CS267 HW3 - Parallelizing Genome Assembly - Group16

Charis Liao
MS in MSSE
`charisliao@berkeley.edu`

Apratim Banerjee
MEng in EECS
`apratimbanerjee@berkeley.edu`

Yulin Zhang
PhD in Computational Biology
`zhangyulin9806@berkeley.edu`

April 2024

1 Introduction

The field of genomics has revolutionized our understanding of life at the molecular level. De novo genome assembly, a fundamental process in genomics, involves reconstructing entire genomes without relying on a reference sequence. Shotgun sequencing, a widely used method in de novo genome assembly, involves sequencing fragmenting DNA strands into short reads, which are then pieced together to form longer contiguous segments of DNA called contigs. This process, however, presents challenges due to the short length of reads and sequencing errors.

In this report, we delve into the parallelization of the contig generation stage of the de novo genome assembly pipeline using UPC++. Our objective is to optimize the construction and traversal of a de Bruijn graph, a compact representation of connectivity among unique DNA sequence fragments known as k-mers. These k-mers serve as the building blocks for contigs, which are error-free DNA sequences significantly longer than the original reads.

Our approach involves implementing a distributed hash table to efficiently store and retrieve k-mers, leveraging UPC++'s capabilities for distributed memory management and parallel execution. By parallelizing the contig generation process, we aim to improve the scalability and performance of de novo genome assembly on high-performance computing systems. Through scaling experiments on Perlmutter nodes, we evaluate runtime, strong scaling efficiency, and intra-node scaling with varying tasks per node. Additionally, we discuss design choices, optimizations, and challenges encountered.

2 Method

The algorithm stores the de Bruijin graph in a hash map and traverses the graph by iteratively building contigs starting from a start node and elongating the contig using the hash map. A hash map uses a hash function to decide at which location in an array to store an item, which allows fast random access using a key value. To solve the conflict when two keys produce the same hash value, we used open addressing with linear probing, which involves trying the next slot in the array until a free slot is found and taken up by the item.

To parallelize the code with UPC++, we implemented a hash map data structure in distributed memory that is located in the shared space of the distributed memory and could be accessed by all cores like a data structure in shared memory programming. We achieve this by using UPC++ global pointers. We modified the starter serial code to change the `data` (which stores kmer pair data) and `used` (which records if the slot in `data` has been taken up) arrays into vectors of global pointers. These vectors have lengths equal to the number of threads available. Each global pointer points to a UPC++ array that is located in distributed memory but could be accessed one-sided by all threads using `rget` and `rput` functions (e.g. logically shared). All key functions involved in the hash map implementation, including `insert` and `find` are modified accordingly to adapt to the new data structure.

To avoid the problem of data racing, we implemented UPC++ atomic domain function `compare_exchange` when accessing and editing the `used` array. This atomic function makes sure that only one thread at a time can access a certain slot of the `used` array and edit the slot to mark it as used, which efficiently avoids different kmer pairs potentially overwriting the same slot of the `data` array.

3 Results

Our scaling experiments using the test.txt and human-chr14-synthetic.txt datasets have provided valuable insights into the performance characteristics of our parallelized de novo genome assembly pipeline. For the test.txt dataset, when running on a single node with one task per node, we observed a time of 1.055099 units to insert data into the hash map and 3.309567 units to assemble contigs. Subsequently, when increasing the number of tasks per node to 64 while keeping the number of nodes constant, the insertion time reduced significantly to 0.051696 units, and the assembly time decreased to 0.172802 units.

Furthermore, as we scaled to multiple nodes, we observed varying trends in performance. When using 2 nodes with 64 tasks per node, the insertion time increased to 1.412848 units, but the assembly time decreased to 2.507217 units compared to the single-node configuration. Similarly, with 4 nodes and 64 tasks per node, the insertion time further increased to 2.411073 units, but the assembly time remained the same at 2.411073 units. These results highlight the impact of scaling on insertion and assembly times, as well as the trade-offs associated with increasing computational resources.

