CS 267: HOMEWORK #3
PARALLELIZE GRAPH ALGORITHMS FOR DE NOVO GENOME ASSEMBLY

Professor James Demmel

April 3, 2016

Sayna Ebrahimi
Fotis Iliopoulos
University of California, Berkeley

Abstract

In this project we investigated the parallelization of graph algorithms for de Novo Genome Assembly. We have built our parallel algorithms based on our modified version of the sequential program for the same problem and implemented them in UPC. We present, evaluate and discuss results for scaling efficiency.

1 Introduction

Graph traversal is one of the thirteen motifs in computational patterns. De Novo genome assembly is one of the most well-known applications of graph traversal algorithms. DNA sequence assembly is the process of aligning and merging pieces of nucleotides from a longer DNA sequence in order to reconstruct the original DNA strand. In the given serial algorithm for this assignment, aligning and merging the k-mers to form *contigs* is done by creating a hash table to construct the de Bruijn graph and traversing it to reconstruct the original DNA strand.

2 Parallelization

For the purpose of this assignment, we parallelized the given serial algorithm and implemented it in UPC. UPC is a programming model of type PGAS (partitioned global address space) which considers a shared memory address space that is logically partitioned to portions accessed by private threads or processors. We have taken advantage of this property and parallelized the serial algorithm by utilizing similar but distributed data structures to be executed by the threads. Therefore, we have defined a distributed hash table and a distributed list of start nodes.

2.1 Data Structures and Algorithm

The pseudocode of the parallel algorithm we implemented is Algorithm 1 below and is consisted of the following steps:

- Reading the data file by evenly distributing the data among threads.
- Graph construction
  - Adding the k-mers to the hash table by each thread
  - Adding the k-mers to the start list by each thread
- Graph Traversal
  - k-mer look up in the hash table by each thread
- Writing the sequences generated by the threads to separate output files

Algorithm 1 Parallel (UPC) Algorithm

```
1: $N \leftarrow$ Reading the number of k-mers
2: $P \leftarrow$ Number of processors
3: LocalBuffer $\leftarrow$ Read(roughly) $N/P$ k-mers
4: Disthashtable $\leftarrow$ Initialize DistHashTable
5: for each kmer in LocalBuffer do                                   ▷ Graph Construction
6:     localStartNodeList $\leftarrow$ Initialize Local List
7:     AddKmerToHashTable(Disthashtable,k-mer, forwardExt,backwardExt
8:     if backwardExt is F then AddKMerToList(localStartNodeList,(k-mer, forwardExt ) )
9: for each (k-mer,forwardExt) in localStartNodeList do                ▷ Graph Traversal
10:    currentContig $\leftarrow$ CreateNewSequence(k-mer)
11:    currentKmer $\leftarrow$ forwardExt
12:    while currentForwardExtension is not F do
13:        AddBaseToSequence(currentForwardExtension,currentContig)
14:        currentKmer $\leftarrow$ LastKbases(currentContig)
15:        currentForwardExtension $\rightarrow$ LookUp(DistHashTable,currentKmer)
16:    PrintContig(currentContig)
```

Here is each item explained in details:

2.2 Reading data file in UPC-IO

We used the UPC-IO (accessed by <upc_io.h>) [1] which is a parallel I/O API for UPC that uses *collective* I/O functions. In UPC-IO the data access is provided in and out of shared and private buffers; thus, private and shared reads and writes are generally supported. Note that in case a thread needs to access a file independently, it can use the regular C/POSIX I/O functions [2] but we used UPC-IO only.

We have considered a collective operation to have each thread reading a block of data into a private buffer from a particular thread-specific offset. The offset should be set by *upc_ all_fseek* function. File pointer modes are specified by the collective *upc_ all_fopen* function. By calculating the number of characters to be read by each thread we can dynamically allocate the space needed for the private buffer and use the *upc_ all_fread_ local* to assign each thread to locally read its own chunk of data. At the end of this process *upc_ barrier* is called to make sure all the threads arrive before moving to the next part which is using the read data. The files closes collectively using *upc_ all_fclose*.

