Project Proposal for CPSC 521: SMOLDD - Sparse Matrix Operation Library for parallel Data Distribution

Cody R. Brown[1][2]

ABSTRACT

SMOLDD is a Sparse Matrix Operation Library implemented with MPI. The module is programmed in Python and uses mpi4py as its message passing backbone. SMOLDD will be useful for problems where large sparse matrices need to be efficiently distributed over a distributed memory cluster. This module will be optimized for use with curvelet values which tended to have specific sparse mappings, and specially designed for serial algorithms with parallelizing sparse matrix operations. Such applications are found in abundance within the seismic processing community where data is typically on the order of terabytes and the sparse optimization community where a large number of sparse matrix-vector operations are performed in iterative methods. Two applications are proposed to test and analyse the module.

MOTIVATION

In the seismic image processing community data is generated through a series of shots over a surface. These shots are pumped into the ground with many receivers measuring the response over time, then repeated over and over. Later these shot gathers are collected and then migrated into an image rendering the sub-surface material. The data generated in this form is often in high dimensions and on the order of terabytes; efficient computation of this data must be performed on a large scale cluster. Furthermore these shot gathers are usally imperfect and plagued with many unwanted artifacts such as missing traces, multiples, low frequency earth background waves, ground roll and noise to name a few.

A common practice is to apply a transformation of the data into another domain to reconstruct or improve image quality. The Seismic Laboratory for Imaging and Modeling (SLIM) (4) at the University of British Columbia (UBC) looks into the use of curvelets for seismic pre-processing. The use of curvelets with seismic imaging is the greatest motivation for the proposed Sparse Matrix Operation Library for parallel Data Distribution (SMOLDD). Curvelets are known to be extremely compressible (decay rapidly) on seismic data (8), however they also add a redundancy factor of around 8 in 2D and up to 24 in 3D (8). With transforming large amounts of seismic data into the curvelet domain, there becomes a pressing need for a sparse matrix library that will be able to distributed the sparse data evenly over the processors and apply common linear algebra operations on these matrices. This becomes a very challenging operation to do namely because of the randomness

[1]email: cbrown@eos.ubc.ca

[2]Seismic Laboratory for Imaging and Modeling,Department of Earth and Ocean Sciences, University of British Columbia, 6339 Stores Road, Vancouver, V6T 1Z4, BC, Canada

of the data populations in sparse matrices; especially with curvelets which tend to have large chunks of zeros as shown in Figure 1. These regions could potentially be mapped to many processors leaving a very unbalanced workload.

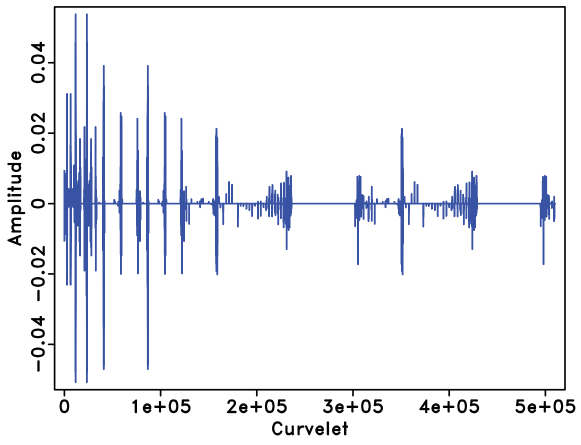

Figure 1: Sparse curvelet vector of a (256,256) image with 32 angles. $\boxed{\text{./curve curve}}$

One further application in the seismic world is to invert these large matrices using iterative solvers. These solvers often take a long time to converge and the sheer number of matrix multiplications involved becomes unrealistic. Improvements in speed for these sparse matrix operations are desirable.

SMOLDD will run on a general purpose distributed memory cluster, such as LIMA (2) a 6xIBM x3550 (288 CPU) Intel cluster with infiniband inter-processor network, located at UBC in the ICICS building.

IMPLEMENTATION AND FOCUS

The main focus of SMOLDD will be data distribution, load balancing, scalability and ease-of-use. As mentioned above the physical distribution of sparse data is a tricky process since sparse data is often hard to map effectively over a wide area of processors. Furthermore with simple operations certain parts of this data need to communicate more than others. SMOLDD will look at techniques proposed in (6; 11) and more recently (10) to tackle this problem. Utilizing these techniques will hopefully achieve our data distributed and scalability related goals for SMOLDD.

