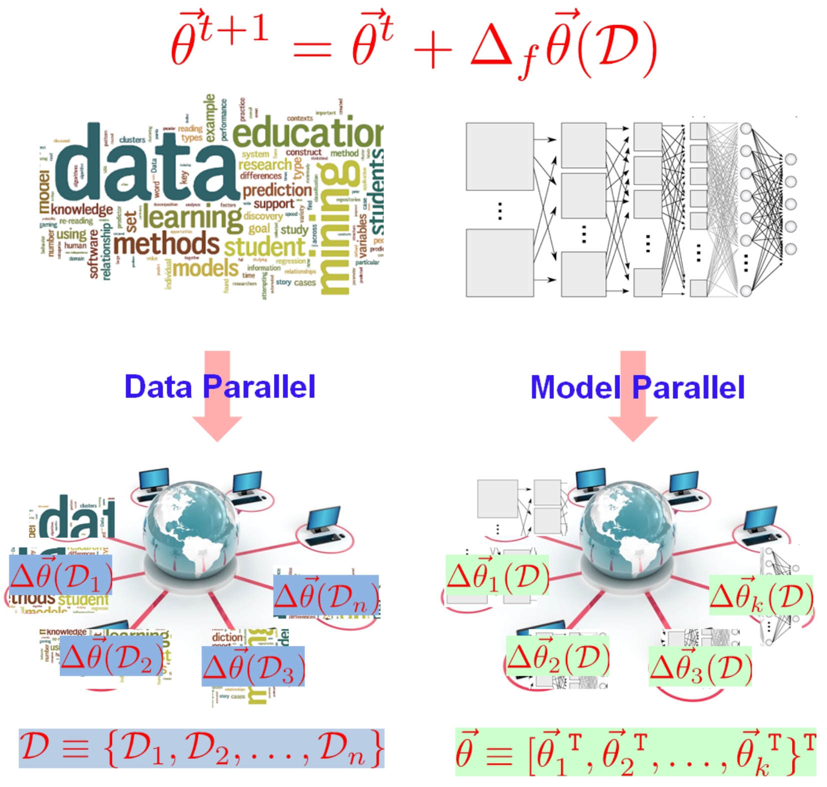
Petuum user manual

Petuum is a distributed machine learning framework. It takes care of the difficult system "plumbing work", allowing you to focus on the ML. Petuum runs efficiently at scale on research clusters and cloud compute like Amazon EC2 and Google GCE.

Petuum provides essential distributed programming tools to tackle the challenges of running ML at scale: **Big Data** (many data samples) and **Big Models** (very large parameter and intermediate variable spaces). Petuum addresses these challenges with a **distributed parameter server** (key-value storage), **a distributed model scheduler**, and **out-of-core (disk) storage**. Unlike general-purpose distributed programming platforms, Petuum is designed specifically for ML algorithms. This means that Petuum takes advantage of data correlation, staleness, and other statistical properties to maximize the performance for ML algorithms.

Petuum is designed around the notions of **data-parallelism** and **model-parallelism** in Machine Learning, as illustrated in the following diagram:


$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$
Data Parallel
Model Parallel
$\Delta\vec{\theta}(\mathcal{D}_1)$
$\Delta\vec{\theta}(\mathcal{D}_2)$
$\Delta\vec{\theta}(\mathcal{D}_3)$
$\Delta\vec{\theta}(\mathcal{D}_n)$
$\Delta\vec{\theta}_1(\mathcal{D})$
$\Delta\vec{\theta}_2(\mathcal{D})$
$\Delta\vec{\theta}_3(\mathcal{D})$
$\Delta\vec{\theta}_k(\mathcal{D})$
$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$
$\vec{\theta} \equiv [\vec{\theta}_1^{\mathrm{T}}, \vec{\theta}_2^{\mathrm{T}}, \ldots, \vec{\theta}_k^{\mathrm{T}}\}^{\mathrm{T}}$

ML programs are built around update functions that are iterated repeatedly until convergence. An update function updates the model parameters and/or latent model states θ by some function $\boldsymbol{\Delta}\theta(\mathcal{D})$ of the data $\mathcal{D}$. Data parallelism divides the data $\mathcal{D}$ among different workers, whereas model parallelism divides the parameters (and/or latent states) θ among different workers. Both styles of parallelism can be found in modern ML algorithms: for example, Matrix Factorization via Stochastic Gradient Descent is a data-parallel algorithm, while LASSO regression via

Coordinate Descent is a model-parallel algorithm. The Petuum parameter server and task scheduler are built to enable data-parallel and model-parallel styles, respectively.

In addition to distributed ML programming tools, Petuum comes with a library of distributed ML algorithms that are used as demo, and at the same time, can be used at massive scale for real Big Data analytics. All programs are implemented on top of the Petuum framework for speed and scalability. Currently, the library includes (and will be steadily enriched):

- Latent Dirichlet Allocation (a.k.a Topic Modeling)
- Least-squares Matrix Factorization (a.k.a Collaborative Filtering)
- LASSO regression
- Deep Neural Network

Petuum comes from "perpetuum mobile," which is a musical style characterized by a continuous steady stream of notes. Paganini's Moto Perpetuo is an excellent example. It is our goal to build a system that runs efficiently and reliably -- *in perpetual motion*.

Installing Petuum

Which Linux distro to use?

Petuum has been tested on Ubuntu 12.04 LTS 64-bit, available at: http://www.ubuntu.com/download/desktop. The instructions on this guide assume a fresh installation.

Petuum should also work on RedHat Linux, but the instructions for installing dependencies will not be the same.

Obtaining Petuum

You can download Petuum as a zip file from GitHub: navigate to http://github.com/sailinglab/petuum, and click on the "Download Zip" link on the right.

Alternatively, you can install git and clone the Petuum repository from GitHub. You will need to set up a GitHub account and upload a public SSH key. Once done, enter the following commands:

```
sudo apt-get install git
git clone git@github.com:sailinglab/petuum.git
```

Software dependencies

Petuum requires the following software:

- g++ 4.8 or later
- autoconf 2.68 or later
- libuuid
- libtool
- openssh server
- libopenmpi-dev (only necessary for STRADS model scheduler)
- libssl-dev (only necessary for STRADS model scheduler)

Commands for installing these software packages on Ubuntu 12.04 can be found below:

Installing gcc and g++ 4.8

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-4.8 g++-4.8
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 50
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 50
```

Installing additional dependencies

```
sudo apt-get install autoconf libtool uuid-dev openssh-server
sudo apt-get install libopenmpi-dev libssl-dev
```

Compiling Petuum

You're now ready to compile Petuum. This requires a working internet connection, as our make system will download additional dependencies. In the Petuum root directory, just type

```
make
```

This process will take between 5-30 minutes, depending on your machine and the speed of your internet connection. To use the STRADS model scheduler, you will also need to