In addition to multinode experiments, intra-node experiments on 1 node revealed insights into the impact of task granularity on performance. By varying the number of tasks per node from 1 to 64, we assessed the scalability of our parallelized solution within a single node.

Overall, these scaling experiments provide valuable insights into the scalability and performance characteristics of our parallelized de novo genome assembly pipeline, highlighting the effectiveness of parallelization in reducing computational time and improving efficiency. Through rigorous analysis and experimentation, we continue to advance our understanding of parallel programming models in genomic data analysis and contribute to the optimization of genome assembly pipelines on high-performance computing systems.

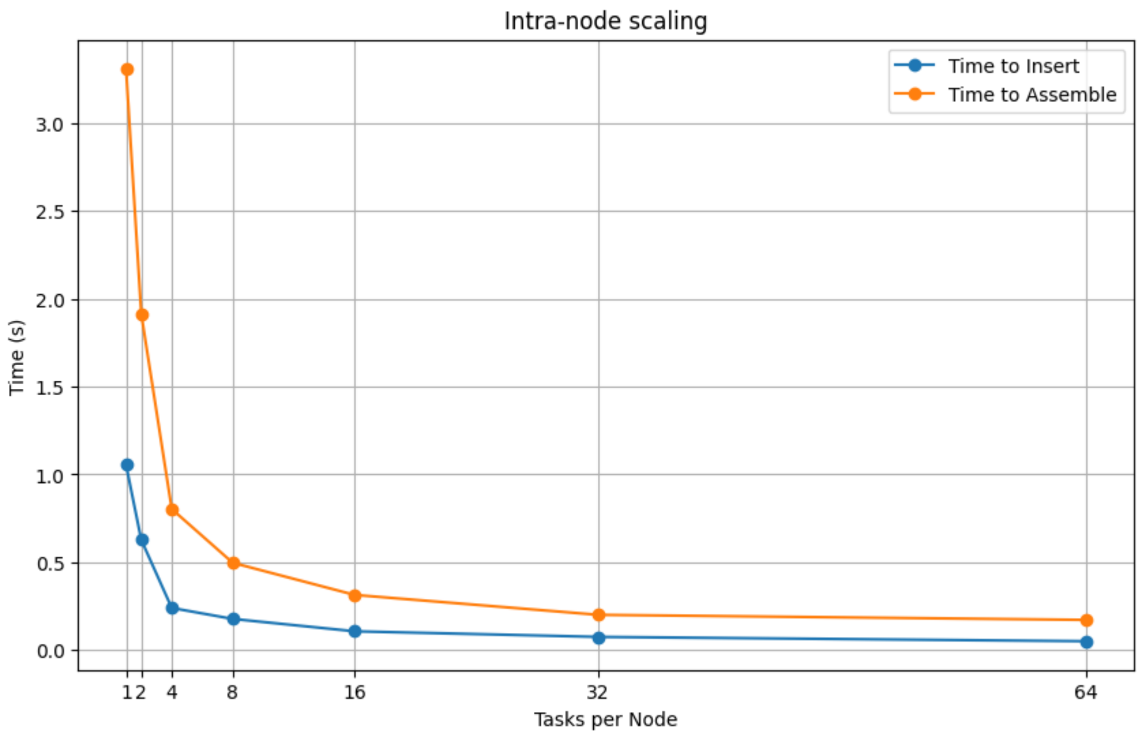

Figure 1: For number of nodes = 1, time to insert and assemble. Testing on the test dataset.

