CS267 HW3: Parallelizing Genome Assembly

Grace Wei, Yinjun Zheng, Emin Burak Onat (equally contributed)

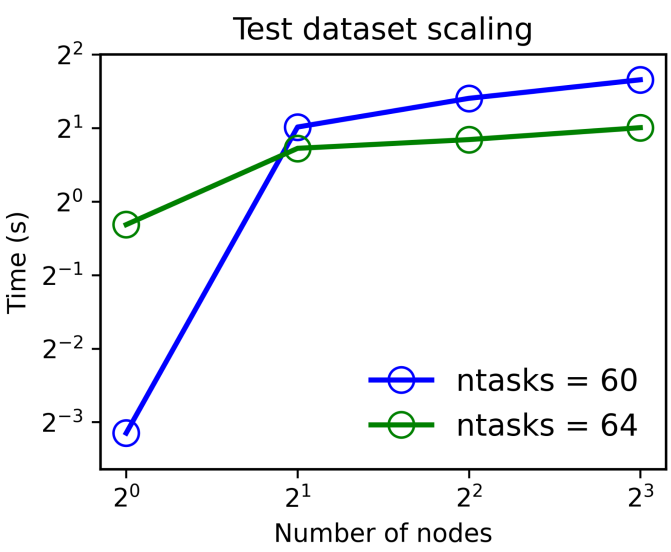
Spring 2023

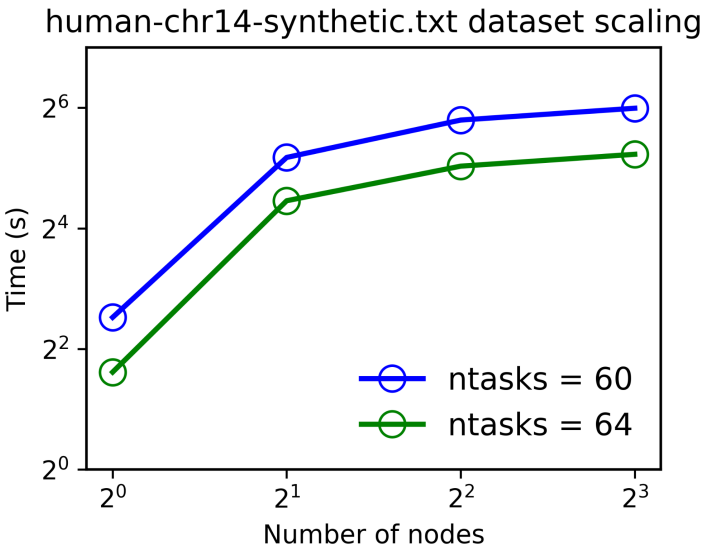
1. Introduction

In this assignment, we implemented an algorithm to parallelize the construction and traversal of the de Bruijn graph of k-mers. We will show how we parallelize the code using UPC++.

