HW 3. Parallelizing Genome Assembly

Nuochen Lyu
Xiaoyun Zhao

Overview

In this assignment we implemented UPC++ to parallelize the genome assembly pipeline. The DNA strands are preprocessed with shotgun sequencing, and the output is a set of unique sequence fragments with length k, which we called k-mers. The k-mers can be presented with de Bruijn graph with k-1 overlapping bases. By traversing the graph we can construct longer DNA contigs, and finally reconstruct the DNA strand. We implemented the parallel construction and traversal of the de Bruijn graph of k-mers.

Implementation

1. Hash Table

We implemented the HashMap data structure:

```
struct HashMap {
  std::vector<upcxx::global_ptr<kmer_pair>> data;
  std::vector<upcxx::global_ptr<int>> used;

  size_t my_size;
  size_t global_size;
  size_t n_proc;

  size_t size() const noexcept;

  HashMap(size_t size);

  // Most important functions: insert and retrieve
  // k-mers from the hash table.
  bool insert(const kmer_pair &kmer, upcxx::atomic_domain<int>& ad);
  bool find(const pkmer_t &key_kmer, kmer_pair &val_kmer);
```

```
  // Helper functions

  // Write and read to a logical data slot in the table.
  void write_slot(uint64_t slot, const kmer_pair &kmer);
  kmer_pair read_slot(uint64_t slot);

  // Request a slot or check if it's already used.
  bool request_slot(uint64_t slot, upcxx::atomic_domain<int>& ad);
  bool slot_used(uint64_t slot);

  // Helper functions to compute index in global memory
  int index(uint64_t slot);
  int offset(uint64_t slot);
};
```

To use UPC++ to parallelize hash table read and write, we create two vectors of *upcxx::global_ptr data* and *used*, where *data* is points to kmer_pair, and *used* to flag whether the slot already stored values in the has table. The usage will be described in later sections. At initialization, each process fills in data and returns the pointer to the global memory based on its rank number, and then broadcast the pointer to other processes.

The most important functions in the HashMap are *insert()* and *find()*. The *insert()* function adds kmers to the HashMap. This is done through *upcxx::atomic_domain*, which is then passed to *request_slot()* function and enables multiple threads to share the global resources without conflicts. When requesting an available slot, we invoke an asynchronous remote procedure call (RPC) to remote processes, and use *wait()* to obtain the value from the returned future object.

The *find()* function retrieves the kmer_pair's value given the key. Our implementation adopts the open addressing with linear probing when there is a collision after first hashing. The same process applies to *find()* too, it keeps searching linearly until it finds the kmer with the correct key in the HashMap.

upcxx::rget() and *upcxx::rput()* are used to read from and write to HashMap's slots. This one-sided communication enables parallelism in a distributed array with shared global references.

2. DNA kmer assemble

The main body of the DNA kmer assembly code does not have much modification. It firstly finds all kmers with 'F'. And at the same time insert all kmers into the hash table. In the next step the program traverses the starting list and assembly the kmers after the starting node.

Since the insert in an atomic operation. In UDP++3.0 version, we need to the atomic domain to manage the atomic operation. While in the previous version the atomic feature is directly accessed. An atomic domain includes the fetch operation and passes the operation handler to the insert method for the hashmap. The read and find operation does not require a lock. In the end the program we need to free the atomic domain by atomic domain destroy function.

3. Impedance

Serialization: We try to create a class that wraps the kmer struct for better implementation. However, this change creates a problem in rget and rput. As the documentation says, in ver3.0 serialization is realized as a new feature. Rget and rput have to accept TriviallySerializable template class. To make a class trivial serialization. We need our user-defined class to be a friend class of access and make a serialize function. The procedure is painful. We need to load the Boost module for the serialization function. Cori does offer Boost module. But the upcxx version is somehow not compatible with the version in Boost. I stop the investigation and use the original kmer class instead.

