Submit your report and source codes on Moodle. We need to be able to build and execute your implementation to receive credit. It should be a zip or tar file of a directory that contains both your report and your Makefile and source code. Spell out in your report what Makefile target we are to build.

To build the program simply type make.

You may need to type source init.sh to get the correct modules loaded.

To run the job, you may type sbatch job-upc.

Contributors:

Sam Pollard
Rory Klein
Albert Andrew Spencer
Brian Francis Lee
Conner Richardson
Robert Birch

A picture describing how the hash table and heap are laid out in memory.
It would describe where the locks are stored, where heaps are located:

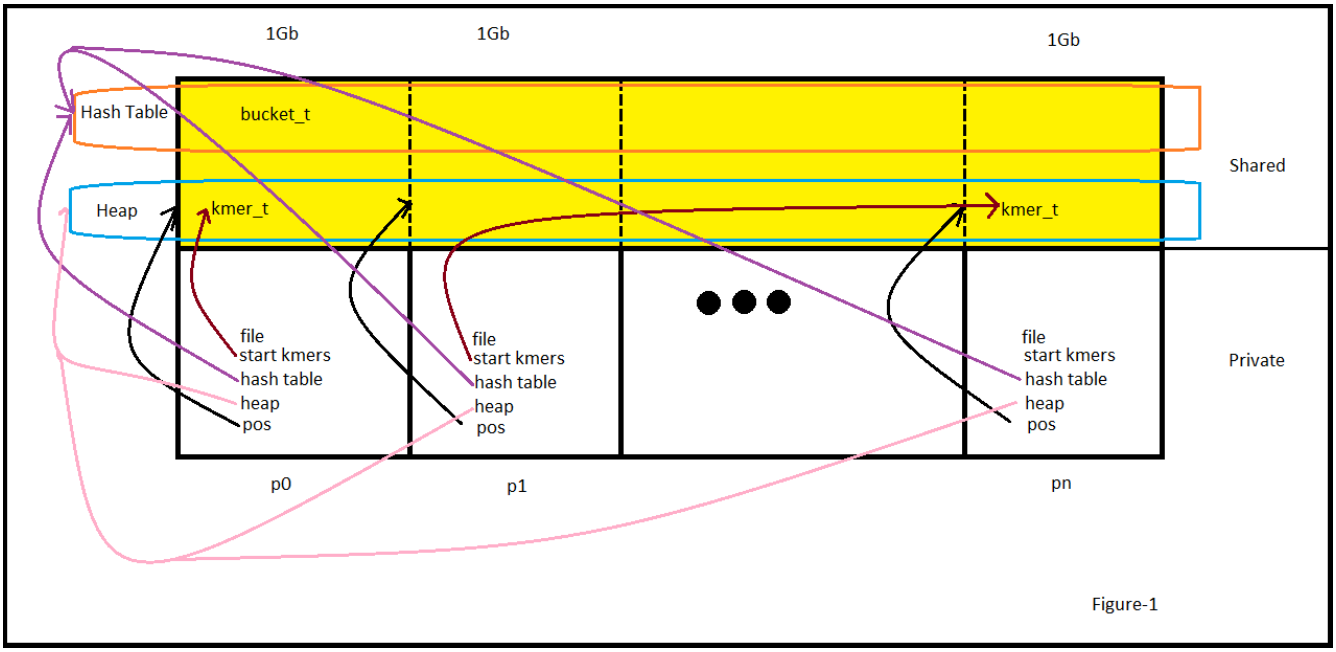


Figure-1

file: each processor gets a private copy of the file handle and reads in its kmers. Example p1 reads in from file[size/N...(size * 2)/N].

start kmers: a linked list of pointers into the heap of all start kmers of this local processor's file portion.

hash table: local pointer to the global Hash Table.

heap: local pointer to global Heap.

pos: offset from heap for the processor's affinity memory.

bucket_t: contains both a shared pointer to a kmer_t and a shared lock for this bucket.

kmer_t: holds a packed kmer and a pointer to the next kmer in the bucket.

