University of California Berkeley

Computer Science Department

CS 267

Assignment 3: Parallelizing Genome Assembly

Submitted by
Cristobal Pais
Alexander Wu
CS 267
March 23rd
Spring 2018

Contents

Assignment 3: Parallelizing Genome Assembly

C.Pais, A.Wu

March 23rd 2018

Abstract

A parallel implementation in C++ using the unified parallel C++ (UPC++) extension for the De novo DNA genome assembly pipeline - assembly of strands of DNA in new genomes, without the use of a reference genome - is developed in the context of distributed memory programs for applications with irregular communication patterns. Single and multi-node strong scaling efficiency is tested on the NERSC's Cori High-performance computer using up to 64 threads and 32 ranks per node respectively. Using a distributed hash table, the parallel implementation is able to obtain strong scaling efficiency factors up to 40% for the single-node experiments and 50% for the multi-node implementation. A comparison of alternative implementation choices and code structure with respect to previous parallel frameworks (MPI, OpenMP) is performed. Future work and optimizations are discussed.

1 Introduction

During the past decades, the study of the genome has been one of the most developed, active, and interesting areas for a variety of scientific research projects. With the continuous improvement of computational power and storage hardware, the possibility of analyzing and studying the genetic code of different species has become a reality, allowing the researchers to gain relevant insights in developing areas such as genetic treatments and genome modification. However, the size of the instances (data), as well as the inherent characteristics of some biological processes, make this analysis a difficult and demanding computational challenge.

In order to tackle it, researchers have invented a series of alternative methods to construct a genome, avoiding problems such as "out-of-memory" crashes when trying to read the whole DNA code in a Naive serial approach. In this project, we will parallelize and optimize one method based on the construction of a graph (called *de Bruijn* graph) where each node corresponds to a small fragment - generated via a shotgun **[3]** sequencing approach - of the original DNA strand, such that we will be able to reconstruct the original chain of nucleotides in a more efficient way. Therefore, as indicated in the problem statement, we will be focusing on the parallel construction and traversal of the *de Bruijn* graph (with a few assumptions and simplifications, as indicated in section 2.3).

The Unified parallel C++ extension (UPC++) is a library that allows an efficient and effective Asynchronous Partitioned Global Address Space (APGAS) - single program, multiple data (SPMD) - programming, where communication is irregular or fine grained. UPC++ parallel programs scaling performance is close to hardware speeds thanks to the utilization of the low-overhead GASNet-EX communication library. Therefore, the code is optimized for running programs under a distributed-memory environment where each thread (rank) has access to local memory.

Using this library to develop a distributed-memory hash table (DMHT), a parallel version of the k-mer traversal algorithm for the *de Bruijn* graph is developed, seeking to obtain the best single and multi-node strong-scaling performance possible in the Cori high-performance computer.

The structure of the report is as follows: In section 2, the main optimization methodology is described as well as the member's contribution and hardware description. Section 3 describes the tested optimization approaches and their potential impact as well as defining and describing the main algorithm developed. In

section 4, results are discussed for both the serial and parallel versions concluding the advantages/disadvantages of each implementation. Section 5 contains a series of short answers to extra credit questions indicated in the problem statement. Finally, section 6 contains the conclusions and future work of the project.

2 Methodology

The main methodology of this project consists of performing an optimization of the original serial hash table implemented for the construction and traversal of the *de Bruijn* graph (in C++ with UPC++) provided with the course material, analyzing the different programming techniques/tricks that can be implemented in order to parallelize it and exploit the UPC++ advantages, obtaining a better and strongly-scalable performance. The performance is measured by calculating the strong-scaling efficiency obtained for different experimental instances ranging from sizes - length of the full chain - $n \in \{3,599;\ 20,654;\ 147,763;\ 996,112;\ 4,623,181;\ 37,738,594;\ 133,587,521\}$ provided with the assignment, where we keep the problem size constant while increasing the number of processes (single-node) or ranks per node (multi-node) allowing us to measure the parallel scalability of the code.

As mentioned, experimental instances differing in the n value are tested, for testing and comparing the performance of our code with respect to the serial implementation. Depending on the techniques applied for optimizing the original algorithm, a series of parameters are optimized for each approach, performing the experiments while keeping the best solutions achieved.

A series of plots - strong-scaling efficiency, speedup comparison, etc. - are generated for each experiment in order to visualize the performance of our parallel implementation focusing on three instances: *test*, *large*, and *human* genome code.

2.1 Contribution

During the development of the project, each member of the team develops its own experiments, sharing his results in a common Github repository for checking the current state of the implementations while re-using code from other members code. In addition, cross-testing of other members' code is performed. After performing a series of experiments, the best implementation is selected as the main submission file.

Based on the experience and preferences of the members of the group, the members' contributions and tasks can be summarized as follows:

- **Cristobal Pais**
  - Main tester.
  - Editor and main writer of the current report.
- **Alexander Wu**
  - Main developer of the distributed hash table for the parallel implementation.
  - Contribution to the report.

2.2 Hardware & Software

The optimization and parallelization of the construction and traversal of the *de Bruijn* graph are developed for a specific hardware and runtime environment from The National Energy Research Scientific Computing Center (NERSC). All experiments, benchmarks, and performance results are implemented using the following hardware and software:

1. NERSC's Cori supercomputer: Phase I

- Intel® Xeon™ Processor E5-2698 v3 ("Haswell") at 2.3 GHz

  - 2 sockets per chip (32 cores per node)
  - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
- Memory
  - 64 KB 8-way set Level 1 cache (32KB instructions, 32 KB data)
  - 256 KB 8-way set Level 2 cache
  - 40 MB 20-way set Level 3 cache (shared per socket)
- Software
  - SUSE Linux version 4.4.74-92.38-default
  - Built with PrgEnv-intel and PrgEnv-gnu (CC)/ gcc version 4.8.5
  - Default Compiler flags: -upcxx-meta PPFLAGS -upcxx-meta LDFLAGS -upcxx-meta LIBFLAGS

2.3 Assumptions

Based on the statement of the problem, our implementation satisfies the following assumptions:

- Input: a set of unique k-mers (sequences of DNA bases of length k).
- The k-mers are guaranteed to be unique and to overlap one another by exactly $k-1$ bases.
- The special base F indicates the beginning or end of a read inside special k-mers.
- Development and application of methods to preprocess and remove potential errors from the short reads are outside of the scope of this project.

3 Solution Description

In this section, we present the main strategies applied for parallelizing and optimizing the original code provided with the statement of the problem.

3.1 Hash Table

Our hash table remains mostly unmodified from the serial hash table except that we use globally distributed arrays for *data* and *used*. The hash table is partitioned across the processes so each of the n processes receives exactly $1/n$ of the arrays backing the hash table.

We default initialize the *used* array of $std :: int32\_t$'s to all 0's. When a process attempts to insert into global memory, it performs a $upcxx :: atomic\_fetch\_add()$ on the slot in $used()$ it is attempting to write to. If the fetch returns 0, then the caller has successfully claimed the spot and performs a remote put of the *kmer_pair* into the corresponding spot in *data*. If the fetch returns a number greater than 0, then the slot has already been occupied, so the algorithm continues to probe until an empty spot is available. Because the hash table is statically sized by the number of lines in our input file, we keep the maximum load of the hash table at 50%, so every insert is guaranteed to terminate.

Our $find()$ algorithm is a simple remote get from the *data* array. It does not even check the *used* array because we are guaranteed that every k-mer will be found in the hash table.

3.2 Other techniques tested

At first, we kept the data distributed but kept k instead of k/n sized buffers in each process. During the $insert()$ step, we would use RPC to send the k-mer to the appropriate node, then add it to a vector local to that node (in a critical section). After all the k-mers have been "distributed", the process would take the k-mers in its own vector and add it to the portion of the global hash table local to the node.

We decided to move away from this strategy because we wanted to avoid RPC (as indicated in the course website on Piazza), and decided to use the built-in atomic functions instead of rolling our own "hacky" solution. The performance turned out to be about the same, but the memory overhead p times more, where p is the number of processes.

3.3 Environment Variables

As discussed in the course's website on Piazza, two main environment variables should be modified in order to be able to run all the experiments (larger instances such as *human* or *large*):

i) $UPCXX\_SEGMENT$: Values above 2048 where enough to avoid memory allocation problems when testing the larger instances such as the humans' 14th chromosome.

ii) $GASNET\_MAX\_SEGSIZE$: After some experiments, values above 8GB are needed for running the larger instances.

Important is to notice that this is one of the drawbacks that the group identified when programming with UPC++: understanding the meaning of the error messages was not as clear as we expected, spending a lot of time debugging and trying to find what part of the code was triggering bad allocation errors, until we realize that it was due to the values of these environment variables.