SMOLDD will be programmed as a Python module. The main purpose behind this is the fact that SPGL1 (5) a tool which will attempt to be optimized through SMOLDD is only released in Matlab. More about SPGL1 will be discussed later, but for now this algorithm has been ported to Python by Sean Ross-Ross and modified by myself at the SLIM laboratory. Since this is what SPGL1 is available in, to optimize the sparse matrix calls, we will need a Python module. Python is quickly becoming the choice for scientific programming in academia because of the free licensing and abundance of free tools available. Python also has a number of parallel tools such as Star-P (1) which seamlessly integrates HPC operations in the background of Python scripts, allowing users to benefit from improved speed and memory with little to no code changes. Python also has several

MPI implementations available, the one which SMOLDD will use will be mpi4py (3) which looks very similar to MPICH code, and uses NumPy data structures for data organization. With such tools it seems clear Python has everything that will be required to effectively implement SMOLDD. Python is also well known for its simple and ease-of-use syntax. This will help us with the SMOLDD module which will primarily be used with serial algorithms wanting to perform parallel operations. This design method will be integrated deeply into SMOLDD and will be discussed a little further.

More details SMOLDD will consider is the sparse matrix representation that it will implemented. Three popular serial choices are Compressed Sparse Column, Compressed Space Row and List of List formats. In a parallel setting many more representations are discussed from quadtrees to specialized data structures. A format that will allow easy distribution and minimal communication will be ideal. Because of the nature of the problem, a static task assignment for the processors will be sufficient, but adaptive tasks (10) for the changes in the sparsity of the matrix will be exploited. This will hopefully dynamically load balance the problem during execution, but may increase communication costs.

One concern that will also be looked at further in the project will be the time to distribute and gather these large sparse matrices. Techniques for keeping the data distributed throughout the processors but still apply concurrent operations will hopefully be looked at. This will greatly affect performance, start-up and wind-down time. The benefits of a decentralized structure for the module are being considered, but a need for a central structure seems to be necessary due to our desire for a simple easy-of-use design goal. This is because to achieve this goal a need to keep the serial algorithm running only on one processor may be necessary, which will then distribute parallelized matrix operations through the library.

FUNCTIONALITY

SMOLDD being a Sparse Matrix Operation library will need to perform the basic linear algebra operations necessary for serial algorithms which would like to perform parallel sparse matrix operations. Table 1 shows a proposed list of initial functions that SMOLDD will support. These tasks will focus particularly on distributing these sparse matrices over an array of processors. More thought will need to go into the background design, but the hope that much of the serial algorithm we want to optimize will remain the same and just a few of the expensive tasks in the algorithm will need to be replaced.

```
% Initial set of proposed functions for the Sparse Matrix Operation Module %

smoldd_init     - Initialize the MPI information for the module.
smoldd_fin      - Finalize the module.

smoldd_dist     - Distribute the sparse data (perhaps internal).
smoldd_gather   - Gather the sparse data (perhaps internal).

mpi_matvecmult  - A sparse matrix-vector multiplication.
mpi_matmatmult  - A sparse matrix-matrix multiplication, easily implement as a number of matrix-vector tasks.
mpi_l2norm      - Calculate the L2-norm of a sparse matrix, used in sparse optimization problems.
mpi_conj        - Applying the conjugate.
mpi_trans       - Applying the transpose of a sparse matrix.

mpi_threshold   - Thresholding a sparse matrix, useful in denoising applications.

Perhaps look into:
mpi_invert      - Inverting a sparse matrix.
```

Table 1: Initial list of routines for the SMOLDD module.

APPLICATIONS

Two applications will be used to test and analyse SMOLDD. Both will be used to compare different accepts of the SMOLDD module. Two different sets of sparse matrices will be used in the testing including curvelet sparse data as well as sparse matrices from the Sparse Matrix Collection (7) at the University of Florida.

The Power Method

The first simple application will be the implementation of the power method. The power method is a very trivial iterative application that locates the most dominant eingenvalue of a matrix. Since it can only find the most dominate, it is not used extensively. One primary application the power method is used within seismic imaging is determining the normalization constant for a large matrix that will be used in an iterative scheme. Certain matrices will become unstable without this step.

To see the importance of the power method, we can look at a more popular application that uses it: Googles' PageRank. Within PageRank the power method is used to find the largest eignvalue of an extremely large yet very sparse matrix. This is a very important but very slow process that is further described in (9). As a side note Google is able to exploit the sparsity of the problem to perform this cheaply. In Google's case, because of the large data used and the extreme sparse nature of this data, SMOLDD will be able to distribute the data evenly and quickly solve these power method problems. The sample code for a serial power method algorithm is shown in Table 2. One goal with SMOLDD is the hope that most of the serial algorithm will remain intact and only the initialization and matrix calls will have to be replaced. This design goal will be tested more thoroughly in the next example, with this example mainly used to implement SMOLDD and see the benefits of parallelizing the matrix operations within a very slowly converging algorithm.