2.3 Creating hash table and adding k-mers

Input file for DNA sequencing problem is usually very large that the hash table cannot hold every permutation of a k-mer. Therefore, we will have a situation that multiple k-mers land in the same bucket. To avoid collision, we assigned a distributed linked list. We first defined a new data structure for our k-mers which consists of the packed k-mer itself and its left and right extensions. The memory heap, hash table, and indices for our linked list are all allocated dynamically using *upc_ all_ alloc* to return private to shared pointers. We initialize the hash table in parallel by using *upc_forall* loop. To avoid race condition, before we can proceed further, we must ensure all threads reach this point, so a *upc_ barrier* function is called. In order to add a k-mer with its extensions to the hash table, we need an efficient way of data transferring between the various combinations of shared and private space which can be executed by all threads. UPC offers three mechanisms of doing so with the following functions:

- *upc_ memcpy*
- *upc_ memput*
- *upc_ memget*

We used the first two in adding k-mers process and the latter in the k-mer look-up process which will be discussed in the next section. After putting the k-mers from private memory to the shared memory by *upc_ memput* and transferring the extensions within the shared memory, we atomically add the kmers to the hash table on shared memory, referenced via a pointer-to-shared. This is a potential case to raise race condition because multiple threads might want to touch the same shared memory location at the same time. To avoid this, we used *type* **bupc_atomicI64_cswap_strict***(shared void *ptr, type oldval, type newval)* which guarantees the atomic read-modify-write operation of data types of int64 accessed by all threads with pointers to shared memory locations. This function atomically sets the location given by the first argument to the value *new val* only if the current value is equal to *oldval*, but return the prior value regardless of whether the write was performed. Similarly, by the end of this process, we have to ensure that all threads finish their task by calling *upc_ barrier* function before we move to the next section.

2.4 k-mer look up in hash table

The graph traversal phase is the process of looking up the k-mers in the distributed hash table and find their extension to keep linking *contigs* and reconstructing the DNA sequences. The look-up process is done by computing the hash value of the k-mer and finding its corresponding bucket to iterate over in order to find the k-mer we are looking for. This part of the algorithm requires a fair amount of communication between threads because it is very likely that the elements of each bucket we iterate over, have an *affinity* to different threads.

3 Computational and "Communication" Motifs

Except for reading and writing the data, the algorithm consists of two important phases of graph construction and graph traversal. In the graph construction phase, the process of adding k-mers to the hash table, requires an atomic synchronization which slows down the computations whereas in the graph traversal phase, the k-mer look-up process requires us to iterate over the buckets and their lists and so no explicit synchronization is required in this phase. This lights up the fact that the graph constructing phase should be the time bottleneck for our algorithm. Results from the experiment also suggest that this phase is the most time consuming part of the implementation.

4 Design Choices and Optimization

Perhaps the most important design choice we had to make is how to handle the race conditions. In a first version of our code we used "locks" to do so, but later we changed to "atomic operations" since we realized that the latter results to a much better performance.

5 Results

In this section we present the results we got from our implementation running on Edison using the small and large input data files as the following:

- Small graph: k = 19, n = 4514197
- Large graph: k = 51, n = 27000544

When the input was the small data set, we only used a single node to run our code, while when the input was the large data set we used multiple nodes. Hence, here are the implemented experiments where P is the number of processors or UPC threads:

- Single-node experiment on small graph: $P = 1 \cdots 24$.
- Single-node experiment on large graph: $P = 1 \cdots 24$.
- Multi-node experiment on large graph: 4-8 compute nodes, $P = 96, 120, 144, 168, 192$.

5.1 Speed Up

We start by presenting our results regarding the Speedup we got in the Single-Node and Multi-Node Experiment, respectively.

5.1.1 Single-Node Experiment (small input)

In figure 1 below we show the speed up of our code (blue dots) compared to the “ideal speed-up" (red dots) where the input data was the small graph Dataset for 1 to 24 processors. It can be seen that while our code is not capturing the "ideal" performance, it very similar to the ideal scenario.

