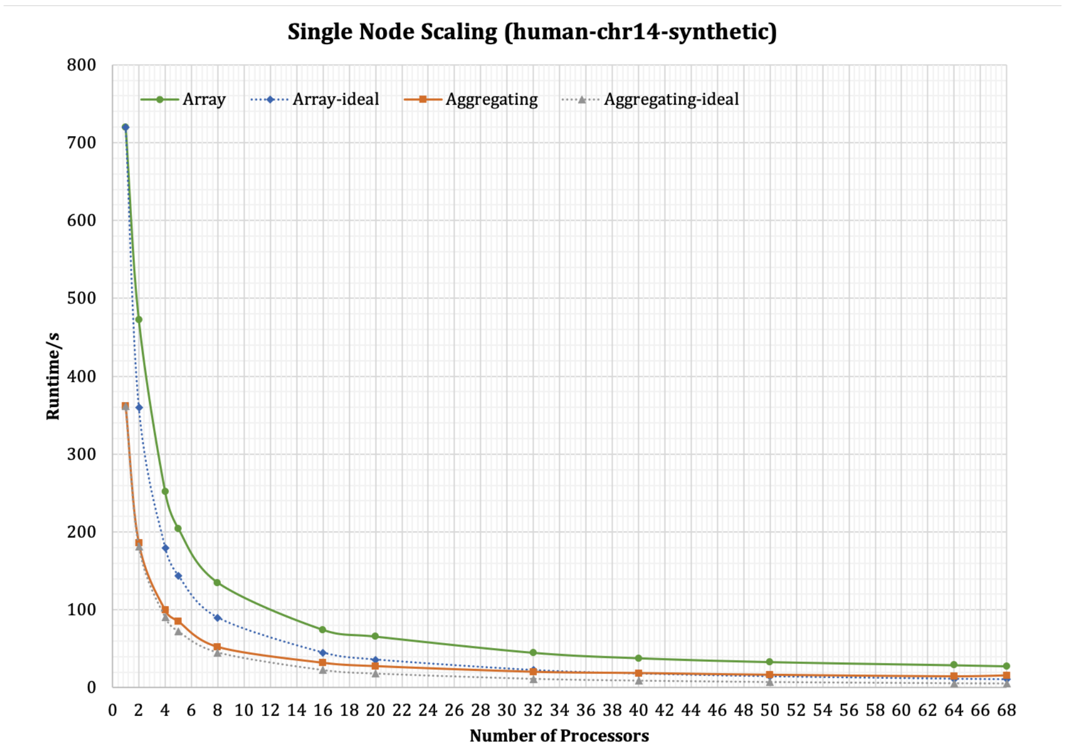


Figure 3: Linear scaling for a single node of all algorithms

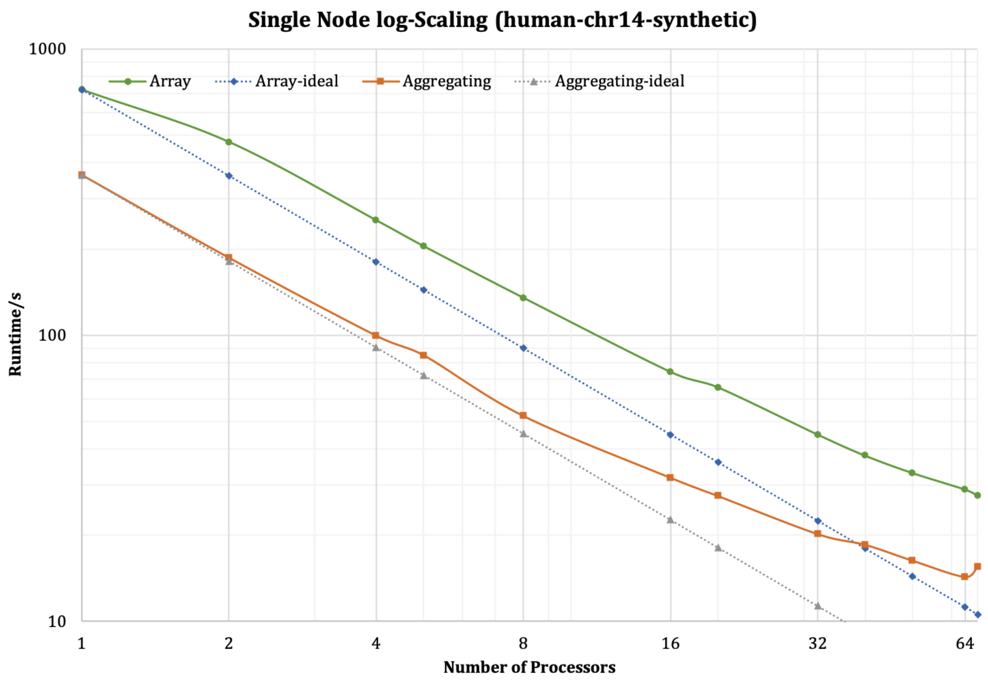


Figure 4: Linear Log scaling for a single node of all algorithms

We also explore the performance of the aggregating stores approach with a smaller dataset. The *test* dataset shows a similar trend similar to the larger dataset as we change the number of processors. To better test its performance on the test data, we divided the time into three categories: reading time, inserting time and total assembly time (from *verbose* output flag). From Figure 4, we can observe that it takes much longer time for our algorithm to read than insert. It may because we need more time to prepare and access the data in the distributed hash table. But we demonstrate high efficiency when inserting data. Thus, the total assembly time is mainly affected by the reading time. To further improve the performance, we can try to explore how to make it read faster.

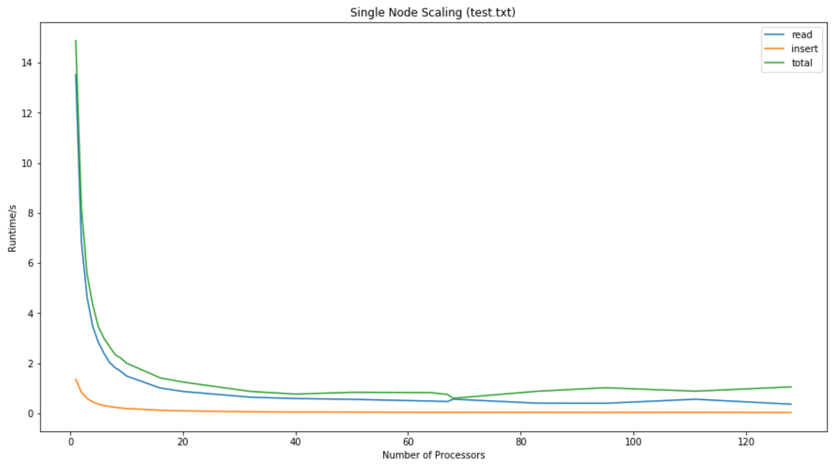


Figure 5: Linear scaling for a single node on test dataset

2. **Multi-Node Runtime Results**
   We tested the results for both algorithms when changing the number of nodes to 1, 2, 4, 8 with 68 processors. Both of the algorithms show a relatively good scaling performance. The aggregating store improved the runtime performance compared to the 2D-Array algorithm in the multi-node setting, almost two times faster. It also shows that the aggregating store matches better to the ideal scaling pattern than the 2D-Array approach.

   We can observe from the figure that when the number of nodes increased from 1 to 2, there is a strange bump in the runtime results. Theoretically, the runtime should be smaller when there are more nodes provided, as parallel efficiency is higher. One of the possible explanations is that it takes more time to allocate data to the nodes while the dataset is big enough when the number of nodes is not greatly increased. The reason why it didn't happen to the aggregating stores algorithm is possibly because of how we reduced communication with more local data structures.

