Parallelizing Genome Assembly with UPC++ Group 30

Brent Thorne Kofi Mireku Puja Madgula

April 13, 2025

Abstract

Genome assembly reconstructs full genetic sequences from short DNA fragments (k-mers) using de Bruijn graph traversal, a computationally demanding process requiring efficient parallelization. In this project, we implement a distributed hash table using UPC++ to optimize k-mer storage and retrieval, leveraging partitioned data structures and batched remote insertions to minimize communication overhead. Our approach assigns k-mers to ranks via a modulo-based hash function, storing them locally in a UPC++ distributed object while enabling synchronous distributed lookups during contig assembly. Through careful optimization of memory access patterns and synchronization mechanisms, we achieve strong scalability for moderate dataset sizes while maintaining correctness and adaptability for larger-scale genome assembly workloads.

1 Introduction

Genome assembly is a fundamental problem in computational biology, requiring the reconstruction of a complete genomic sequence from short DNA fragments known as k-mers. This process relies on de Bruijn [1] graphs, where k-mers form vertices, and their overlaps define edges, enabling the identification of contiguous sequences (contigs). Due to the sheer volume of k-mers and complex data dependencies, genome assembly is computationally demanding, making parallelization critical for scaling performance. [5] [2]

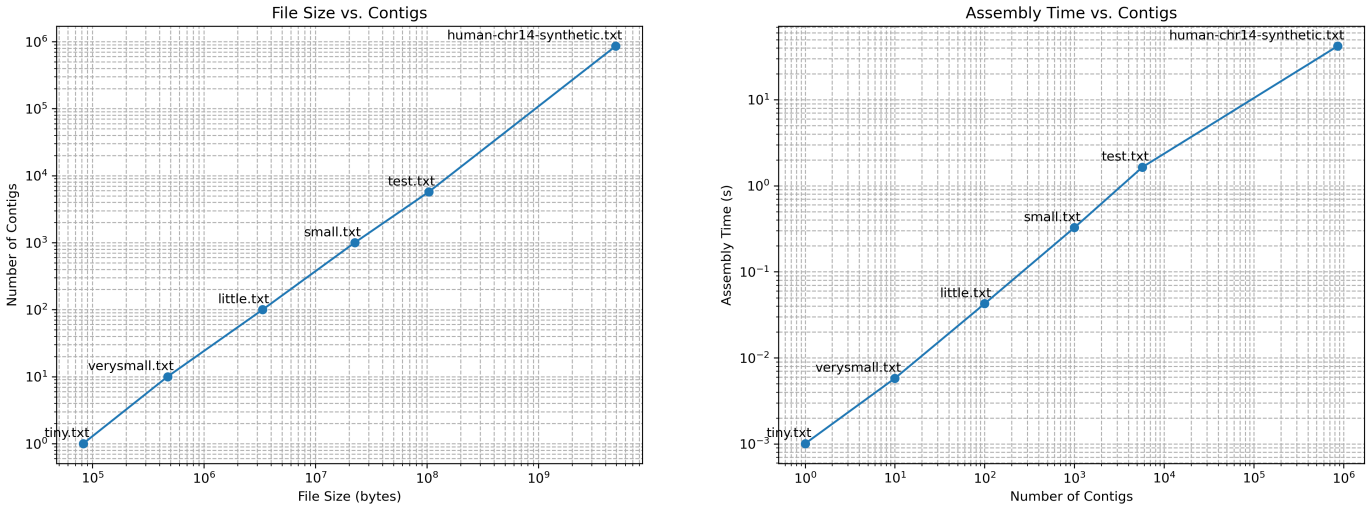
This project implements a distributed hash table using UPC++ to optimize k-mer insertion and traversal. Our approach partitions k-mers across processes using a modulo-based hash function, distributing the data among ranks to minimize contention and memory overhead. The hash table is wrapped in a UPC++ distributed object, allowing batched remote insertions via RPC to reduce communication costs. Contig assembly follows the de Bruijn graph structure using synchronous distributed lookups, ensuring correctness in extending sequences while maintaining efficiency. Through careful data partitioning, synchronization control, and memory access optimization, our implementation achieves strong scalability for moderate-sized genome assembly tasks while remaining adaptable for larger datasets.