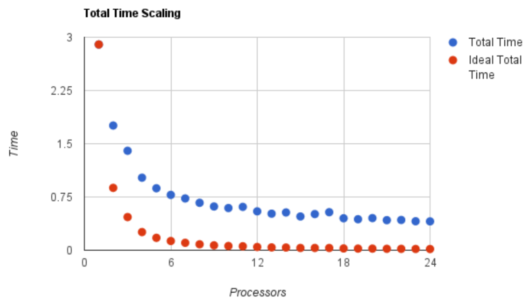

Figure 1: Single Node Experiment Speed Up

5.1.2 Multi-Node Experiment (large input)

In figure 2 below we show the speed up of our code (blue dots) compared to the “ideal speed-up" (red dots) where the input data was the large graph running on $96, 120, 144, 168, 192$ to processors, and to $4, 5, 6, 7, 8$ nodes, respectively. We note that for threads of more than 120, we do not see an excellent scaling. We think this is because that we are entirely bounded by communication. Adding more more computer power does not contribute towards faster computation, adding more threads can only increase communication or in the best case maintain the same amount of communication.

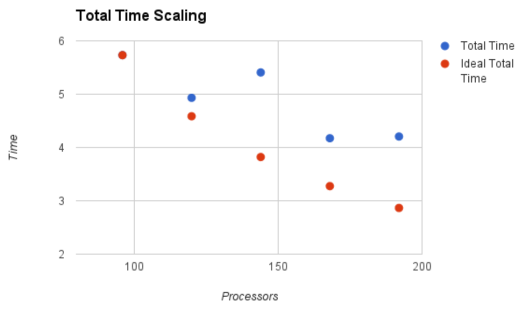

Figure 2: Multi-Node Experiment Speed Up

5.2 Scalability: Construction and Traversal of the Graph

In this subsection we present the results regarding the scaling of our code separately for the construction of the graph part and the traversal of the graph part. We also discuss the relative cost of each part.

5.2.1 Single-Node Experiment (small input)

In figures 3 and 4 , we present the scaling of the construction and traversal part of the graph in the Single-Node experiment, respectively, and we compare them to the "ideal" performance. In figure 5 we present the *average* percentage of time spent in each part. We see that in both parts our code has a behavior close to the ideal.

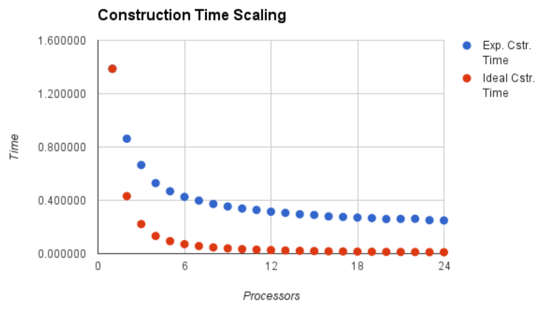

Figure 3: Construction Time Speed Up

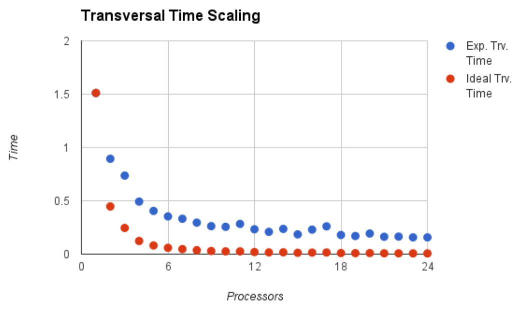

Figure 4: Traversal Time Speed Up

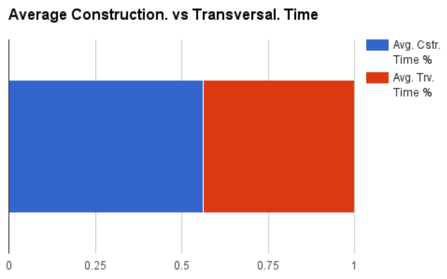

Figure 5: Average Percentage

5.2.2 Multi-Node Experiment (large input)

In figures 6 and 7 , we present the scaling of the construction and traversal part of the graph in the Multi-Node experiment, respectively, and we compare them to the "ideal" performance. In figure 8 we present the *average* percentage of time spent in each part. Similarly, having more computational power causes more communication between processors and impedes scaling.