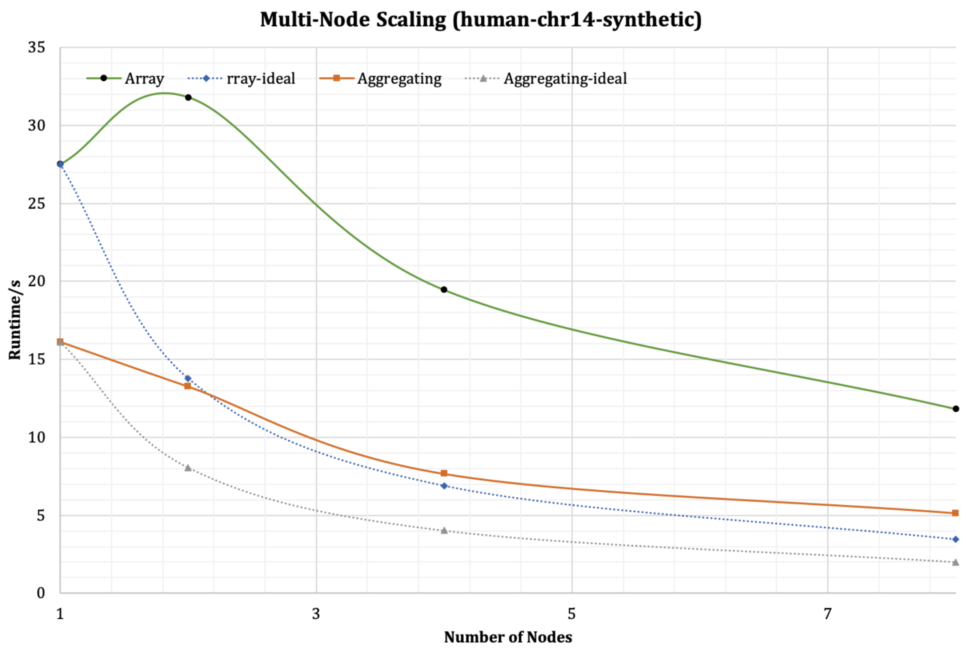


Figure 6: Linear scaling for multiple nodes of all algorithms

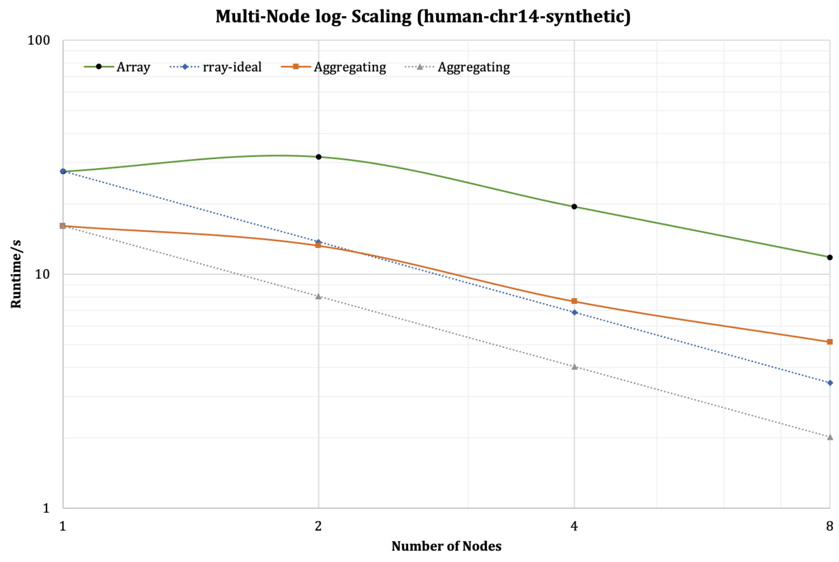


Figure 7: Linear log scaling for multiple nodes of all algorithms

3. **Effect of Strong Scaling**
The strong scaling efficiency results show that both the algorithms have a feasible trend for both the single node scaling and multi-node scaling. As we increase the number of processors and nodes, the strong scaling efficiency tend to be lower. We hope our parallelization can follow the ideal scaling pattern, but we can't achieve it in reality. There are many possible factors that will affect our parallel efficiency, such as the memory/cache limitations, system properties, communication efficiency, etc.

