Final Project Progress Report

Jared Roesch and Tristan Konolige

As we discussed in our original proposal the goal of our project is the implementation of parallel and distributed sparse matrices in Haskell. With the hopeful goal of using them to encode graph algorithms. Our primary contribution will be the engineering effort of required to correctly implement parallel and distributed matrices themselves. We chose Haskell because we wanted to strike a balance between abstraction and performance. Overall our goals have not changed much, but we have dramatically changed our strategy to realize them. We originally thought that Repa, a Haskell library for parallel arrays, would be the core of our project but we have abandoned it after realizing that Repa's design is centered around dense arrays and would make be awkward to shoehorn sparse arrays into Repa. This happens to be the case because Repa is designed around dense arrays and much of its optimizations would not benefit sparse ones without significant changes to the code.

Repa was very interesting to us because of its novel approach to stream fusion. A technique that allows the compiler to fuse loop bodies into single iterations over a structure (in this an array), coupled with their multiple array representations which allow the programmer to specify how new arrays are computed, copied, ect. We fortunately learned a lot from studying Repa's design so the time was not a complete waste. As well general processes described in the authors' papers have helped inform our overall approach to the project.

Having spent a lot of time attempting to implement our project on top of Repa we decided to change gears. We began investigating different sparse representations and looking around in the literature to see what had been tried. We spent a chunk of time investigating space filling curves and we feel that they will play an important role in our eventual distribution of matrices. At the same time we discovered that Edward Kmett an active member of the Haskell community had recently been working on a derivation of sparse matrices as well. He has employed some interesting ideas from space filling curves, and has a nice prototype. We decided to piggy back on his efforts and use them as a base for improving upon. This also enabled us to give back to the OpenSource community as well as finish our project.

Edward's initial prototype of the matrices were missing some keys pieces so we added them. We first spent some time updating the test suites so that we case validate the correctness of our implementation. Next we got a benchmark suite running so that we can accurately chart the performance of our code as we evolve it. The benchmark suite tests multiple configurations of sparse against the BLAS and compares the performance. Our main intention was to have sparse matrices that work well in the sequential, parallel, and distributed setting and Edward's only work sequentially.

Our primary method of whether this is a success will be taking our current benchmark suite and scaling both the input sizes and number of cores. For us a performance win will showing significant speed up in the parallel and distributed setting. In order to show this for a large number of cores and nodes we really felt that we needed Triton to show our scaling. We have been running locally but 8 cores is not enough to accurately discuss our results.

In order to do this we needed to be able to run Haskell code on Triton. The first important piece important piece we nailed down was how to run Haskell on Triton.We did a little bit of investigation and determined that the OS and architecture of the super computer. We built a version of the Glasgow Haskell compiler for it, the set up a system for compiling and deploying code. This allows us to easily build and test code locally then transplant it to Triton for further testing and benchmarks.

Another challenge we faced was how to coordinate between all the nodes participating in the computation. We figured out how MPI does this through the careful coordination of environment variables and other per node system settings, and reverse engineered it for our purposes. We are going to rely on this information to coordinate the distribution of work across the nodes. We are experimenting with using a space filling curve to determine how to split the data. There are some potential challenges here with properly chunking the work since it is not as simple as the dense case where we just even partition it.

Inn terms of milestones related directly to the project we have finished two important pieces that are essential to finishing the project we have established a base representation based on matrices in morton order. We have figured out how to both run code and deploy code on Triton as well as figuring out how to piggy back on MPI's in place information for distribution. We have laid the ground work for the parallel and distributed versions. We have begun implementing parallel operations over the matrices by using a thread gang to have each operation as parallel as possible. We do this by spawning one fixed sized gang at the beginning of every computation and using it for all subsequent computations. This allows for us to get optimal use out of the machine without trashing the cache and incurring context switches for multiple gangs. For the distributed part we have taken some of the expertise we gained implementing Paxos in Haskell the previous quarter and applying to our matrices. We have build a simple system for communicating pieces of a data structure to a distributed computation and will build upon that to write high level distributed versions of our operations that decompose into locally parallel operations, that then decompose into the sequential versions on a per thread basis. The recursive nature of this makes implementing the pieces separately very easy. We also suspect a significant amount of time will be spent tuning our final implementation for optimal performance.