2 Performance Results and Analysis

2.1 Impact of Contig Count and Assembly Complexity

Figure 1(a) illustrates the relationship between file size and contig count, as well as the impact of increasing contig counts on assembly runtime in the serial implementation. As expected, larger files

such as *human-chr14-synthetic.txt* produce significantly more contigs than smaller ones like *tiny.txt* or *verysmall.txt*. This growth reflects the increased complexity and data volume inherent in larger genomic datasets, making parallel computing crucial for handling large-scale genome assembly.



(a) Filesize vs Contigs

(b) Assembly time vs Contigs Count (Serial)

Figure 1: File sizes and Assembly Times vs Contig count (serial)

Figure 1(b) further illustrates that larger numbers of contigs correlate with longer processing times. On a log-log scale, the relationship appears roughly linear, suggesting that the assembly time increases proportionally with the number of contigs. This result confirms that the complexity of the dataset plays a major role in determining performance, justifying the need for distributed parallelism in handling large genomic inputs.

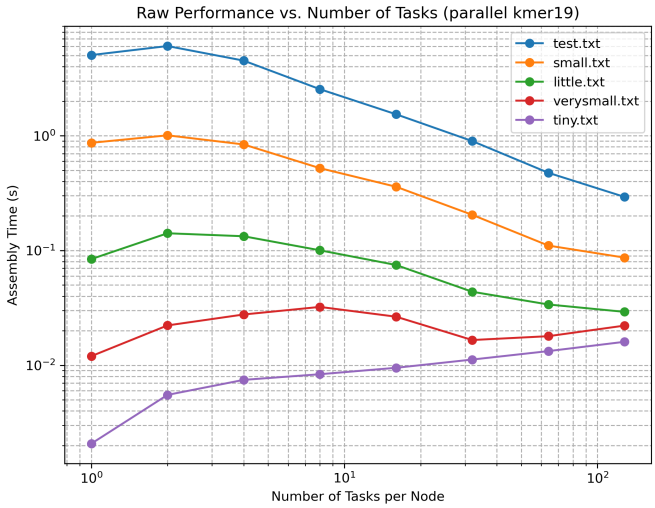
2.2 Raw Performance Trends

Our implementation of distributed genome assembly shows significant improvements in execution time when transitioning from a serial approach to parallel execution. The timing results demonstrate clear trends in insertion and assembly phases across varying levels of parallelism.

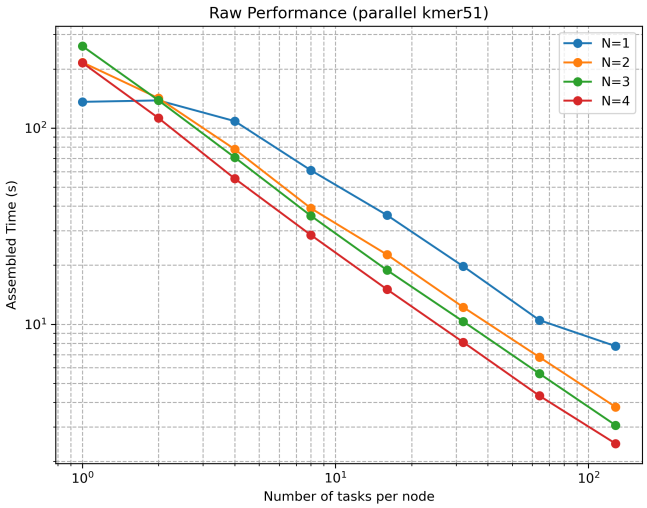
For insertion, increasing the number of parallel tasks leads to substantial reductions in runtime. Small datasets exhibit minor performance improvements, while larger datasets showcase significant speedups. The transition from a serial single-task execution to high task counts (e.g., 64 or 128 tasks per node) shows that insertion can be up to **ten times faster**. However, at higher levels of parallelism, the speedup plateaus, indicating potential contention or communication overhead.

Assembly time follows a similar trend but with more gradual improvements. While more tasks generally reduce execution time, k-mer traversal is inherently more compute-heavy than simple data insertion, meaning that speedup is not always linear. For larger datasets, particularly with $k = 51$, total assembly time decreases significantly as parallelism is introduced, dropping from **136 seconds in a single-task run** to **7.7 seconds at 128 tasks per node**.