```
cd src/strads/
make
```

Again, this will take around 5-30 minutes to compile. In a later section, we'll explain how to compile and run Petuum's built-in apps, as well as how to write your own applications.

Installing Petuum on multiple machines

If you just want to use Petuum on one machine, you may skip this section.

Petuum is designed to be used in clusters with shared network directories that are visible to all machines. If your cluster does not have a shared directory, do not worry - Petuum will still work, but you'll need to take extra steps.

In either case, all machines must be identically configured: they must be running the same Linux distro (with the same machine architecture, e.g. x86_64), and the dependencies described above must be installed on every machine. The machines need not have exactly identical hardware, but the Linux software environment must be the same.

If you have a shared network directory

You need to copy or git clone Petuum into your shared network directory, and compile Petuum and the Petuum apps from one machine. All machines must mount the shared network directory at exactly the same path.

When running Petuum apps, you need to place the server configuration files and any input data files into your shared network directory, so that they are visible to all machines at exactly the same path.

If you don't have a shared network directory

You need to copy or git clone Petuum onto every machine, at exactly the same path (e.g. /home/username/petuum). All machines must compile Petuum and the Petuum apps separately.

When running Petuum apps, the server configuration files and input data files must be present on every machine, at exactly the same path.

Ports that must be open

If you have a firewall, you should open the following ports.
SSH port: 22
Petuum apps: 9999 and 10000 (you can change these)

Setting up SSH

Petuum uses SSH to start jobs. You will need an SSH key, and you will need to setup password-less key-based authentication on each machine.

If you don’t have an SSH key

Create a public-private keypair on your work machine:

```
ssh-keygen
```

Adding your SSH public key to each machine

You’ll need to append the contents of your public key file, `~/.ssh/id_rsa.pub,` to the `~/.ssh/authorized keys` file on each machine. For example, you can upload your public key to `~/.ssh/id_rsa.pub` on each machine, and then run:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Before proceeding, you should SSH into each machine to test that password-less key-based authentication is working. Petuum will not function properly if passwords need to be entered.

Cloud compute support

Petuum should run in any Linux-based cloud environment that supports SSH. We recommend using Ubuntu 12.04 LTS 64-bit.

Amazon EC2

If you wish to run Petuum on Amazon EC2, we recommend using the official Ubuntu 12.04 LTS 64-bit Amazon Machine Images provided by Canonical:
http://cloud-images.ubuntu.com/releases/precise/release/

Petuum Apps

What apps does Petuum come with?

Petuum comes with several ready-to run applications:

- **Latent Dirichlet Allocation**, a.k.a Topic Modeling
- **Least-squares Matrix Factorization**, a.k.a Collaborative Filtering
- **LASSO Regression,** Parameter Server version (for Big Data problems with many samples)
- **LASSO Regression,** STRADS version (for Big Model, high-dimensional problems)
- **Deep Neural Network**

Instructions on how to compile and run each application follow. First, we describe how to generate server configuration files for each app.

Creating Parameter Server configuration files

Most Petuum apps (except for LASSO STRADS version) use a common Parameter Server configuration file format:

```
0 ip_address_0 10000
1 ip_address_0 9999
1000 ip_address_1 9999
2000 ip_address_2 9999
3000 ip_address_3 9999
...
```

The placeholders `ip_address_0`, `ip_address_1`, etc. are the IP addresses of each machine you want to use. If you only know the hostname of the machine, for example `work-machine-1.mycluster.com`, use ‘host’ to get the actual IP:

```
host work-machine-1.mycluster.com
```

Each line in the server configuration file format specifies an ID (0, 1, 1000, 2000, etc.), the IP address of the machine assigned to that ID, and a port number (9999 or 10000). Every machine is assigned to one ID and one port, except for the first machine, which is assigned two IDs and two ports because it has a special role.

What if I want to run those apps on one machine?

Simply use the server configuration file at `machinefiles/localserver`, which looks like this:

```
0 127.0.0.1 9999
1 127.0.0.1 10000
```

A word of caution - please read this!

The Petuum apps that use the Parameter Server cannot be run on the same machines, unless you use server configuration files with different ports.

If you ever find that you cannot run an application - especially if you see error messages with a “Check failure stack trace” - the cause is likely another running (or hung) Petuum app that is using the same ports. You can forcibly kill other Petuum applications by using the scripts

```
scripts/kill_*.sh
```

For example, to kill all instances of Matrix Factorization, use

```
scripts/kill_matrixfact.sh <petuum_ps_hostfile>
```

Note that you cannot crtl-c to terminate a Parameter Server application, because they are invoked using background SSH commands. If you wish to terminate a running Parameter Server application, you must use its kill script.

Latent Dirichlet Allocation

Topic model, a.k.a Latent Dirichlet Allocation (LDA), automatically find the topics, defined as weighted set of vocabularies, and the topics of each document from a set of documents. Here we are solving LDA using Gibbs sampling. The common practice in scaling up Gibbs sampling is to partition the documents onto different machines, and make word-topic counts globally accessible to all workers. Note that in this scheme the document-topic counts are local. Here we instead we store the document-vector in the parameter server as global parameters and store word-topic counts as local statistics. We call this scheme "word sampler", to distinguish it from the "document sampler" where documents are the unit of partition. Our implementation uses a fast sampling procedure similar to Yao et al (2009).

Processing Data

We provide the 20-news-groups dataset as a demo. They are stored in

```
datasets/lda_data/20news.dat
datasets/lda_data/20news.vocab
```

Each line in the .dat file is a document in LibSVM format:

```
label word_id:count word_id:count ...
```

We ignore the first column labeled category (first column). The .vocab file is the original words (line i in 20news.vocab has word index i). In this example we will run LDA on two machines. We first process it into a format suitable for word sampler:

```
# Under Petuum directory, compile and run preprocessor to make 2 partitions
cd apps/tools
make
mkdir datasets/processed
sh scripts/run_lda_data_processor.sh datasets/lda_data/20news.dat
datasets/processed/20news 2
```

This produces the following files:

```
datasets/processed/20news.0
datasets/processed/20news.1
datasets/processed/20news.map
```

20news.0 and 20news.1 are two partitions of the original documents. Furthermore, the first line of each is the number of documents in corpus (thus they are the same). Each line that follows has the following format:

```
word_id doc_id:count doc_id:count ...
```

So “3 5:4 2:3” means vocabulary 3 occurs 4 times in document 5 and 3 times in document 2. 20news.map maps the vocabulary indices in original .dat (column 1) to indices in the output (column 2). Now we are ready to run LDA.

Compiling
From the Petuum directory, run the following commands to compile the sampler:

```
cd apps/lda_word_sampler/
make
```

This will put the the sampler binary in `apps/lda_word_sampler/bin/`. Make sure you create a server file, say, `machinefiles/servers` which has two machines. For example:

```
# content of machinefiles/servers
0 10.1.1.245 10000
1 10.1.1.245 9999
1000 10.1.1.246 9999
```

Note that for LDA to work properly, each machine’s ID needs to be 1000 apart (except the 0 and 1). You can test that the app works on two machines (with 4 worker threads) for 100 topics over 10 iterations with 0 SSP staleness using:

```
# run experiment with 4 worker threads on each machine
sh scripts/run_lda_word_sampler.sh datasets/processed/20news
datasets/processed/20news.map 100 10 machinefiles/servers 4 dump 0
```

`dump` is the prefix to output files (the word-topic and doc-topic tables). If all goes well, you should something like (with prefix removed):

```
client_1) Initializing topics..
client_1) Start Collapsed Gibbs Sampling
client_0) Start Collapsed Gibbs Sampling
client_0) Iter: 1 likelihood_part_1 -22810.9
client_1) Iter: 1 likelihood_part_1 -40336.7
client_1) Iter: 1      Took: 3.84615 sec       Throughput: 42883.7
token/(thread*sec)
client_0) Iter: 1 likelihood_part_2 -1.74982e+07
client_0) Iter: 1      Took: 4.33541 sec       Throughput: 37975
token/(thread*sec)
...
client_0) Writing output to /home/wdai/petuum-ps/dump.word_topic.0 ...
```