A description of your distributed data structures and parallel algorithms:

For a description of the distributed data structure, see the figure above. We parallelized the serial implementation by splitting up the number of kmers. Each processor is given N/P kmers, where N is the total number of kmers and P is the number of processors. Each thread is then in charge of inserting their kmers to the global hash table. Each thread also builds a startNodeList when they come across a kmer with a left extension of 'F'. After the hash table is built each thread loops through their startNodeList and performs the De Brujin Graph Traversal algorithm. Each thread then outputs the contigs they've generated to their own file.

A description of the computational and "communication" motifs of the parallel algorithms:

Our approach attempts to minimize communication by assuming the start kmers are evenly distributed within the input file. In the test case this resulted in reasonable load distribution.

Computation was divided completely among the threads by the number of start kmers each thread read in from the input file.

Potential race conditions only cropped up when building the graph. These were addressed by creating a lock for each bucket in the hash table. This caused an extra memory overhead but gave fine granularity when locking.

A description of the design choices/optimizations that you tried and how did they affect the performance:

At first we tried a global lock which controlled the entire hash table. (BS) This would have the lowest number of locks but effectively serializes the graph creation. Another idea we explored was to lock the hash table on a per-processor level. However, this would cause additional complication because the program would have to be aware of how the hash table was laid out.

The layout of the hash table is round-robin with a block size of a single bucket. This should be as good as any layout since the nature of the hash table means the data locality is impossible to predict.

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:

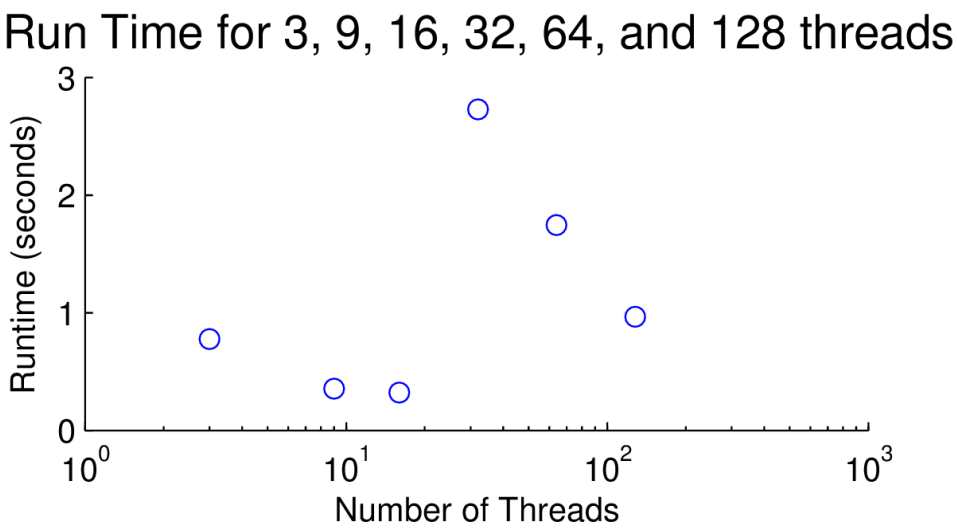
One scalability issue is for a very large number of processors it gets difficult to ensure the number of start kmers will be evenly distributed among the threads.

A discussion on using UPC for such an application with the underlying computational motif.

UPC was difficult to work with because of the obfuscation that occurs when translating a upc file to the intermediate form via upcc. This makes debugging incredibly difficult because it is difficult to determine where in the original upc code the error occurred.

A discussion on how would you implement the same parallel algorithms in a two-sided communication model (e.g. by using MPI).

The difference would be relatively difficult because receiving data would require a send/receive from two processors, and most of the time with the graph traversal would require both the thread which requires the hash value and the thread with the stored value to be aware of the send/receive pair, and also whether this message is even required if the hash lookup happens to be local.



This program had strange scaling in terms of the number of threads. This could be due to a couple of factors. One is that the load imbalance may be off significantly. Since the total number of kmers may not be evenly divisible by the number of threads, the last thread has to pick up stragglers. At a certain number of threads this could be very large. Another factor is at a certain point, the program could jump to multiple machines. This would cause the machines to have to send messages to each other which could explain the enormous jump in runtime.