# Convex Optimization for Big Data T-61.6020

# Guidebook to use Spark on Triton

Example implementation of *Shotgun* algorithm

Kunal Ghosh, 546247 | Jussi Ojala, 544605

## Contents

# 1 Introduction

The password "big data" is often defined with triple V, Volume, Velocity and Variability. In the context of this report we concentrate problems related to the size of the data (Volume). This report guides you through elementary steps to deal with such a problems in Aalto University computer environment. The essence of the report is to help to utilise Aalto Triton queueing system for distributed calculations with Spark. Throughout the report we use parallel co-ordinate decent (convex) optimisation algorithm "Shotgun" as an guiding example.

The report is organised so that we first briefly introduce the general use case scenarios when it is useful to change from the single computer calculations to the distributed calculations, which can be allowed by using some cluster computing system. Then we have general introduction to the Aalto cluster computing environment and it's queueing system, called Triton. After getting familiar with Triton we set the Triton system for distributed calculations with Apache (Py)Spark implementations. We go this setting through step by step. At the end we are ready to go to the details of the *Shotgun* Spark implementation and run this distributed algorithm in Triton system.

The report contains the user guide to set up the Saprk to Triton environment. However the aim of the project was to first enable the distributed optimisation of the example algorithm, called *Shotgun* [Bradley]. We selected simple L1 (Lasso) regularised linear regression model for our study case. This is another of the example cases shown in the original paper introducing "Shotgun" algorithm. As shown in the paper, this convex optimisation problem can be parallelised, but the algorithm is intended for multicore parallelism (i.e. no time delay between parallelised calculations). Our multicore matlab implementation can be found in Appendix C. In this paper we will use the algorithm in distributed calculations i.e. allowed delays between processes. The synchronization details are internal to `Spark`, however, it maintains the logical order in which the operations (on the data) have been implemented. The Spark implementaion can be found in Appendix A.

In the change from multicore matlab implementation to the PySpark implementation, the first step was to implement algorithm in Python and then do the conversion to PySpark to enable distributed calculations. The python implementation is included in Appendix B. The second step is the essence of this report and that is to show how to set up this distributed calculations (Spark implementation ) to normal queueing system (Triton). The report includes detailed instructions to set up distributed Spark implementation to that environment. For short example calculation we selected the same data from the original paper [Bradley] called *Mug32singlepixcam*. The data size is not "big" however it is used for the testing purposes. This enables us to see that implemented algorithm gives similar results as produced in original paper.

# 2 Use case - When do you need this setup

In our case it is assumed that the data size is the origin of the problems. The limitation of single computer comes either due to storing capabilities or computing time (when either the algorithm is complex or the dimension of the data is very high). In this case the size of the data and the number of cores in available computer defines when it is useful to change the setup from the one computer to several computer. As long the memory in the computer is not the bottleneck and the cores in the computer provide enough parallelism that the calculation time is reasonable there is no reason to change the setup.

However when the time or memory start to be bottleneck that is usually the time to change. In our example this means changing from Matlab multicore programming to PySpark implementation of distributed computing. It would have been fine to do the original implementation with (Py)Spark to get the benefits of the multi-core parallelism, however often the initial implementation and debugging of the code is easyer with some other programming language. This was the reason why we first version was coded with matlab (there were no intention to distributed calculations by then). The time for Spark is often when the original programming environment meet it's limits in parallelism.

Moreover the cluster's like Triton are used only when the dataset you want to work with doesn't fit into the memory of a single computer or the algorithm needs more parallelism than can be found in one computer to work in sufficient time.

However, its fine (and advisable) to use a smaller dataset when testing your (Py)Spark implementation and when testing the cluster setup.