4 Results & Discussion

In this section, we present the main results obtained from a series of experiments including the optimization techniques introduced in section 3.

4.1 Preliminary results: Vanilla version

Using the vanilla serial hash table implementation provided with the assignment statement, we obtain a series of preliminary results for benchmarking our optimized code.

4.1.1 Execution time analysis

When running the AS-IS/Vanilla version of the code provided with the assignment, we can clearly see in Figures 1 and 2 that the execution is not very efficient when increasing the size of the instance n: we can expect that solving larger instances (real-life instances) than the 14th human chromosome example - $n = 133,587,521$ - will lead this implementation to very long execution times and potentially "out-of-memory" problems due to the size of the whole DNA chain (remembering that the human instance represents the genome contained in only one of our chromosomes).

Figure 1: Vanilla serial hash table implementation for the de novo Genome assembly using the *de Bruijn* graph algorithm (all instances).

From the plot, we can see that most of the time (around 90%-95%) is spent performing the graph traversal operation while the graph construction needs a very limited amount of time to be completed - as expected from the problem statement. In Table 1 we can see the details of the results obtained for all instances $n \in [3599, 133587521]$.

In order to perform meaningful comparisons with respect to our parallel implementation performance, we will focus the analysis in the following sections on the three main instances: *test*, *large*, and *human*. The correctness of our implementation has been tested via the *test* flag provided with the original code of the project, without noticing any error during our runs.

Table 1: Serial hash table implementation for all instances with detail of execution times.

Instance	**Construction** [s]	**Traversal** [s]	**Total** [s]
Very Small	0.000	0.008	0.008
Tiny	0.000	0.002	0.002
Little	0.004	0.051	0.055
Small	0.036	0.354	0.390
Test	0.488	3.166	3.654
Large	3.258	17.155	20.413
Human	10.962	57.482	68.444
Average	2.106	11.174	13.281

Based on the results, we can see that the largest instance (human 14th chromosome data) needs around one minute and ten seconds of execution time for getting the full DNA chain while the *large* and *test* instances need only 20 and while 3.6 seconds for running the algorithm. Therefore, our parallel code should be able to improve this performance after adding a certain amount of threads to our algorithm.

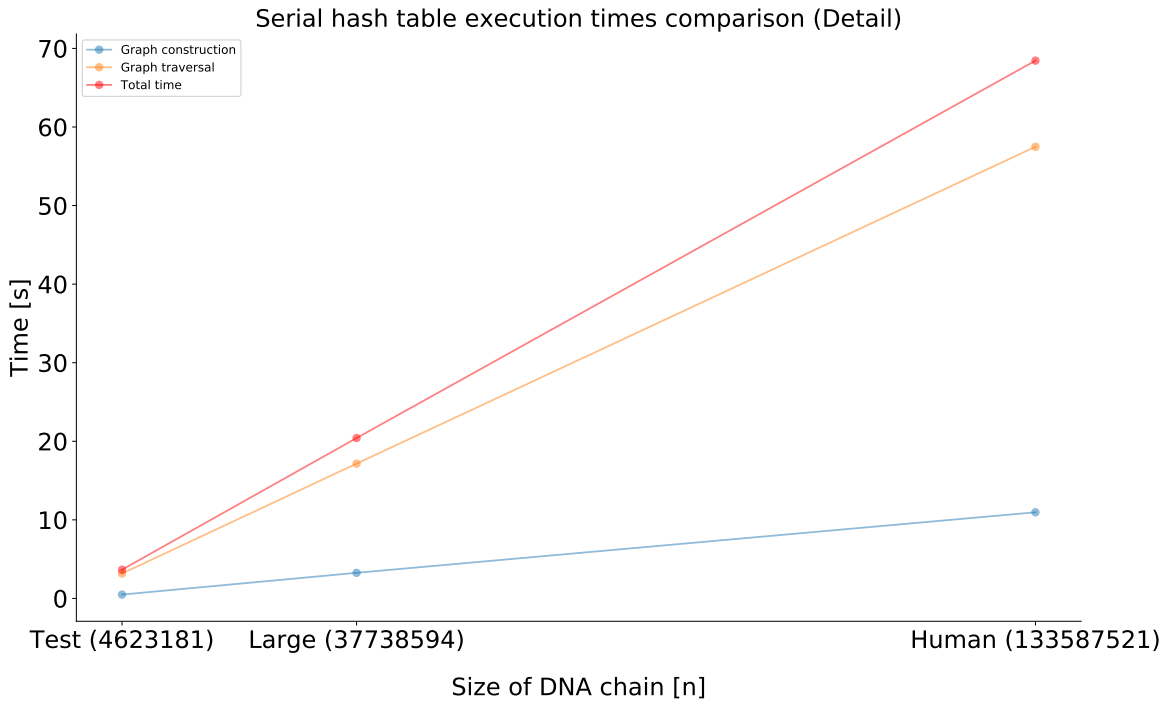


Figure 2: Detailed results for instances *Test, Large, and Human* using the Vanilla serial hash table implementation.

4.2 Optimized Parallel implementation

Based on the ideas discussed in section 3, we implement our simple but effective parallel hash table using UPC++ in order to perform a series of test using multiple threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ inside a single node of Cori as well as using 32 threads per node N in a multi-node setting, testing $N \in \{1, 2, 4, 8\}$.

Two main performance metrics are analyzed: (1) Execution times and (2) speedup factor/efficiency of our parallel implementation. In addition, a comparison with the performance of the original serial approach provided with the statement of the assignment is developed, identifying the t is developed, identifying the threshold (total number of parallel threads) at which our parallel implementation is able to beat the serial code's performance.

4.2.1 Execution times

In this section, we compare the performance of our parallel implementation in terms of execution times (construction & chain assembly times) for a different number of threads running inside the same node (single-node analysis) as well as in multiple nodes.

Single-Node

Looking at the results from Figures 3, 4, and 5 we are able to identify the following main patterns: (1) execution times with one thread under our parallel implementation are significantly worse than the ones obtained by the serial vanilla implementation for all instances, indicating that our parallel hash table is not useful and efficient (not suitable) for a serial implementation due to the $find()$ and storing logic implemented in our distributed hash table: writing to global memory is more expensive than writing to local one as well as the fact that we use atomic operations to avoid synchronization, (2) using two threads tends to be worse than only one, due to communication overhead (communication is costly), and (3) adding more and more threads allows us to obtain better performances - decreasing times - across all instances but in a decreasing rate fashion: differences are negligible between 32 and 64 threads as expected since each node of Cori counts with 32 cores, being useless to add extra threads for boosting the performance after this threshold. Largest improvement tends to happen near 4-8 threads.

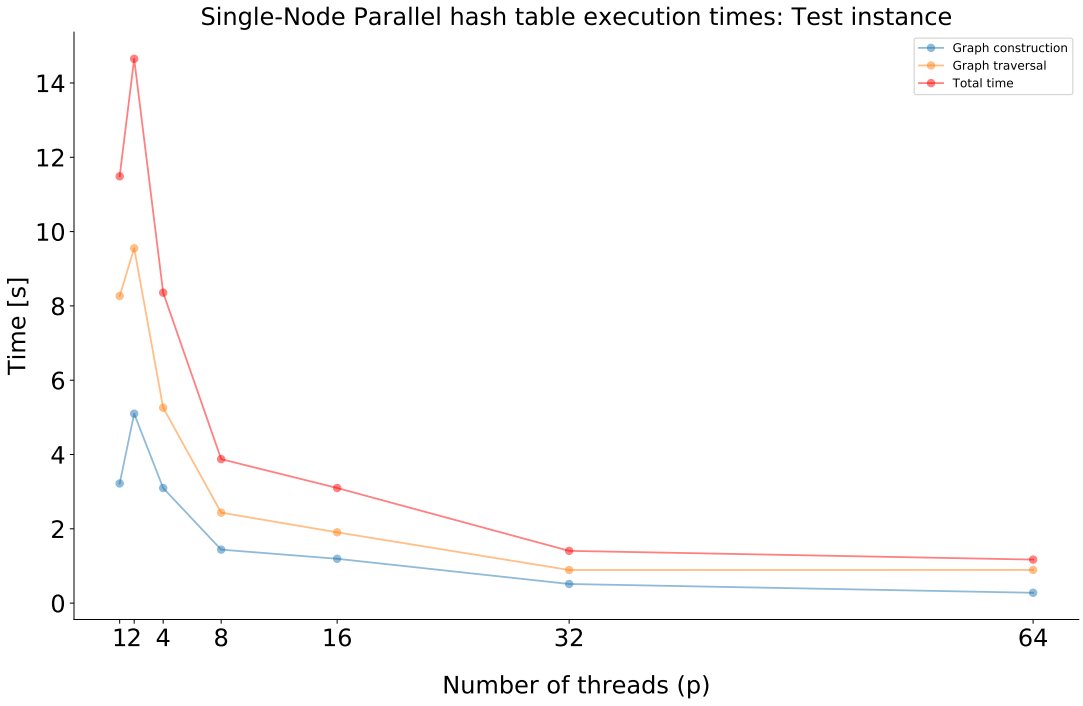


Figure 3: Test instance execution times for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running in a single node.

