By: Hallvard Jore Christensen, Ingeborg Ødegård Oftedal, Paul Philip Mitchell and Kyrre Laugerud Moe

CS290B: Design Document HW4

As the homework asked for we have focused on reducing network communication, making the system do asynchronous network communication, and let the system utilize all cores available on the system.

Our system already had a computer proxy (ComputerProxy) that served as a mediator between the Space and the Computer(’s). In order to make each computer utilize all available cores, we toyed with the idea of letting each computer core register to the space. This would effectively mean that a computer with 4 cores would register 4 times. The problem with this is that the space would have to allocate 4 computer proxies, instead of one. This means that the space will become a bottleneck faster as the system scales. Because of this we chose to keep the current structure, and instead let the computer handle concurrent computation. This also seemed like a natural choice, considering implementation of prefetching. What follows is a description of the control flow:

First the computer (ComputerImpl) registers in the space, and is allocated a ComputerProxy. ComputerProxy does three things. Firstly it tries to retrieve a Closure from a LinkedBlockingQueue. When a Closure is retrieved the closure is added to a list of retrieved closures, for fault handling. ComputerProxy then tries to execute on the ComputerImpl. The execute method adds the retrieved Closure to a local LinkedBlockingQueue in the ComputerImpl. After this it checks whether the the local queue is bigger than the prefetch limit. At initiation the ComputerImpl checks how many cores are available, and initiates one CoreHandler for each core. These handlers simply executes cilk threads as they become available in the queue. This implementation is in our opinion a good solution, as it by its nature is modular, allows for asynchronous communication, multi-thread execution and flexible prefetching. By setting the prefetch limit to 0, prefetching, and asynchronous communication from the Space to the ComputerImpl is effectively turned of.

In order to ensure that fault tolerance is maintained, a list of retrieved closure is kept by the ComputerProxy. When a cilk thread has successfully finished it sends a call back to space, This notifies space that the closure/thread was successfully executed. The closure id is then added to a HashSet, representing all finished closures/threads. If the computer throws an exception back to the ComputerProxy, ComputerProxy’s list of Closures is check against the HashSet of completed closures. Any closure that is not found in this set is re-entered into Space’s own queue of ready closures.

In order to decrease communication between space and computer’s, universal unique identifier (UUID) is used to generate id for closures. This allows closures to generate their own id’s upon creation, without consulting a centralized entity, such as space. The specific method that is used is java's own `UUID.randomUUID().toString();`, which is called in the constructor of Closure.

As described above communication from SpaceImpl to ComputerImpl is inherently asynchronous if pre-fetching is allowed. To ensure asynchronous from ComputerImpl to SpaceImpl, we introduced a SpaceProxy. SpaceProxy implements the same interface as SpaceImpl, namely Space. If the system is run without asynchronous mode and a method is called, SpaceProxy calls the same method in SpaceImpl. If asynchronous mode is on, method calls, and their arguments are wrapped in a QueueTicket, and inserted into a LinkedBlockingQueue, before the method returns. At initiation in asynchronous mode, a separate thread is started. This retrieves QueueTickets and calls the coresponding method, with it's argument, on SpaceImpl.