CS240A: Final Project

Jared Roesch and Tristan Konolige

Introduction

In deciding on a topic for our project was hard as we wanted something interesting, and challenging. We wanted there to be sufficient motivation for continuing with it in the future. Given our backgrounds and interest in functional languages, we wanted to do something that brought together ideas from programming languages, optimization, and functional programming in a single project.

As we evaluated different ideas it quickly became obvious that matrix multiplication is a critical piece to solving many problems. Therefor targeting matrices would be incredibly useful enabling one to write algorithms a large set of linear algebra problems, and by extension Linear Algebra problems. This impression was reinforced both from the material we covered in the class, and our previous experiences with scientific computing.

As we discussed in our original proposal the goal of our project was to enable both parallel and distributed matrix operations in Haskell. We chose Haskell because we wanted to strike a balance between abstraction and performance. It is good for this for a few reasons:

- very powerful optimizer
- ability to write low level code
- type safe ptrs
- history of optimization
- mature networking and collection libraries

We also enjoy dragging as much theory as possible back into the “real world” and see if we can apply it solve useful problems. We copied this approach from the author of sparse who attempts to do this asm much as possible with the projects he works on.

We used this as a guide for building the component pieces of our project.

Our primary contribution is the engineering effort to correctly implement distributed matrices themselves in Haskell. The rest of the paper is dedicated to our research in this area, our implementation efforts, and our evaluation of what we finished by the end of the quarter.

Background

Haskell programmers are few and far between so we want to highlight the reasons we choose Haskell and the features that helped us most in our implementation.

Haskell

Haskell the language began as a though experiment of taking a pure, lazy language design and trying to produce something useful from it. Its long history has been documented in A History of Haskell [1], which talks about the various design choices that had been made over the first two decades of its life. The primary authors have also had a strong interest in enabling parallel performance.

Purity

Purity is one of the initially puzzling features of Haskell. It provides some important properties:

```
- referentially transparent
- immutable data structures
- no global state
```

Laziness

Laziness is another important property of Haskell that allows us to express a level of control over evaluation that is not normally available to a programmer.

Monads

Monads are one of the other tools that make Haskell a great tool for writing code quickly and correctly. Monads are a structure from Category Theory but in practice are a useful abstraction (design pattern) for writing functional code. Monads are built upon two other simple primitives Functor, and Applicative. Haskell has a concept of "type classes" which are open interfaces that can be implemented at any time, by any type. Each has a simpe definition that is provided below.

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

class (Applicative m) => Monad m where
    return = pure
```

[1] http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf

```
(>>=) :: m a -> (a -> m b) -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

This declares that for thing that for a Functor f if you provide me a transformation from a f parametrized by a to one parametrized by b. For example IO (the Monad representing input/output effects) forms a Functor: `fmap (+1) [1,2,3] == [2,3,4]`. Type classes allows one to write generic code over these interfaces and reuse it for free. For example there is a function forM that we make great use of that is a generic monadic loop, and for any thing that forms a monad ever will be able to use this piece of code. The next step Applicative is an important structure for expressing computation, and represents contextual computations where the arguments are contextual but do not have sequential contraints. Monads are the most interesting abstraction here and provide a computational context, and sequencing.

Monads also have the added property that they are an type algebraic way to annotate effects. The Haskell `do` notation is a simple sugar over the Monadic operations (>>=), and (>>).

Concurrency

Haskell has great builtin support for concurrency. They provide a scheduler and runtime for lightweight threads. This makes it really easy to manage concurrency, and makes the overhead of creating another thread very low. [2]

Parallelism

Through this experimentation there has been a few interesting outcomes. Simon Marlow a principle researcher and maintainers of Haskell and has spent much of his time making Haskell a viable language for parallel evaluation and parallel computation. There main evolutions has been the light weight threading and scheduling primitives like MVars, Chan, speculative parallelism in the form of the parallel library, and basic primitives for parallelism such as sparks. [3] [4].

Stream Fusion

Quad tree

We took ideas from __ that inspired our quad tree representation, this is something that could be possibly tweaked but it seemed like a good way to tackle the problem initially.

[2] http://www.haskell.org/ghc/docs/7.6.2/html/libraries/base/Control-Concurrent.html
[3] http://chimera.labs.oreilly.com/books/1230000000929/index.html
[4] http://www.haskell.org/haskellwiki/Research_papers/Parallelism_and_concurrency

Related Work

There is quite a bit of related work providing simple APIs for highly performant matrix operations. The prior work on this idea helped inform and guide our research, experimentation and implementation.

COMBINATORIAL_BLAS

We are in the same space COMBINATORIAL_BLAS [5] Ideas from CombBlas, and quad trees.

The Combinatorial BLAS is an extensible distributed-memory parallel graph library offering a small but powerful set of linear algebra primitives specifically targeting graph analytics.

Repa

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 is an interesting because its novel approach to array representation and stream fusion. This technique 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.

Sparse Matrices in Haskell

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

[5] http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/

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.

Dense Matrices in Haskell

For dense matrices there is `hmatrix` a Haskell wrapper around LAPACK, BLAS, and GSL. This seems to be the most mature matrix library in Haskell, and provides correct, efficient operations on dense matrices. [6]

Design

Our eventual design was focused on using quad trees as our distributed representation. We picked it as our mechanism for distributing matrix chunks as it provided easy reasoning about the distribution of a matrix, and allowed for us to write SIMD operations.

We wanted to be able to do SIMD style operations so that it was easy for the eventual programmer to write algorithms without having to worry about the low level details. Part of the way we did this was with a Monad that represented distributed computations. This both allowed us to differentiate local operations from distributed operations, but also hide the plumbing of the distribution from the user.

Implementation

- general approach, tried a few alternatives
- in HS, ,different pieces, primitives provided by HS
- mvars, chans, light-weight threads
-

Our implementation has three key components: - distribution, and serialization - representation - operations

[6] https://github.com/albertoruiz/hmatrix

Distribution

- must solve some basic distributed system problems
- quote papers
- SIMD approach, computations on matrices must be completely distributed 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.

Representation

As we mentioned above we used a quad tree representation for our distributed matrices, but we had to handle the distinction between locally present (i.e concrete) matrices vs. ones that were located on remote machines. Our representation split out our representation into two different types. Concrete ones:

We then used them - quad tree, recursive - concrete vs. distributed - specialized representation and specialized operation - experiment with fusion/delayed representation

Operations

- SIMD approach, computations on matrices must be completely distributed
- multiple, transpose, ect all fall out easily and recursively thanks to repr, and synchronization prims

Evaluation

- graphs!!?!?1

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.

We were able to both get a copy of GHC (Our Haskell compiler) on to Triton. Allowing code to be run and deployed one Triton. We then piggy backed on MPI's infrastructure

Future Work

We built a very simple message passing system for enabling process communication across nodes, but we hope to be able to bring more process synchronization, and communication primitives to it. Currently we treat processes as distinct entities regardless of whether they running locally or remotely. It would be better to treat work allocation as a per machine quantity and use threads locally instead of distinct processes. We began by attempting to implementing parallel operations on matrices by using a thread gang to have each operation as parallel as possible. We took this idea from Repa, and it would be good to be able to bring this parallel power down to the individual matrix level. Initially we thought it was impossible to shoehorn a sparse matrix into repa, we think that it would now be possible, and if not, at least adapting the approaches taken in Repa to our concrete matrix representation, and

Conclusion

When compared to our original milestones we accomplished the important ones. We assembled the essential pieces needed to perform both dense and sparse matrix operations serially. We built on finishing the project we have established a base representation based on matrices in morton order. We were able to both run and compile code on Triton. 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