```
client_0) Writing output to /home/wdai/petuum-ps/dump.doc_topic ...
Done! Shutting down.
```

Output format

The doc-topic table has the following format:

```
doc_id topic_id:count topic_id:count ...
```

The word-topic table has the following format:

```
word_id topic_id:count topic_id:count ...
```

Note that the word topic table is divided in M files, where M is the number of partitions/machines.

Terminating a hung app

If the LDA should hang (or you just want to stop it prematurely), you can kill it by running

```
scripts/kill_lda_word_sampler.sh <path_to_serverfile>
```

Otherwise, the LDA will terminate naturally when it is done.

Least-squares Matrix Factorization

Given an input matrix X (with some missing entries), the Matrix Factorization app (MF for short) learns two matrices L and R such that L*R is approximately equal to X (except where an element of X is missing).

If X is *N-by-M*, then L will be *N-by-K* and R will be *K-by-M*. Here, K is a user-supplied parameter (the “rank”) that controls the accuracy of the factorization. Higher values of K usually result in a more accurate factorization, but at the cost of additional computation.

MF is commonly used to perform Collaborative Filtering, where X represents the known relationships between two categories of things - for example, X(i,j) = v might mean that “person i gave product j rating v”. Because we might not know all possible relationships in X, we can use MF to predict the unknown relationships. Specifically, we can predict any missing entry X(i,j), using the learnt matrices L and R:

X(i,j) = L(i,1)*R(1,i) + L(i,2)*R(2,i) + … + L(i,K)*R(K,i)

Compiling

From the Petuum directory, run the following commands:

```
cd apps/matrixfact/
make
```

This will put the MF binary in `apps/matrixfact/bin/`. You can test that the app works on one machine (with 2 worker threads) using:

```
scripts/run_matrixfact.sh datasets/matrixfact_data/9x9_3blocks 3 100 mf_output
machinefiles/localserver 2 0
```

If the program completes successfully, you will see 2 output files:

```
mf_output.L
mf_output.R
```

In both files, you should see that lines 1-3 are similar, lines 4-6 are similar, and lines 7-9 are similar. For example, `mf_output.L` might look something like this:

```
0.308813 0.963073 -0.555122
0.308813 0.963073 -0.555122
0.308813 0.963073 -0.555122
1.29682 -0.362354 0.602023
1.29682 -0.362354 0.602023
1.29682 -0.362354 0.602023
```

```
0.50998 -0.943248 -1.35629
0.50998 -0.943248 -1.35629
0.50998 -0.943248 -1.35629
```

You won't see the exact same numeric values in different runs (that's expected behavior, since there's more than one correct answer). All you need to confirm is that each group of 3 lines is similar.

Input data format

The MF app takes a sparse matrix as input:

```
row_0 col_0 value_0
row_1 col_1 value_1
row_2 col_2 value_2
...
```

This says that the matrix has elements (row_0,col_0) = value_0, (row_1,col_1) = value_1, and so on. If a matrix element is not explicitly specified, it is treated as a missing value - this is NOT the same as a zero value. In Collaborative Filtering, missing values are those that you want to predict, e.g. "How much will user x like product y?"

Running the app

The syntax for running the MF app is:

```
scripts/run_matrixfact.sh <datafile> <K> <iters> <output_prefix>
<petuum_ps_hostfile> <client_worker_threads> <staleness>
```

The arguments are as follows:

`<datafile>`
Path to input data file.

`<K>`
Factorization rank.

`<iters>`
Number of iterations to run: one iteration = touch every matrix element once.

`<output_prefix>`
The factor matrices L and R will be written to `<output_prefix>.L`, `<output_prefix>.R`.

`<petuum_ps_hostfile>`
Petuum server configuration file.

`<client_worker_threads>`
How many worker threads to use on each machine.

`<staleness>`
Set this to 0 unless you want to use the SSP consistency model; see the Parameter Server tutorial for details.

Terminating a hung app

If the MF app should hang (or you just want to stop it prematurely), you can kill it by running

```
scripts/kill_matrixfact.sh <path_to_serverfile>
```

Otherwise, the MF app will terminate naturally when it is done.

Output data format

The MF app outputs the two factor matrices L and R to `<output_path>.L` and `<output_path>.R`, as whitespace-separated arrays of floating-point values.

`<output_path>.L` will contain N lines, with K numbers on each line.
`<output_path>.R` will contain M lines, with K numbers on each line (it is the transpose of R).

LASSO Regression (PS version)

Given a design matrix X (possibly with some missing entries) and a response vector Y (no missing entries in Y), the Lasso app estimates one vector (regression coefficient vector) BETA, where coefficients corresponding to the features relevant to Y become nonzero. If X is *N-by-M and* Y is *N-by-1*, then BETA will be *M-by-1*. Lasso is commonly used to perform feature selection, where X represents the M features of N samples, and Y represents an output of interest for the same N samples.
The Lasso algorithm optimizes the following objective function:

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} \frac{1}{2}\left(Y(i) - \sum_{j=1}^{M} X(i,j)\beta(j)\right)^2 + \lambda \sum_{j=1}^{M} |\beta(j)|$$

where lambda parameter determines the level of sparsity of BETA vector. The larger the lambda, the larger number of zero coefficients in BETA.

Compiling

From the Petuum directory, run the following commands:

```
cd apps/lasso_ps/
make
```

This will put the Lasso binary in `apps/lasso_ps/bin/`. You can test that the app works on one machine (with 2 worker threads) using:

```
scripts/run_lasso_ps.sh datasets/lasso_data/X100 datasets/lasso_data/Y100 100
100 0.2 100 lasso_output machinefiles/localserver 2 0
```

If the program completes successfully, you will see one output file:
`lasso_output.BETA`

The simulation data (datasets/lasso_data/X100, and datasets/lasso_data/Y100) were generated with non-zero coefficients at 0,2,4,6,8 indices. Thus, you will see that 0,2,4,6,8 lines are non-zero in lasso_output.BETA output file.

For PS Lasso, we recommend not to increase the number of machines/threads too high (e.g. keep the number of concurrent updates less than 100). If dimension of X is very large (e.g. M > 10,000,000), we recommend to use Lasso developed under STRADS. Notice that when running coordinate descent Lasso using many machines/threads, errors may occur due to the concurrent updates of coefficients, leading to conversion to a non-optimal solution (Please see Bradley et al. ICML 2011 for details.) In general, as the number of concurrent updates increases, the error increases. STRADS is designed to control such parallelization errors.

Input data format

For Lasso, there are two input files, and both take a sparse matrix format as input:

```
row_0 col_0 value_0
row_1 col_1 value_1
row_2 col_2 value_2
...
```

This says that the matrix has elements (row_0,col_0) = value_0, (row_1,col_1) = value_1, and so on. If a matrix element is not explicitly specified, it is treated as a missing value - this is NOT the same as a zero value, as missing values are not considered in Lasso computation. Note that for input data file for Y, column index should always be zero because it is a vector.

Running the app

The syntax for running the Lasso app is:

```
scripts/run_lasso_ps.sh run_matrixfact.sh <design_matrix_datafile>
<response_vector_datafile> <num_rows> <num_cols> <lambda> <iters>
<output_prefix> <petuum_ps_hostfile> <client_worker_threads> <staleness>
```

The arguments are as follows:

`<design_matrix_datafile>`
Path to input data file for design matrix.

`<response_vector_datafile>`
Path to input data file for response vector.

<num_rows>
Number of rows in design matrix and response vector. The number of rows in X and Y must be the same.

<num_cols>
Number of columns in design matrix

`<lambda>`
Regularization parameter

`<iters>`
Number of iterations to run: one iteration = touch every coefficient in BETA once.

`<output_prefix>`

The regression coefficient vector BETA will be written to `<output_prefix>.BETA`.

`<petuum_ps_hostfile>`
Petuum server configuration file.

`<client_worker_threads>`
How many worker threads to use on each machine.

`<staleness>`
Set this to 0 unless you want to use the SSP consistency model; see the Parameter Server tutorial for details.

Terminating a hung app

If the Lasso app should hang (or you just want to stop it prematurely), you can kill it by running

```
scripts/kill_lasso_ps.sh <path_to_serverfile>
```

Otherwise, the Lasso app will terminate naturally when it is done.

Output data format

The Lasso app outputs one regression coefficient vector BETA to `<output_path>.BETA` as a column vector.

`<output_path>.BETA` will contain M lines (M is the dimension of X), with 1 number on each line.

LASSO Regression (STRADS version)

Parallel distributed coordinate descent based lasso is implemented with Petuum STRADS that schedules model parameters to update in parallel while avoiding parallel error that results from running CD algorithms in parallel. For performance comparison, we provide two version of lasso - one with sophisticated STRADS scheduling service and another without random scheduling.

Please note:

To run the STRADS Lasso application, at least three machines are required: one worker, one scheduler, and one coordinator.

Compiling Lasso with scheduling service

From the Petuum directory, run the following commands:

```
cd src/strads/
make

That command will downloads and compile all thrid party libraries and two
versions of lasso. You will see three executables

- dist-schedlasso
  Distributed lasso with scheduling service
  Depending on user configuration, it support out of core feature
- dist-noschedlasso
  Distributed lasso without scheduling service. Instead, random and uniform
  scheduling is applied.
```

Compiling Lasso without scheduling service

From the Petuum directory, run the following commands:

```
cd src/strads/
make noschedlasso // see parameter configuration section
```

Input data format

The input data consists of design matrix A (N data samples with M dimension) and output Y vector (N by 1 vector) and output data is coefficient vector (M by 1 vector).

- Design Matrix A

The format of the input matrix A is as follows PBFF(Petuum Binary File Format), our proprietary binary format, for faster uploading. For your convenience, we provide a conversion tool that convert sparse matrix in the standard matrix market format into PBFF.

For conversion, run the following commands

```
cd strads/
./mmt2bin <mmt-file name> <PBFF filename>
```

- Observation Matrix Y
The file format of the observation Y vector contains one output per line without indexing.

Configuration File

STRADS configuration file should be provided. By default, it’s stored in strads/config directory. There are five user configurations.
- maxiter : maximun iteration (phase) to run
- timelimit : maximum time limit to run
- dfrequency : frequency of objective value calculation in terms of iteration
- lambda : lasso lambda
- outputdir : directory to store output files
  This directory can be on network file system or on local storage space of coordinator machine.
  (coordinator machine is the machine with largest MPI rank in your MPI cluster.)

Command Line

For your convenience, we provide python script.
-d : design matrix file name
-o : observation vector file name
-c : configuration file name
- m : the number of worker machine
- t : the number of threads in worker machine
- s : the number of machines for scheduling and coordinating
- p : the number of threads in scheduler machine
    <Caveat: to run STRADS, you need at least three machines for worker, coordinator and scheduler respectively.>
- f : machinefile
- b/-r : size of one phase (the number of model parameters to update in parallel.
  -b and -r should be the same in current version.)
-a : the model size (the number of coefficients in lasso case)
-g : the number of out of core partitioning. (should be larger than or equal to 4)
-e : minimum phase per out-of-core partitioning switching
- i : length of initialization step.
    (if set to 1.2 and the number of coefficients 1M, initialization step will touch 1.2M model parameters.)

For your conveinence, we provide a script run.py

Sample data set

We provide a tiny sample set only for verification. It has 100K dimension, and 50000 with 500K non zero entries. Caveat is that this is too small data for performance comparison. Please, try bigger data set in the Sailing Web site for performance comparison. The benchmark data set for performance comparison is at “http://cogito-b.ml.cmu.edu/stradsdata/data.tar.gz“. This data repository has two sample data sets with 1million and 10 mllion features with 50K samples.

Verification experiment

For verification, run the following steps. We assume that you do not change configuration file and run.py script.

```
Step0) cd src/strads
       make
       *if you have done compilation, you can skip step 0.

Step1) Create mach.vm with exactly 3 MPI machines’ IP addresses,
       one per line.

Step2) cd src/strads/sample
       tar -xvzf X.bin.tar.gz
       tar -xvzf Y.txt.tar.gz

Step3) cd src/strads
       python run.py
```

It will run dist-schedlasso with X.bin and Y.txt in strads/sample data. Result should be similar to the below. Due to the randomness in thread switching, numbers will be slightly different.

```
[coordinator] start lasso iterative update

iteration 1000 obj: 0.3504857764 nz: 27481  elapsed: 5.135013 sec
objt(0.001172)-compt(5.133841) updated params(48945)
iteration 2000 obj: 0.3080116593 nz: 41519  elapsed: 9.975659 sec
objt(0.001359)-compt(9.973128) updated params(97928)

[Coordinator] Initial step is done. Summary: 45412 nz touched var: 120011
[Coordinator] Dynamic scheduling starts!

iteration 3000 obj: 0.2871748594 nz: 45140  elapsed: 15.430288 sec
objt(0.001384)-compt(15.426373) updated params(146765)
iteration 4000 obj: 0.2771896725 nz: 42888  elapsed: 21.400474 sec
objt(0.001369)-compt(21.395190) updated params(195436)
iteration 5000 obj: 0.2724347776 nz: 41503  elapsed: 27.120842 sec
objt(0.001371)-compt(27.114187) updated params(244089)
```

```
iteration 6000 obj: 0.2690880803 nz: 41208  elapsed: 32.809291 sec
objt(0.001350)-compt(32.801286) updated params(292670)
iteration 7000 obj: 0.2664247011 nz: 41334  elapsed: 38.458423 sec
objt(0.001347)-compt(38.449071) updated params(341105)
iteration 8000 obj: 0.2638648408 nz: 41739  elapsed: 44.088898 sec
objt(0.001354)-compt(44.078192) updated params(389440)
iteration 9000 obj: 0.2617938127 nz: 42284  elapsed: 49.691579 sec
objt(0.001358)-compt(49.679515) updated params(437790)
iteration 10000 obj: 0.2599395938 nz: 42774  elapsed: 55.198662 sec
objt(0.001358)-compt(55.185240) updated params(486065)


