Final Project Proposal: A framework for writing parallel and (doubtfully) distributed graph algorithms

Jared Roesch and Tristan Konolige

Introduction

As Haskell programmers we have developed an appreciation for the balance between abstraction and high performance. Often times these two goals are at direct odds, for example the goal of MPI is to not give up performance for the sake of abstraction. We are greedy indviduals and want to have both. One possible approach to this is to build a language that only allows one to express operations that are inherently performant, parallel, distributed, or whatever criteria one is interested in. This is the appraoch taken by MapReduce which only provides the primitives that they can successfully distribute. In their case they have only support `map` which emits a (K, V) pair and then `reduce` which applies a function over all values that map to a specific key. This programming style has allowed multiple projects to build higher level programming abstractions on top, like Scoobi and Scaling. The challenge of making the programming style usable is then put on the library author who is the one that must worry about the restrictions of the core language.

The current state of the art in large scale parallel computing is filled with approaches that are built upon the abstraction capibilities of C which are poor. We really wanted abstraction and expressivity without giving up performance and so we decided on Haskell is the perfect solution for reaching our goals. It has been made performant in vareity of situations from string operations (recent paper from intel), SDN controllers, and more. We could of alternatively used C++ but Haskell has properties that make our goals much easier to actualize.

An area of active research is the automatic parallelization of languages like Haskell. Haskell is unique in the fact that it is a purely functional and lazy. These properties enable types of reasoning that are not easily performed on other languages. Haskell is filled with many approaches that enable automatically parallel code such as `accelerate` a DSL for GPU programming, 'repa for writing code with parallel arrays, as well as approaches to automatica parallelism such as data parallel haskell (for SIMD style programming), the Par monad for speculative paralleism, as well it offers both lightweight user space threads and bound OS threads. The finally cherry on top is Haskell's mature FFI for interfacing with foreign code allowing one to write critical components in C or C++ if needed, exposing pointers and other low level machinery to the programmer.

We would like to extend and explore the space of graph algorithms similar to Combinatorial BLAS but exploit the improved type level reasoning and fusion properties Haskell enables. We are planning on using Repa as a basis providing operations on parallel arrays. Repa provides a vareity of tools for dealing with Arrays and allows for the specification of the underlying representation and allow for the compiler to perform the optimization. Using Repa provides a battle-tested framework for dealing with parallelism and arrays, but their work still has limitations. The largest being that Repa does not provide support for nested parallelism or the distribution across multiple nodes.

We propose a novel extension on top of repa jokingly named doubtfully distributed arrays which will provide primitives for manipulating distributed arrays. We are visualizing our distributed arrays as a type of Repa array repreprsentation with some special properties. We all make the assumption that each node will be running identical code like OpenMPI and other frameworks. We will then implement some combinators over these distributed arrays that will allow each node to run locally parallel code on a slice of the array. Repa provides functionality for manifest and delayed representations, the manifestation of a distributed array which will result in the entire distributed array being pulled into memory. This is important as this will allow compatibility with existing functionality.

Once we are satisfied with our implementation of distributed arrays we want to work on a small set of primitiives that allow one to easily express graph algorithms. We will then demonstrate examples of sequential code and parallel code and discuss the speed up.