## 2.1 How big is your data ?

As we explained earlier the limits of data size to change between single computer to cluster based approach depends on the available resources. Aalto university IT provides quite large hosts like `brute.aalto.fi` and `force.aalto.fi` each with 256GB of RAM and 32 Cores. It is possible to run `MATLAB` or (Py)Spark on these hosts. However the resources of these computers are shared by all Aalto students and staff. If your code would use significant portions of the resources of the above two machines, for a long period of time, you should consider using Triton instead. Before starting to use Triton environment, read the Triton Usage Policy ![1] and test your implementation with smaller data.

### 2.1.1 Data for example implementation

For testing purposes we used the data `Mug32singlepixcam` which is small data 410 rows with 1024 features. The shotgun is designed for this kind of data that have more features than samples. This also allowed us to compare the results to one test result from the original paper.

We have briefly discussed how to handle large datasets (which do not fit into the memory of a single computer) here.

# 3 Triton - A brief Introduction

Triton is a collection of heterogeneous computer nodes (servers). By heterogeneous, we mean, some of the servers have large RAM, some have lots of CPU cores, some servers have GPU (Graphics Processing Unit) cards connected to them, etc.So, one can run a wide variety of different types of computations on the Triton cluster.

To get started with Triton CSC department has collected parctical information starting from, how to log-in to Triton to several details in *Triton quickstart guide*[2]. When the initial step is done helping guidance can be found from Aalto-IT triton wiki[3].

In the following subsections we go through essential issues to understand how to start use the Triton cluster. First we connect to the Triton Loging host and clarify few essential terms issues in the Triton

[1] https://wiki.aalto.fi/display/Triton/Triton+usage+policy

[2] http://science-it.aalto.fi/wp-content/uploads/sites/2/2016/05/SCiP2016_summer.Triton_intro.pdf

[3] https://wiki.aalto.fi/display/Triton/

queueing system. When the main interface is clear we use simple example to show how to describe and send a `Job` to Triton. We also clarify how the `Job` is proceeding inside Triton and where you can find the output.
But before start to implement, read the Triton Usage Policy ![1]

### 3.1 Interacting with Triton

In this section we briefly introduce high-level structure of typical queueing system. The aim is to go through the steps to interact with Triton queueing system and go through most important logical components of Triton. Figure 4 shows the tree main component: Loging host, task manager and computer resources. In following we clarify interaction with them and remind how those components see data and needed software.

- **Connect Triton:** First `ssh` into the Triton **Login Host** from a laptop or a desktop computer connected to the Aalto network.
  - Open a terminal program and enter `$ssh username@triton.aalto.fi`
- **Specify Computing needs:** After connecting the Triton **Login Host** you need to specify computing resource needs to Triton task manager, called `SLURM`. `SLURM` is an open-source task-management system employed in many similar clusters like Triton.
  - **NOTE:** You must not run any CPU / Memory intensive applications on the login Host. It is typically only used for submitting jobs to SLURM and monitoring the submitted jobs. These job submission and monitoring is done using special slurm specific commands.
- **Use allocated computers:** Triton resources includes a cluster of Heterogenous computers. These are Individual computers which are all connected to triton. You can access these only after you have submitted a job to SLURM, in which you have specified what sort of computers you need and for how long, and SLURM has allotted these computers to your job.
  - Once your allotted time is over, you can no longer access computers you were allotted earlier, unless SLURM allotts them to you again in a following new job request.