Legend:
Obj : objective value, nz: the number of non-zero coefficients,
objt: time spent to calc objective value, compt: time spent to update coefficients.
```

Experiment output file

STRADS generates two output files. one file with “.log” extension records performance log and another with “.beta” extension stores beta coefficient values when it reaches max iteration. The files are created in the directory that you specify as -outputdir option in the configuration file.

Experiment with no-scheduling

To run dist-noschedlasso, you can execute run-nosched.py

Experiment with out-of-core feature

To run dist-schedlasso with out of core, you can use run.py. To enable out of core, you should set -g option to larger than or equal to 4.

Manual termination of hang out experiments

To run STRADS test, there should no other running instance of STRADS in a cluster. If STRADS process hangs out, you will have to terminate them by kill or pkill command manually and make sure that there is no running instance of STRADS before starting a next experiment.

Deep Neural Network

We build DNN on top of Petuum parameter server for multi-class classification. DNN takes in a D-dimensional feature vector and outputs a multinomial distribution over L labels. The activation function of hidden units is sigmoid. The activation function of output units is softmax. The error function is cross-entropy. We use Backpropagation and stochastic gradient descent to train the network.

Compiling

From the Petuum directory, run the following commands:

```
cd apps/dnn/
make
```

This will put the DNN binary in `apps/dnn/bin/`. You can test that the app works on one machine (with 2 worker threads) using:

```
scripts/run_dnn.sh 2 0 machinefiles/localserver
datasets/dnn_data/para_example.txt datasets/dnn_data/dnn_data.txt
datasets/dnn_data/dnn_model.txt
```

If the program completes successfully, you will see the saved model in :
`datasets/dnn_data/dnn_model.txt`

Input data format

The input data consists of N data samples. Each data sample has a feature vector and a class label.
The format of the input data is as follows:

```
N D
label : feature_vector
label : feature_vector
...
```

The first line contains two integers N and D. N is the number of training data samples and D is the dimensionality of feature vector. In the following N lines, each line corresponds to one data sample. Each line contains the class label and a feature vector of length D. Class label and feature vector are separated with `:`

Running the app

The syntax for running the DNN app is:

```
scripts/run_dnn.sh <num_worker_threads> <staleness> <hostfile>
<parameter_file> <datafile> <modelfile>
```

The arguments are as follows:

`<num_worker_threads>`
number of worker threads in each client

`<staleness>`
Set this to 0 unless you want to use the SSP consistency model; see the Parameter Server tutorial for details.

`<hostfile>`
Petuum parameter server configuration file.

`<parameter_file>`
Neural network hyperparameter configuration file. Refer to the exemplar file /datasets/dnn_data/para_example.txt for how to configure these parameters.

`<datafile>`
Containing the input training data.

`<modelfile>`
Learned model will be saved into this file.

Output data format

The DNN app saves learned weight matrices and bias vectors into a single file. A network with L layers has L-1 weight matrices: W1 (between layer 1 and layer 2), W2 (between layer 2 and layer 3), …. , and also L-1 bias vectors B1 (in conjunction with W1), B2 (in conjunction with W2), …
The format of the output file is:

```
rows_of_W1  cols_of_W1
values_of_W1

rows_of_W2  cols_of_W2
values_of_W2
```

```
...

...

...

length_of_B1
values_of_B1

length_of_B2
values_of_B2


...

...

...
```

In this file, we first store W, then store B. The order is W1, W2, …, B1, B2…
To store a W, we first store its number of rows and columns, then store the matrix.
To store a B, we first store its length, then store the vector.

Terminating a hung app

If the DNN app should hang (or you just want to stop it prematurely), you can kill it by running

```
scripts/kill_dnn.sh <path_to_serverfile>
```

Otherwise, the DNN app will terminate naturally when it is done.

Writing your own Petuum Apps

The Petuum framework contains two C++ libraries for building new Machine Learning apps

- Parameter Server
- STRADS Model Parameter Scheduler

For each library, we provide a short tutorial and brief API reference. Power users who wish to learn more are encouraged to view the header files in `/src/`, and the Petuum apps in `/apps/`.

Have questions? Visit our Google Group at:
https://groups.google.com/forum/#!forum/petuum-user

Parameter Server

The Petuum Parameter Server allows any thread in any machine to read/write ML algorithm parameters as if they were locally stored on that machine. This enables a “distributed shared memory” style of programming, in which the programmer does not have to think about how to transmit information between machines; (s)he only has to query the Parameter Server as if it were a local array or dictionary data structure.

Unlike other key-value stores such as memcached, the Petuum Parameter Server features a novel consistency model called Stale Synchronous Parallel (SSP), designed to ensure ML algorithms execute both fast and correctly. SSP guarantees that a worker will see all parameter updates from other workers within the last S iterations, where S is a user-defined threshold. When S = 0, SSP is equivalent to Bulk Synchronous Parallel (BSP) execution. As S is increased, Petuum’s Parameter Server can rely more and more on locally cached parameters - this prevents the network interface from becoming a bottleneck, resulting in faster ML algorithm execution.

Together, the Petuum Parameter Server’s key-value store interface and SSP consistency model allow distributed ML applications to be written almost as easily as single-machine multi-threaded programs. When using the Petuum Parameter Server, there is (usually) no need for concurrency primitives like mutexes and barriers, or distributed communication schemes such as message passing and remote procedure calls, because the Parameter Server takes care of all communication and synchronization needs. The Petuum apps built on the Parameter Server are good examples of this simplicity in action.

At its core, the Petuum Parameter Server is driven by 3 basic commands:

- Reading table elements
- Adding to or subtracting from table elements
- Advancing to the next iteration

In the following tutorial, we will explain how to set up the Petuum Parameter Server in a distributed, multithreaded environment, as well as how to use these 3 operations within worker threads.

Tutorial foreword

This tutorial assumes the reader is familiar with basic multithreaded programming in C++, using C++11 threads, pthreads or Boost threads. We assume that the reader knows how to create new threads, and wait for them to finish (joins). We do not assume familiarity with other parallel programming primitives such as locks, mutexes and semaphores, as they are usually unnecessary when writing ML algorithms on the Petuum Parameter Server.

Setting up the Parameter Server

A distributed, multithreaded program consists of *processes* and *threads*: every machine has one program process, which in turn has multiple program threads. When you run a C++ program, its `main()` function is executed by the primary process thread. You can then create additional threads using pthreads or another threading library, and wait for them to complete their task using a *join*.

The Petuum Parameter Server (PS for short) is always configured in the primary process thread; this needs to be done before other threads access the PS. Here is an example PS configuration within `main()`:

```
// Configure Petuum PS
petuum::TableGroupConfig tgc;
// Read server configuration file
petuum::GetHostInfos(FLAGS_hostfile, &tgc.host_map);
petuum::GetServerIDsFromHostMap(&tgc.server_ids, tgc.host_map);
// Global params
tgc.consistency_model = petuum::SSP;
tgc.num_total_server_threads = FLAGS_num_clients;
tgc.num_total_bg_threads = FLAGS_num_clients;
tgc.num_total_clients = FLAGS_num_clients;
tgc.num_tables = 3;
// Local params
tgc.num_local_server_threads = 1;
tgc.num_local_bg_threads = 1;
tgc.num_local_app_threads = FLAGS_num_worker_threads + 1;
tgc.client_id = FLAGS_client_id;
tgc.local_id_min = tgc.client_id * 1000;
tgc.local_id_max = tgc.client_id * 1000 - 1;
// Configure PS row types
petuum::TableGroup::RegisterRow<petuum::DenseRow<float> >(0); // Register dense rows as ID 0
petuum::TableGroup::RegisterRow<petuum::SparseRow<float> >(1); // Register sparse rows as ID 1
// Start PS
petuum::TableGroup::Init(tgc, false); // Initializing thread does not need table access
```

Configuration is done through a `TableGroupConfig` object that contains most of the information needed to start the PS (except individual table configuration, which will be covered in the next section). We now explain the important parts of the configuration:

`petuum::GetHostInfos(FLAGS_hostfile, &tgc.host_map);`
`petuum::GetServerIDsFromHostMap(&tgc.server_ids, tgc.host_map);`
These lines load the server configuration file: in this case, at the path specified by the string `FLAGS_hostfile`. Change `FLAGS_hostfile` to whatever you need.

`num_local_server_threads`
`num_local_bg_threads`
Both of these should be set to 1. Other values are possible, but they require an advanced understanding of the server configuration file format, which is out of the scope of this tutorial.

`num_local_app_threads`
If you are creating P additional threads (within this machine only) on top of the primary process thread, set this to P+1.

