CS267 - HW3: Parallelizing Genome Assembly

Chongye Feng* Thiago Bergamaschi†

March 2024

1 Introduction

This write-up outlines our approach, implementation, and evaluation of HW3 for CS267: Parallelizing Genome Assembly, focusing on implementing a distributed hash table. We discuss the data-structure implementation strategies employed, as well as minor optimizations.

2 Approach

Our approach broadly followed the techniques outlined in recitation, in implementing the distributed hash table by leveraging atomicity in the insertion function. Only minor modifications to the application code (kmer_hash.cpp) were needed, so they were omitted. Here we focus on the description of the hash table.

2.1 The Distributed Data Structure

Is, broadly speaking, a hash map stored in partitions across multiple nodes. Each local map maintains two vectors,

- **The Data**, a upcxx:global_ptr which is used to index each node/ranks vector of kmer_pairs.
- **The Used List**, a upcxx:global_ptr which is used to index each whether the bins of the table are occupied.

In the subsequent subsections we discuss how to maintain the data-structure in a race-free manner.

2.2 External-facing Operations

Following the starter code, our hash table supports two external-facing operations: Insert, and Find. Here we briefly discuss what they do and the helper functions they rely on, described in more detail in the next section.

- **Insert.** Once the desired hash location is computed, the insertion protocol runs a **request_slot** query, to verify if the associated location indexed by the hash has been occupied. If not, it issues a **write_slot** query. If the request slot fails, we simply run a naive linear probing scheme to find the next available location.
- **Find.** Iteratively checks the slots in the table, running **read_slot** queries and comparing the desired kmer key to that read from the data-structure query.

*chongyef@berkeley.edu.
†thiagob@berkeley.edu.

2.3 Helper Functions

As discussed, we require three helper functions: **request_slot**, **write_slot**, and **read_slot**. In order to avoid data-races during insertions, we make crucial use of an upcxx atomic domain, initialized with the hash table.

- **request_slot** queries are made to the Used vectors, which via fetch_add calls, allow us to increment the hashed entries atomically.
- **write_slot** queries to the Data vector are supported under the guarantee that **request_slot** was successful, and are implemented using Rput calls. Under this guarantee, no further atomicity is needed.
- **read_slot** queries to the Data vector are supported using Rget calls. In the application, **Find** calls are made only after all insertions are performed, and there are no deletions. Thereby, races are benign, and no further atomicity is needed.

3 Scaling Experiments

We conducted scaling experiments to evaluate the performance of our parallel genome assembly implementation on 19-mers and 51-mers datasets across 1, 2, 4, and 8 nodes with 60 and 64 tasks per node. These experiments aimed to assess the impact of node count and task distribution on initialization and total execution times.

3.1 19-mers Dataset

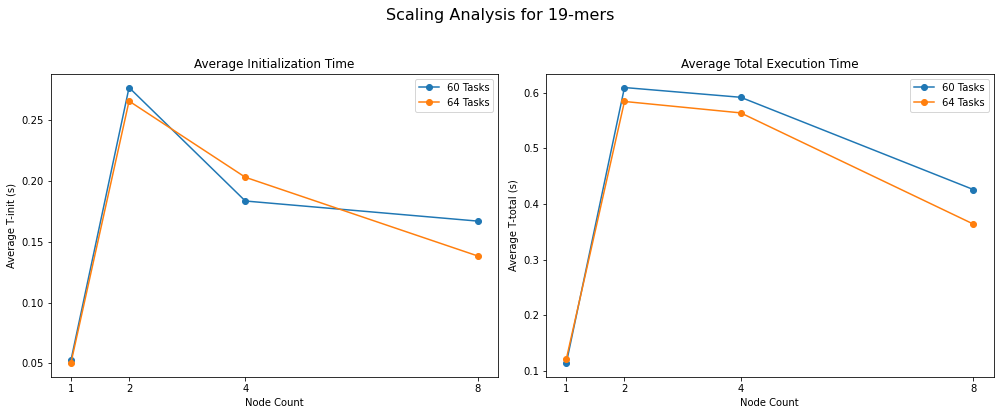

Figure 1: Scaling analysis for the 19-mers dataset, showing initialization and total execution times for different node counts with 60 and 64 tasks per node.

For the 19-mers dataset, we observed the following trends:

- **Initialization Time (T-init)** increased with the number of nodes, with a significant rise when scaling from 4 to 8 nodes, suggesting a non-linear scaling behavior possibly due to inter-node communication overhead.
- **Total Execution Time (T-total)** also increased with the number of nodes, but a notable decrease for 8 nodes indicated improved efficiency at larger scale operations.
- Execution times for 60 and 64 tasks per node were similar, with 64 tasks showing slight performance improvements in certain node configurations.

3.2 51-mers Dataset

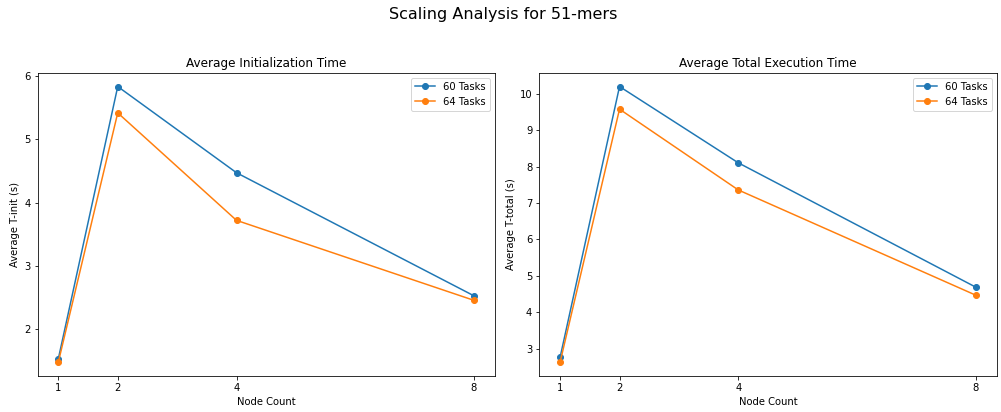

Figure 2: Scaling analysis for the 51-mers dataset, showing initialization and total execution times for different node counts with 60 and 64 tasks per node.