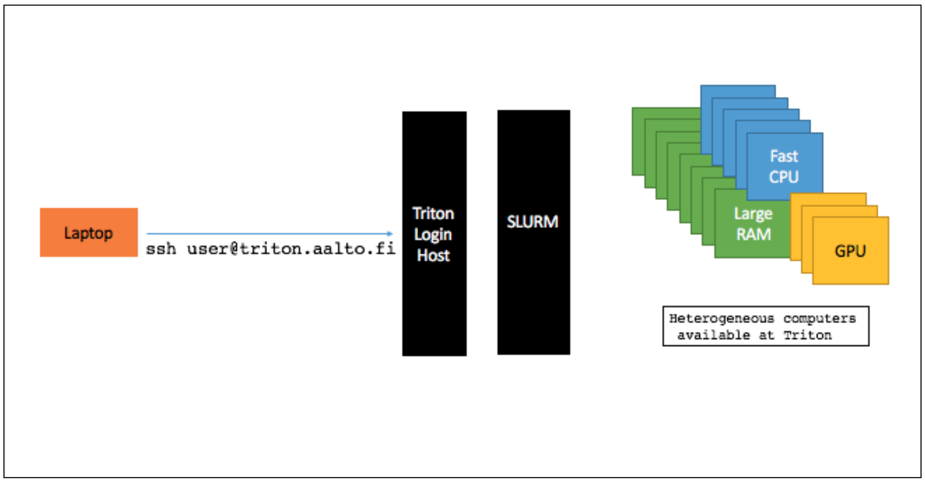


Figure 1: A block diagram depicting the Triton main logical components that user interacts.

After logging process to Triton is clear it is worth to clarify how the data and needed software are handled in Triton.

- File System, how the data is handled
  - Each host i.e. the `Triton login host` and the hosts that SLURM allots to your job have a fast local storage mounted as the `tmp` folder in the root directory. Consider this as a fast harddrive or SSD connected locally to that computer. Use this when you want to store some temporary files during computations.
  - Each host also has an environment variable `$WRKDIR` which is a folder mounted on a network file system, and is available at the same location in all hosts. You are recommended to `cd` into this folder when you log into Triton and treat that as your workspace.
    - **NOTE** that since its a network folder, it is synchronized between all the hosts and hence it is slow to access.
- Software : You might want to use various software packages (e.g. a compiler) when using Triton. You can use the `module` command to access various pre-installed packages. Some commonly used commands:
  - `module list` : To list all installed software packages.
  - `module spider <name>` : To search for a software package. e.g. `module spider numpy`
  - `module load <module name>` : Load a module after which you can start using the softwares bundled in the module.

## 3.2 Computing with Triton

When you want to run a computation on Triton you do it by submitting a *Job* to `SLURM`. However you can also use Triton interactively by requesting SLURM an interactive session. These interactive sessions are mainly intended for debugging your *Job* scripts.

A *Job* script is a (Unix/Linux) shell script. Below a very simple example script named `request.slrm`:

```
#!/bin/sh
#SBATCH --time=00:05:00
#SBATCH -N 3
#SBATCH --ntasks-per-cpu=10

srun /bin/echo hello world
```

Listing 1: `request.slrm`

A few things to note:

- We choose to give the script a `.slrm` extension just to distinguish it from other shell scripts. But it could also be `.sh` for example.
- As in all shell scripts the first line is in the comment for unix shell and specify the executable shell interpreter(sh). For more information read the wikipedia article on `UNIX Shebang`[4]
- The lines in the script beginning with a `#SBATCH` specify our *Job* configuration and have a special meaning for `SLURM` when we submit the script to it. For example in the script above:
  - `#` at the beginning of the SBATCH indicates that its a comment for the unix shell. However, it has a special meaning for `SLURM`.

[4] https://en.wikipedia.org/wiki/Shebang_(Unix)

  - `#SBATCH -time=00:05:00`
    Indicates (to `SLURM`) that we want to run the job for 5 minutes. Once `SLURM` allocates some resources to this *Job* it will revoke the resources and kill the job after 5 minutes.
  - `#SBATCH -N 3`
    Indicates that this *Job* must be run on 3 hosts.
  - `#SBATCH -mem-per-cpu=10`
    Indicates that we want 10 MB memory per core.

- **`SLURM` will put your job on hold** until all the resources you have requested for are available before it will launch your script. So be judicious with the amount of resources you request !