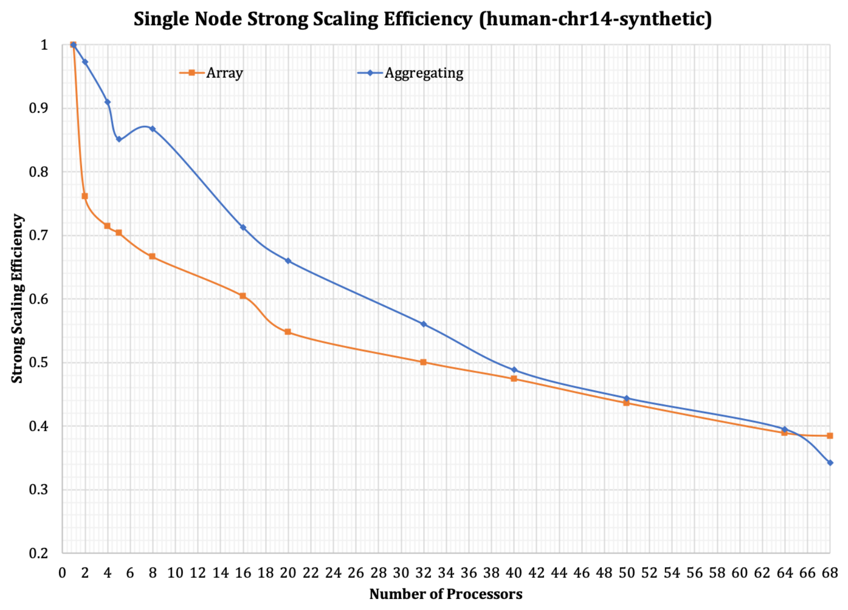


Figure 8: Single Node Strong Scaling Efficiency

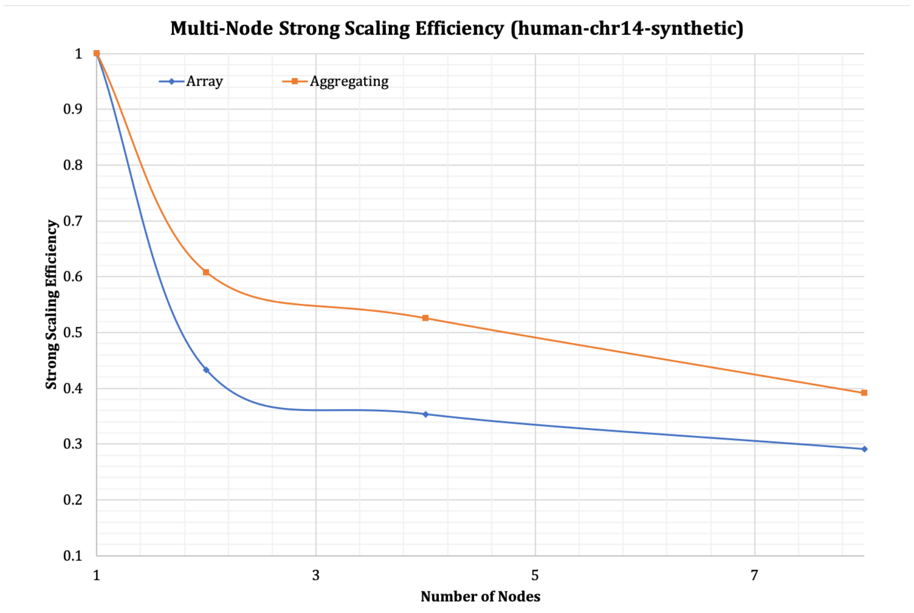
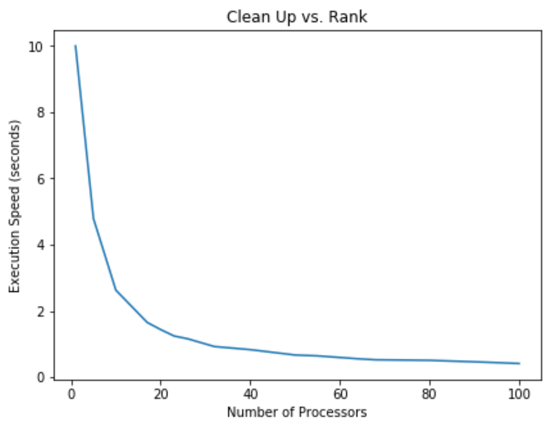
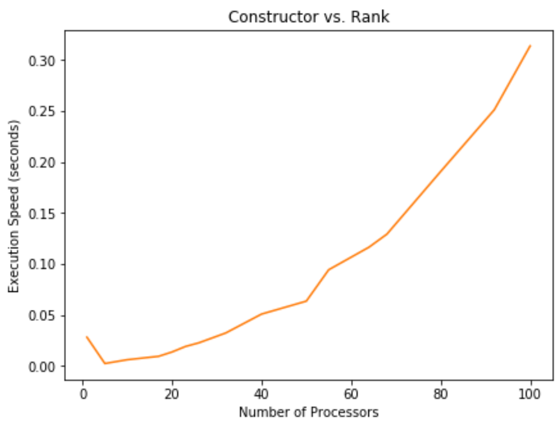


