CS 267: Homework 3

Richard Barnes, Danny Broberg, Jiayuan Chen

April 5, 2016

1 Introduction

This report describes our implementation of a de novo genome DNA assembly program using Unified Parallel C (UPC). UPC is an extension of the C language, outfitted for a Partitioned Global Address Space (PGAS) programming model. This model associates memory physically with specific processes but enables shared access. This bears syntatic similarities to OpenMP and other shared memory programming models.

Our paper is organized as follows:

1. description of the computational resources
2. a section that describes the general serial algorithm approach
3. a section that describes the general extension to UPC
4. evaluation of the code's performance

2 Machine Description

For our codes we use NERSC's Edison machine. The machine has 5,576 compute nodes. Each node has 64GB DDR3 1866 MHz RAM and two sockets, each of which is populated with a 12-core Intel "Ivy Bridge" processor running at 2.4 GHz. Each core has one or two user threads, a 256 bit vector unit, and is nominally capable of 19.2 Gflops. Notably, for our purposes, each core has its own L1 and L2 cache. The L1 cache has 64 KB (32 KB instruction cache, 32 KB data) and the L2 cache has 256 KB. The 12 cores collectively share a 30 MB L3 cache. The caches have a bandwidth of 100, 40, and 23 Gbyte/s, respectively.[1] Both the L1 and L2 caches are 8-way and have a 64 byte line size.[2]

3 The Algorithm

Sequencing long strands of DNA is difficult and time-consuming. For strands beyond a certain length it may not even be technically feasible. Therefore, faster and more scalable methods have been developed. Shotgun de novo genome assembly breaks a long piece of DNA into many shorter segments. These segments can be easily sequenced.

[1] http://www.nersc.gov/users/computational-systems/edison/configuration/
[2] http://www.7-cpu.com/cpu/IvyBridge.html

The problem then is to reassemble the original DNA by fitting the pieces together. In the simplified data set given here, the data takes the form of contiguous sequences of 19 or 51 bases (henceforth, a "kmer") with known prefixes and suffixes. We are guaranteed that each kmer is unique. Together, the kmers a De Bruijn Graph.

Our program will construct this graph and traverse its connected components. To do this, a hash table will be used to store nodes of the graph with hashed kmers acting as keys. The values of the hash table will correspond to the prefix and suffixes and, thereby, serve as edges linking the nodes together.

Our algorithm begins by detecting how many kmers are in the input. It then allocates a shared memory hashtable three times larger than this number. Members of the hashtable are stored via open addressing with linear probing: this is wasteful of memory, but greatly simplifies the data structure, thereby reducing the potential for errors. Using a hashtable of size $3N$ ensures that occupancy stays well below 70%, a point at which performance has been empirically shown to degrade significantly.

Kmers are hashed using Dan Bernstein's `djb2` hash function. Tests using the small data set with Austin Appleby's `MurmurHash3` didn't show a significant reduction in collisions.

Additionally, a local array of pointers to the hash table is allocated to be 1/50th the number of kmers. As the process reads kmers and adds them to the hash table, it makes a note of which kmers have prefixes indicating that they begin sequences and uses this local array to store that information. These starting-kmers (smers) form the starting points of the graph traversals which generated the contiguous (output) sequences. One might worry that storing smers locally would result in them being unevenly distributed; however, this is not a problem if we assume (a) that the smers are distributed evenly throughout the file and (b) that the smers, on average, are part of components of approximately equal size. These assumptions are reasonable because it is possible to randomize the order of the lines in the input file, which means that smers are evenly distributed and, therefore, component sizes should therefore be as well. For the purposes of the assignment, we do not randomize the input, but do verify that the assumption is at least nominally fulfilled by the test data provided.

4 UPC approach

The first thing to be done in the extension to UPC was to rewrite the packing and hashing functions for a PGAS framework. This gives us an initial list of start kmers (Elaborate on changes for these functions).

Next a function was built for finding the next kmer, so that graph traversal can occur in the partition global address space. This allows one to get to the point of allowing contigs to be generated(Elaborate on changes for this function).

At this point a choice needed to be made for the communication of PGAS threads. We decided to read input into a circular buffer that all the threads could then have blocks of dedicated address space in. (insert better explanation of how this buffer works?)