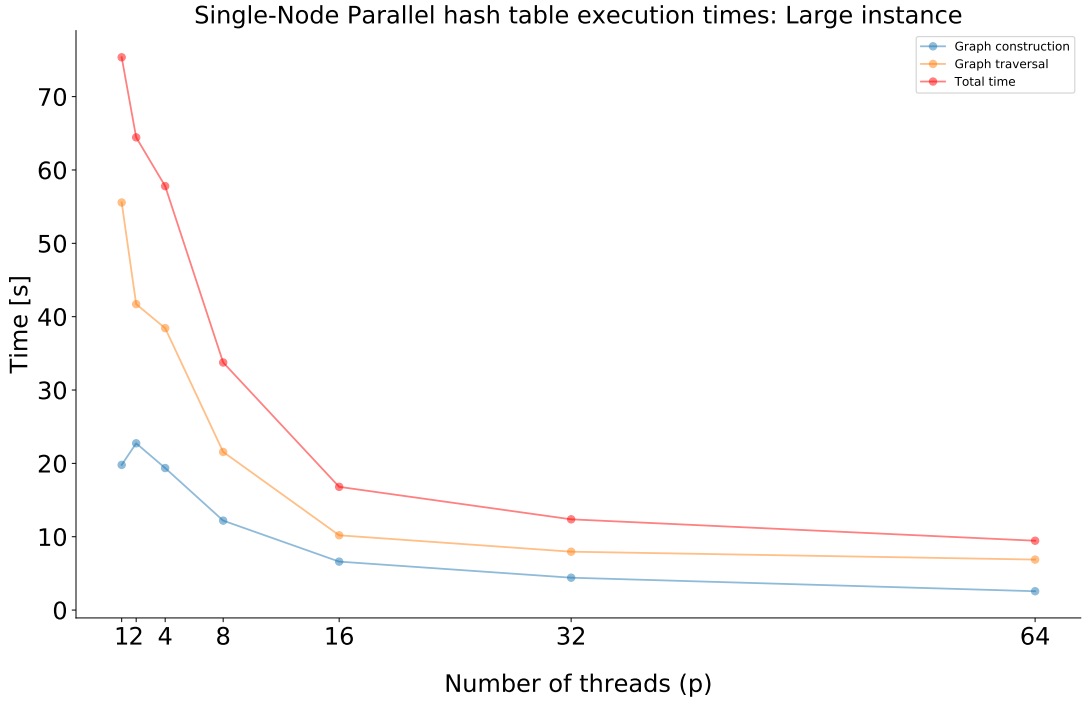


Figure 4: Large instance execution times for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running in a single node.

Comparing the results of our parallel implementation with the serial code performance (see Table 2 for details), we can see from the plots that at least 8, 16, and 32 threads are needed for the *test*, *large*, and *human* instances, respectively, for obtaining better execution times than the provided serial version, indicating us how relevant is to take into account the communication cost when developing and testing parallel implementations.

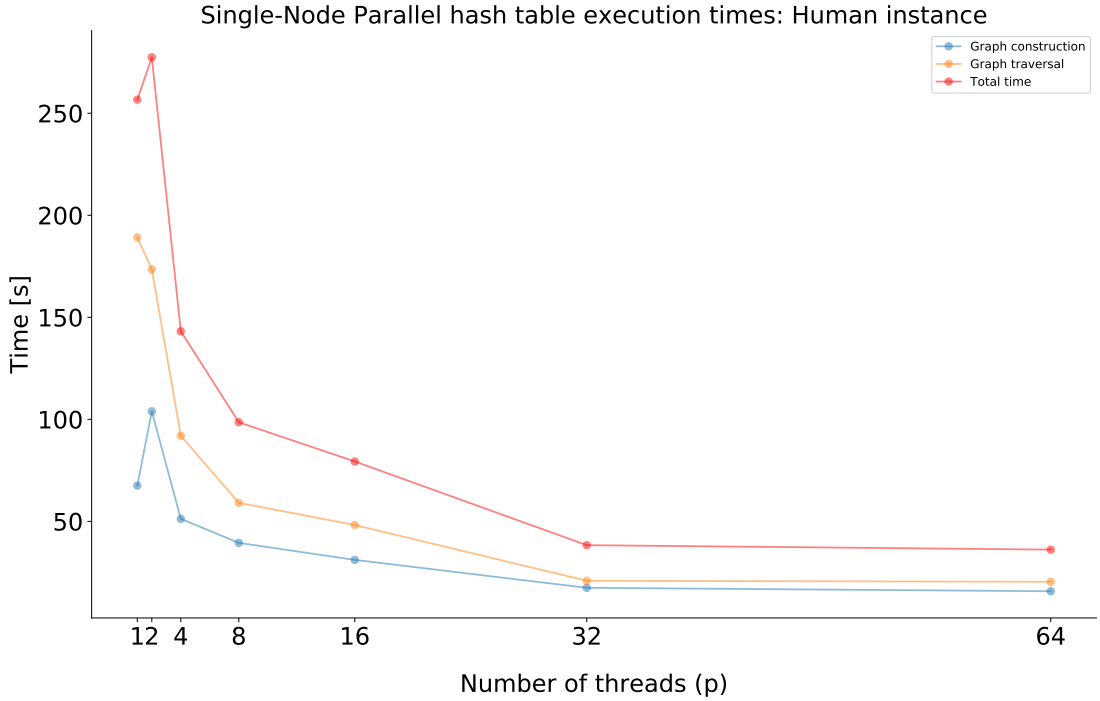


Figure 5: Human instance execution times for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running in a single node.

Based on the comparison results from Table 2, we can see that the best results for the single-node experiments are obtained for the *test* instance, reaching execution times around 30% of the original time

needed by the serial implementation, followed by a 46% and a 53% for the *large* and *human* instances, respectively. In addition, we are able to see that the decreasing pattern is very similar across all three instances when adding more threads inside the same node: an initial loss in performance is obtained when adding threads due to the communication overhead and then the performance is improved as we add more threads to the execution, being able to beat the serial implementation when using at least 8 threads (in the *test* instance).

No erratic patterns are identified besides the initial performance loss due to communication overhead. Interesting is to note that in contrast to the serial version where the graph traversal operation was the bottleneck of the algorithm, the execution time of our parallel implementation is almost evenly divided into construction and traversal operations due to the new hash table distributed logic.

Table 2: Single-Node Experiments: ratio of parallel/serial execution times. Best results are in green, worst in red.

Threads	Human	Large	Test
1	3.75	3.69	3.14
2	4.05	3.16	4.01
4	2.09	2.83	2.29
8	1.44	1.65	1.06
16	1.16	0.82	0.85
32	0.56	0.61	0.39
64	0.53	0.46	0.32

Multi-Node

In this case, we can see from the results in Figures 6, 7, and 8 that the execution times patterns are very similar to the ones obtained for the single-node experiments: a significant performance loss is experienced when adding a second node (64 threads) with the *test* and *human* instances due to communication overhead while the *large* instance shows a monotone decreasing pattern. Then, a similar performance is obtained with 128 threads (4 nodes), reaching the best performance when using 8 nodes with a total of 256 threads.

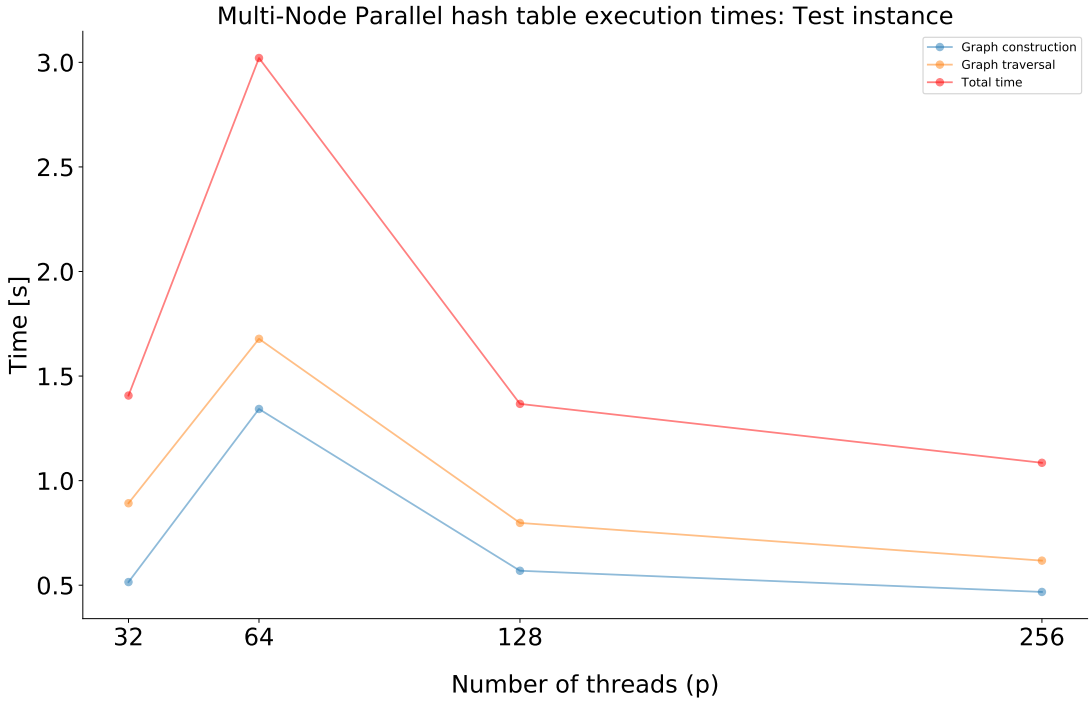


Figure 6: Test instance execution times for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