Figure 2 presents raw performance results for varying task counts across datasets, illustrating the impact of increasing parallelism. We used the optimal node configuration (N = 4) to showcase our raw performance for kmer 19 (Figure 2(a)). All nodes are shown for kmer 51 (Figure 2(b)).



(a) kmer19 (per file/Node = 4)



(b) kmer 51 (per node count) (human-chr14-synthetic.txt)

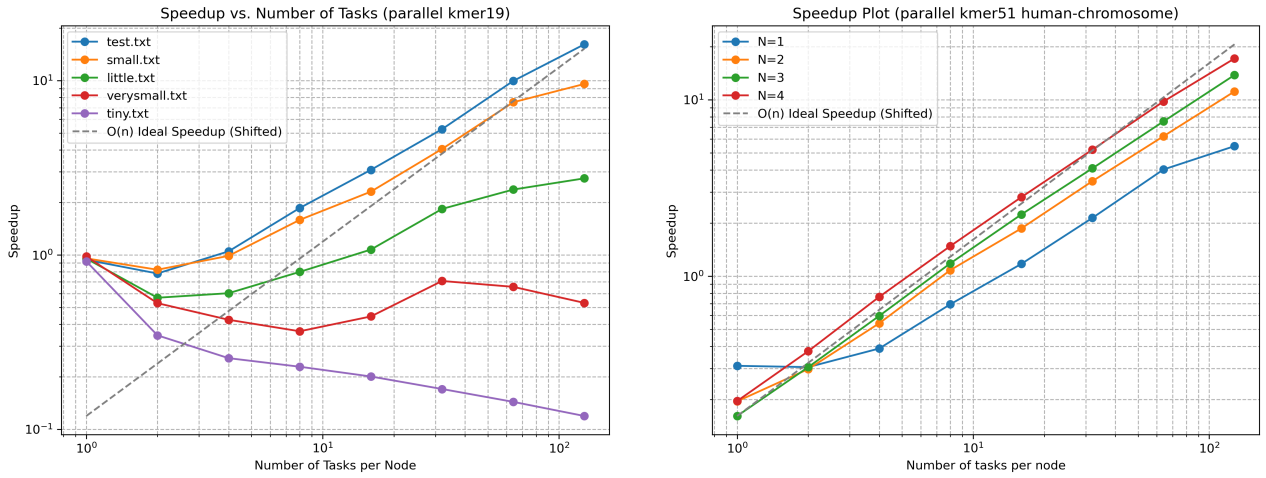
Figure 2: Raw Performance (parallel)

2.3 Speedup and Scalability

Our analysis indicates strong scaling properties, especially for insertion operations. The performance gains from parallelism follow the expected trend of increasing efficiency with more tasks, but diminishing returns emerge beyond 64-128 tasks per node.

For assembly operations, scalability is more variable. While increasing the number of tasks generally results in lower assembly times, nonlinear scaling effects appear. Some configurations exhibit higher assembly time at intermediate values of N (e.g., $N = 2$ and $N = 3$), suggesting that inter-process synchronization or communication contention may limit efficiency. Despite these variations, the largest dataset (*human-chr14-synthetic*) benefits significantly from high parallelism, improving from **42 seconds (serial)** to **2.46 seconds (N=4, 128 tasks per node)** for assembly time.

Figure 3 illustrates the speedup trends across different configurations, showing how our implementation scales relative to the baseline. We used the optimal node configuration (N = 4) to showcase our speedup for kmer 19 (Figure 3(a)). All nodes are shown for kmer 51 (Figure 3(b)).



