CS240A: Final Project

Jared Roesch and Tristan Konolige

Introduction

Deciding on a topic for our project was hard as we wanted something that was both 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:

- 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 by being pure. We are now able to draw the distinction between procedures and functions. In Haskell every function is in fact a function (i.e a mapping from domain to codomain) and is only determined from its arguments. This means we have some nice properties like referential transparency meaning we can always substitue a function body for a function invocation while maintain correctness. This means that reasoning about function invocation and flow of control becomes simple as pure functions cannot mutate their input parameters. Purity also means that data structures are immutable by default, as once they are constructed there is no way to modify them. This also means that there is no global state reducing the ability for the programmer to write programs that rely on fragile global initialization.

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. It enables us to easily write algorithms that compose, and evaluate things like the computation of an expensive list in parallel. For example:

```haskell
expensiveComputation = ...

main = do
  expensiveList <- sequence $ replicate 100000 expensiveCompuation
  forM_ (partition 1000 expensiveList) $ forkIO \piece ->
    return $ deepseq piece
```

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

This allows us to build a really computationally expensive list of possibilities and then distribute the work across multiple threads, since the evaluation doesn't happent until deepseq.

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 simple definition that is provided below.

```
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
    (>>=) :: 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, for example fmap allows us to transform an action which produces a string into one that produces an integer.

```
readInt :: IO Int
readInt = fmap read getLine
```

that reads a string into one that reads an integerType 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

Stream fusion is a technique for automatic deforestation or the removal of unnecessary intermediate data structures. This is very effective in Haskell as we can safely rewrite these operations without worrying about correctness. One can accomplish stream fusion by transforming sequences into a stream type and applying functions the streams instead of the original data structure. We then provide GHC a rewrite rule that describes how to rewrite multiple stream operations into a single one. A simple fusion rule is the one for map:

```
map f . map g = map (f . g)
```

This means that we can transform most operations on sequences into a single invocation of stream, and unstream, meaning we only apply the fused operations to each element precisely once. Increasing the constant factors of our code by quite a bit. There are quite a few papers on this, but the most well known is Coutts et al[5] with further improvements from Mainland et al[6].

[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
[5]http://metagraph.org/papers/stream_fusion.pdf
[6]http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf

Quadtree

We spent time reading the literature on using space filling curves for matrix representation, and developed quite a few interesting insights about the problems. Bader and Heinecke[7] proposed using a block-recusive deconstruction of the matrix ordered on peano curves, where each level is divided into nine blocks. We found this to be an elegant construction which would allow us to have a recursive data type with of leaf nodes of varying type.

Each of these leaves could be handled by an external library, so we wouldn't have to reimplement concrete matrix primitives. Furthermore, it allowed us to structure our matrix algorithms recursively which is a natural fit for Haskell. The original purpose of the peano ordering was to increase cache locality. However, since we are distributing our matrix across multiple nodes, it received negligible benefit from the peano ordering.

Once we dropped the peano ordering, it made more sense to use a quadtree because it was simpler to use for our purposes, and so we replaced the three by three representation described in the paper, with the quadtree.

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 [8], which is a distributed-memory parallel graph library that is very much trying to solve the same sorts of problems we were tackling. It provides a set of linear algebra primitives that are specifically useful for targeting graph analytics.

Repa

Repa inspired our project greatly and we thought initially that we would use it as the core of our project. Repa is Haskell library for parallel arrays, would have be the core of our project, but we have abandoned it after realizing that Repa's design was centered around dense arrays and would make be awkward to shoehorn sparse arrays into Repa.

[7]https://para08.idi.ntnu.no/docs/submission_155.pdf
[8]http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/

Repa was 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.

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 community in the process.

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 evolved it. The benchmark suite tests multiple configurations of sparse against BLAS and compares the performance. This of course being the sequential case for matrices.

Dense Matrices in Haskell

For dense matrices there is `hmatrix` a Haskell wrapper around LAPACK, BLAS, and GSL. This seemed to be the most mature matrix library in Haskell, and provides correct, efficient operations on dense matrices. [9]

Design

Our eventual design was focused on using quad trees as our distributed representation. We choose it 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. There are some limitations that we haven't solved such as doing uneven distribution, and dealing with a custom representation for rectangular matrices (for now we can treat them as a sparse square matrix).

This design helped us do SIMD style operations, which was important to us because we wanted the algorithm author be able to use the operations without

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

having to worry about the low level details. The quad tree made this easy because we were able to specify which parts each process had to do computation for without having to deal with specialized representations.

The other tool that helped was the use of a Monad to encapsulate the bookkeeping necessary to deal with interprocess communication, and the book keeping needed for the distributed monad to work. This also allowed for us to differentiate between local computations, distributed computations, and ones that required requests, while all keeping it hidden from the user.

For example our multiplication on matrices simply boiled down to these lines for dealing with remote pieces:

```
dmult' l @ (Remote pid quad) mat = do
    !lmat <- requestMatrix pid L quad
    dmult' lmat mat
dmult' mat r @ (Remote pid quad) = do
    !rmat <- requestMatrix pid R quad
    dmult' mat rmat
```

Implementation

We first considered a few different approaches to implementing our project, from binding to MPI, to using Cloud Haskell, and more. These all seemed like more complicated approaches so we decided to use implement our own framework in order to do message passing.

As we discussed in our related worked we used many of the primitives in Haskell to great effect. In particular we used `MVar`s, `Chan`s and Haskell's light weight thread primitives.

Our implementation can be broke into three key components:

- distribution, and serialization
- representation
- operations

Distribution

We built a small framework for distributed processes that allowed for easy, well ordered communication between processes. This allowed for us to spin up a bunch of proceses with just a few lines of code, and build all of our abstractions on top of this.

This framework only supported the concept of a registry, a set of processes that are registered, and ways to communicate between them. We used these primitives to implement some common distributed systems algorithms for reaching consistent global states. We used these algorithms to ensure that system synced up at the operation boundaries in order to maintain the correctness of our operations, without a system is vulnerable to race conditions that would result incorrect answers.

Representation

As we discussed in our related work and design, and related work we decided upon using a quad tree representation in our distributed matrices.

In the process of implementation we decided that we wanted to enforce a stratification between concrete and distributed matrices so we split our original data type definition into two separate types. One for concrete and one for distributed.

This actually turned out to be a big win, as we were able to rewrite our operations on concrete matrices to be total functions, and specialize then just for the concrete (Zero, Sparse, Dense cases) meaning we didn't have to have failure conditions for something that shouldn't fail like matrix multiplication.

We then were able to write simpler algorithms for the distributed cases.

Our concrete representation:

```
data CMat a = Dense (D.Matrix a)
            | Sparse (S.Mat a)
            | Zero
            deriving (Show, Generic)
```

The distributed case:

```
data DMat a = DMat !Int !(DMat a) !(DMat a) !(DMat a) !(DMat a)
            | Remote !DT.PID ![Q]
            | Concrete !(CMat a)
            deriving (Show, Generic)
```

We also made the simplifying assumption that all of our leaves are the same size meaning we didn't have to deal with odd multiplication cases. We can build recursive algorithms that were easy to reason about.

We implemented some very simply fusion for the concrete multiplication cases allowing us to optimize that case, but we did not end up having as much time to experiment with distributed fusion.

Operations

The foundation we used for implementing distributed computations was `sync` which allowed us to synchronize computation around a piece of data and require the distributed system to reach a consistent global state before continuing. `sync` also require our computation to be expressed in a Requests monad which makes it explicit that the computation
needs outside information.

Since we took a SIMD approach the operations provided by our library are very easy to use and work much like normal operations, except they live in our computational context that represents distributed computations. This was nice because some operations like transpose did not need to be distributed and we were able to reflect that in our type signature like so:

```
transpose :: (S.Arrayed a) => DMat a -> DMat a
transpose (Concrete (Dense smat)) = Concrete $ Dense $ D.trans smat
transpose (Concrete (Sparse smat)) = Concrete $ Sparse $ S.transpose smat
transpose (Concrete Zero) = Concrete Zero
transpose (Remote pid quad) =
    case quad of
      [B] -> Remote pid [C]
      [C] -> Remote pid [B]
      a -> Remote pid a
transpose (DMat mask tl tr bl br) =
    DMat (mask' mask) (transpose tl) (transpose bl)
                      (transpose tr) (transpose br)
  where
    mask' m = (m .&. 1) + (m .&. 8) +
              (if thirdQ m then 1 else 0)*2 +
              (if secondQ m then 1 else 0)*4
```

If you look carefully you will see that we can do this transpose on any process without needing to communicate.

Implementing new operations is quite easy, especially if they are composed of our primitive ones. For example our implementation of `ddot` for conjugate gradient was very simple:

```
ddot :: DMat Double -> DMat Double -> Distribute (DMatMessage Double) Double
ddot x y = (transpose x) .* y >>= topleft
```

Evaluation

We were able to both get a copy of GHC (Our Haskell compiler) on to Triton. Allowing code to be easily run and deployed one Triton, with little extra effort.

We also piggy backed on MPI's existing infrastructure for communication to allow us to run on more than Triton node for benchmarking.

Our method of benchmarking our implementation was very simple: did it scale?

We got our system running on Triton and were able to benchmark it, but the results showed that we still needed to do more performance tuning in order to get it truly competitive. We also encountered a bizarre bug in which the program seemed to transform from running fine to hanging after bumping up the input size. Resultantly we couldn't get running on matrices pasts sizes of 40,000 as the program hung and continued running many minutes past the time it took to do 40,000.

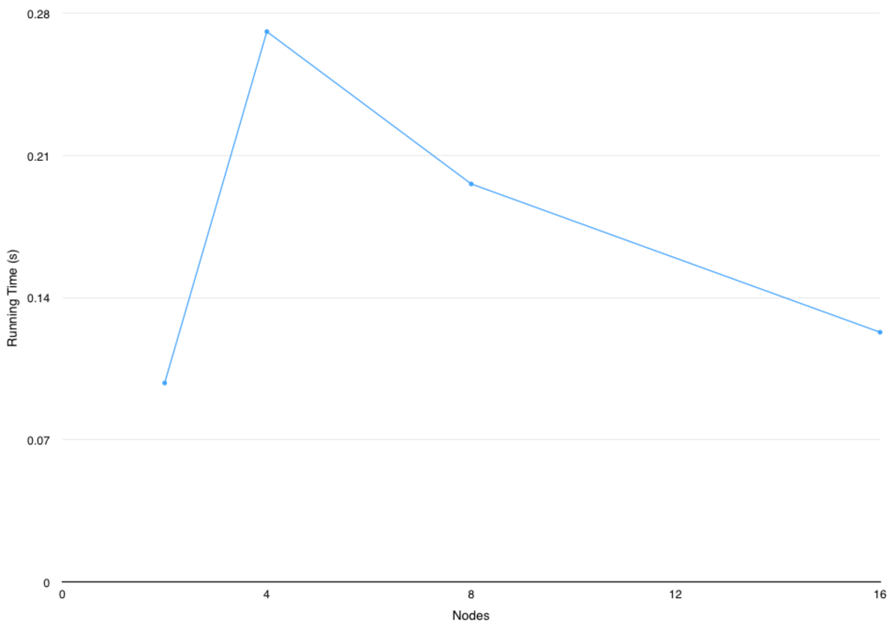

Figure 1: Scaling Performance

Included is a graph of performance on relatively small sized matrices (20,000 non-zeros). Performance is optimal at two nodes, most likely because we have high communication overhead. However, from 4 nodes to 16 nodes, our matrix multiplication scales relatively well.

Overall we feel that our subpar performance was due to a couple possible problem areas:

- serialization
- redundant data fetches

- too many messages
- compiler annotations, and tricks

For serialization we quickly enabled serialization for all of the required data types, but we didn't invest significant time into optimizing this, and it may of cost us extra precious time on top of our other inefficiencies.

We did not have time to implement a batched request system and so we were probably unnecessarily fetching data more than once meaning wasted time in communication, and serialization.

We used a simple algorithm for synchronizing processes, and we probably could of used a more sophisticated scheme such as those used in Dynamo or Riak.

We used the simple annotations, and tricks for performance in Haskell, but it is likely there are more we could apply to our code.

We were disappointed that we couldn't see our matrices scale to bigger input size because our only real result was the communication overhead was too high and ended up slowing us down much more than expected.

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 looking at our original milestones we accomplished the important ones. We assembled the essential pieces needed to experiment, and optimize a framework for doing matrix operations. We implemented routines for working with both dense and sparse matrices serially. We built on an existing open source library and improved a the implementation of the sparse matrices. For distribution we have taken the expertise we gained implementing Paxos in Haskell and applied it our implementation efforts this quarter. We have built a simple system for communicating pieces of a data structure to each process, but there is huge room for improvement in optimizing the overhead of message passing. We are pleased at how our representation decomposes recursively as it allowed us to implement the pieces separately very easy. We were able to both run and compile code on Triton, and demonstrate that we were at least able to achieve some of our initial performance goals.