Later we considered the option of enabling parallel read-in capabilties, since at this point the UPC code did a sequential read in and then distributed the input accross the partitioned global memory space. This means that the only benefit from using PGAS came from having access to a large memory space, but no improvements in speed could be gained yet.

To perform a parallel read-in, we hoped to involve atomics on the hash table - and we consider the use of conditional swap as a method for implementing this. However, atomic procedures are

	Alg. **??** (packing sequences)
1	*Function* for initial lookup of kmers
2	*Function* to convert FourMer to packed
3	*Function* to pack sequences
4	*Function* to unpack sequences
5	*Function* to compare packed sequences

	Alg. **??** (kmer hashing)
1	*Function* to create hash table and preallocate memory
2	*Function* to compute hash sequence values
3	*Function* to return hash value of a kmer
4	*Function* to look up Kmer and return a pointer to it
5	*Function* to add a kmer to hash table
6	*Function* to add a kmer to start list
7	*Function* to deallocate memory buffers
8	*Function* to deallocate hash table memory

	Algorithm **??** (contig generation)
1	*Function* for time keeping
2	*Structure* for Kmer data
3	*Structure* for startkmer
4	*Structure* for buckets
5	*Structure* for hash table
6	*Structure* for working memory buffers
7	*Function* for getting number of Kmers in file

Table 1: Functions and Structures included in Algotithms **??**- **??**

not traditionally used for individual bytes and we were not able to find a Berkeley UPC atomics library. We therefore considered using UPC locks instead - which has it's own difficulties due to the inability to lock an openly addressed hash table. As an alternative strategy we decided to partition the table into blocks of 100 do locking on the individual blocks. (Insert code to describe how this works)

This methodology works only if the probability of a lock collision is relatively small. (Insert part about the relative probability of collisions existing for certain hash tables... ?)

After testing the code further at this point, numerous deadlocks were witnessed. To address this issue, we decided to try synchronized rotating access to chunks of the table - which resulted in an intiial code that was faster than the serial version. (Insert a figure comparing initial performance against serial version?)

In the later stages of the code, local representations of the hash tables were created to avoid synchornization issues. We also began to wonder if things could be made faster with linked lists, which would have $\circ(1)$ insertion time and $\circ(BinN)$ retrieval time, as opposed to open hash table approaches which have $\circ(BinN)$ for both. If the circular buffer method was still applied to linked lists, then the approach could remain lock free.

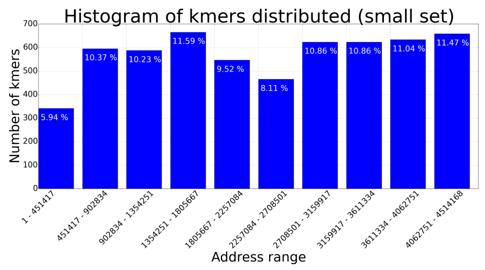

Figure 1: Distribution of start kmers to address space (Short data set)

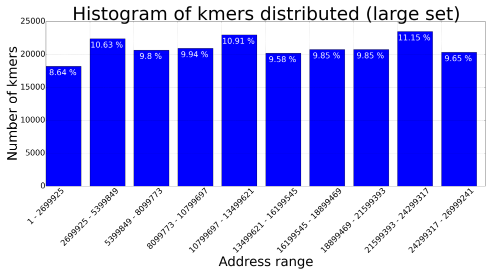

Figure 2: Distribution of start kmers to address space (Long data set)

figures/plots that could exist:

- timing plots?

List of suggested thigns to include from homework statement:

- A description of your distributed data structures and parallel algorithms
- A description of the computational and communication motifs of the parallel algorithms.
- A description of the design choices/optimizations that you tried and how did they affect the performance.
- A description of how you avoided race conditions.
- Speedup plots that show how closely your parallel code approaches the idealized p-times speedup in the two experimental scenarios described in the previous section.
- Discussion of the scalability and relative costs of the parallel graph construction and traversal algorithms.
- A discussion on using UPC for such an application with the underlying computational motif.
- A discussion on how would you implement the same parallel algorithms in a two-sided communication model (e.g. by using MPI).

5 Performance

- Simple compared results against serial implementation.

6 Summary