CX: A Scalable, Robust Network for Parallel Computing

Several trends in technology like bandwidths getting cheaper, computations getting faster, non-decreasing communication latency, gives new opportunities. Technology can make technically infeasible “internet computations” possible, which is infeasible for a network of supercomputers. The maximum “infeasible” problem size keeps increasing for non supercomputers. A goal is exploiting the computational capacity.

The CX project, is an open extensible **C**omputation e**X**change. It can be instantiated several ways, privately, publicly and can also be application specific. The goals of the system is as followed; The system should support large internet computations, reduce human administrative costs and reduce application design costs. The system must adapt to its circumstances, e.g available various producers and consumers, priorities, computations of various lifetimes. Consumer discriminating should be supported, as well as no human intervention for upgrades after system installation. It should hide fault tolerance and communication for the users, have a simple API and a computational model that supports decomposing and composing of tasks.

It is not an easy task to achieve these goals when it comes to issues like performance and complexity. A subsystem presented in this article, *Production Network* service, focus on design with respect to administrative and application complexity as well as performance. Administrative complexity is managed by the Java programming system, and the application complexity is managed by a simple general API. This article will focus on CXs Task Server, the Producer and the Consumer.

CXs computational model, focusing on having the communication latency in relation to the execution speed. Similar to Cilk Threads, the computation model is a dag of non blocking tasks.

CXs “task server”, is used for application communication with the system. Where the decomposition occurs, depends on the application programmer. Communication can also be batched. The “task server” holds tasks stored by the consumer and sends callback for the results, the producers take the tasks and perform computations.

The time complexity are of low time for scheduler operations, in the number of producers and tasks it is O(1). The scheduler must therefore be general. The system must tolerate any component fault, as well as be high performance and scalable. For the computation progress, any compute producers fail must be transparent. No human intervention should be required from recovering from server failure. The entities relevant for the article is the Consumer, Producer, Task Server and the Producer Network. Java is used because of its VM, its industrial strength, its JIT technology and many programmers like to use Java.

Design principles that needs to be followed concerns scalability, requirements for achieving high performance and the design of the computational part of the system. For scalability, the components must follow the principle for consuming resources. It must happen independently of the components of the system, or else a bottleneck will occur. For high performance, each producer must finish the consumers job before it is “free” again. The design of the computational part consist of an isolated cluster and a producer network. For making the design scale and be fault tolerant, the producer network is being used. In the isolated cluster, a computation start by the consumer putting the computations “root” task in a task server. The servers proxy gets automatically downloaded when a producer registers with a server. The tasks gets removed from the server when it is completed by the main proxy server, thus transaction are not needed. Intermediate results gets put in the server by the producer from the decomposition.

For hiding the communication latency for producers and their servers, they use task caching. *Ready task heap* is maintained by the proxy sending tasks to the server and forward arguments, and is based on two components; number of times a task has been given, that is, if Task A has been assigned more than task B, then Task A has a higher priority.

When the amount of tasks its below a watermark level in the proxies task cache, a copy of the tasks its pre-fetch from the server. The pre-fetched task is among the fewest assigned tasks and the lowest level in the dag. After completion it gets removed from the heap.

Before a tasks is ready, it get puts in a collection of unready tasks. When its ready, it gets put in the ready task heap, and can be pre-fetched. The task graph is a DAG and the spawn graph is a tree, giving the task an unique identifier because of the path from the root to the subtasks making it possible to delete duplicates.

The server has to balance the workload between the producers, to make sure the fastest available producers is not idle. This is done via pre-fetching, making it possible for producers to steal tasks spawned from other producers when they deplete their task caches.

Bottlenecks are avoiding by the server networks because the functionality of the isolated cluster each server retains. The diffusion of tasks is similar to the way the producers handles it. Ready tasks is moved, and downloaded tasks do not get moved to other task servers. When a task is complete, it gets removed. Other than delays caused by requests og receive latency, a design goal is to have no communication delays between producers and their tasks server.

For fault tolerance, the article propose several methods. The servers state must be recoverable in order of toleration of server failure. This can be done by replication of the servers state to other servers. By organizing the servers in a *sibling-connected fat tree*, the state of the server can be replicated. The siblings works as a *mirror group* for each server, and each time a state changes, it gets mirrored. This fault tolerance makes it possible for the network to recover itself, even when it's a sequence of failures.

The distribution of code strategy is as followed; the amount of producers who downloads the code must be bounded to scale in the distribution scheme, thus the download points must increase accordingly. Task class files can be downloaded from a task server.

Several experiments was done on the Departmental Linux cluster. From the experiments they elaborated an ideal speedup general expression, Tp(w). Work w on a vector p of processors. They figured out a load generator, for arduous the DAG they used F(n) computation. They used a technique based on two short executions where they could estimate execution time on long sequences.