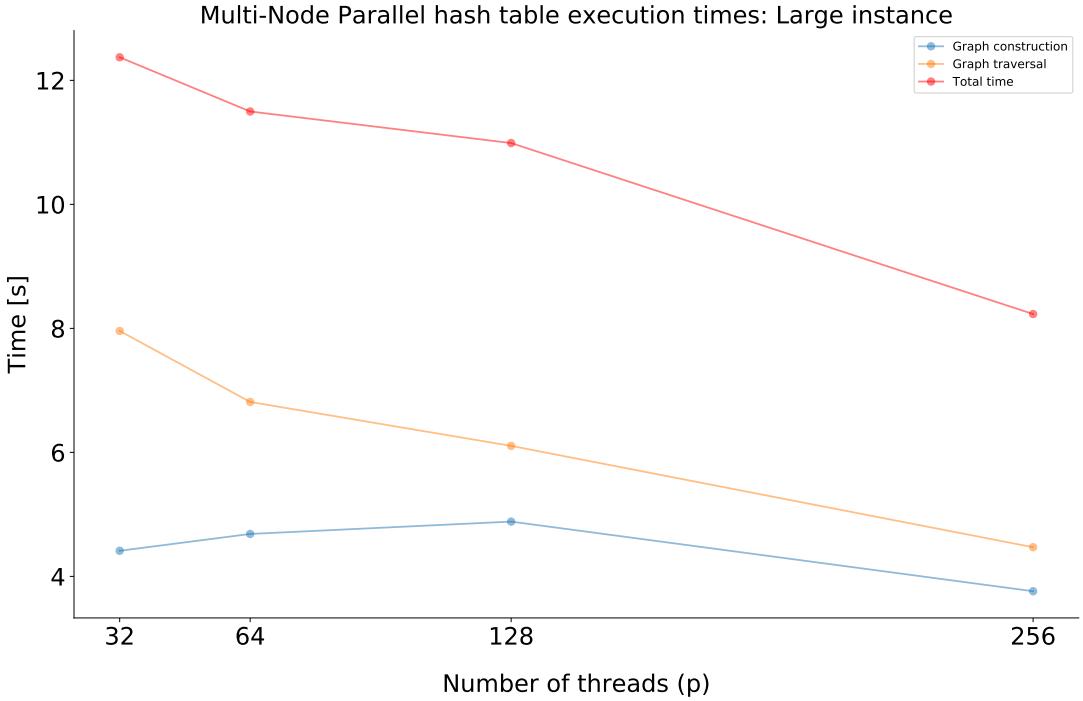


Figure 7: Large instance execution times for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

Comparing the results of our multi-node parallel implementation with the serial code performance (see Table 3 for details), we can easily see - as expected - that all execution times are better than the ones reached by the serial version. However, using less than 256 threads does not give a significant performance boost due to the already mentioned communication overhead. In addition, we can notice that using 64 threads with 2 nodes obtains worse performance than using 64 threads in a single-node fashion due to communication issues.

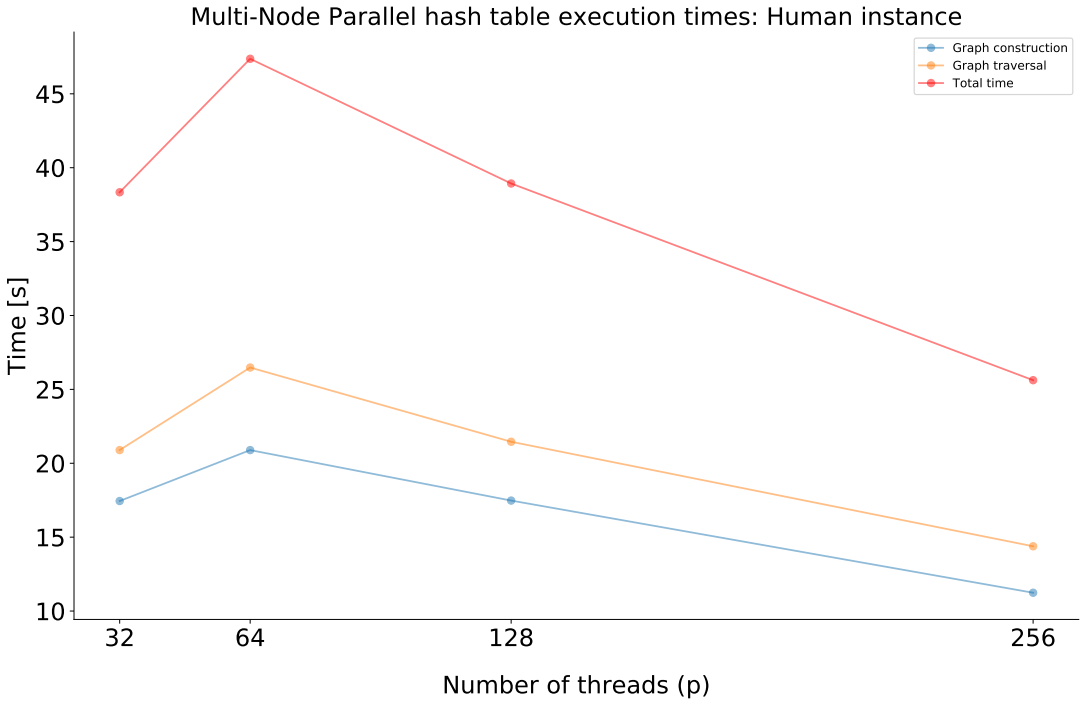


Figure 8: Human instance execution times for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

Therefore, our multi-node parallel approach is not suitable for the genome instances tested when the number of nodes used is less than 4: in that situation, a single-node approach using 64 threads is able to obtain better results - mainly due to communication overhead - without incurring a larger usage of resources (with the corresponding time on queue in an HPC).

Table 3: Multi-Node Experiments: ratio of parallel/serial execution times. Best results are in green, worst in red.

Threads (32 per node)	**Human**	**Large**	**Test**
32	0.56	0.61	0.39
64	0.69	0.56	0.83
128	0.57	0.54	0.37
256	0.37	0.40	0.30

Again, as with the single-node experiments, we can see that the construction and traversal times are very similar in both magnitude and behavior, a situation that was not the case for the serial version where the traversal time was the dominant term of the total execution time - as expected - of the algorithm. The explanation is the same as before: due to the structure of our distributed hash table, we expected to increase the construction time of the algorithm.

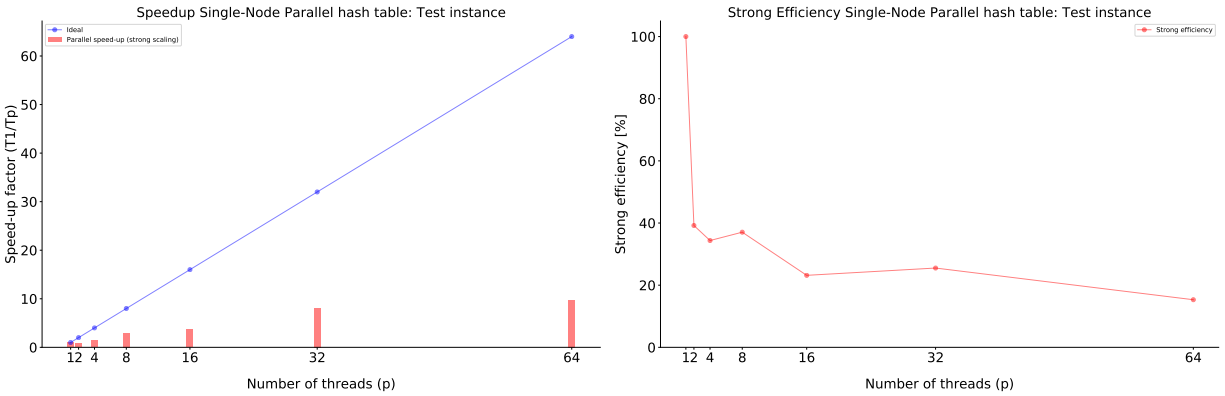
4.2.2 Speedup analysis

One of the classic challenges when implementing a parallel approach for a particular problem is to be able to reach good scalability of the speedup factors when solving different instances (n), used as key indicators of the performance of the overall parallel implementation. Therefore, the main objective of our implementation is to be able to obtain a significant performance boost with respect to the optimized serial version. However, since this is our first non-embarrassingly parallel project and based on the execution times obtained and discussed in the previous section, we expect to obtain strong efficiency factors as low as 20%.