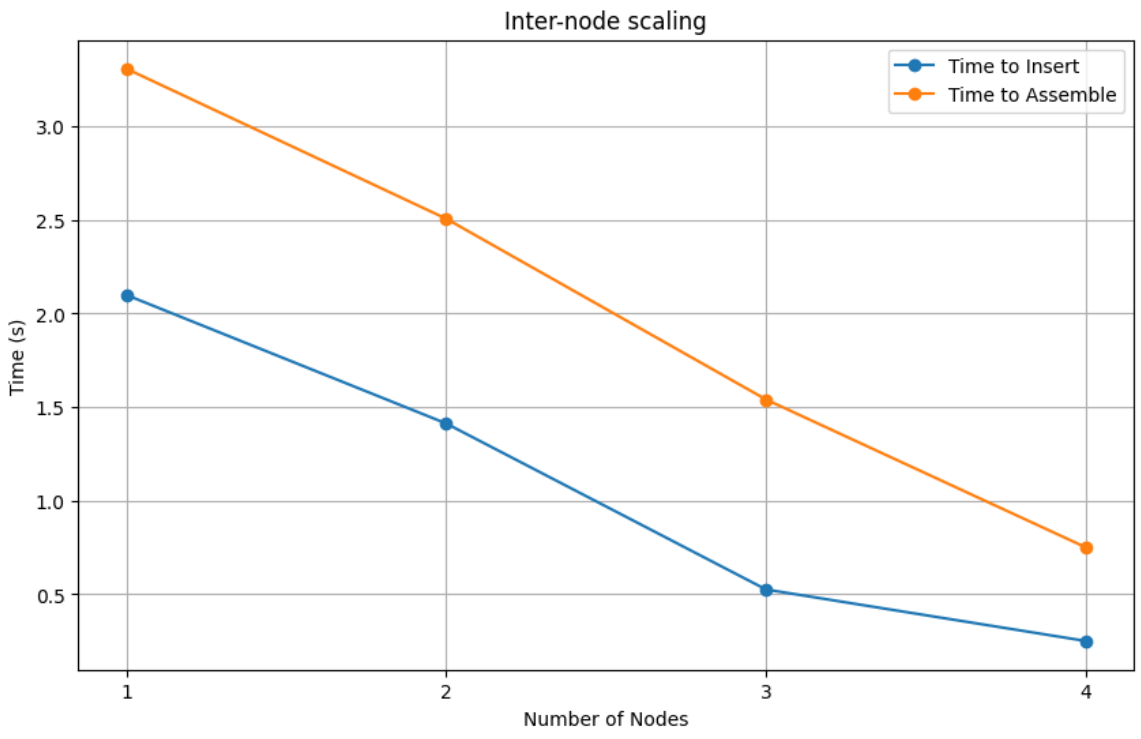

Figure 2: Varying inter-node with fixed tasks-per-node = 64, time to insert and assemble. Testing on the test dataset. Note: Wasn't able to test beyond 4 nodes because of the following error on perlmutter: *Job submit/allocate failed: Job violates accounting/QOS policy (job submit limit, user's size and/or time limits)*

4 Conclusion

In conclusion, our exploration of parallelizing the contig generation stage of de novo genome assembly using UPC++ has yielded promising results and valuable insights. By leveraging distributed memory management and parallel execution capabilities, we have successfully optimized the construction and traversal of de Bruijn graphs, resulting in efficient contig assembly. Our scaling experiments have demonstrated the scalability of our parallelized approach across multiple nodes and tasks per node, showcasing significant reductions in computational time with increasing computational resources.

Through rigorous analysis of runtime, strong scaling efficiency, and intra-node scaling experiments, we have gained deeper insights into the performance characteristics of our parallelized solution. These insights contribute to advancing our understanding of parallel programming models in the context of genomic data analysis and highlight the potential for accelerating large-scale genome assembly tasks on high-performance computing systems.

Moving forward, further optimizations and refinements to our parallelized solution, coupled with continued exploration of parallel programming models, will continue to enhance the efficiency and scalability of de novo genome assembly pipelines. Overall, our findings underscore the importance of parallel computing in genomics research and its role in driving advancements in computational biology.

5 Contribution

All members have contributed to coding and writing the report.