- **`srun`** `<command>` indicates the command that slurm will run as separate process.
  - Typically, `<command>` is another shell script which is executed on each of the hosts assigned to your Job.
  - Intuitively, it is like logging into all of the hosts and executing the `<command>` on each of the hosts.So, you need to take care of how you want to share data between the processes running on all the hosts.
  - A typical script invoked using `srun` could logically have the following parts:
    1. Set up your processing environment (e.g. using `module load <name>` which loads the necessary software on each host).
    2. Download the necessary datasets.
    3. Run a (python) script which processes the downloaded data.
    4. Clean-up tasks after the script is done executing. (e.g. delete the downloaded dataset or temporary files created during execution).

The process to submit the *Job* to the SLURM and it's execution is described in Figure 2. From Triton Loging host the *Job* script is submitted to SLURM with the `sbatch <script_name.slurm>` command. This command returns a `JOB ID` specific to your submission. The job itself is now in the SLURM queue. You can query the current status of your job using the `slurm q` command (more information at[5]). Once the job is completed you will find a output file in your current working directory. The file has `JOB ID` as the filename and a `.out` ending extension. This will contain all the output that your job writes to `STDOUT`[6].
Excellent place to look more information about these is from the Triton User Guide[7] and Aalto Triton FAQ[8] maintained by Aalto Science-IT.

## 4 Setting up Apache Spark on Triton

`Apache Spark (Spark)` has three main components: The master node, the slave node(s) and the data stores. The slave nodes do the distributed calculations and the master node is responsible for "control and coordination" of a spark setup and doesn't itself do any computation. The data store contain the spark data wich can be saved with different types of API's such as sql, data set, data frames or RDD's.

We will go through the setting up the Spark to Triton in step by step. First we connect to the Triton and download the Spark somewhere to be available by all the hosts. This is one example how to do this, later when the spark is available in Triton you can use the module command like in any software.

[5] https://wiki.aalto.fi/display/Triton/Slurm+status+commands
[6] Seehttps://en.wikipedia.org/wiki/Standard_streamsStandardoutput
[7] https://wiki.aalto.fi/display/Triton/Triton+User+Guide
[8] https://wiki.aalto.fi/display/Triton/Triton+FAQ

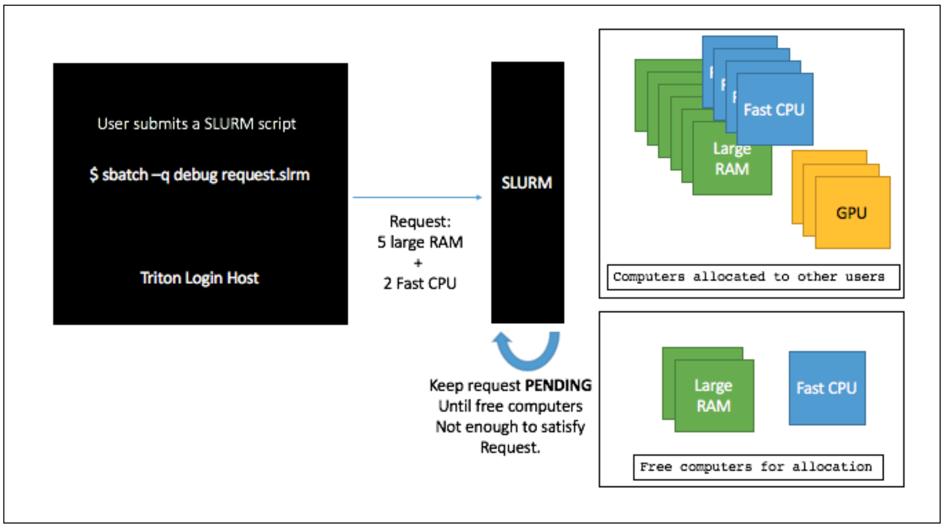


Figure 2: Illustration of the scenario where a user requests SLURM for 5 computers with large memory (RAM) and 2 computers with Fast CPUs. There are specific ways to make requests like these, see details from the User Guide. However, in this example case the request is kept pending by SLURM since there are no free resources which fully satisfy the user's request. **NOTE:** the command `sbatch -q debug` (which would be introduced in the next few steps) requests hosts from the *debug* queue, which has a restriction that jobs in this queue cannot be requested for more than 10 minutes, i.e. its for debugging only.