Single-Node

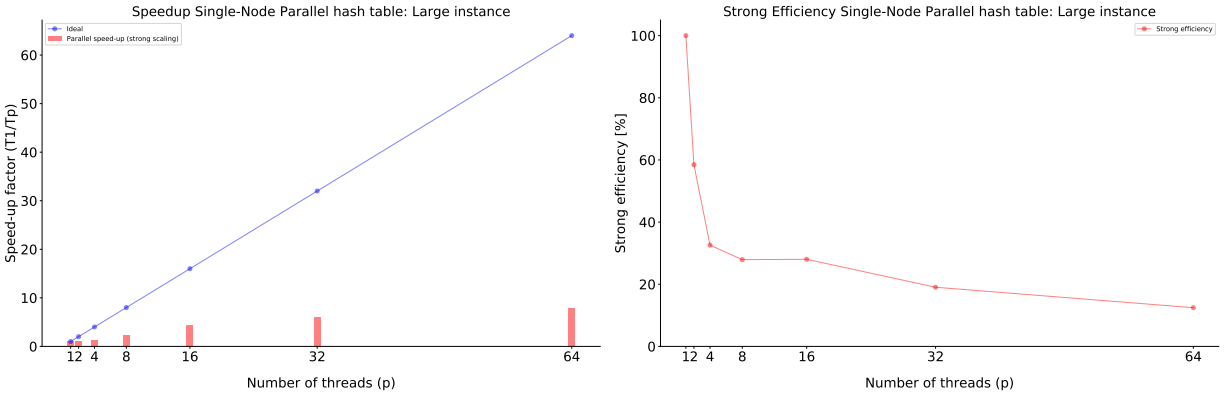
After generating the speedup and strong-scaling efficiency plots for the three experimental instances *test*, *large*, and *human*, we can easily check our initial expectations: speedup factors, as well as strong-scaling efficiencies, are very low with respect to the ideal benchmark, obtaining speedup factors below 10 times and average efficiencies around 20%-30%.

As we observed in the previous section, adding more threads leads to better execution times but following a decreasing rate pattern: extra threads are not significantly improving the performance of the algorithm after a certain threshold, leading to poor strong-scalability of our code. The same pattern can be seen across the three experimental instances - see Figures 9, 10, and 11 - where the speedup factors are pretty static (flat) and the efficiency is clearly decreasing with the number of threads.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 9: Test instance speedup factors & strong scaling efficiency (w.r.t. 1 thread) for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node.

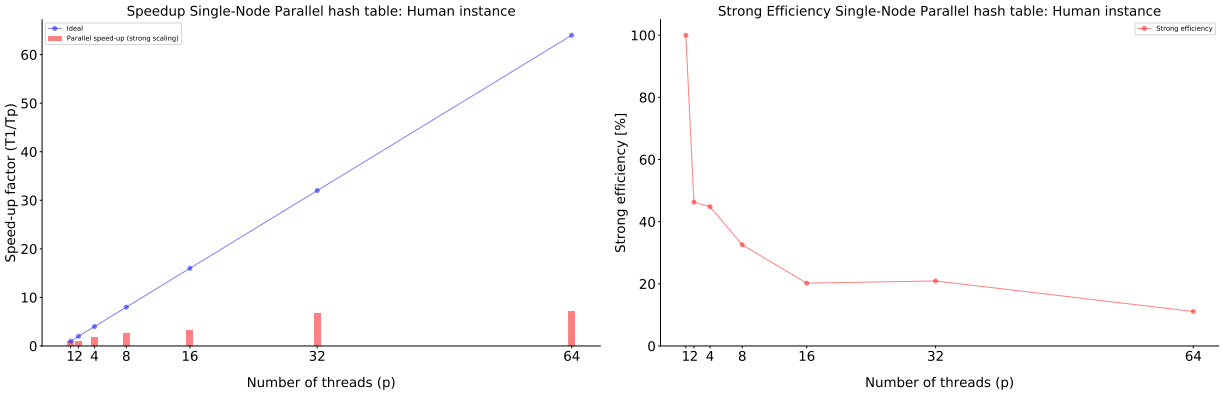


(a) Speedup factors (b) Strong-scaling efficiency

Figure 10: Large instance speedup factors & strong scaling efficiency (w.r.t. 1 thread) for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node.

Based on these results, we can see that the best strong-scaling efficiency values are attained around 2-8 threads with values between 30% to 50% for our single node experiments. The main reason for

such a low strong-scalability has been already discussed in previous sections: due to the fact that the *insert*() operation takes a lot of message overhead as well as the synchronization that must be made to assure the correctness of the code, scalability of our implementation is significantly affected, leading to its actual poor performance, as expected.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 11: Human instance speedup factors & strong scaling efficiency (w.r.t. 1 thread) for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node.

From the results in Table 4, we can see that average strong-scaling efficiencies, as well as average speedup factors, are pretty similar across the three experimental instances leading us to conclude that our code presents a consistent scalability performance, not significantly dependent of the size of the instance to be executed.

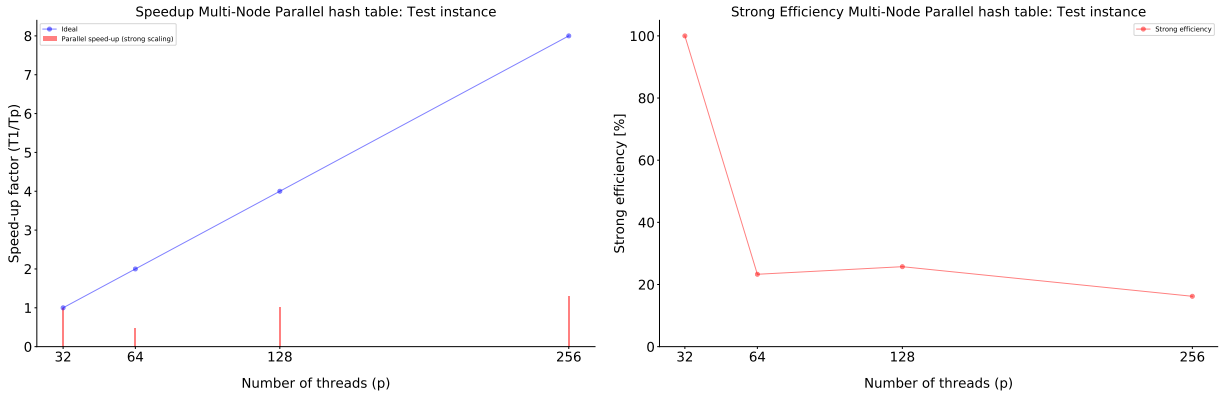
Table 4: Average speedup factors and strong-scaling efficiency for experimental instances running on a single-node

Performance Metric	**Human**	**Large**	**Test**
AVG Strong-scaling Eff. [%]	39.408	39.787	39.233
AVG Speedup Factor	3.335	3.465	3.971

Multi-Node

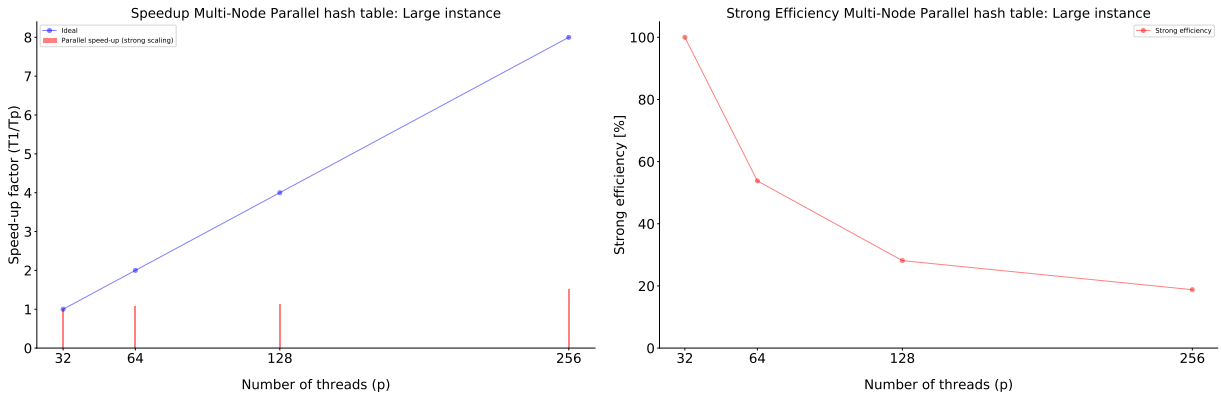
Following the results from Figures 12, 13, and 14 we can see that the pattern is identical to the one observed in the single-node experiments: (1) Speedup factors (calculated with respect to the one node and 32 threads run) tend to be very flat when including more than 32 threads and thus, (2) strong-scaling efficiencies are significantly decreasing when adding more and more nodes/threads. Again, the *large* instance is the one with the (slightly) best performance in terms of average strong-scaling efficiency, reaching a value around 50%, a 9% better than the same metric for the *test* instance and around 4% better than the results obtained for the *human* chromosome data.