2. Scaling Experiments

2.1 Multinode Experiments



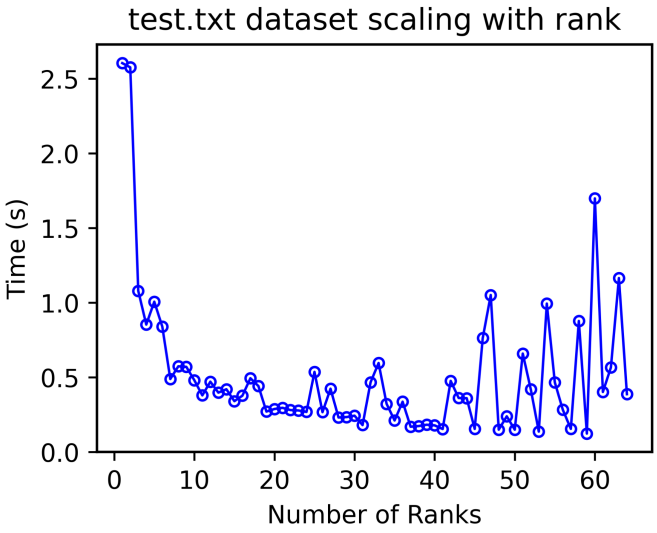


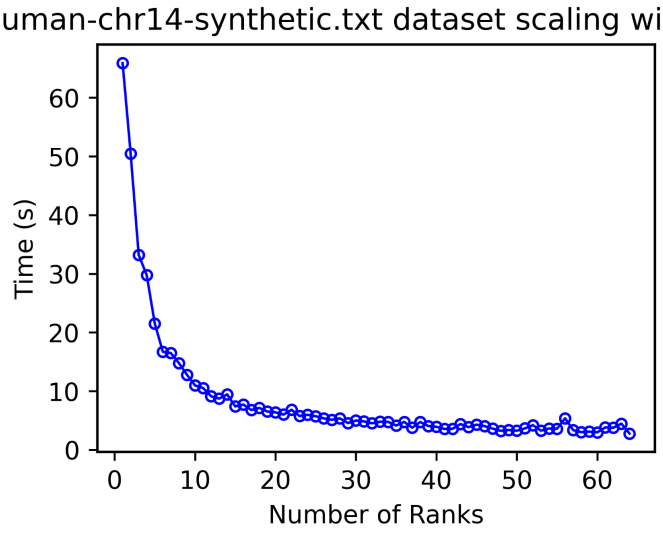
Above, we plot the runtime of the parallel program for test.txt and human-chr14-synthetic.txt as we increase the number of nodes from 1 to 8. We see how performance varies with ntasks_per_node = 60 and 64. We observe that for both datasets, increasing the number of nodes increases the performance runtime. This is likely because our implementation does not use a buffer approach for our implementation. A buffer approach would allow us to optimize the communication by holding more elements before sending out. Another reason for this increase in runtime

is because the CPUs on Perlmutter are very fast, making the runtime more dependent on network overhead.

Additionally, we observe that 64 processors tends to have better performance than 60 processors for both datasets, except for one node on the test dataset. This may be because for smaller datasets, less tasks are needed, and having more processors requires more communication, which could slow down the program. Additionally, we observe that the test dataset performs better than the human-chr14-synthetic dataset. It also appears to have less increase in runtime per node (after the first node). We hypothesize this may be due to the increased network overhead for the larger dataset.

2.1 Intra-node Experiments





Above, we plot how the overall runtime of the program varies with the number of processors used, from 1 to 64 processors. Here, we observe that in general for both datasets, increasing the number of ranks improves the program performance. However, we see that the timing is much more variable for the smaller dataset, especially at higher ranks, compared to the larger dataset, which has a pretty smooth decreasing exponential curve. This is because additional distribution of the smaller dataset amongst the ranks is unnecessary and only increases communication, causing the performance to be more variable at higher numbers of ranks.

3. Data Structure

To parallelize the code, we modify the *data* and *used* data members of the hash map to refer to distributed objects and then modify the insert() and find() methods accordingly. We implement a distributed array in UPC++ by creating a C++ vector of

upcxx::global_ptrs that point to arrays allocated in the shared segment on each rank. We then view the distributed array as one logically contiguous array and use arithmetic to write to the appropriate location in shared memory.

3.1 C++ vector of global_ptrs

Our hashmap comprises two C++ vectors of upcxx::global_ptrs of kmer_pairs and integers, replacing the original vectors *data* and *used*, respectfully. In the hashmap initialization, we divide up the number of elements to be hashed amongst the number of processors, with each processor receiving offset = (*global_size* - 1/ *num_ranks)* + 1 elements. We add the remainder of the work to the final processor. The *offset* describes the distance between the starting position of one processor and the one before it. We utilize the offset to determine algebraically the exact location in the distributed array to read from and write information in. Thus, each processor can access its first location in shared memory via data[(hash%global_size) / offset], and the location of any slot via data[(hash%global_size) % offset]. This is identical to what was done in HW2-1 and HW2-2 to divide up work amongst processors.

3.2 local_data, local_used, fut_data, fut_used

To initialize our distributed hash table, we locally create an upcxx array of global pointers to kmer_pairs (local_data) and integers (local_used). We then loop through all the processors, and broadcast each local set of pointers to shared memory. We then loop through all the processors and accumulate the data, making use of futures (fut_data, fut_used) for asynchronous communication (will be discussed in detail in Section 4).

4. Algorithm and Optimizations

4.1 Atomic Domain

We created an atomic domain for shared ints that work with compare_exchange and load operations. This prevents race conditions from occurring in the code without losing too much in computational speed, due to the atomic operations' leverage of processor support. In our code, the shared integers are the values of the *used* datatype, processors are constantly checking whether a slot is occupied. We use the compare_exchange operation in the HashMap's *insert()* method when we are checking

if a slot is used and change the value of used if needed. We use the atomic load operation load in the helper function *slot_used()* to check whether the slot is used.