```
num_total_server_threads
num_total_bg_threads
num_total_clients
```
If you are using M machines in total, set all of these to M.

```
num_tables
```
A table is essentially a 2-dimensional global array stored in the PS. Set this to the number of tables your application requires.

```
client_id
```
This number is used to identify machines. It must match the order of IP addresses in the server configuration file. For example, say your server configuration file looks like this:
```
0 10.1.1.0 10000
1 10.1.1.0 9999
1000 10.1.1.1 9999
2000 10.1.1.2 9999
3000 10.1.1.3 9999
```
Then the machine at `10.1.1.0` must have `client_id = 0`, the machine at `10.1.1.1` must `have client_id` = 1, and so on. This is usually accomplished by passing the client_id in as a command-line argument. We'll talk about how to do this later, in the section on scripts.

```
consistency_model
```
This should always be `Petuum::SSP`, which corresponds to the Stale Synchronous Parallel consistency model. It is possible to write other consistency models for the PS, but that is out of the scope of this tutorial.

```
local_id_min
local_id_max
```
Leave these exactly as the example says. Other values are possible, but they require an advanced understanding of the server configuration file format, which is out of the scope of this tutorial.

```
petuum::TableGroup::RegisterRow<petuum::DenseRow<float> >(0);
petuum::TableGroup::RegisterRow<petuum::SparseRow<float> >(1);
```
These lines are used to register dense table rows (vectors) and sparse table rows (maps) with the PS. The argument is a user-chosen integer ID for the row type - in this example, dense floating-point rows have ID 0, while sparse floating-point rows have ID 1. Later, when you configure individual tables, every table must choose exactly one row type to use. Note that dense rows are faster but require more memory, while sparse rows are slower but are very memory-efficient if most table elements are zero. You can also register integer rows by changing `<float>` to `<int>`. You can even register both floating-point and integer rows, using

different IDs - for example, `RegisterRow<petuum::DenseRow<float> >(0)` adnd `RegisterRow<petuum::DenseRow<int> >(2)`.

`petuum::TableGroup::Init(tgc, false);`
Starts the PS with configuration `tgc`. The second argument is a bool that specifies whether the primary process thread needs table access. If your primary process thread is not doing any computation, you must set it to `false`. Otherwise, the PS will expect `clock()` (i.e. advance to next iteration) calls from the primary process thread. If you are also using the primary process thread to do computation, then set it to true. In general, we recommend leaving all computation to additional worker threads, and seeting this argument to false.

Configuring Parameter Server tables

Once the PS has been set up, we need to supply details for each table in the program. Here is a 3-table example taken from the Matrix Factorization application:

```
// Configure PS tables
petuum::ClientTableConfig tc;
tc.table_info.row_type = 0; // Dense rows
tc.oplog_capacity = 100;
// L_table (N by K)
tc.table_info.table_staleness = FLAGS_staleness;
tc.table_info.row_capacity = FLAGS_K;
tc.process_cache_capacity = N_;
petuum::TableGroup::CreateTable(0,tc);
// R_table (M by K)
tc.table_info.table_staleness = FLAGS_staleness;
tc.table_info.row_capacity = FLAGS_K;
tc.process_cache_capacity = M_;
petuum::TableGroup::CreateTable(1,tc);
// loss_table (1 by total # workers)
tc.table_info.table_staleness = 0;  // No staleness for loss table
tc.table_info.row_capacity = get_total_num_workers();
tc.process_cache_capacity = 1;
petuum::TableGroup::CreateTable(2,tc);
// Finished creating tables
petuum::TableGroup::CreateTableDone();
```

Each table are configured through a `ClientTableConfig` object, with the following members:

`table_info.row_type`
Determines what row type this table uses. In our earlier example, 0 = dense rows (vectors) and 1 = sparse rows (maps).

`oplog_capacity`
Advanced PS option out of the scope of this tutorial. Just set it to 100.

`table_info.table_staleness`

SSP staleness setting for this table. Set to 0 for Bulk Synchronous Parallel execution. Values > 0 will improve the throughput of the parameter server (more ML algorithm iterations per second), at the cost of slower ML algorithm progress per iteration. For ML applications involving 10+ machines, there is a sweet spot for `table_staleness > 0` that maximizes progress per second (which is = iters per second * progress per iter). If you are using only a handful of machines, `table_staleness = 0` is usually the best choice.

`table_info.row_capacity`
Determines the number of columns per row in this table. Only affects dense rows; no effect on sparse rows. Note that you don't need to set the number of rows in the table; you can always access any row ID that is a 32-bit signed integer.

`process_cache_capacity`
Determines how many of this table's rows to cache locally. Higher values improve performance, particularly when worker threads are expected to access many rows at random. However, this increases memory usage on the machine. For small ML problems, you can just set this to the total number of rows used by the ML problem. If you do not expect the machine will have enough memory to cache all rows, then you should lower this value.

`petuum::TableGroup::CreateTable(table_id,tc);`
Creates the table specified by `tc`. The first argument is the table ID >= 0, which will be used to access the table later. Note that all machines must set up their tables in the exact same way.

`petuum::TableGroup::CreateTableDone();`
Call this once all tables have been created. Tables cannot be accessed before this has been invoked.

Setting up worker threads to use the Parameter Server

Once the PS and its tables have been set up, worker threads can start using the PS tables by calling

```
petuum::TableGroup::RegisterThread();
```

Before a worker thread terminates, it should also call

```
petuum::TableGroup::DeregisterThread();
```

If you need to access tables from the primary process thread, it should call:

```
petuum::TableGroup::WaitThreadRegister();
```

This is not necessary if `petuum::TableGroup::Init()` has been called with false as its 2nd argument. Never call `WaitThreadRegister()` from a worker thread.

Accessing tables from worker threads

Before reading and writing to tables, we must first set up an accessor object, like this:

```
petuum::Table<float> table = petuum::TableGroup::GetTableOrDie<float>(0);
```

The template type (<float> in this example) must match the table's underlying row type. The argument (0 in this example) is the table ID from `CreateTable(table_id,tc)`. If you have multiple tables, you can declare multiple table accessors.

Caution: table accessors are thread-specific. Each thread must create its own accessor, and cannot share it with other threads. For example, it is wrong to create a single accessor in a global variable or object, and then share it among all threads.

Reading, writing and advancing iterations in worker threads

Once you have the table accessor, you can fetch a table row using the following code:

```
petuum::RowAccessor row_acc;
table.Get(row_index, &row_acc);
auto& row = row_acc.Get<petuum::DenseRow<float> >();
```

The first line declares a row accessor `row_acc`, and the second line connects it to the table, where `row_index` specifies which row you want. Finally, the third line gets a read-only reference `row` to the underlying table row; make sure you correctly specify the row type (`petuum::DenseRow<float>` in this case).

Caution: row accessors are also thread-specific. Do not attempt to share them between threads. The safest policy is to always create a new accessor every time you need one.

You can now use `row` like a read-only array, e.g.