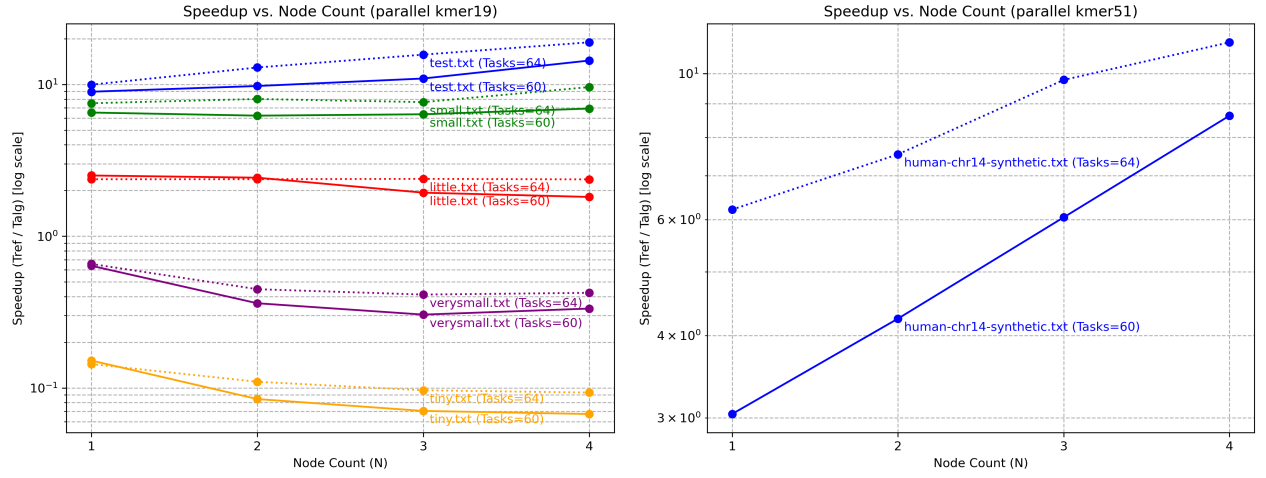
(a) Speedup kmer 19 (Parallel/Node = 4)

(b) Speedup kmer 51 (Parallel)

Figure 3: Parallel Speedup

2.4 Notable Observations

Insertion bottlenecks appear at low parallelism levels, where execution times can be up to **10× slower** compared to high-parallel configurations. This highlights the advantage of batching RPCs to minimize communication overhead. However, scaling insertion beyond **64-128 tasks** yields diminishing returns, suggesting possible saturation effects in communication efficiency. Assembly time follows a nonlinear scaling pattern, influenced by synchronization and memory access overhead. For larger datasets, higher parallelism provides notable speedups, but for medium-scale problems, contention and inter-task coordination introduce variability in performance. Figures 4 further illustrate speedup comparisons across different node configurations.



(a) Speedup vs. Nodes (kmer 19 w/60 and 64 tasks)

(b) Speedup vs. Nodes (kmer 51 w/60 and 64 tasks)

Figure 4: Parallel Speedup vs Nodes Comparisons (60 and 64 tasks)

2.5 Limitations and Resource Availability

We were unable to allocate 8 nodes due to the infrequent availability of resources, which delayed job scheduling. Therefore, results are presented for up to 4 nodes only. Despite this limitation, the strong scaling behavior and efficiency of our distributed genome assembly method remain evident across the tested configurations.

3 Implementation Description

Our distributed genome assembly algorithm builds upon the original design, significantly improving scalability and efficiency. The primary enhancements include restructuring the hash table, optimizing communication patterns, and refining synchronization mechanisms to minimize computational overhead. [4] [3]

3.1 Distributed Hash Table Design

The initial implementation relied on a globally shared array for k-mer storage, leading to bottlenecks in rank 0 and inefficient remote memory accesses. To improve scalability, we transitioned to a fully distributed hash table, where each process manages its partition using UPC++ `dist_object`. K-mers are assigned to ranks based on a hash function, `hash(key) % world_size`, ensuring balanced storage while reducing contention.

To optimize insertion efficiency, we implemented **batched remote procedure calls (RPCs)**. Instead of sending individual RPCs per k-mer insertion, our algorithm groups k-mers by destination rank and sends them collectively in a single request. This approach reduces network latency, lowers communication overhead, and improves performance for larger datasets.

3.2 Contig Assembly and Synchronization

Genome assembly follows the de Bruijn graph model, where k-mers are extended until terminal nodes are reached. We enhance traversal efficiency by performing local lookups whenever possible while issuing targeted RPCs only for remote accesses. Although asynchronous lookups were considered, we opted for synchronous `find()` calls to ensure correctness and maintain stability.