4.2 Asynchronous barriers (futures, wait)

Additionally, we opt to use all asynchronous barriers rather than synchronous barrier() to optimize our code. We employ futures to first collect the data from all the processors during initialization of the data structure. Each future holds the value of the broadcast as well as the state. We then load the future data into the distributed array using fut_data[i].wait() and fut_used[i].wait().

We also use asynchronous barriers when we put and get (using rput() and rget()) the kmer_pair into the array by using wait(), for example upcxx::rput(kmer, read_slot_data(slot)).wait(). This means that each local portion of the distributed array can continue to be modified by a single processor until communication is necessary, wherein then there will be a block.

4.3 Relaxed memory order

The memory order specified in our atomic operations specifies how memory accesses are to be ordered around an atomic operation. The default memory order is sequentially consistent ordering. We employ relaxed ordering in our approach. In relaxed ordering, there is no synchronization or ordering constraints imposed on other reads or writes. We actually tried a few different orderings, but they didn't seem to make a significant difference. The timing was quite inconsistent depending on the node used, so we just stuck with the relaxed memory order because theoretically, it is the most basic and efficient atomic lock.

5. UPC++ Implementation, MPI and OpenMP Comparison

5.1 UPC++ Implementation

The hash table is divided among multiple ranks (processes) using the UPC++ library. The hash table has various methods for inserting, finding, and managing k-mers. The implementation uses atomic domains to handle synchronization and atomically manage shared integer values.

kmer_hash.cpp reads k-mers from a file, and inserts them into the HashMap, and assembles contigs from those k-mers. The program uses parallelism with UPC++ to distribute the work across multiple ranks. It initializes the hash table with a size proportional to the number of k-mers and the number of ranks. The k-mers are read from the input file, and the program identifies starting k-mers for contig assembly.

After reading the k-mers, the program inserts them into the hash table. The insertion is parallelized by distributing the k-mers among the different processes. The assembly process uses a parallel Breadth-First Search (BFS) approach to expand the contigs. The program maintains a queue to handle the BFS expansion, and it sends RPC (Remote Procedure Call) requests to other processes to access the k-mers from the hash table. The contig assembly process continues until no more k-mers can be added to the contigs.

5.2. MPI Implementation

If we were to implement the same functionality using MPI instead of UPC++, we would first replace the UPC++ initialization and finalization calls with their MPI equivalents, namely MPI_Init and MPI_Finalize. Next, we would substitute UPC++ process information functions with MPI_Comm_rank and MPI_Comm_size to obtain rank and number of ranks. To manage data distribution, we would use regular C++ pointers instead of UPC++ shared pointers and employ a combination of MPI communicators and collective and point-to-point communication operations to handle distributed data structures. For synchronization, we would use MPI_Barrier in place of upcxx::barrier(). Additionally, we would replace UPC++ Remote Procedure Calls (RPCs) with MPI point-to-point communication operations, using appropriate message tags to distinguish message types, and manually handle message reception and corresponding function execution. Finally, we would use MPI Remote Memory Access (RMA) operations for atomic operations, such as MPI_Win, MPI_Win_create, MPI_Win_lock, MPI_Win_unlock, MPI_Put, MPI_Get, MPI_Accumulate, and MPI_Fetch_and_op.

5.3. OpenMP Implementation

If we were to implement the same functionality using OpenMP instead of UPC++, we would need to consider that OpenMP is designed for shared-memory parallelism, whereas UPC++ and MPI are meant for distributed-memory systems. With that said, we can still use OpenMP to parallelize our code, but it would be limited to a single shared-memory machine.

First, we would replace the UPC++ initialization and finalization calls with the appropriate OpenMP directives. Next, we would modify our code to remove any references to distributed memory and UPC++ shared pointers, since OpenMP operates in a shared-memory environment.

To parallelize the code, we would use OpenMP directives such as #pragma omp parallel, #pragma omp for, and #pragma omp single, to control the flow of parallel execution. We would also need to manage the data distribution manually, partitioning the data structures among the threads and ensuring proper synchronization to avoid race conditions.

For synchronization between threads, we would use OpenMP constructs like #pragma omp barrier, #pragma omp critical, and #pragma omp atomic. In place of UPC++ Remote Procedure Calls (RPCs), we would rely on regular function calls, since all threads share the same memory space and can directly access each other's data.