Figure 9: Multi-Node Strong Scaling Efficiency

4. **Profiling**

In addition to the performance results, we also profiled the execution speeds of different methods. These numbers are generated from runs on the human dataset with a single node. The ranks we tested were [1, 5, 10, 17, 20, 23, 26, 32, 40, 50, 55, 64, 68, 80, 92, 100]. The execution speeds for each method are determined differently. The constructor and cleanup methods are only called once per process, so we took the average of the runtimes across all processes. The execution speeds for read_slot are the average of all the runtimes for each time that method is called for a single process. We retrieved these times with the ‘chrono‘ library and print statements. The insert, read, and total times were taken from the ‘verbose‘ output. The runtime results are displayed in the following graphs, with the X axis being the rank, and the Y axis being the execution speed in seconds.





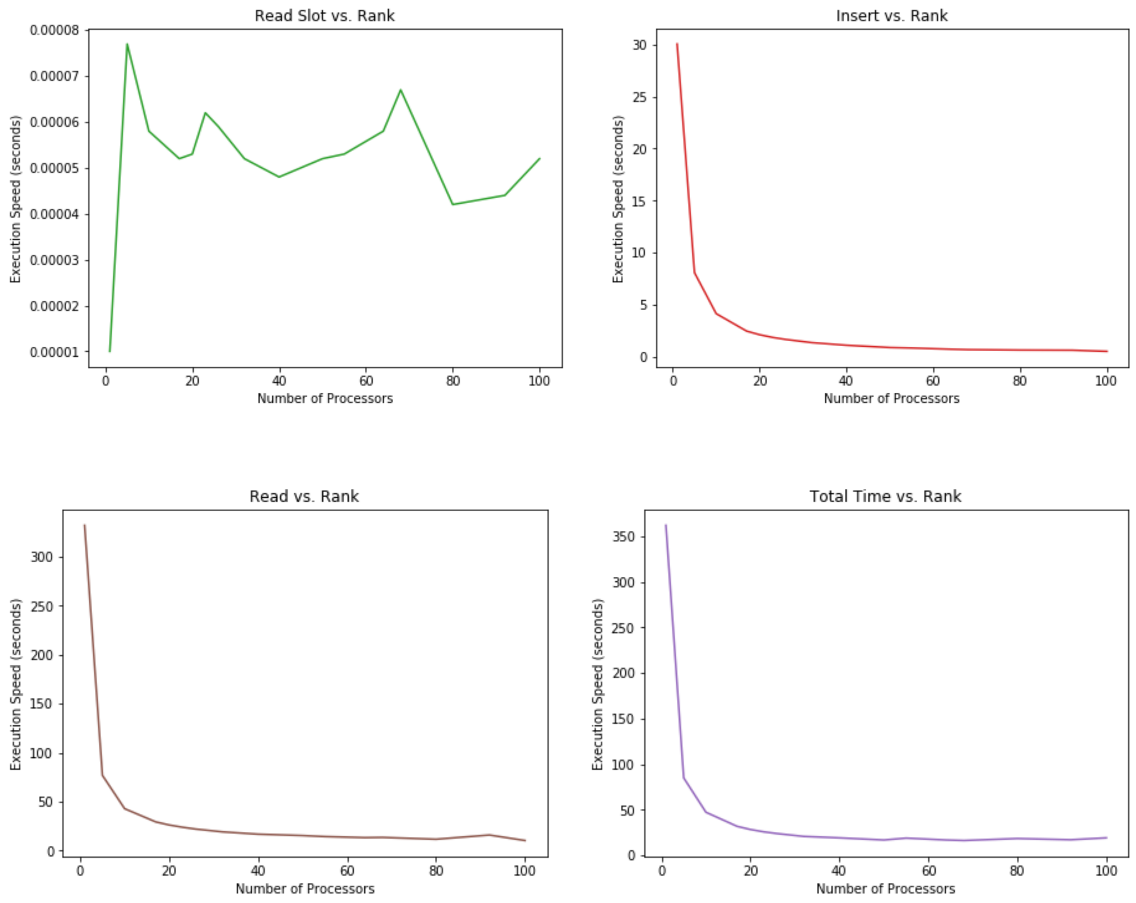


Figure 10: Execution speeds of different methods from human data runs

We think the trends of the above graphs allow us to draw strong conclusions about how much time is spent per method. First and foremost, as the number of processes involved increases, it takes longer for the constructor to complete. This makes sense, since the sizes of the local data and used arrays, in addition to the assortment of stack and cache values, all depend on the number of processes. Initializing lengthier vectors should take longer per process. The linear nature of this graph also makes sense, since the relationship of vector length and rank is linear. The counterpart to the constructor is the cleanup method, which decreases as the number of processes increases. When there's a few number of processes, each individual process is handling a lot more k-mers. As a result, each process should take longer propagating all k-mers stored in the local stack, cache, and data to the global distributed hash table. As rank increases, each process is responsible for fewer k-mers, hence the decrease in time. This reason also applies to the 'read' graph. The 'insert' time also decreases as the rank increases. The reason for this is similar to why cleanup's runtime decreases, with an extra facet. Before the aggregating store modification, insertion costs decreased in a more linear fashion, since each insertion was more naive, and all processes would insert every k-mer to the distributed hash table. However, because aggregating stores makes insertions for k-mer's not assigned to the process much cheaper, the decrease is more logarithmic than linear.

