Design document for CS290B, homework 4

By Torgeir Lien and Oeyvind Rein

Intro

The project is built around three main modules, API, system and problem specific tasks with clients. Tasks and their corresponding clients can use the methods found in API to interact with the compute system. The API has four components which deals with different aspects of the compute system which the task programmer can use, namely; DAC, shared, space and task. The programmer can use the utility of the API to divide and compute tasks distributed and concurrently.

The compute space it self consists of four main parts: client, space, worker and task. Clients send large tasks to space which then distributes these tasks to workers which execute the tasks. The tasks can and should be designed to split themselves up in smaller parallel pieces so that they can be executed on many computers in parallel.

Infrastructure

Proxies are implemented as interfaces between the modules in the system to make inter-thread and remote communication as efficient as possible. The only blocking call in the system is the space.take which is sent by the client. Proxies themselves do however have blocking calls, but that doesn't affect the efficiency of the system.

Shared objects are updated in the space when the received object is detected to be 'newer'. The shared objects are currently only distributed when a 'old' object is received in the space, then the newest object is returned to the sending worker. Workers can ask for new objects whenever they want. A better solution would be to distribute objects to all workers whenever a new object is detected by the space. The shared objects are generic and the interfaces are implemented by the task computer so that some custom features can be made.

Fault tolerance is achieved by looking for remote exceptions between workers and spaces. In the case of a remote exception from a worker, the space will simply drop it and not try to reconnect. The workers have to register to the space themselves. If a worker is dropped the task it was working on will be reassigned to an operational worker.

Implementation

The DAC uses a DAG model where nodes are threads and edges are dependencies. DAC uses two types of tasks, one for computation and one for composition. Computation tasks are spawned with 'spawn' , composition tasks are spawned with 'spawn_next' and a number of inputs which it has to wait for before starting its computation. This functionality is sewed together with closures and continuations. The tasks programmer does not have to worry about how these are connected together, they are assigned implicitly by looking at which tasks and worker is spawning and using spawn_next.

Generic objects are used to interface between modules where it is possible in order to make the system as general and applicable as possible. The task programmer will have to deal with type checking of the objects which are being sent around.