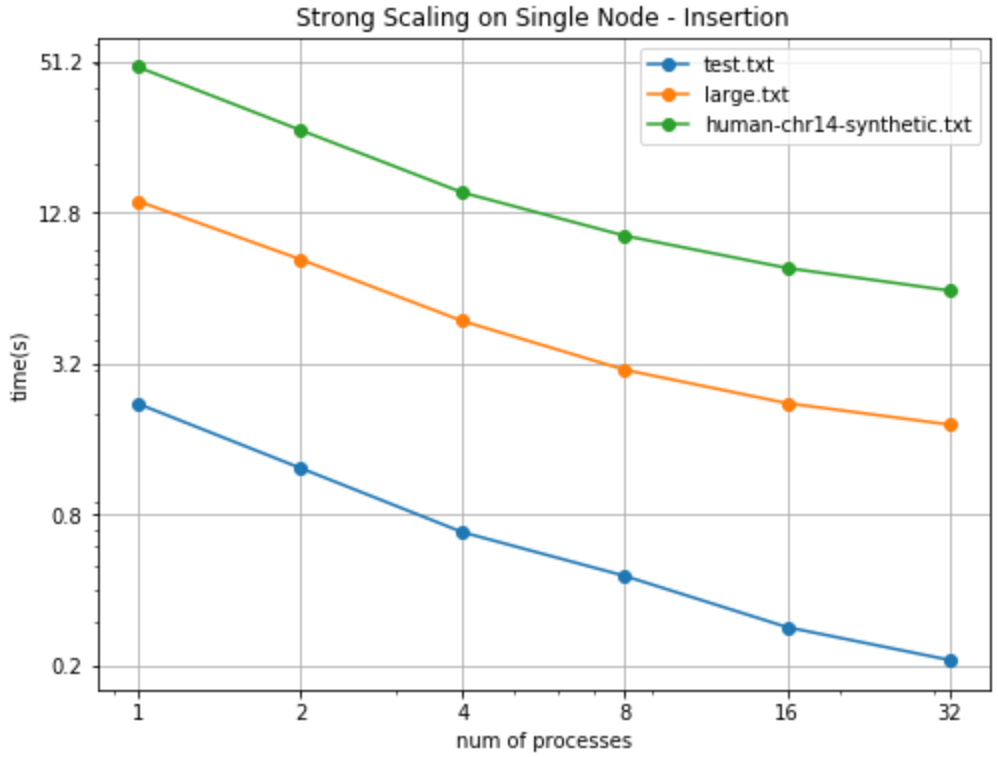
4. Other trials

We try to add O3 flag for the compilation. It does not speed much. It is mainly because of there not much vector operations. The main computation happens in accessing and reading the hashmap.

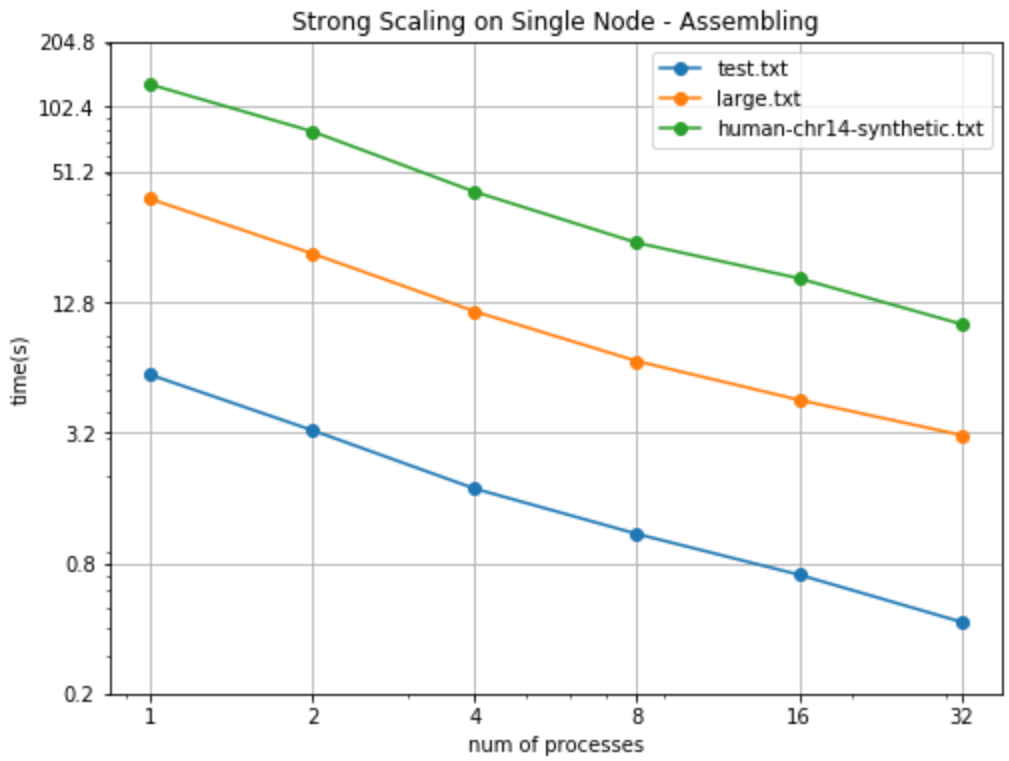
We see that GNU support openmp with the -fopenmp flag. But we did not figure out the openmp version. For the insertion, the atomic operation for UPC does not lock the thread operation in openmp. Also the C++ for syntax is a little different for the openmp parallel for. We try to make locks for the hashmap but we do not have enough time to complete.