To maintain consistency across ranks, we employ **global synchronization** via `upcxx::barrier()`. Synchronization barriers ensure that all processes complete insertion and assembly phases before advancing, preventing premature reads or updates that could lead to incorrect genome reconstruction.

3.3 Algorithm Overview

The genome assembly process follows a structured sequence of steps to ensure efficient parallel execution. First, each process performs **preprocessing and partitioning**, where k-mers are read from the input dataset and distributed across ranks using a hash-based partitioning scheme. This ensures an even workload distribution and optimizes memory usage. Next, **distributed hash table insertion** takes place, where k-mers are inserted using batched remote procedure calls (RPCs), significantly reducing network communication overhead compared to one-by-one insertions. During the **contig assembly** phase, start nodes–k-mers marked with a backward extension identifier 'F'– are identified as entry points for contig construction. The traversal then extends through distributed

lookups, fetching k-mers either locally or via targeted RPCs, continuing until a terminal node is reached. Finally, the **output and performance evaluation** step records execution time metrics and writes the assembled contigs to output files for downstream analysis. This refined approach eliminates bottlenecks from the starter code, improving memory efficiency and enabling scalable parallel genome assembly.

4 Optimizations

To improve efficiency and scalability in distributed genome assembly, our implementation incorporates several key optimizations tailored for UPC++. These optimizations focus on balanced workload distribution, efficient memory access, minimized communication overhead, and controlled synchronization.

4.1 Hash-Based Partitioning

To distribute k-mers evenly across ranks, we employ **hash-based partitioning**, where each k-mer is assigned to a process using the hash function `hash(key) % world_size`. This ensures a balanced workload, reduces contention, and enables efficient memory usage across distributed processes. By eliminating the need for a central data structure, our approach significantly improves scalability while preventing bottlenecks caused by excessive remote accesses.

4.2 Distributed Hash Table Storage

Each rank maintains its portion of the hash table using UPC++ `dist_object`, providing localized storage while allowing for efficient remote access. This design eliminates global memory congestion, ensuring that each rank independently manages its assigned k-mers. The distributed object mechanism enables direct, one-sided communication, reducing overhead compared to standard message-passing approaches.

4.3 Batched Remote Procedure Calls (RPCs)

To optimize k-mer insertion and retrieval, we implement **batched RPCs**, where k-mers destined for the same rank are aggregated and transmitted in a single remote procedure call. This reduces the number of network messages exchanged, minimizing latency and overall communication costs. Instead of performing individual insertions that would overload the system with small messages, batching operations enable more efficient processing and lower synchronization overhead.

4.4 Optimized Lookup Strategy

For efficient genome assembly traversal, we differentiate between local and remote lookups. **Local queries** are performed directly within the rank's stored partition, eliminating unnecessary network communication. When a k-mer resides on a different rank, a **targeted UPC++ RPC** is issued to retrieve the required data. By prioritizing local operations and limiting remote calls to essential cases, we minimize communication overhead while preserving fast query times.

4.5 Synchronization and Consistency Control

To maintain data consistency across ranks, we enforce **global synchronization via** `upcxx::barrier()` at key points in execution. Barriers ensure that all processes complete insertion and assembly phases before proceeding, preventing premature reads or writes that could lead to incorrect genome reconstruction. Synchronization is particularly important after k-mer distribution and contig assembly to maintain coherence across all ranks.

4.6 General Design Considerations

Our optimizations align with the broader goal of scalability and correctness in distributed genome assembly. The use of a decentralized distributed hash table avoids reliance on a single coordinating process, reducing contention and enabling efficient parallel processing. Remote memory operations are streamlined using UPC++'s lightweight communication model, ensuring minimal synchronization overhead while allowing rapid access to remote partitions. By batching insertions, prioritizing local queries, and enforcing controlled synchronization, our approach achieves significant improvements in computational efficiency. Additionally, future work can explore asynchronous lookup optimizations and memory layout refinements to further enhance performance.