The 51-mers dataset demonstrated:

- A **significant decrease in Initialization Time (T-init)** as the number of nodes increased, with the most substantial improvement observed when moving from 4 to 8 nodes.
- A clear trend of **reduced Total Execution Time (T-total)** with more nodes, emphasizing the advantages of scaling up resources for larger datasets.
- Minor variations between 60 and 64 tasks per node, indicating that the number of tasks per node had a less pronounced impact compared to the number of nodes.

4 Intra-node Scaling Analysis

Intra-node scaling experiments were performed to assess the performance impact of varying the number of tasks per node, using the 19-mers and 51-mers datasets. These experiments help understand the scalability and efficiency of the parallel genome assembly process within a single compute node.

4.1 19-mers Dataset

The 19-mers dataset exhibited notable trends in performance as the number of tasks per node increased:

- **Initialization Time (T-init)** and **Total Execution Time (T-total)** both showed a decrease as the number of tasks per node increased, indicating effective utilization of node resources.
- The rate of performance improvement tended to diminish beyond a certain point, suggesting that there are limits to the benefits gained from increasing tasks per node, likely due to the overheads or resource saturation.

4.2 51-mers Dataset

The 51-mers dataset presented a different scaling behavior, particularly in terms of memory requirements:

- There was a **sharp decrease in execution times** when the number of tasks increased from 1 to 10, highlighting the efficiency of parallel processing in handling larger workloads.

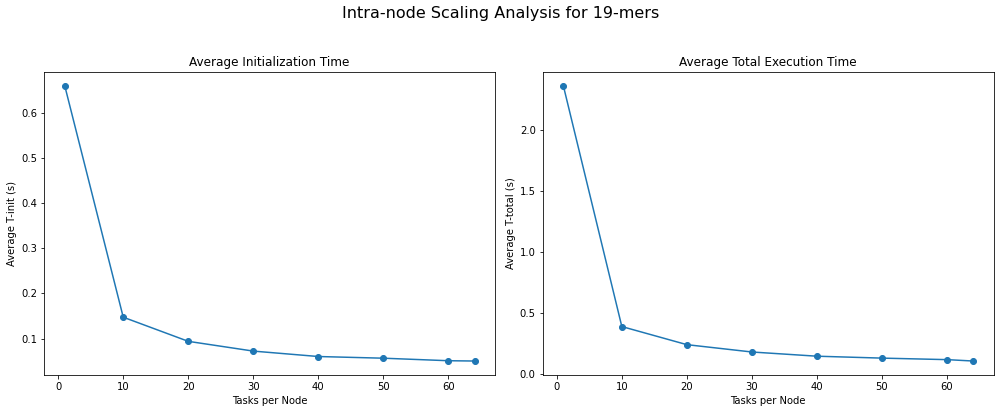

Figure 3: Intra-node scaling analysis for the 19-mers dataset, illustrating the change in initialization and total execution times with varying tasks per node.

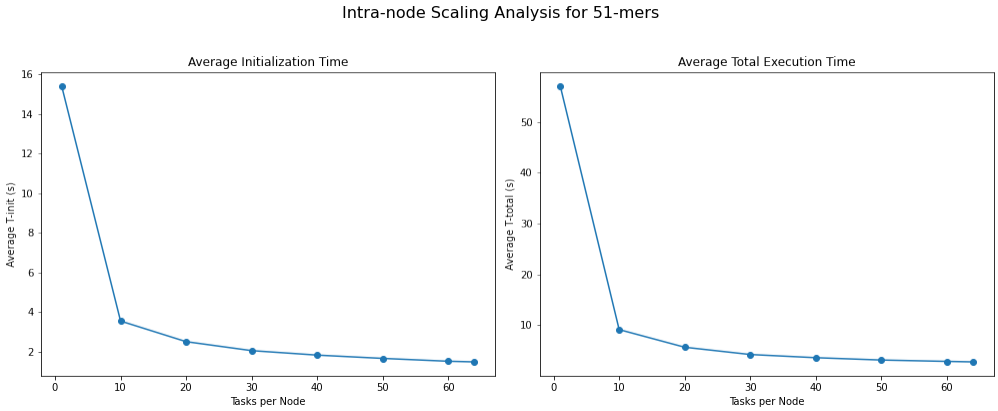

Figure 4: Intra-node scaling analysis for the 51-mers dataset, showing the impact of increasing tasks per node on performance.

- Beyond 10 tasks, performance gains were more incremental, which indicates a plateau in scalability within the node.
- The experiments revealed substantial memory demands, necessitating an increase in the shared heap size to 4 GB to avert out-of-memory errors.

4.3 Shared Memory Challenges and Observations

The intra-node scaling experiments underscored the significant memory requirements of the genome assembly task, especially for the 51-mers dataset. To accommodate the high memory usage, the shared heap size was increased to 4 GB. This adjustment highlights the importance of memory management in parallel computational tasks and suggests that future optimizations should consider both computational and memory resources to improve scalability and performance.

5 Conclusion

Our scaling experiments for 19-mers and 51-mers datasets highlighted the importance of balanced task distribution and efficient memory management in distributed genome assembly. The findings indicate that while our implementation shows effective parallelism, optimizing memory usage and minimizing parallelization overhead are crucial for enhancing scalability and performance.