Scaling Experiments

We first run the strong scaling experiment for **insertion** on a single node for all three datasets (test.txt, large.txt, human-chr14-synthetic.txt):



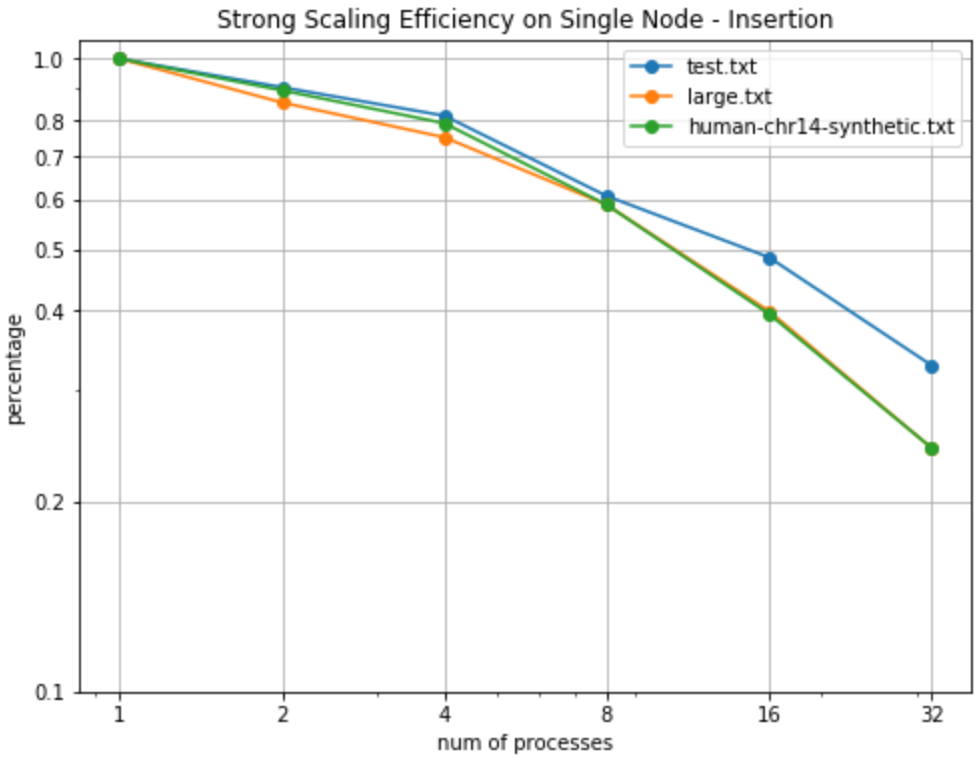
And here's the result for **assembling** on single node:

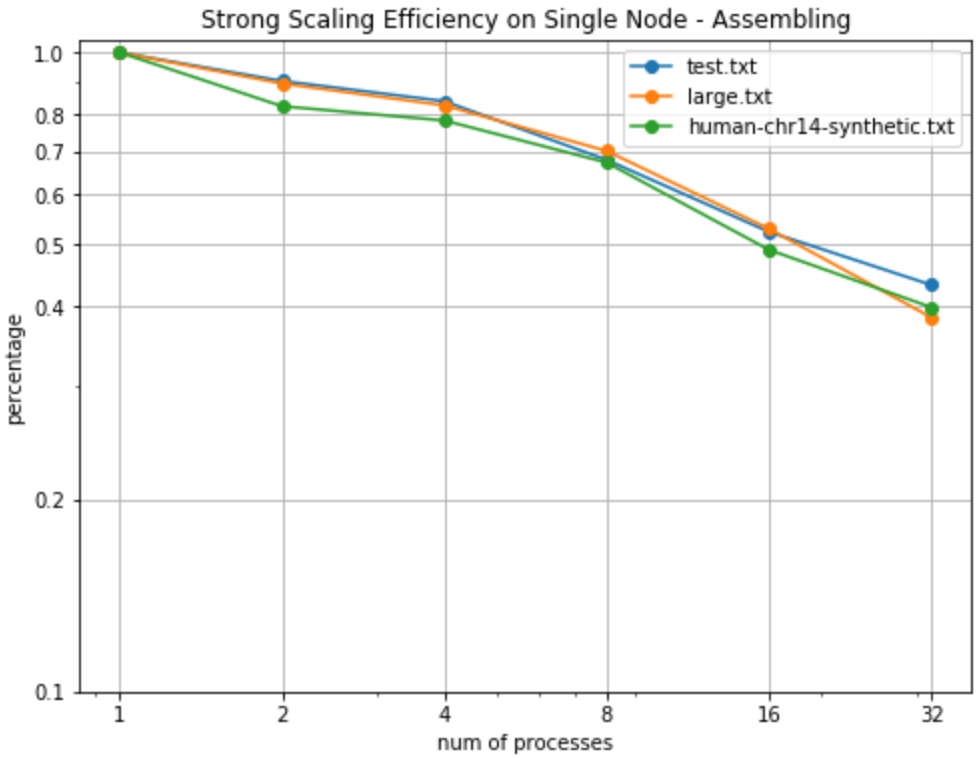


A clear linear pattern can be seen in the above plots, with the number of processes increases, both insertion time and assemble time decrease accordingly. And the parallel code works well for all three datasets.

We also plot for strong scaling efficiency, which is calculated as:
Strong scaling efficiency = t1 / (N * tN) * 100%

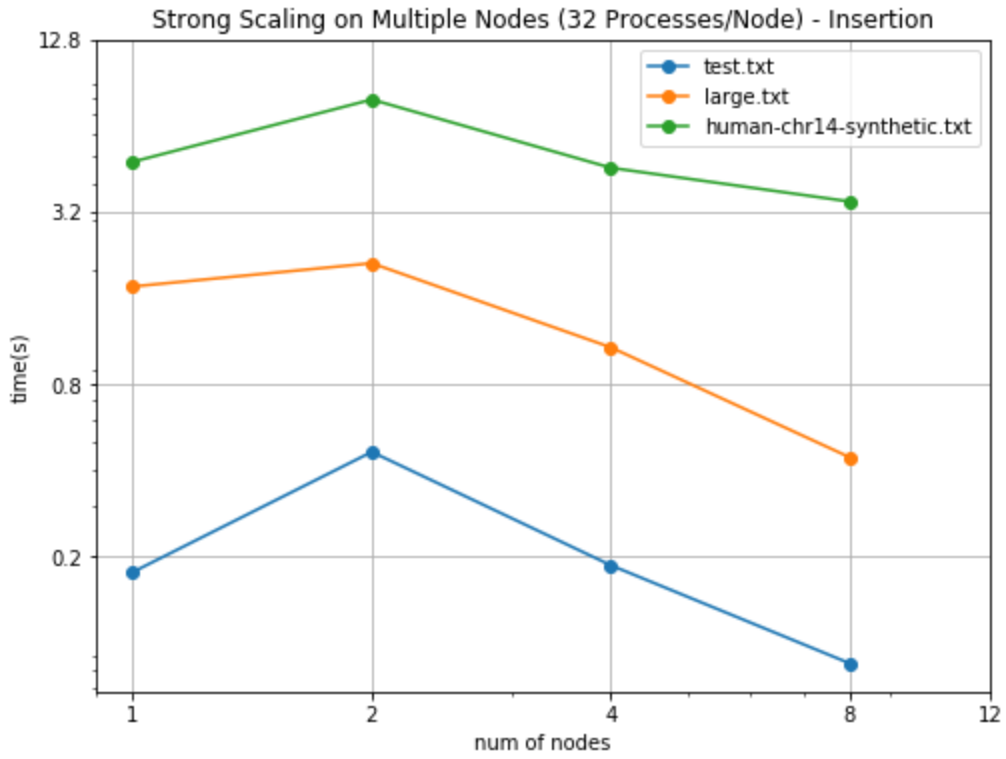
And here's the result:

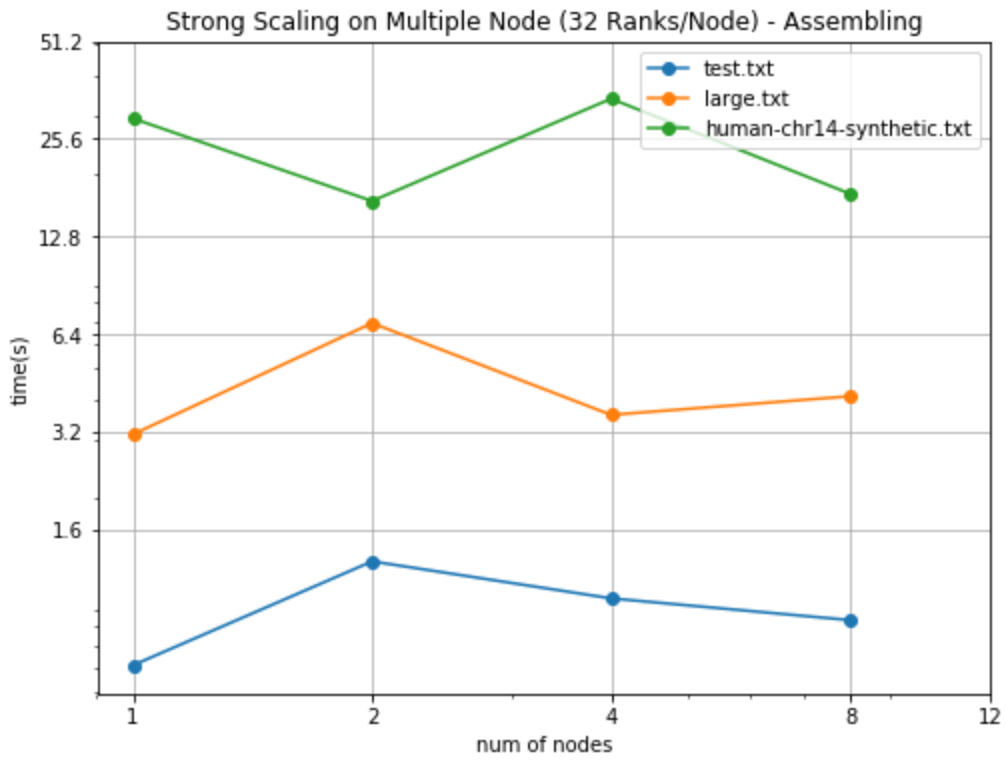

Strong Scaling Efficiency on Single Node - Insertion
test.txt
large.txt
human-chr14-synthetic.txt
percentage
num of processes
1.0
0.8
0.7
0.6
0.5
0.4
0.2
0.1
1
2
4
8
16
32


Strong Scaling Efficiency on Single Node - Assembling
test.txt
large.txt
human-chr14-synthetic.txt
percentage
num of processes
1.0
0.8
0.7
0.6
0.5
0.4
0.2
0.1
1
2
4
8
16
32