However since there are quite frequently available new versions of Spark it is useful to understand this approach as well.

The second phase is to launch the Spark master in the Loging host. When launching the master we allocate the ports to master interface and the master web browsing interface. To be able to follow further Spark actions the WebUI should be available in your computer, this you can do via "ssh tunneling". When this is done you can further proceed by sending the SLURM request to get the hosts for the Spark slaves. When sending the request, remember that Triton release your resources as soon as the scrip has been executed or the allocated time is expired. We put our scrip to sleep for a while to keep our slave host alive as long as we allocated time to the Triton hosts.

In the end we show how to run the famous word count example in Triton environment, where the python code for that comes with (Py)Spark package. We use the Spark streaming interface to read the data and feed the data to the python script. Now depending which kind of data you are going to analyse with your Spark implementation it can be either downloaded or read from the file on other server or other data storage system like hadoop. If the data is "small enough" like in our example the network drive is visible to all the host and spark takes automatically care participation of the data to all the slaves. In this simple "word count" example we just use the tiny data in the network drive.

Although, there might be other better ways to setup `Spark` on Triton we found this approach as easiest.

1. Ensure that you are logged into the Triton login host. Now create a folder under the network drive `$WRKDIR` e.g. `$WRKDIR/shotgun-spark`.

2. Begin by first downloading `Spark` from the apache.org website and then extract the archived file inside the folder you created e.g. `$WRKDIR/shotgun-spark/`. In our case we now have the following directory structure `$WRKDIR/shotgun-spark/spark-2.0.1-bin-hadoop2.7`

   - Recall that all the files inside `$WRKDIR` are synchronized between all the Triton computers. So, our newly created folder will be available on all Triton computers now. That's why, in this setup, we don't use a datastore like `HDFS` but make use of `$WRKDIR` as our datastore.

- **NOTE**: Henceforth `$WRKDIR/shotgun-spark/spark-2.0.1-bin-hadoop2.7` is referred to as `$SPARK_DIR`

3. run the `hostname` command to get the *hostname* of the Triton Loging host. We will need it later and refer to it as `<hostname>`

4. We will run our master node in the Triton Loging host and we launch the master node by running the script `$SPARK_DIR/sbin/start-master.sh -host <hostname> -port 7077 -webui-port 8080`

   **NOTE:** It is possible that someone else is using the port 7077 for the master node and the (default) port 8080 for the `Spark` Web-UI, in which case the `start-master.sh` script might fail to start. Try different port number in that case.

   **NOTE:** If the Master node started successfully, you should see a message like

   ```
   starting org.apache.spark.deploy.master.Master,
   logging to /scratch/work/username/spark-2.0.1-bin-hadoop2.7/logs/
   spark-ghoshk1-org.apache.spark.deploy.master.Master-1-login2.int.triton.aalto.fi.out
   ```