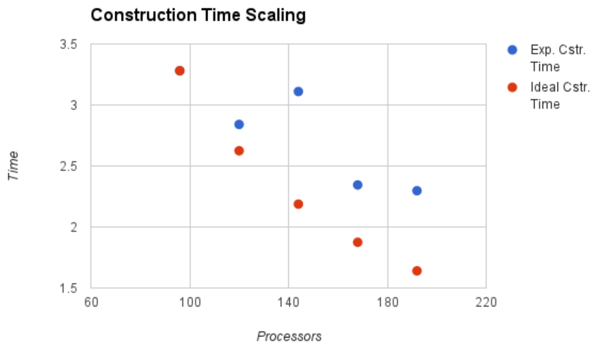

Figure 6: Construction Time Speed Up

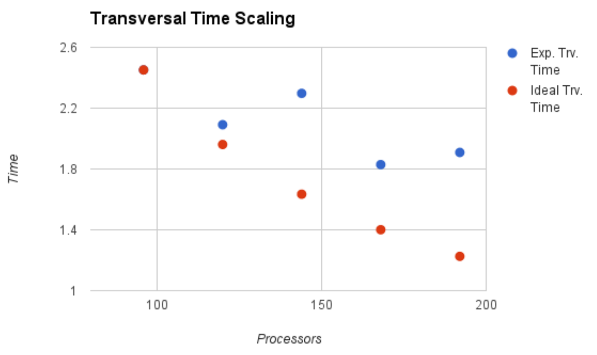

Figure 7: traversal Time Speed Up

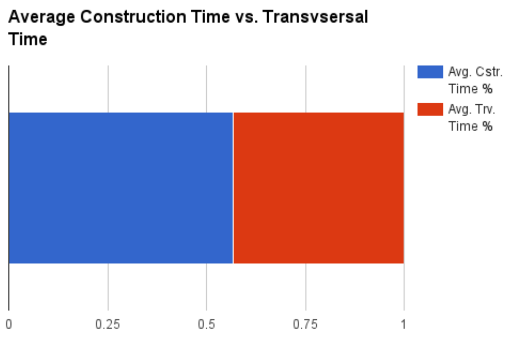

Figure 8: Average Percentage

6 Discussion

6.1 Implementation in two-sided communication model

There are at least two ways of implementing the parallel algorithm to a two-sided communication model, depending on whether the data (the graph) can fit to the memory of a single node, or not.

In the first way, where we assume that the graph does fit to a memory of a single node, we could have each processor constructing some part of the graph separately, and then merge all these parts using one root thread. For the traversal part, we could broadcast the graph to all threads and have them work in parallel. This way we have minimum communication between threads and we expect the algorithm to scale greatly.

In the second way, the graph does not fit into the memory. In this case each processor would again reads some portion of the input and builds its own part of the graph. However, at the traversal step, each processor would have to send a query to each other node in order to look up the next k-mer each time (assuming it doesn't find it in its own local, hash table). Of course such a solution requires a lot of communication and we expect that it wouldn't scale in a good way. The performance would possibly be improved in the case where there was some "locality" in the data, in the sense that we could partition them in a way so that each processor, most of the times, would only have to look into the local hash tables of neighboring processors.

Since in our application we saw that the data most of the times cannot fit into the memory of singe node, it is evident that the "partitioned global address space" model is a much better choice than the two-sided communication model.

7 Future Work and Conclusion

UPC is an efficient programming model which considers distributing the work by giving copies of the code to be executed by private threads which have access to shared memory data structures. Fully understanding the data allocations in both shared and local spaces is essential for UPC programmers while there are relatively less number of resources/tutorials available compared to other programming models.

In this project we gave a straightforward implementation of the parallel algorithm without using any involved design choices. As future work, we could consider using more sophisticated techniques such as "work stealing" between processors as well as a better algorithm that exploits the potential underlying locality of the data, to avoid communication between processors.

References

[1] T. El-Ghazawi, F. Cantonnet, P. Saha, R. Thakur, R. Ross, and D. Bonachea, “UPC-IO: A parallel I/O api for UPC,” *V1. 0*, 2004.

[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: distributed shared memory programming*, vol. 40. John Wiley & Sons, 2005.