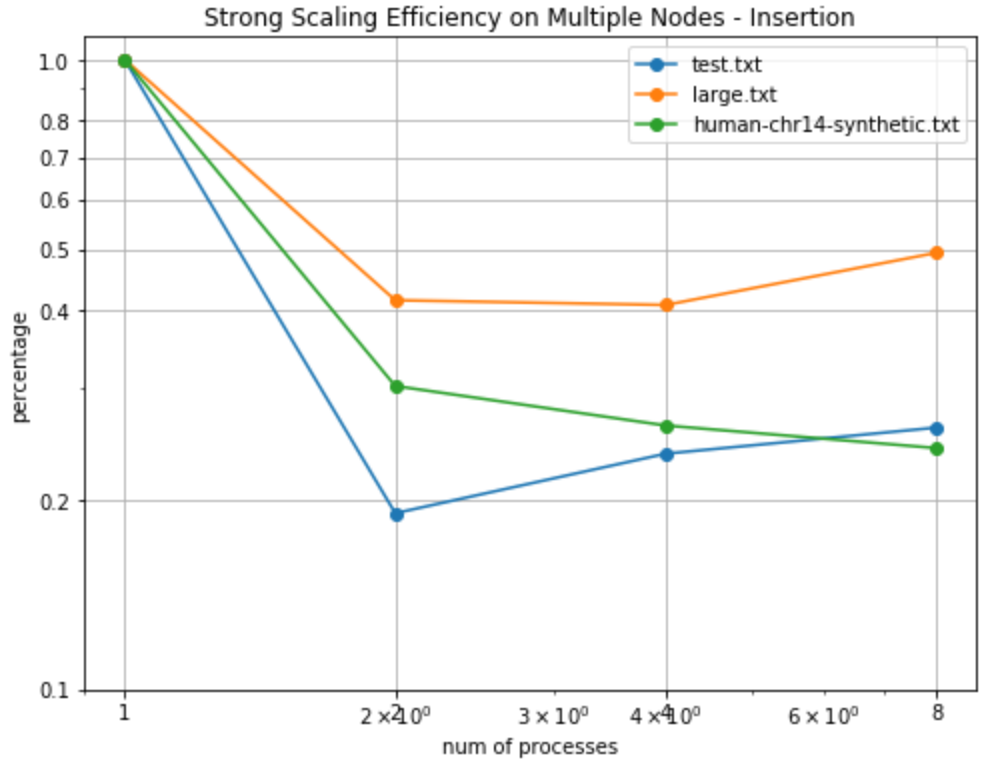
As shown in the plot, efficiency starts to drop as number of processes increase, which could be caused by large communication overhead in parallelization. All three datasets exhibit similar behavior, and for insertion the overhead seems to have a bigger impact when the number of threads is large than 8.

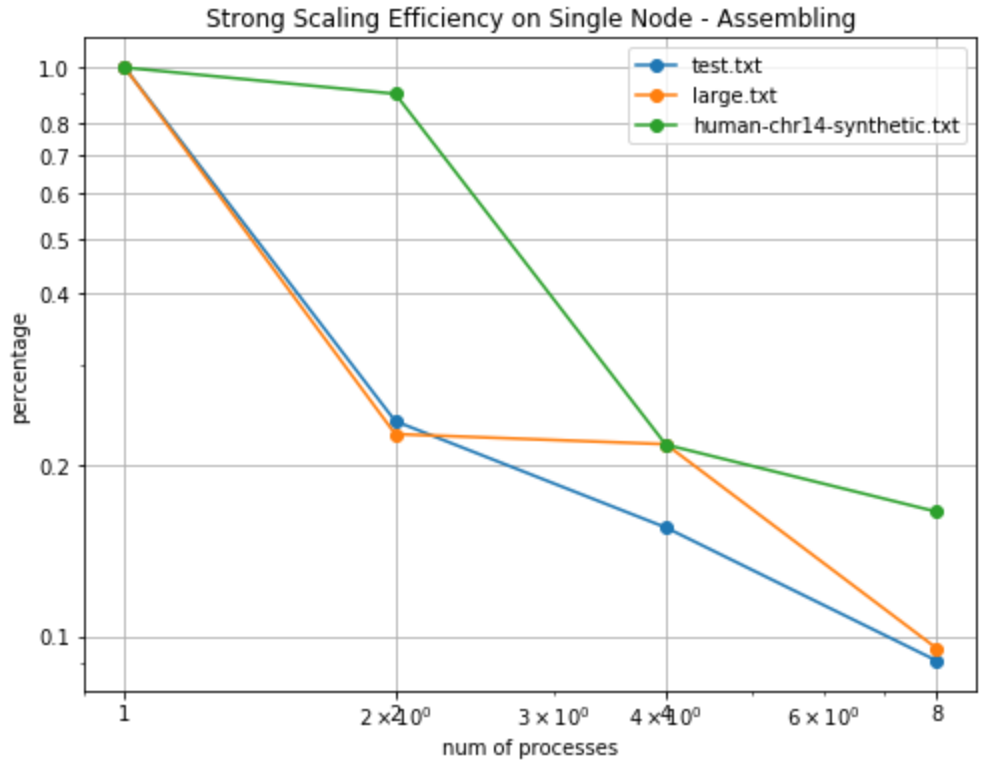
Results for multi-nodes experiments are shown below:





And the strong scaling efficiency looks as follow:





Discussion

1. Implementation using MPI

The distributed memory structure is actually similar to what we have in UPC++, however, there isn't a shared global memory for separate ranks, therefore the information about the location of different key value pairs need to be communicated among ranks. One way to do this is to have a rank dedicated to assigning location information, whenever there's an update in the hashtable, the dedicated rank needs to broadcast the update to all ranks, and the rank with the actual data responds by making the update.

2. Implementation using OpenMP

OpenMP should have a more straightforward way of update the hashtable. Since all processes share the same memory, there's no need to communicate about where the data is located. A lock should also be adopted to avoid racing conditions.

Extra Credit

Short Answer - Contig F Extensions

What if we did not have the base F marking the beginning and end of contigs? Suppose instead we had a random base as an extension whose corresponding k-mer did not exist in the dataset. How would this affect the contig generation algorithm? How would it affect your implementation?

If we did not specify “F” to be the base marker. To identify a starting/ending kmer, we cannot just check whether it is an “F”. It can be any words even alphabet “A” “T” “C” “G” (if the kmer starts/ends with ATCG does not exist in the database). Therefore we need to check the hashmap and see whether it exists in the dataset. This would slow down the speed since we need to find every beginning/end of contigs by traversing the whole file. The traversing could be down by UPC in parallel but it is still much slower.

Short Answer - Load Balancing

Suppose start k-mers are unevenly distributed throughout the dataset, and this creates a load balancing problem. How might you deal with this?

If the k-mers are unevenly distributed throughout the dataset. The segment of contig would be very different in length. Thus some part of the hashmap would be filled by a very long contig. Where some part of the hashmap would be filled by lots of tiny contigs. This discrepancy creates the load-balance problem when we assembly the DNA in the second step. Since in our implementation, each part of the hashmap is evenly taking charged by one processor. There would a situation where the contig is very short and it falls within one part of the hashmap. So when we are constructing that short contig. Only one part of the hashmap is visited frequently. That means only one UPC process is running while the others are just idle. If the contig is very different, this unbalance problem would always appear. It wastes the power of processors and slows down the program.

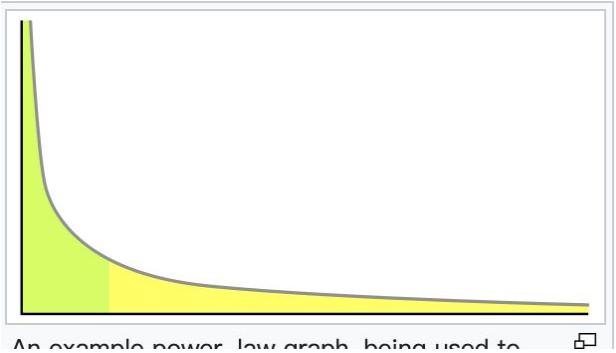
To solve this problem, we need a more even contig distribution. We could preprocess the file and cut the long contig smaller or makes the smaller contig longer by merging them together with some dummy node. We could also do padding for the short contigs. Moreover, we could improve the hash function and let it assign kmers randomly. Or we could increase the size of the hashmap to force the contig distributed evenly.