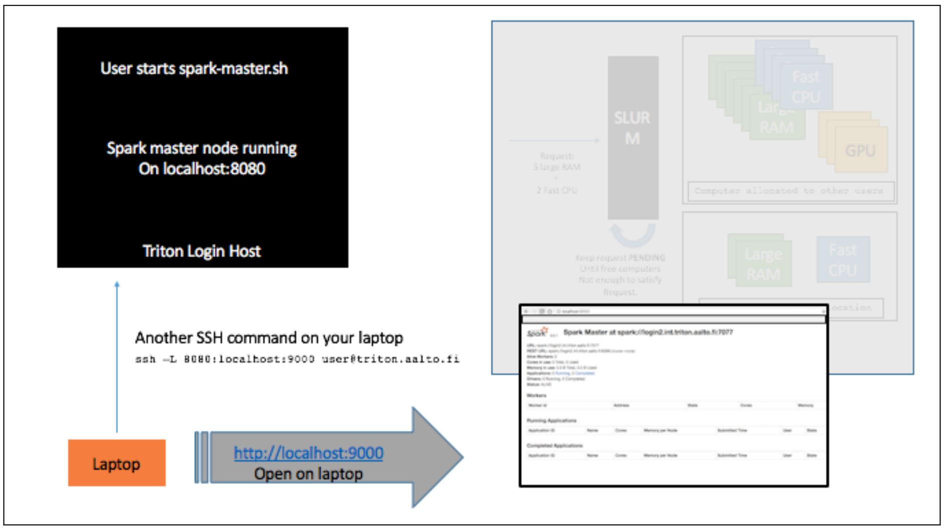
5. We would now want to setup an *SSH-Tunnel* from our laptop or desktop to the Triton login host (Where the `Spark` Master node is running). This will allow us to monitor the Master node and observe when the slaves have connected to the master and are executing jobs etc. From your laptop or desktop run the following command `ssh -L 8080:localhost:9000 <username>@triton.aalto.fi`

   **8080** This is Web-UI port used when setting up the `Spark` Master Node. If you used a different port, replace 8080 with the port you used while starting the Master node.

   **<username>** This is thet `username` you use to log into Triton.

   **NOTE:** If the `ssh` command fails to execute then you should try changing the port 9000 to something else.

6. If the previous step was successful (if the command executed without any error), then you should now be able to view the `Spark`**Master WebUI** by navigating to the url `http://localhost:9000` in any web-browser on the laptop or desktop where you executed the ssh command,in the previous step. *Remember*, if you used a port other than 9000 in the previous step then you should replace 9000 in this step with that same port number. This process is also illustrated in Figure 3.


User starts spark-master.sh
Spark master node running
On localhost:8080
Triton Login Host
Another SSH command on your laptop
Laptop
http://localhost:9000
Open on laptop
SLUR M
Fast CPU
Large RAM
GPU
Spark Master at spark://login2.int.triton.aalto.fi:7077

```
ssh -L 8080:localhost:9000 user@triton.aalto.fi
```

Figure 3: Illustration of the SSH-Tunnel setup.

7. Now, load all the software packages (Triton modules) your job would need to be available for all the hosts. As explained earlier you can search modules available in Triton with *spider* command. In our case we loaded the following:

   - `module load Python/2.7.11-goolf-triton-2016b`
   - `module load numpy/1.11.1-goolf-triton-2016b-Python-2.7.11`
   - `module load scipy/0.18.0-goolf-triton-2016a-Python-2.7.11`

8. Now we will create a `SLURM` script (For more details see[9]) which will request for some Triton hosts from `SLURM` and then launch the `Spark` slave processes on these hosts.

   **NOTE:** These slaves would connect to the `Spark` Master node which we had set up earlier.

   These *Jobs* are send to SLURM in a scrip. For that create a file (say slave-request.slrm) and copy the following lines into it: In previous script we specify the Triton queue to be *patch* queue.

```
#!/bin/bash
#SBATCH -t 00:10:00
#SBATCH -N 2
#SBATCH --ntasks-per-node=1
#SBATCH -p batch

srun bash slave-invoke.sh
```

Listing 2: `slave-request.slrm`

However when you are debugging your scrip the *debug* queue might be faster (but limited with resources and times to use). Next we create another script called `slave-invoke.sh`, which defines the *Job* we are running in each slave.

```
export SPARK_DIR=$WRKDIR/shotgun-spark/spark-2.0.1-bin-hadoop2.7
hostname
sh $SPARK_DIR/sbin/start-slave.sh spark://login2.int.triton.aalto.fi:7077
# the script will keep running for 10 minutes
# (i.e. you will have the slave for 10m)
sleep 10m
# after that kill the slave
sh $SPARK_DIR/sbin/stop-slave.sh
```

Listing 3: `slave-invoke.sh`

In First script `#SBATCH -t 00:10:00` This tell `SLURM` that we want our job to run for 10 minutes. Remember that once `SLURM` allocates hosts to your job, the hosts remain allocated until the time (here 10 minutes) expire, or your script (job) terminates, whichever is first. So you will notice that we make our `slave-invoke.sh` script to `SLEEP` for 10 minutes (using the `sleep 10m` command).

