Note:

Hi 267 Staff,

I've unfortunately had a fairly difficult time with this project - I had a submission deadline last week and another one by the end of next week. Furthermore, the partner I had for the last homework decided to work with his lab mate, so I've been working alone on this project. I recognize that none of these are valid excuses, but I am willing to accept whatever penalty is inflicted.

I'm going to submit my solution and design by this deadline, and, after my other deadlines, I intend to finish this project, create the graph, and write a full report (along with some extra credit like the batching). I hope that I'm able to receive some credit for the work when I finish it.

Thanks,
Ankit Mathur

Scaling Experiments

Still to come :(

Implementation and Optimizations

My design seeks to split the data evenly across the n nodes. As such, we house a data structure for the hashmap at node 0.

Setup

Upon initialization, the hashmap is initialized with an array that contains n global pointers to segments, along with n global pointers that point to int arrays that are the same size as the segments. These integer arrays are 0 or 1 depending on whether that space in the hashmap is used. If that space is used, then the array has a 1-value.

At setup time, each node needs to allocate the two arrays for the data and the used-values. Then, each node needs to update the hashmap data structure's list with these pointers. This needs to be done in an atomic fashion, but UPC++ allows atomic operations only on integer arrays (not arrays of kmer_pair). Therefore, we create a hashmap owner variable. Each process spin-waits until it gets ownership of the hashmap, at which point it adds its global pointers to the list. Once this process is done, each node fetches the fully updated version of the hashmap. Since all the of the data structures in the hashmap are global pointers, these pointers never change, and over the course of the actual code, the functions in the hashmap are implemented to perform operations by performing remote gets (when necessary). As such, the data structure never needs to be re-fetched from node 0, which would make node 0 the bottleneck.

Inserts

During normal operation, we need to handle concurrency issues on the inserts. The granularity of locking that I've chosen as of now is at the level of each node's segment. The lock acquisition scheme uses spin waiting on the value of an integer array. This inflates node utilization, which is unfortunate, but UPC++ does not offer any native signaling scheme that I can identify. It's possible I can use remote_fetch_add for this, but I have not had time to explore that.

Otherwise, inserts occur the same way that the starter code did them. The request_slot function has been updated to use locking, and the write_slot function has been updated to perform remote writes when necessary

Finds

Finds are a similar operation as inserts. We don't require concurrency for reads as of now.

The check_if_used function has been updated to use remote calls.

UPC++

Using UPC++ was a relatively painful experience. I personally don't feel this framework is quite ready to be used in a class project, given that the number of examples is essentially limited to what's the in the v1.0 docs.

That being said, the abstraction offered by UPC++ was outstanding. If I were using MPI to solve this problem, I'd have to write the messages that do puts and gets, which would require a lot more lines of implementation. This hits at the fact that the abstraction of a global pointer in a distributed system is quite useful.

OpenMP would really struggle to help with this problem because it doesn't really have functionality that allows for communication across the threads as easily. As such, we'd have to find ways to batch the finds and inserts and we could parallelize those operations. Memory bandwidth and contention may become an issue with this approach.