```
float new_value = row[column_index] + delta;
```

If row is a `SparseRow<>`, you can also iterate over nonzero elements (zero elements are automatically deleted by the PS), using the following syntax:

```
auto iter = row.cbegin();
while (!iter.is_end()) {
  // Fetch value = row[column_index]
  int column_index = iter->first;
  float value = iter->second;
  ... // Do something with column_index and value here
  iter++;
}
```

Writing to a table is done via:

```
table.Inc(row_index, column_index, delta);
```

This will add the (possibly negative) value `delta` to the (`row_index`,`column_index`)-th element in `table`. If you want to replace the element rather than add to it, all you have to do is subtract off the old value:

```
table.Inc(row_index, column_index, delta - row[column_index]);
```

Every time the thread is done with one iteration's worth of work, it should call

```
petuum::TableGroup::Clock();
```

This tells the PS that the thread is ready to exchange its table updates with other threads. `Clock()` is a core part of the Bulk Synchronous Parallel and SSP communication models that the PS is built upon. Failing to call `Clock()` will result in the program stalling indefinitely.

Most ML algorithms are iterative in nature, and thus have an obvious iteration/clock boundary. When in doubt about the clock boundary, a natural strategy is to call `clock()` every x data points or variable updates, where x is the same across all threads.

Scripts for starting Parameter-Server-based programs

We use bash scripts to start up PS programs across multiple machines. For example, the Matrix Factorization application uses `scripts/run_matrixfact.sh`:

```
#!/bin/bash

if [ $# -ne 7 ]; then
 echo "Usage: $0 <datafile> <K> <iters> <output_prefix> <petuum_ps_hostfile> <client_worker_threads>
<staleness>"
  echo ""
  echo "This script runs Matrix Factorization solver on <datafile> for <iters> iterations, and"
  echo "outputs the rank-<K> factor matrices to <output_prefix>.L and <output_prefix>.R."
  echo ""
  echo "<datafile> should have one line per non-missing matrix element, and every line should"
  echo "look like this:"
  echo ""
  echo "ROW COL VALUE"
  echo ""
  echo "Where ROW and COL are 0-indexed row and column indices, and VALUE is the matrix"
  echo "element at (ROW,COL)."
  echo ""
  echo "Other arguments:"
  echo "<petuum_ps_hostfile>: Your Petuum Parameter Server hostfile, as explained in the manual."
  echo "<client_worker_threads>: How many client worker threads to spawn per machine."
  echo "<staleness>: SSP staleness setting; set to 0 for Bulk Synchronous Parallel mode."
  exit
fi

data_file=$(readlink -f $1)
K=$2
iters=$3
output_prefix=$(pwd)/$4
host_file=$(readlink -f $5)
client_worker_threads=$6
staleness=$7

progname=matrixfact
ssh_options="-oStrictHostKeyChecking=no -oUserKnownHostsFile=/dev/null -oLogLevel=quiet"

# Find other Petuum paths by using the script's path
script_path=`readlink -f $0`
script_dir=`dirname $script_path`
project_root=`dirname $script_dir`
prog_path=$project_root/apps/matrixfact/bin/$progname

# Parse hostfile
host_list=`cat $host_file | awk '{ print $2 }'`
unique_host_list=`cat $host_file | awk '{ print $2 }' | uniq`
num_unique_hosts=`cat $host_file | awk '{ print $2 }' | uniq | wc -l`

# Kill previous instances of this program
echo "Killing previous instances of '$progname' on servers, please wait..."
for ip in $unique_host_list; do
 ssh $ssh_options $ip \
    killall -q $progname
done
echo "All done!"

# Spawn program instances
client_id=0
for ip in $unique_host_list; do
 echo Running client $client_id on $ip
  ssh $ssh_options $ip \
    GLOG_logtostderr=true GLOG_v=-1 GLOG_minloglevel=4 GLOG_vmodule="" \
    $prog_path \
    --hostfile $host_file \
    --datafile $data_file \
    --output_prefix $output_prefix \
    --K $K \
    --num_iterations $iters \
    --num_worker_threads $client_worker_threads \
    --staleness $staleness \
    --num_clients $num_unique_hosts \
    --client_id $client_id &
  # Wait a few seconds for the name node (client 0) to set up
  if [ $client_id -eq 0 ]; then
   echo "Waiting for name node to set up..."
    sleep 3
  fi
 client_id=$(( client_id+1 ))
done
```

The script does the following things:

1. Read in command line arguments for the Matrix Factorization application
2. Parse the Petuum PS server configuration file
3. Kill previous instances of the Matrix Factorization application
4. Use SSH to start the Matrix Factorization application on each machine in the server configuration file, with the proper client ID

We strongly recommend modifying the script file for your particular application, rather than attempting to create one from scratch. This will allow you to avoid configuration issues, particularly with the server configuration file and assigning client IDs.

Making your own PS program

The easiest way to make your own PS programs is to modify an existing Petuum PS application - that way, most of the setup and scripting is already in place. For example, to use the Matrix Factorization application as your starting point,

1. Copy `/apps/matrixfact/` to `/apps/yourapp/`
2. Copy `/scripts/run_matrixfact.sh` to `/scripts/run_yourapp.sh`
3. Modify the Makefile, cpp file, and bash script to suit your needs

Parameter Server Fault Tolerance

The parameter server provides fault tolerance via checkpointing. The parameter server preserves its states by periodically taking snapshots and store them in a high-performance, persistent key-value store. This allows the parameter server to recover from failures like process crashes. We do not yet provide failure detection and automatic recovery. Thus users that need fault tolerance should do failure detection and recovery by themselves. Also, the applications are responsible for preserving their local states (variables that are not stored in the parameter server).
Since checkpointing involves writing to persistent storage, users may see degraded performance in their applications compared to the non-fault-tolerant parameter server.
We encourage users who require strong fault tolerance contact Petuum developers for further assistance.

To get the fault tolerance parameter server:

```
sudo apt-get install git
git clone git@github.com:sailinglab/petuum.git
cd petuum
git checkout release_0.2_ft
```

The fault tolerant parameter server follows the same setup procedure as the regular parameter server, but it requires 3 additional configuration parameters in `TableGroup`:

`snapshot_clock`
This parameter determines at which clocks the snapshot is taken. For example, if `snapshot_clock` is set to `N`, snapshots will be taken at clock `N, 2*N, 3*N` and so on. Under BSP, snapshot taken at `N` contains updates from all workers produced from exact clock `0` to clock `N - 1`. Under SSP, the snapshot may contain updates generated in up to clock `N + staleness + 1`.

`snapshot_dir`
The directory where the snapshots are saved. If application needs to tolerate permanent storage failures, the snapshots should be stored in fault tolerant storage systems, such as RAID or HDFS.

`resume_clock`
From which snapshot to resume the parameter server. Set this parameter to 0 if it is not recovery.

The parameter server comes with a HelloWorld demo for fault tolerance. It is not a real application, but it demonstrates how to use the fault tolerance APIs in an application.

In HelloWorld, each application threads write to the parameter server in each clock and performs sanity check to verify the writes are properly stored and propagated.
If there's no error, HelloWorld prints "`Helllo World!`" to the terminal in the end.

In order to run the HelloWorld application, you need a host file. Note that this is different from the the Parameter Server Configuration file discussed before. This configuration file consists of a list of unique IP addresses, like:

```
192.168.0.1
192.168.0.2
...
```

First, we need to create a server file using `/scripts/mk_server_file.py`
In Petuum root directory, run the script:

```
./scripts/mk_server_file.py <host_file> <output_server_file> <num_app_threads>
```

`<host_file>`
The host file described above.

`<output_server_file>`
Name of the output server file.

`<num_app_threads>`
Number of application threads to run in each process, the number must be at least 1.

Then we can run HelloWorld using `/scripts/run_helloworld.sh`
In Petuum root directory, run:

```
./scripts/run_helloworld.sh <host_list> <server_list> <client_worker_threads>
<staleness> <snapshot_dir> <snapshot_clock> <resume_clock> <num_clocks>
```

`<host_list>`
The host file described above.

`<server_list>`
The serve file described above.

`<client_worker_threads>`
The number threads to run in each HelloWorld application process. Must match the number of application threads set when creating the server file.

`<staleness>`
Staleness of SSP.

`<snapshot_dir>`
Directory to store snapshots.

`<snapshot_clock>`
The `snapshot_clock` parameter in `TableConfig`.

`<resume_clock>`
From which snapshot to recover the parameter server. Set to 0 if it is not recovery.

`<num_clocks>`
How many clocks to run the helloworld application.

Users may simulate a crash by killing an arbitrary HelloWorld process on any host. Once the failure is discovered, the user may kill the entire system and resume the system from the latest snapshot.

To verify the system works, we provide a sample host file: `machinefiles/localhost` for you to start with, which contains a single IP.

Create server file:

```
./scripts/mk_server_file.py machinefiles/localhost machinefiles/localserver 1
```

This creates a server file: machinefiles/localserver.

```
./scripts/run_helloworld.sh machinefiles/localhost machinefiles/localserver 1
0 snapshot 2 0 10
```

This runs HelloWorld for 10 clocks under BSP and takes a snapshot every 2 clocks. Snapshots are stored in `snapshot` directory.

```
./scripts/run_helloworld.sh machinefiles/localhost machinefiles/localserver 1
0 snapshot 2 0 10
```

This resumes the HelloWorld process from clock 2 using snapshots stored in snapshot directory.

STRADS Model Parameter Scheduler

Introduction to model parameter scheduling

STRADS provides model parameter scheduling service to accelerate convergence speed while doing dependency checking to reduce parallel. The key observation behind STRADS is that model parameters require different amount of workload to converge. Therefore, scheduling more promising parameter more frequently will improve convergence speed.
To scale up with the size of model and the number of machines, STRADS deployes an array of scheduler machines that are in charge of weight sampling and dependency checking to make sets of independent parameters which maximizes contribution to convergence if updated in parallel. Each scheduler machine is given a disjoint partition of model parameter space. A set of such independent parameters to update in parallel is called “phase”.
Phases from schedulers are pooled in the coordinator machine. The coordinator dispatches phase to worker machines. Worker machines are give a partition of samples and do make partial results for a given partition of sample. Partial results from worker machines are delivered to and aggregated by the coordinator machine. With aggregated values, the coordinator get fresh values of model parameters and send updated priority information (sampling weight information) to the scheduler machine that is in charge of that phase. To prevent starvation, the coordinator pulls phases from schedulers in round-robin.

Introduction to out-of-core (OOC) of STRADS

To support extremely large data not fitting into aggregated memory of a cluster, STRADS supports out-of-core primitives. STRADS partition model space into N disjoint partitions and update model parameters in K-th partition while uploading K+1 th partition from disk into main memory. When uploading K+1 th partition is done, STRADS switches to K+1 partition and initiates K+2 th partition loading. To overlap parameter updating and partition loading, partition loading is conducted by a background I/O thread. For the integrity of updating parameters, coordinator enforces a global barrier across workers and scheduler whenever partition switching happens. Therefore, STRADS guarantees all workers and schedulers works on the same model partition.

Programming with STRADS scheduler

When STRADS starts, it will automatically partition samples in worker machines and models in scheduler machines for a given user configurations (the number of worker machines, threads per machine, the number of schedulers and threads per machine) and input data (the number of samples and the number of features). Users are free of model/data partitioning. The main programming efforts will be spent on developing algorithm specific update routine, message encapsulation, and asking for scheduling service and updating the priority of a parameter as an algorithm proceeds.

Structure of STRADS Apps.

STRADS app consists of three tasklets deployed in coordinator, scheduler, and worker machine(s). Pseudo code of each tasklet are followed.

Pseudo code of Coordinator :

```
prepare scheduler and workers to start work.
while(1){
   Receive phase information from an array of scheduler
   Send the phase with fresh model parameter from the previous phase
   Collect partial results from the workers
   Aggregate the partial results and
   Calculate fresh value of model parameters in the phase
   Send new priorities of model parameters to a scheduler
}
```

Pseudo code of Scheduler :

```
Get partitioning information and loading data partition from disk
while(1){
   Do weight (priority) based sampling and find a set of parameters
   Filter out too strongly dependent parameters
   Send a phase to the coordinator
   Receive new priorities of the parameters in the phase.
   Update priorities
}
```

Pseudo code of Worker :

```
Get partitioning information and loading data partition from disk
while(1){
   Receiving a phase to update in parallel
   Update local status (residual in lasso case) with fresh values in the Phase
   Make a partial result for the set of model parameters in the Phase
   Send the partial result to the coordinator
}
```

Function APIs of STRADS

- syscom_async_recv_amo_malloc(comctx *ctx, void *buf);
  Receive a phase from an array of scheduler machines
  This is asynchronous call. If there is no phase in communication stack, NULL will be returned.
  ctx should be threadctx->comh[SGR] to receive a packet from a scheduler machine.
  ctx should be threadctx->comh[WGR] to communication with worker machines.

- syscom_send_amo(comctx *ctx, threadctx *tctx, void *pos, command cmd, int partno)
  Send a phase to worker machines.
  This is buffered send command. ctx should be threadctx->comh[WGR] to send a packet to a scheduler. ctx should be threadctx->comh[SGR] to send a packet to a scheduler machine.

- psched_process_delta_update(list *weight, threadctx *tctx, double min_unit, int phasesize)
  Send a list of priorites to a scheduler together with minimum priority for sampling and desired size of a phase. The minimum priority should be larger than 0 to prevent starvation of model parameters with zero valued priorities. Call of this function means giving a token to allow a a scheduler to make a next phase in the future.

- psched_send_token(threadctx *tctx, int schedid)
  Send a token without priority information. The main use of this function is to conduct initialization step that does not run scheduling and sweep out all parameters to find initial priorities.

- send_plan_wait_for_loading(threadctx *tctx, dpartitionctx *dpart, int machines, samples, features);
  Send data partitioning scheme and data loading command to N machines and wait for the destination machines to complete partition loading. This is blocking call.

- send_oocplan_no_wait_for_loading(threadctx *tctx, dpartitionctx *dpart, int machines, samples, features)
  Send data partitioning scheme and data loading command to N machines and do not wait for the completion of partition loading. This is non-blocking call for out of core features.

Apps with STRADS

As an example, current release includes coordinate descent Lasso with L1 regularization. But application of STRADS are not limited to Lasso. Various coordinate descent based algorithms are supposed to benefit from STRADS scheduling.