The only inconsistent phenomenon we noticed was how inconsistent the read_slot data run time appears to be at face value. Our original hypothesis was that the execution speeds should be consistent because the read_slot operation is simply a wrapper for an rput call to send data from the stack to the data variable. It shouldn't be influenced by the number of nodes or ranks. This theory also applies to the other helper functions (i.e. write_slot, slot_used, request_slot - not included to cut down on redundancy). We noticed that for all four, the graph was fluctuating in nature. We think that this fluctuation can be attributed to how we profiled. If we averaged all the runtimes across all processes of every call to these methods, we would expect the graph to look much flatter and much more consistent. Also, the scale of the Y-axis is on the order of 1e-5.

The last thing our profiling experiments address is the absolute times of each method. In the lifecycle of a run, it seems like the most time is spent in the following order: Read, Insert, Clean Up, Constructor, Helper Methods. In summary, as the number of processes increase, the [clean up, insertion, read] times decrease, while the constructor time increases per process.

Two side notes: We originally also wanted to include metrics generated from running the test.txt dataset, but because the dataset is much smaller, the execution speed trends across runs were much less pronounced. Second, our final submitted version does not include the profiling code, since we felt that the additional code would hurt the performance.

V. Design Choice Discussion

UPC++ Influences

Working with a language that could take advantage of both distributed and shared memory was a pretty interesting task, especially after writing code with frameworks that worked with only one or the other. One of the small but relevant details, which was similar to CUDA and MPI, was the lack of dynamic block sizes. Our takeaway from this is that it seems like dynamic memory data structures in general are frowned upon within the HPC community because of the variability it introduces that prevents one from taking advantage of optimizations founded on invariants. Therefore, after the data structure sizes are defined in the constructor, we only do reads and writes to these structures, no mutations of the data structures themselves. If dynamic memory structures were available, we'd likely create the local data, stack, and cache variables on a case by case basis within a conditional in the ‘insert‘ method. This would, in theory, cut down on the memory needed, at the expense of additional overhead incurred by maintaining dynamic memory structures.