In second script first line we define new variable to save some space later. Seconly we print the hostname and execute the slave starting comand where we give masters position as input argument.

Note we put the second script to sleep for a while (which we want to ensure that SLURM host is available) and after sleep it will kill the slaves (this will make it sure that even we would loose the connection to host of the slave the salve would be terminated after the sleep).

[9] https://wiki.aalto.fi/display/Triton/Running+programs+on+Triton

Ensure that both the scripts `slave-request.slrm` and `slave-invoke.sh` are in the same directory. This is because in the last line of `slave-request.slrm` we execute `srun bash slave-invoke.sh` which expects the slave-invoke script to be present in the same directory as `slave-request.slrm`, if we invoke the script with the full path `srun bash /full/path/to/slave-invoke.sh` then `slave-request.sh` need not be in the same working directory.

9. Now we have to submit this *Job* to `SLURM` from Loging host with command line command:

```
sbatch slave-request.slrm
```

Listing 4: Command to submit a *Job* to SLURM

Note that, on submitting the job you will get a *jobID* remeber this number.
You can now check the status of your request by running the **slurm q** command to see the complete `SLURM` job queue (including your job) or by running `slurm j <jobid>`. You can find more information about managing your jobs on triton here[10]

10. We then test the setup by submiting an example job (here we run the *Word Count* demo from the `Spark` examples folder) by executing the following command(s):

```
export SPARK_DIR=$WRKDIR/shotgun-spark/spark-2.0.1-bin-hadoop2.7
sh $SPARK_DIR/bin/spark-submit \
--master spark://login2.int.triton.aalto.fi:7077 \
--executor-memory 5G \
$SPARK_DIR/examples/src/main/python/wordcount.py \
$SPARK_DIR/LICENSE
```

Listing 5: Command to submit a job to `SPARK`! This submit the job to spark specify the master, the slave memory allocation, python script and the data location (Note: This is not submitted to `SLURM`)

`export SPARK_DIR=$WRKDIR/shotgun-spark/spark-2.0.1-bin-hadoop2.7` This is done to avoid writing long paths.

`spark://login2.int.triton.aalto.fi:7077` The master node is running on this `hostname` (Triton login host) and port 7077 (Use the same `hostname`:port you used while setting up the master node.)

`executor-memory 5G` Executor memory is set to `5 GB` here. As a rule of thumb, allocate execute memory approximately twice[11] your RDD storage needs. Note in our example we only use tiny tex file the allocated memory is way too big, but serve as a example to any practical Spark implementation.

11. The example job should now execute when `SLURM` allocates hosts. The hosts would then be setup as slave nodes, connect to the master node and execute the jobs. You should be able to view the status of your master node (and the slaves / jobs) on your browser by navigating to the `Spark` **Master WebUI** as described in a previous step.

## 5 Running the *Shotgun* algorithm on Triton

The "Shotgun" Spark implementation can be found in Appendix A. We have included necessary comments to the code, so that it would be easy to follow. Here we briefly go through the steps to run it

[10] https://wiki.aalto.fi/pages/viewpage.action?pageId=116668503
[11] http://stackoverflow.com/a/26578005

in Triton. We show that the distributed parallelism follows the results that the paper derives i.e. the the number of iterations drops linearly with the number of parallel calculation [Bradley]. The optimal value $P$ will tell the maximum parallelism that theoretically follow the linear trend.

1. As described in the previous section. Run the master node and request slaves for 30 minutes (You can modify this time based on your needs). You would need to change `#SBATCH -t 00:10:00` to `00:30:00` in `slave-request.slrm` and change `sleep 10m` to `30m` in `slave-invoke.sh`
2. Run the `Spark` script `RDD-impl-1d.py` (See Appendix A) from the `Triton` login host by running the following command. **NOTE:** You might need to change executor memory based on the dataset used.