4.7 Alternative Implementation Using MPI or OpenMP

If implemented using **MPI**, data distribution would rely on explicit message passing via `MPI_Scatter` or `MPI_Broadcast` to assign k-mers across processes. Instead of UPC++'s `dist_object`, each rank would manage its own hash table and perform insertions through `MPI_Send` and `MPI_Recv`, or leverage non-blocking communication (`MPI_Isend`/`MPI_Irecv`) to reduce synchronization delays. Contig assembly would involve direct rank-to-rank queries using point-to-point messaging, requiring careful optimization to prevent excessive communication overhead. `MPI_Barrier` would ensure global consistency at key computation phases.

Alternatively, if implemented using **OpenMP**, genome assembly would be parallelized within a shared-memory system. A single hash table would be maintained across threads, protected by `#pragma omp critical` sections or atomic updates to prevent race conditions. K-mer insertions and lookups would be executed in parallel using `#pragma omp parallel for`, distributing workloads efficiently across CPU cores. Contig assembly would benefit from `#pragma omp task` directives, allowing multiple sequences to be constructed concurrently. Synchronization would rely on `#pragma omp barrier` to coordinate thread execution.

While MPI enables scalability across multiple nodes, OpenMP is best suited for shared-memory environments where communication costs are lower. Our choice of UPC++ leverages one-sided communication advantages, eliminating explicit message passing while allowing direct access to remote partitions. Future implementations could explore hybrid MPI+OpenMP strategies to further optimize memory usage and parallel execution.

5 Conclusion

Our implementation of distributed genome assembly using UPC++ demonstrates significant scalability and efficiency improvements over the baseline approach. By leveraging a distributed hash table with partitioned key-space and batched remote procedure calls (RPCs), we effectively reduced communication overhead and improved performance across both insertion and assembly phases.

Analysis of execution time shows that **batched insertions** contribute the most to overall speedup, particularly at higher task counts, where communication overhead is amortized efficiently. The use of **synchronous remote lookups** ensures correctness during contig assembly but introduces synchronization delays, particularly at moderate scales. While insertion exhibits near-linear scalability up to 128 tasks per node, contig assembly performance is affected by inter-process coordination and communication overhead.

Overall, this work highlights the suitability of UPC++ for large-scale distributed genome assembly. The combination of efficient partitioning, one-sided communication, and synchronization mechanisms ensures correctness while enabling parallelism. Future optimizations could focus on incorporating **asynchronous lookups** to further reduce latency, minimizing synchronization points, and refining memory layouts to maximize data locality. These improvements will enhance the scalability and efficiency of genome assembly for even larger datasets.

6 Team Contributions

Brent led the UPC++ design and implementation, taking primary responsibility for coding the distributed hash table, optimizing communication via batched RPCs, and ensuring overall system correctness. He conducted performance tuning, profiling, and scalability testing across datasets, and generated the final performance plots and analysis presented in the report.

Puja contributed to testing and validation efforts, helping ensure correctness across datasets and configurations. She also assisted in identifying useful performance metrics and selecting plots that best illustrated key trends in the results.

Kofi contributed to testing and results collection, helping validate runtime behavior and ensure the accuracy of performance data. He also participated in reviewing experimental output and contributed to the analysis of timing and scalability trends.

References

[1] Aydin Buluc et al. "Scalable High-Performance Genome Assembly using UPC++". In: *Proceedings of SC15* (2015). URL: https://people.eecs.berkeley.edu/~aydin/sc15_genome.pdf.

[2] et al. Buluc. "Parallelizing Irregular Applications for Distributed Memory Scalability: Case Studies from Genomics". In: *Technical Report* (2020). URL: https://escholarship.org/content/qt1400c4rh/qt1400c4rh_noSplash_a3cca84fb0d7680a6f6809fecedc182f.pdf.

[3] Lawrence Berkeley National Laboratory. *UPC++ Distributed Map Documentation*. 2025. URL: https://bitbucket.org/berkeleylab/upcxx/src/master/example/prog-guide/dmap.hpp.

[4] Lawrence Berkeley National Laboratory. *UPC++ System Documentation: Perlmutter*. 2025. URL: https://bitbucket.org/berkeleylab/upcxx/wiki/docs/system/perlmutter.

[5] et al. Li. "Why are de Bruijn graphs useful for genome assembly?" In: *PLoS Computational Biology* (2017). URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC5531759/.