Another design choice influenced by UPC is the exhaustive way in which we broadcast every global pointer to every process. Because of this, during run time, every process has a pretty comprehensive view of all the data, where it's located and how to go about mutating or reading that data. We initially considered whether more optimizations can be made by trying to cut down on the number of ‘upcxx::broadcast‘ calls that we made. For instance, perhaps some form of preprocessing or sorting of the k-mers could cause them to be distributed such that certain processes didn't have to be aware of certain local data arrays and pointers. We got this approach to work for smaller datasets, but because our idea relies on assumptions about the amount of data / k-mers each process is responsible for, we couldn't come up with a general solution. To put it simply, broadcasting local pointers and vectors to every process was necessary to making sure the algorithm was generalizable to any dataset.

Initially, our types for the 'used' and 'ptr' arrays were vectors of booleans instead of ints. However, running this code in the release environment was much slower than in debug. Changing these to be an array of integers led to much faster execution times.

OpenMP & MPI

Since UPC++ is somewhat of a hybrid of distributed and shared memory, implementing this same problem in either MPI or OpenMP would require rewriting of different portions of the code. UPC++ is also designed to interoperate with MPI and OpenMP, so there may also be room for hybrid solutions.

Implementing this same problem in OpenMP would probably result in performance similar to the 2D Array Algorithm without aggregating stores. Since OpenMP operates in a shared memory environment, we could imagine speeding up the looping that occurs in the 'kmer_hash' file with several 'pragma omp parallel' and 'pragma omp for' directives. Since OpenMP performs multi-threading as opposed to multi-processing, we could still simulate some of the aggregating store architecture, such as the assignment of k_mers to specific process IDs, or in the case of OpenMP, thread IDs. However, because threads would be sharing the same memory, the idea of having "local copies" of data per thread would not be as feasible in this framework. It could still work, but performance would likely suffer from cache coherence. In fact, it might be better to simplify the code and do away with the "aggregating stores" approach if written in OpenMP. This would also mean that the cleanup method is non-essential. Furthermore, writes to the distributed hash table would likely have to be placed in a "critical" section to avoid any data race conditions. Some optimizations can be made by wrapping code that only needs to be run once in an 'omp single' directive, such as the constructor.

We think that a rewriting of aggregating stores + 2D arrays with MPI would be much more feasible. Since MPI is a shared nothing architecture, we could use the gather and aggregate functions to distribute k-mers to individual nodes to be worked on. However, these nodes would not be able to perform writes directly to the distributed hash table result until the gather step, and writing to a stack for k-mer's not assigned to a process / node might be unwieldy because of its inability to take advantage of shared memory settings (at least in earlier versions of MPI). The insertion and k-mers processing step would likely have to be rewritten in a while loop. Within the loop, the k-mers are redistributed to each node. Each process would process its list of k-mers similar to the existing UPC approach. When the results are gathered, the while loop should only terminate if the stack/cache of every process is empty, meaning every process was assigned the correct set of k-mers. The code would then enter a second phase of processing the contigs. In a nutshell, the MPI implementation would likely involve separating the read, insertion, and contig processing steps into three isolated stages that would occur one after another. Each stage, the task is distributed to multiple nodes, then gathered. The code only moves on to the next stage when the gathered result is satisfactory.

Since UPC is compatible with both OpenMP and MPI, an interesting next step would be to explore writing a hybrid program. We believe that our existing implementation, mainly the ‘hash_map.hpp‘ code, takes advantage of multi-node scaling and distributed memory quite well. We’d probably focus more on OpenMP, especially within the ‘kmer_hash‘ file, to see if we can speed up some of the processes occurring within shared memory.

Reference

UPC++ v1.0 Programmer’s Guide Revision 2020.3.0
Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly
Scalable Parallel Algorithms for Genome Analysis by Evangelos Georganas