Roman Kazarin
Michael Nekrasov

Homework 4 Design Document

Changed/Added Classes:

api.Task:

We leave it up to the application programmer to determine how caching and space-runnable tasks are handled. This is done in the implementation of a Task. We now add an isCachable method to mark if Computers can store this Task for quick lookup next time the same task is referenced with the same inputs. We add an isShortRunning method to mark if tasks will quickly terminate and so should be prioritized to be run on the space, reducing the latencey of sending them to Computers

system.SpaceImp:

Our Space implementation incorporates the idea of “space-runnable tasks”. Therefore in the constructor of the Space, we provide a parameter *numLocalThreads,* which tells the space how many *ComputeNode Threads* to run within the space. The setTask method is called by a Client on the original task: the task is added to the waitingTasks queue. One change from our previous implementation is the creation of a Proxy for each computer. A Proxy object has a Collector and Dispatcher object, which both work to implement Task prefetching. A dispatcher runs in its own thread and will populate a computer’s blocking task queue with as many tasks as the *prefetchBufferSize* variable allows (set when running a computer). A Collector also runs in its own thread and will process completed task results and pull them from a Computer. To minimize latencey for short running tasks, the Dispatcher for the ComputeNode running on the thread will only pull Tasks that self identify as *shortRunning*. This restriction is lifted if there are no non-space ComputeNodes connected.

system.ComputeNode:

A ComputeNode contains a *tasks* queue, a *results* queue, a number of *ComputeThreads*, and a *cache.* Unless explicitly specified, the number of threads each computer will spawn to perform tasks will be handle is equal to the number of cores the OS exposes to the JVM. In the run method of a *ComputeThread,* each thread will keep taking tasks from the *tasks* queue and calling call() on them, placing the result into the *results* queue. If the task allows caching, and caching is enabled on the ComputeNode, a result is cached if it is an actual value, and not a further decomposition of tasks.

Design Discussion:

Our goal for this assignment was to modify our infrastructure to reduce communication latency, implement some form of caching, and take advantage of multi-core processors of our ComputerNodes. In order to reduce communication latency, we implemented Task prefetching. Computers can now store a queue of tasks and are sent new tasks, while they executing. Now, both the Space and the ComputerNode no longer have to wait for each other: there are always tasks ready for the ComputerNode to execute and there are always Results ready for the Space to collect.

Our next modification was caching. Caching is by default disabled for TSP and enabled for FIB by the task implementation via API methods. We also allow disabling caching via command line arguments on a per computer basis. The Computer will first check if this task has already been cached: if so, it will retrieve it and put the result into the *results* queue. If not, it will be added to the *tasks* queue. Caching as it is currently implemented is only done where tasks return a value (not a decomposition). For Fib this means that only the Fib(1) Fib(0) and the Add operations are cached. Tasks like Fib(2) are still decomposed into Fib(1), Fib(0), Add. Ideally in future implementations caching would allow top level calls that produce a result to be cached. For example When calculating Fib(6). it decomposes into multiple Fib(4) sub trees, ideally fib(4) would be computed once and cached.

To further reduce amount of network delay we implemented on-space compute threads. Tasks can self identify as being short running. The space prioritizes executing short running tasks locally, so as not to have to wait to deliver them to remote Compute Nodes. For example in TSP, while the long-running permutation checking is always delivered to remote computers, the comparison between results a fast operation is prioritized to be done locally.

Our final modification was taking advantage of a ComputerNode's multi-core processor. Instead of having to start multiple JVMs on a single machine to utilize all the cores, a single ComputeNode will spawn multiple threads to fill the cores. This reduces the number of objects that a Space has to keep track of increasing the scalability, and allows a shared state between the threads. Each ComputeNode has a shared cache between all it's threads and a shared results and tasks queue.