Adapting Algorithms

The proposed approach requires one to view an algorithm as a finite state machine, where the separation between states is governed by the need of synchronizing some state variables. The bulk synchronous parallel model facilitates many of these goals, hence we based our approach on it.

**Bulk Synchronous Parallel**

A BSP [1] based program consists of a series of supersteps, each divided into three stages:

- **Concurrent computation**
- **Communication**
- **Barrier synchronization**

One of the main advantages of this scheme is elimination of race conditions (related to message passing) and deadlocks, by avoiding circular data dependencies. Another advantage is that the BSP program structure can be naturally exploited for fault tolerance.

Previous experience with existing BSP-based frameworks showed that most current solutions do not satisfy our requirements for scientific computing applications [6].

**NEWT**

Models such as MapReduce [5] and Pregel [2] have previously used BSP to craft transparently fault tolerant distributed computing frameworks. However, in these cases fault tolerance relied on restricted programming models. We propose a more open ended approach, which uses a concept similar to continuation passing.

The program under NEWT is a mapping of functions and their labels, with each function having a explicitly defined continuation – the next function in the sequence. Additionally a state object is defined, which is passed from function to function. The rest is handled by the framework.

**Introduction**

We looked for a distributed computing framework that could be utilized for running iterative scientific computing algorithms in the cloud, but most solutions proved lacking in some respect. This is why we decided to create our own with the following goals:

- Provide automatic fault recovery.
- Retain the program state after fault recovery.
- Provide a convenient programming interface.
- Support (iterative) scientific computing applications.

The proposed approach requires one to view an algorithm as a finite state function to function. The rest is defined, which is passed from function in the sequence.

Additionally a state object is having a explicitly defined mapping of functions and their passing.


**Running on Hadoop YARN**

Fault-tolerance requires a scheduling mechanism and a resilient storage environment. These can be provided by the Apache Hadoop project, since with the introduction of YARN, Hadoop cluster resources can be utilized by frameworks other than MapReduce. As such the following components were used in implementing NEWT:

- Apache Hadoop YARN (Yet Another Resource Negotiator) – provides resource management, process scheduling and monitoring. [3]
- Apache HDFS (Hadoop Distributed File System) - provides a highly fault-tolerant distributed file system, that was designed to be run on commodity hardware.
- Apache MINA - provides message passing through socket connections. [4]

**Evaluation**

We’ve run scaling experiments on the prototype, showing it scales as well as MPI for coarse grained iterative algorithms (PAM), but, comparing to MPI, has a bit too much synchronisation overhead for fine grained ones (CG), which leaves room for more optimization (such as an unbuffered communication mode).

Additionally we measured the overhead of the framework’s fault tolerance mechanisms.

**References**