As with the single node experiments, speedup factors are pretty similar across the three instances when comparing the times with respect to the one node and 32 threads run ($-N$ 1 $-p$ 32). This pattern is expected based on the results with the single node experiments when using 32 and 64 threads. From Table 5 we can see that there is no significant speedup improvement when adding more nodes/threads, leading to a very poor strong-scalability performance of our code, although decreasing the total execution time of the algorithm.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 12: Test instance speedup factors & strong scaling efficiency for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

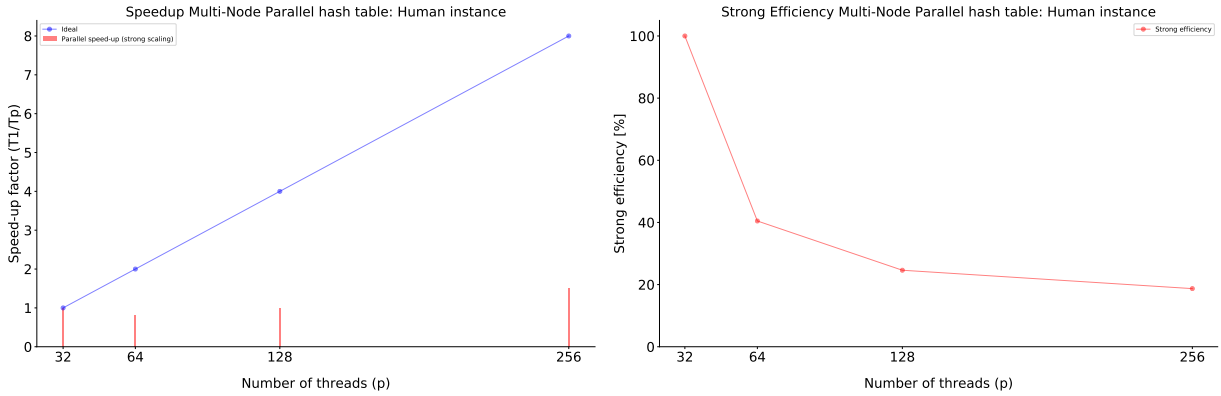


(a) Speedup factors (b) Strong-scaling efficiency

Figure 13: Large instance speedup factors & strong scaling efficiency for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

Therefore, we are able to conclude that although our current parallel hash table implementation using UPC++ is able to reduce the execution times for the genome problem by using multiple threads on a single node or multiple nodes and threads inside a high-performance computer, we were not able to obtain good performance in terms of the strong scalability/efficiency of the code mainly due to the following reasons:

1. Lack of experience with UPC++: a better distributed hash table structure could be made by taking advantage of the strong points of the UPC++ extension. As a first approach to UPC++, it has been a challenge to the team's members to obtain better performance.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 14: Human instance speedup factors & strong scaling efficiency for different number of nodes & threads $p \in \{32, 64, 128, 256\}$ running on multiple nodes.

Table 5: Average speedup factors and strong-scaling efficiency with respect to 32 threads results in a single-node for experimental instances running on multiple-nodes.

Performance Metric	**Human**	**Large**	**Test**
AVG Strong-scaling Eff. [%]	45.947	50.183	41.305
AVG Speedup Factor	0.947	1.176	1.072

2. Communication is costly: overhead and synchronization steps make our code not completely efficient when handling the data and performing the *insert* operations.
3. This is the first class project dealing with a non-embarrassingly parallel code, leading to a more complicated analysis and parallelization of the original code.

4.3 UPC++: comparison with MPI and OpenMP implementations

In this section, we include a discussion of how using UPC++ led us to our final design choices as well as how we would have implemented a similar algorithm using previously applied parallel frameworks such as MPI and OpenMP.

4.3.1 UPC++

UPC++ limits us to performing RPC's on other nodes and carefully maintaining our memory in the global segment. We chose to avoid the former because we felt that an implementation could be done without RPC, synchronizing instead with atomic operations. Rather than explicitly sending memory buffers like we do in MPI, in UPC++, that is done through writing to global memory. Whereas in MPI we would do an explicit 2-way message pass to request memory (with perhaps background threads), in UPC++, we are allowed a simple one-way remote put and remote get that exploits the runtime to allow for synchronous retrieval of memory in the global segment, even if it physically exists on another node.

4.3.2 MPI

Based on our previous experience with MPI in the previous project (particle simulator assignment), we notice that the main structure of the code should not be very different from the one used with UPC++. The graph implementation and initialization logic will likely remain the same, with the only exception that specific calls to $MPI\_Rput()$ - puts data into a memory window on a remote process and return a request handle for the operation - and $MPI\_Rget()$ - gets data from a memory window - for a one-sided communication approach will be pertinent.

Regarding the graph traversal implementation, the idea would be the following:

1. Each processor will send $MPI\_Rget()$ requests to the relevant running processors in order to get access to the k-mer objects, corresponding to one for each contig object that it is being processed on the current processor.
2. An $MPI\_Waitany()$ - waits for any specified MPI Request to complete - call will be performed inside a loop in order to wait for all the requests to be sent before proceeding to the next iteration (synchronization step).
3. We repeat step 1 until no more requests are performed. If no more requests, the breaking condition is achieved, exit.

Thus, the implementation would not be particularly challenging since the main structure of the code would be similar to the one developed using UPC++. In addition, our previous experience with MPI (project 2), as well as our experimentation with RMA one-sided communication schemes, would help us to obtain an effective and efficient code implementation.

4.3.3 OpenMP

In this case, we have a shared-memory approach instead of a distributed memory. We would similarly attempt to partition the file read across threads. We would probably carefully initialize the hash table, using either atomics like we did here, or using explicit locking to avoid race conditions on writes to the hash table. The main work loop would simply be a work-sharing construct across the start nodes (that could be stored in a global vector).

5 Extra credit questions

5.1 F markings

If we did not have the base F marking the beginning and end of contigs, then given starting at random contigs, we could just find forward and backward extensions until we hit a hash table miss. This hash table miss is equivalent to marking the beginning and end of a contig (this is under the assumption that every k-mer is still unique).

5.2 Load balancing

Dynamic load balancing on start k-mers is not a difficult task: we essentially set up a global work queue of start k-mers and a global counter. The work queue will consist of a global array (perhaps located on a master node) of all the start nodes. A simple counter can tell you what the "next" start node to conquer is. We modify the global counter with an $atomic\_fetch\_add()$.

5.3 Parallelizability of Graph Traversal

Suppose that the underlying *de Bruijn* graph consists of p connected components (p is the number of processors) whose length (number of vertices) follows a power law distribution. How well does your parallel code perform on this input? Can you do better (algorithmically)? What if it consists of a single connected component?

5.3.1 Power Law

Due to the structure of the problem, this case is a special case of the single-component problem discussed in the following section since we can make the expected depth to be as close as the value of n as we want, obtaining an asymptotic complexity for the graph traversal phase of $O(n)$ (with n the length of the graph).

Therefore, we can use the strategy proposed for the single component case in the following section in order to obtain better results than with our optimized parallel code for the simple instances provided with the project's statement.

5.3.2 Single component

In this case, since we have that the underlying *de Bruijn* graph consists of only a single connected component, we would not be able to take full advantage of our parallel implementation: the serial version will be more suitable to solve this problem unless the full graph does not fit into memory, where the parallel approach will have an advantage, besides the fact that the parallel structure is not being exploited.