Short Answer - 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

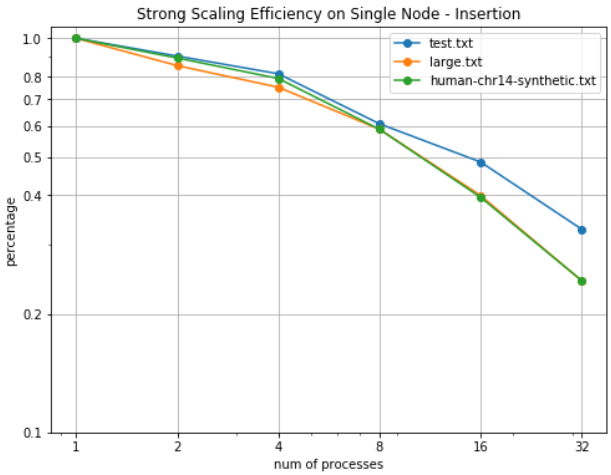
code perform on this input? Can you do better (algorithmically)? What if it consists of a single connected component?

When the underlying de Bruijn graph contains p connected components, as p increases, the length (number of vertices) will decrease, and there will be idle resources when p increases to a certain number.



An example power law graph being used to

The graph above shows the power law distribution, and based on the plot for strong scaling efficiency (see below), we know that the performance starts to drop at a certain number of processes partially due to the communication overhead among processes.



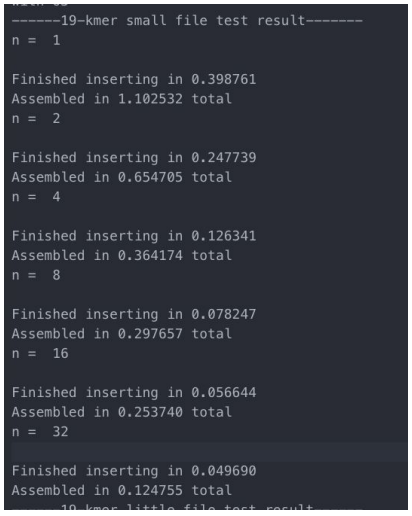
We need to calculate the critical point at where the performance for parallelization is at peak without compromising efficiency. For example, from

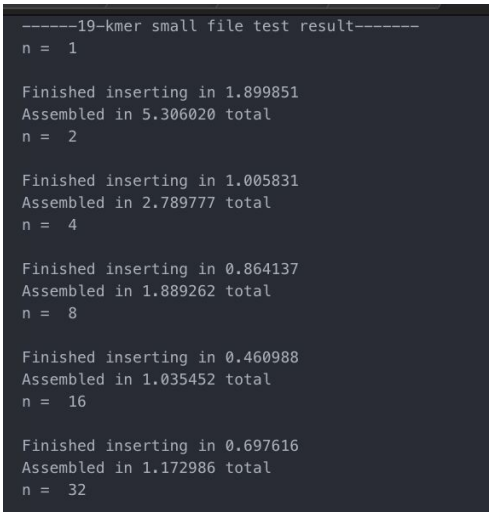
the above efficiency graph we can roughly tell when the number of processes is larger than 4, the efficiency starts to drop.
For a single connected component, its resource will be pulled fully on executing a task, thus can reach the highest efficiency (100%), but the overall runtime cannot beat parallelization due to the limited computation resource.

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?

Haswell KNL





We use a simple module swap craype-haswell craype-mic-knl

But does not change any code. The code in Haswell environment runs about four times slower in the KNL environment. In KNL, the code still scale but the average speed is much slower.

Experiment - RPC Hash Table

It's possible to implement a very trivial hash table in UPC++ which uses remote procedure calls to trigger an insert() or find() operation in a local std::unordered_map on a remote process. Implement this. How do the results compare to the hash table that you implemented? Discuss why the performance is or is not different. Does the way you're using the network affect your results? Any differences on Cori Phase II, which has Intel Xeon Phi KNL processors?

Since we still use the global pointer in UPC. Thus a hashtable or a local hashtable is very similar. The speed should be the same or in the same scalar. Rget and Rput are still applied. However, the hashtable we implemented using linear probe hash function which is very different from the hash function in c++ unordered map, which use bucket sort. Also, the size of the unordered map is very different from our naive hash table.