SPGL1 in Python

The second application will be the optimization of SPGL1 (5) which is developed at UBC. Due to the complex nature of the SPGL1 code a hope to fully parallelize the algorithm will be dismal. However using SMOLDD a hope to parallelize the expensive operations of the algorithm should show a large benefit in overall performance, especially in relation to large sparse matrices. SPGL1 is perfect for using our optimization since all the expensive matrix operations are timed and have been extracted into one function shown in Table 3. Optimized and parallelizing this function will be the task for our second application.

REFERENCES

[1] Star-P: Interactive Supercomputing, 2004. `http://www.interactivesupercomputing.com/products/`.

[2] LIMA: Laboratory for Imaging and MAthematics HPC Cluster, October 2008. `http://www.lima.icics.ubc.ca/`.

[3] MPI for Python, September 2008. `http://mpi4py.scipy.org/`.

[4] SLIM: Seismic Laboratory for Imaging and Modeling, October 2008. `http://slim.eos.ubc.ca/`.

[5] E. v. Berg and M. P. Friedlander. SPGL1: A solver for large-scale sparse reconstruction, June 2007. `http://www.cs.ubc.ca/labs/scl/spgl1/`.

[6] T.-R. Chuang, R.-G. Chang, and J. K. Lee. Sampling and analytical techniques for data distribution of parallel sparse computation. In *In 8th SIAM Conference on Parallel Processing for Scientific Computing*, page 8, Minneapolis, Minnesota, USA, 1997. SIAM.

[7] T. Davis. The university of florida sparse matrix collection, September 2008. `http://www.cis.ufl.edu/research/sparse/matrices/`.

[8] F. Herrmann, D. Wang, G. Hennenfent, and P. Moghaddam. Curvelet-based seismic data processing: a multiscale and nonlinear approach. *Geophysics*, 73(1):A1–A5, 2007.

[9] A. N. Langville and C. D. Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.

[10] S. Lee and R. Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 195–204, New York, NY, USA, 2008. ACM.

[11] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

```
#!/usr/bin/env python
"""
  power_meth.py
  =======
  (c) Cody Brown, 2008
  Email: cbrown@eos.ubc.ca

DESCRIPTION:
    Power Method Implemention
        This is a simple sparse power method implementation
        to test a Sparse MPI Matrix Operation Library.

REQUIREMENTS:
    MADAGASCAR - Seismic processing package, used for external data (Sparse Matrix)
        (http://reproducibility.org/)
    NumPy - scientific module
        (http://numpy.scipy.org/)
    SciPy - scientific module
        (http://www.scipy.org/)
    mpi4py - MPI implmentation for Python
        (http://mpi4py.scipy.org/)
    SMOLDD - Sparse Matrix Operation Library for Data Distribution
        (CPSC 521 Project by Cody Brown)
"""
usage = "\npower_meth.py [options]\n"

import os
import rsf as sf

import numpy as np
from time import time

#My Sparse Matrix Operation Library (Python Module) and others which will be needed.
from smoldd import smoldd_init smoldd_fin mpi_matvecmult,mpi_l2norm,mpi_conj,mpi_trans
from scipy import sparse
import mpi4py as mpi

def main(A=None,iter=50):
    # Create some noisy data to start our initial guess.
    x0=np.random.randn(A.shape[0],1)

    tStart=time()

    #Start of generic power method
    for it in range(iter):
        x00 = np.dot(K,x0)
        x0 = x00/np.linalg.norm(x00)

        x0T = x0.conj().transpose()

        mu = np.dot(x0T,x00)/np.dot(x0T,x0)
        print 'Iter: %d   Time: %f   Largest Eigenvalue: %f\n' %(it,np.abs(mu),time()-tStart)

if __name__ == '__main__':
    # Import the Sparse Matrix A from rsf
    par = sf.Par()

    iter = par.int("iter",50)
    input  = sf.Input(par.string("A"))

    A = np.zeros((input.int("n2"),input.int("n1")),'f')
    input.read(A)

    main(A,iter)
```

Table 2: Generic serial Power Method written in Python.

```
        print " %-20s:  %6i %6s %-20s:  %6.1f" % \
              ("Newton iterations", nNewton, "", "Mat-vec time (secs)", info['timeMatProd'])
        print " %-20s:  %6i" % \
              ("Line search its", nLineTot)
        print ""

    return (x,r,g,info)


def Aprod(x,mode):
    global __A
    global nProdA
    global nProdAt
    global timeMatProd

    tStart = time()

    if mode == 1:
        z = numpy.dot(__A,x)
        nProdA = nProdA + 1
    elif mode == 2:
        z = numpy.dot(__A.T,x)
        nProdAt = nProdAt + 1
    else:
        raise Exception('Wrong multiplication mode!');

    timeMatProd = timeMatProd + time() - tStart

    return z


def project(x,lam):
    global timeProject

    tStart = time()
    x = numpy.mat(oneProjector(x, lam))
    timeProject = timeProject + time() - tStart
```

Table 3: Aprod function from the SPGL1 algorithm.