Carlos Moustafa (aem278)
aem278@perlmutter

Design

For my implementation, I chose to distribute the arrays *data* and *used* across the ranks using UPCXX's global_ptr construct. Then, to ensure that each rank has access to call Remote Procedure Calls on other ranks, I used the *distr_object* type.

Insert

For the insert function, I kept the linear probing logic the same. The main difference is that the probing and insertion happens in a Remote Procedure Call on one of the ranks. I determine which rank to perform the insertion on using the hash of the kmer.

In particular, the rank the insertion is performed on is kmer_hash % rank_n. So, assuming the hash function is collision-resistant, then our assignment of kmers to ranks is also collision-resistant (i.e. a "good/uniform" assignment of kmers).

Then, for avoiding race conditions, I used upcxx's atomic_domain. The specific atomic domain operation I used was compare_exchange, which takes as input (1) a global ptr to an integer and (2) two integers. Atomically, it checks if the value of (1) equals the first integer, and if so, writes the second integer to overwrite (1). I used this in this context by updating the *used* array atomically to reserve a slot. If the value of *used* is 0 at the slot of interest, then it atomically writes a 1 to it, which reserves the slot for the insertion. Otherwise, if there was already a 1 at that location, the linear probing continues.

For actually writing to the slot in the global_array after an unused slot has been reserved, I simply downcast the global pointer to a local C pointer. This is legal because we know the rank executing the remote procedure has affinity to the global_array (else, we wouldn't have called the procedure on that rank in the first place).

Find

Similar to insert, most of the probing logic is the same, except it is all wrapped in a Remote Procedure Call on the rank determined by the same hash function kmer_hash % rank_n. It is important that the function used to determine rank of the RPC is the same as the one used in insert, else we would be searching for the desired kmer on a rank that would never have inserted it in the first place.

Carlos Moustafa (aem278)
aem278@perlmutter

Since find is a read-only operation, there was no concern of race conditions and so I did not use any atomic_domain here.

For checking if a slot is reserved, I downcast the *used* global_array to a C ptr and check if the value at the slot is non-zero (which it would be only if it was previously reserved atomically in a previous insert Remote Procedure Call).

As a hack, when the desired kmer is not found in the HashMap, I return a kmer with forward and backward extension equal to 'N'. Since N is not a valid extension, the calling rank can check if the forward and backward extensions equal 'N', and if so, the caller knows the find operation was unsuccessful and can return false.

Optimizations

One optimization I considered for the find function, is early-stopping to avoid unnecessary operations. The default find function supplied with the serial code checks every single slot in the underlying data array in the worst case. Most of these checks would be unnecessary and wasted computation. Since we know we never delete entries from the HashMap after insertion, it is safe to stop probing at the first iteration that an unreserved slot is encountered.

This is because, if an unreserved slot k is encountered and we have not previously found the kmer of interest, we can be certain we won't find it anywhere else in the array, since if this entry actually was inserted previously, it would have been inserted at slot k or earlier.

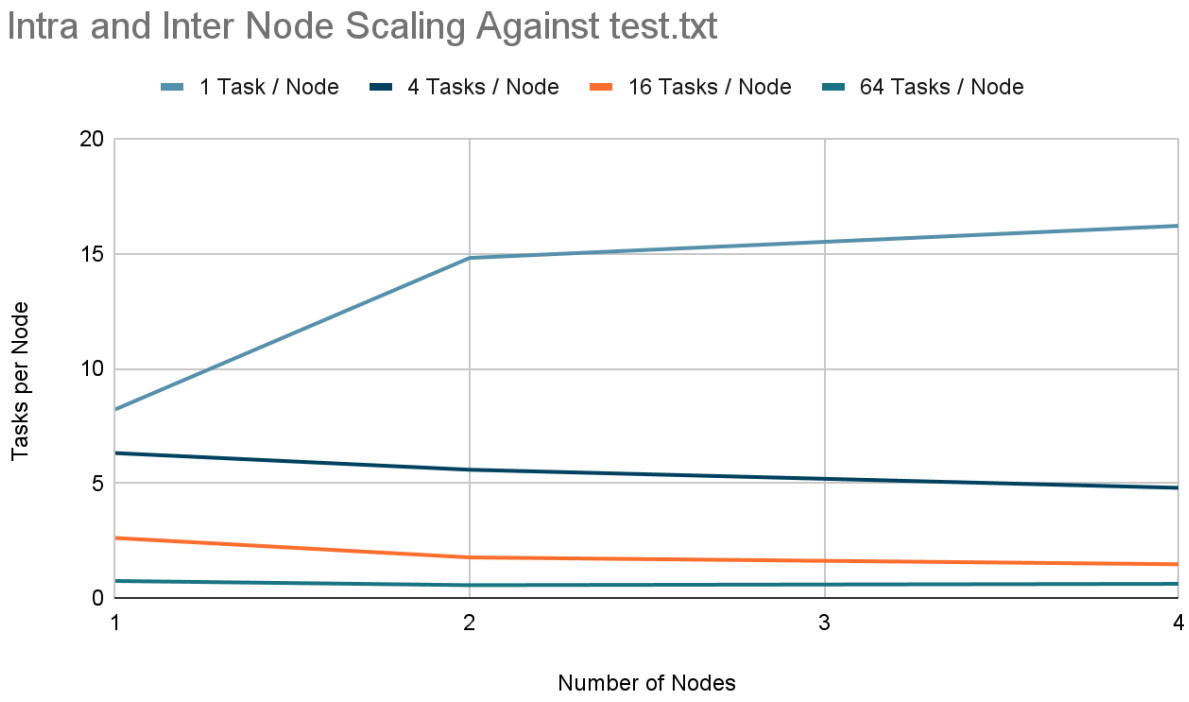
Another optimization I considered was batching inserts, by grouping kmers by affinity and performing only at most M Remote Procedure Calls, where M is the total number of ranks. Due to time constraints, I did not end up completing this implementation.

Carlos Moustafa (aem278)
aem278@perlmutter

Experiments

Here, I note that Perlmutter did not allow me to allocate more than 4 nodes at a time:
salloc: error: Job submit/allocate failed: Job violates accounting/QOS policy (job submit limit, user's size and/or time limits)

# Nodes	1 Task / Node	4 Tasks / Node	16 Tasks / Node	64 Tasks / Node
1	8.221346	6.319331	2.621774	0.749291
2	14.818570	5.593907	1.775792	0.566961
4	16.217535	4.806097	1.478111	0.624190



Carlos Moustafa (aem278)
aem278@perlmutter

Discussion

The use of UPC++ impacted my choice of using the dist_object type. Coming from background of being familiar with distributed systems, I had to spend extra time understanding the UPC++ constructs and what messages it would be sending between processes under the hood. If I were using MPI or OpenMP, I would have implemented this using explicit message-passing and synchronization.