Based on our background and lecture notes, it is possible to use the following approach for effectively handling this situation:

1. Each processor/rank/thread can start processing the data from a random assigned k-mer. Once initialized, each process can start processing neighbors in both directions.
2. In order to decrease the overlap between processors and merge all the work performed by each of them, each k-mer should only be assigned to an individual process (its "owner"). Therefore, each k-mer processed by a certain process is marked as a non-free k-mer preventing other processes to go further in that direction.
3. Then, when all processes are done - no feasible direction exists - the ending points of each process' contig object is identified and a sorting & merging operations are performed - in one of the processes - in order to obtain the final sequence.

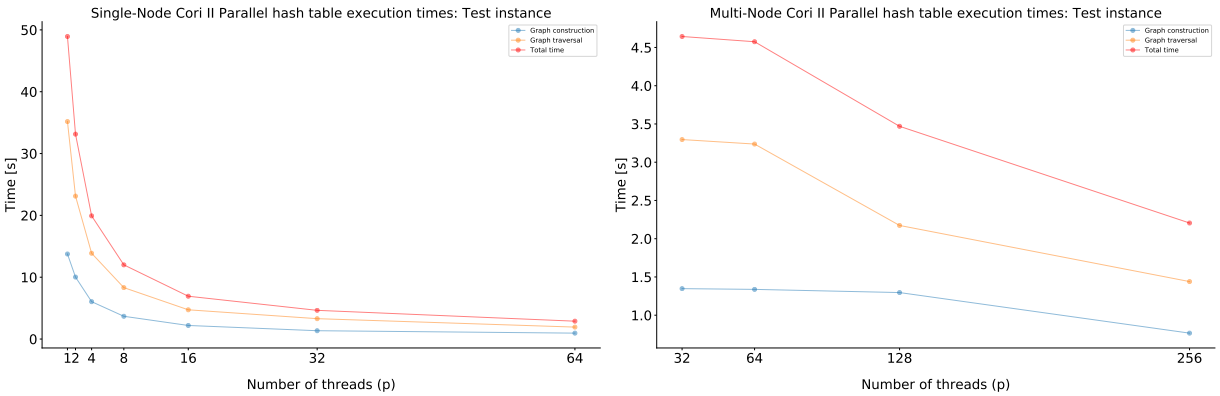
5.4 Experiment - KNL

Recompile your code for Cori Phase II, which has Intel Xeon Phi KNL processors, and run your scaling experiments again. How does the performance compare to that on Cori Phase I, which has Intel Haswell processors? How might your results be influenced by the CPUs or the networking hardware?

Test Instance

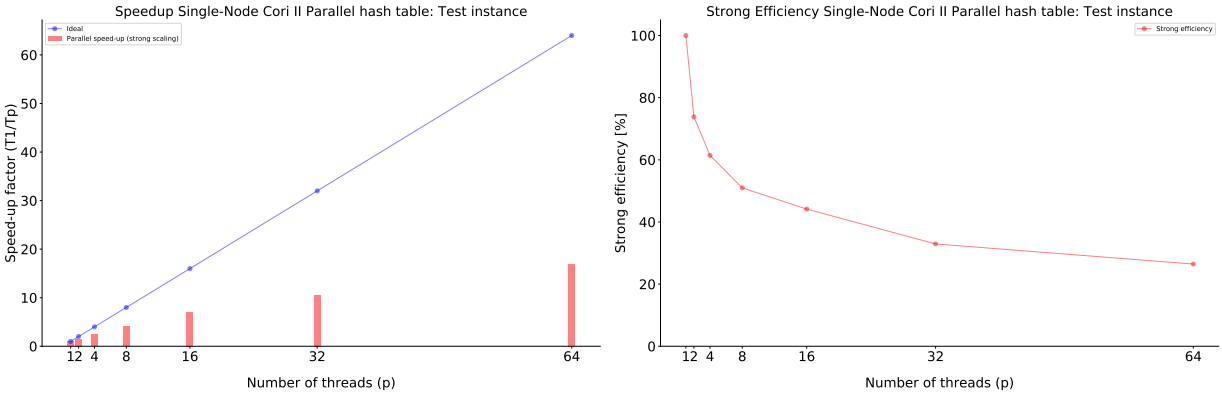
Based on the results obtained when running our parallel code re-compiled for Cori Phase II hardware, we are able to see the main differences in performance in Figures 15, 16, and 17. Comparing the execution times obtained in Cori Phase I, we can clearly see that the performance of our code in Cori Phase II is worse in both the single and multi-node approaches.

When running our parallel implementation in a single-node fashion with a unique thread, we see a difference of 38 seconds (12 vs 50 approximately) in the total execution time. Similar differences' magnitudes are kept among all different number of threads. The smallest performance gap is achieved when comparing both versions using 64 threads: Cori Phase I version reaches a total execution time around 2 seconds while the compiled code for Cori Phase II is executed in 3 seconds. Although the difference in performance, it is interesting to realize the fact that we do not see any performance loss when adding more threads in Cori II - in contrast to what we observed in Cori I - obtaining a very smooth decreasing curve for all the components of the total execution time.



(a) Execution times Single-Node (b) Execution times Multi-Node

Figure 15: Test instance execution times for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node in Cori II (a) and multi-node setting (b).



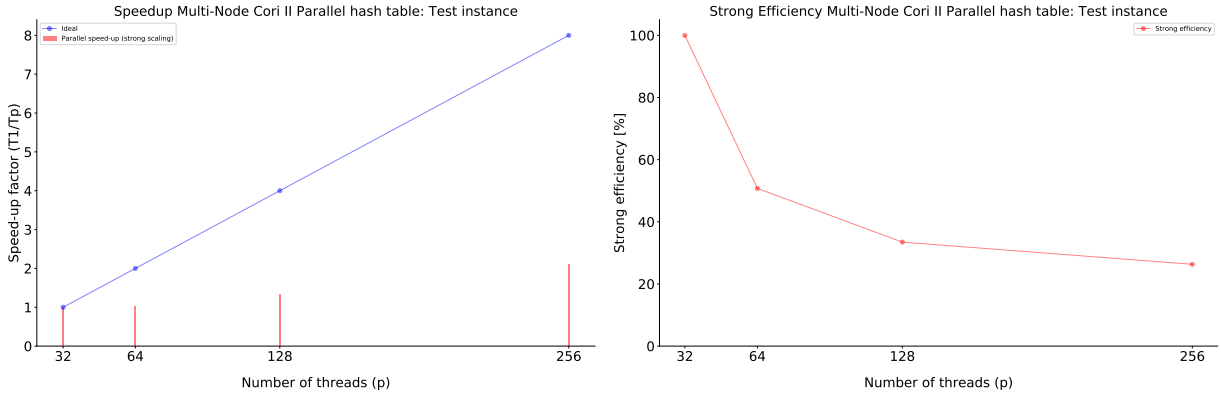
(a) Speedup factors (b) Strong-scaling efficiency

Figure 16: Test instance speedup factors & strong scaling efficiency (w.r.t. 1 thread) for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node in Cori II.

Following the previous discussion, we analyze the speedup and strong-scaling efficiency obtained in Cori Phase II. From the results, we can see that in this case, we are able to obtain better performance in both

indicators, mainly due to the smooth pattern observed in the execution times: an average speedup factor of 6.22 vs the original 3.97 as well as an average strong-scaling efficiency equal to 55.67% vs 39.23% for the single-node implementation and 1.36 vs 0.94 and 52.63 vs 41.3 for the multi-node version, respectively.

Therefore, we have that the performance of the *test* instance on Cori Phase II has the following characteristics: (1) Worse average execution times than the ones obtained with the Cori Phase I hardware/network and (2) significant better average strong-scaling efficiency as well as speedup factors due to the smooth behavior of the execution times when adding more nodes/threads.



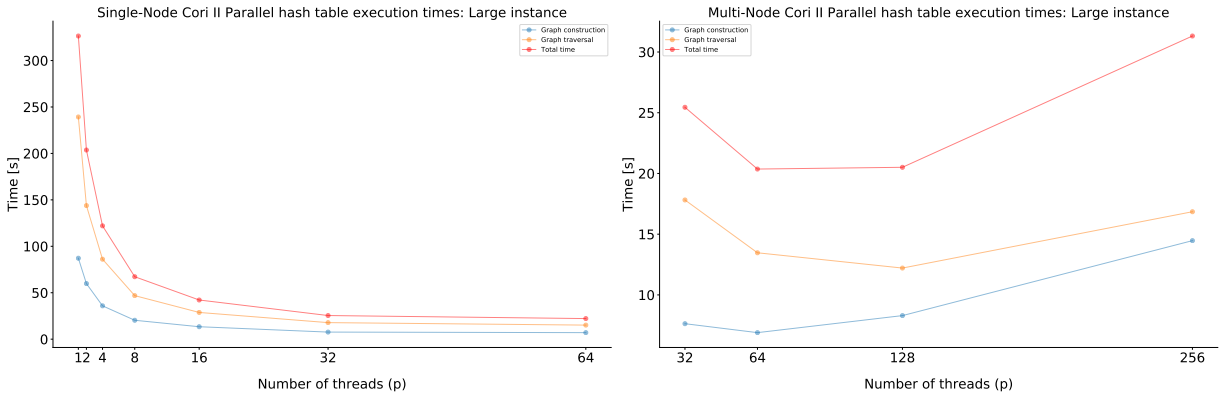
(a) Speedup factors (b) Strong-scaling efficiency

Figure 17: Test instance speedup factors & strong scaling efficiency (w.r.t. 32 threads) for different number of threads $p \in \{32, 64, 128, 256\}$ running on a multi-node setting in Cori II.

Large instance

Similarly to the previous instance, we can see (Figure 18) that in this case, the performance obtained in Cori Phase II when executing the *large* instance is significantly worse - in total execution time - than our original results in Cori Phase I: more than 300 seconds are needed in Cori II when using one thread while only around 75 were needed in Cori I. As before, the performance gap is a constant, reaching its minimum value when using 64 threads in parallel (22 seconds vs 10 seconds approximately).

Again, performance is improved with the addition of extra threads. However, we can see in Figure 18 (b) how adding more nodes/threads is leading to a significant performance loss of our algorithm: 256 threads leads to worse results than 32, 64, and 128. This is mainly due to communication overhead between nodes, a pattern that was not present in Cori I.



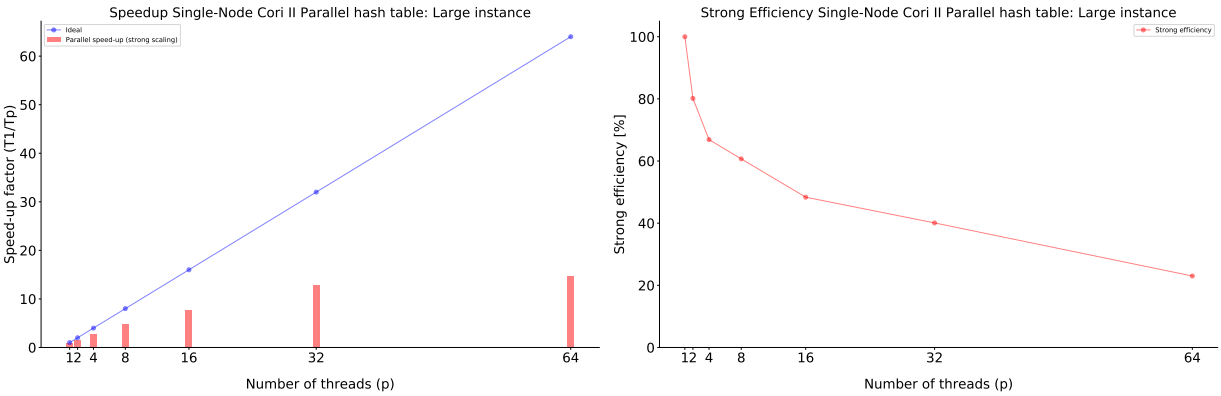
(a) Execution times Single-Node (b) Execution times Multi-Node

Figure 18: Large instance execution times for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node in Cori II (a) and multi-node setting (b).

Looking at the results of Figures 19 and 20, we analyze the speedup and strong-scaling efficiency obtained in Cori Phase II. From the results, we can see that in this case, we are able to obtain better performance in both indicators with the single-node implementation but worse scalability results when running the algorithm in

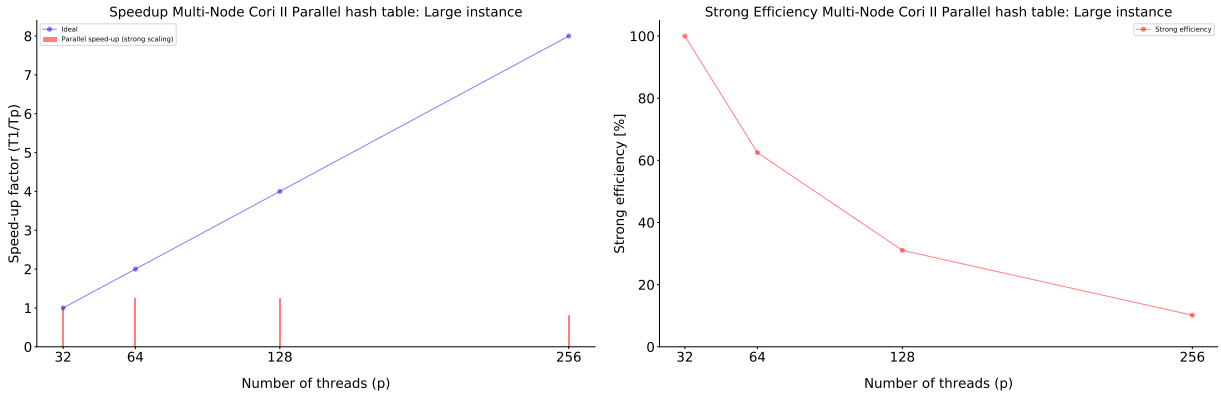
a multiple-node fashion: an average speedup factor of 6.48 vs the original 3.46 as well as an average strong-scaling efficiency equal to 59.88% vs 39.78% for the single-node implementation and 1.07 vs 1.17 and 50.91 vs 50.18 for the multi-node version, respectively.

Therefore, we have that the performance of the *large* instance on Cori Phase II has the following characteristics: (1) Worse average execution times than the ones obtained with the Cori Phase I hardware/network and (2) better average strong-scaling efficiency as well as speedup factors on a single-node experiments and worse results for these metrics when running the code in multiple-nodes.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 19: Large instance speedup factors & strong scaling efficiency (w.r.t. 1 thread) for different number of threads $p \in \{1, 2, 4, 8, 16, 32, 64\}$ running on a single node in Cori II.



(a) Speedup factors (b) Strong-scaling efficiency

Figure 20: Large instance speedup factors & strong scaling efficiency (w.r.t. 32 threads) for different number of threads $p \in \{32, 64, 128, 256\}$ running on a multi-node setting in Cori II.

Human instance

Due to time constraints, we were not able to obtain all the results in Cori Phase II for the *human* chromosome instance and thus, results are not included. However, based on the results obtained with the *test* and *large* instances, we would expect a similar performance pattern: worse execution times and better speedup factors and strong-scaling efficiency.

6 Conclusions & Future Work

1. A first approach with the unified parallel C++ (UPC++) extension is done, complementing our learning process about different parallel programming frameworks. The main advantages of using the UPC++ extensions lie on its simple and straightforward syntax as well as the powerful management of resources under the hood. However, we would be able to obtain better performance in future implementations thanks to this first experience, taking advantage of the extension strongest characteristics.
2. When developing parallel algorithms it is important to take into account the potential communication overhead that the code will experience. Depending on the structure of the problem, a serial implementation or a shared-memory approach instead of a distributed-memory implementation can be more suitable for the problem under study.
3. A non embarrassingly parallel project has been successfully implemented for the construction and traversal of the *de Bruijn* graph in the context of DNA assembly. This characteristic makes this problem a challenging one after mainly working with easily decomposable (parallelizable) problems up to this date.
4. The effects and impacts on the algorithm performance of the communication overhead can be clearly seen in the obtained results for both the single and multi-node implementations. Based on the obtained results, the parallel implementations of our algorithm are not able to beat the serial Vanilla code when using the same number off threads (1). Furthermore, we needed more than 4 threads to obtain similar results to the Naive implementation being able to clearly identify the impact of the communication overhead experienced by the code.
5. Different network structures and hardware specifications should be taken into account when developing parallel code. Depending on the characteristics of the running environment, significant differences in the overall performance of the code (execution time and/or scaling efficiency) can be seen, as with Cori Phase I and Phase II.
6. Further work must be done in the design of the distributed hash table in order to exploit the performance of the parallel code. The current implementation is not able to get more than 50% of strong-scaling efficiency (best results in a multi-node approach) and we are not able to exploit the available hardware by adding more threads/nodes for the processing. Therefore, processing larger instances will become a challenge after some threshold of n due to the limitations of our current code.
7. An MPI implementation using the structure described in section 4.3.2. would be useful for gaining strong-scaling performance thanks to our previous experience with this parallel framework, allowing us to have more control of the threads interaction and overall algorithm performance.

7 References

1. Lecture notes and presentations from UC Berkeley, CS267, Spring 2018.
2. Bachan, J., Bonachea, D., Hargrove, P. H., Hofmeyr, S., Jacquelin, M., Kamil, A., ... & Baden, S. B. (2017, November). The UPC++ PGAS library for Exascale Computing. In Proceedings of the Second Annual PGAS Applications Workshop (p. 7). ACM.
3. Zerbino, D. R., & Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research, 18(5), 821-829.