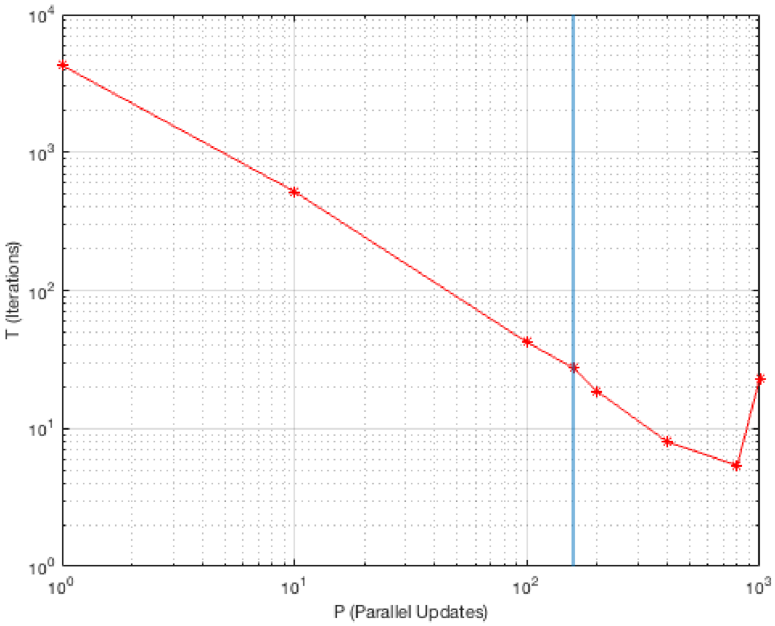


Figure 4: *Shotgun* Implementation in `Spark`. Number of iterations taken for the algorithm to converge (Y-axis) vs The number of parallel updates per iterations (X-axis). We see a same linear trend (in log scale) in the graph as observed by [Bradley]. The blue line corresponds to P = 158 the theoretical `P`-optimal in our case. The above plot is for the `Mug32SinglePixCam` dataset.

3. We would like to point out that, in the script `RDD-impl-1d.py` the data matrix `A` is loaded from a matlab binary file (`Mug32_singlepixcam.mat`) into the memory of one host as a NumPy object. It is then preprocessed before creating an RDD. This is possible only because we had a small dataset. For large datasets (which don't fit into memory on a single machine) you could consider the following steps:
   - Convert the data to a standard text file (or a CSV file) using another script which reads the file line by line (or using Spark or Hadoop).
   - Load the data from the text file into a spark `RDD` (for example using the `sc.textFile(<path to text file>)` function call. Where `sc`is the sparkContext).
   - You might also need to modify `RDD-impl-1d.py` to take into account that the variable `A` is no longer a NumPy array but a Spark RDD.

### 5.1 Background for the algorithm

For reference we have also included a serial version of the Shotgun algorithm implemented in Python and the NumPy library (See Appendix B). Our fist implementation of the algorithm was in Matlab (see Appendix C). We believe the pure Python (with NumPy) and Matlab implementations might be useful for someone trying to understand the algorithm.

The matlab code uses the `parfor` command (parallel for loop) for the $P$ updates in each iteration. In `Spark` there is no special command to run the updates in parallel. Lines 124 to 129 in `RDD_impl_1d.py` (Appendix A) first get the $P$ random indexes (we assume reader's familiarity with the algorithm) which would be updated and then perform the update, the parallelisation is implicit.

The Python (with NumPy) code (Appendix B) doesn't have any parallelisation. This is not a problem, if one just wants to test the algorithm, since here the algorithm's performance is measured in iterations taken to converge to optimal. If the `P` updates in each iteration are in parallel then the total execution time is reduced.

The time comparison of the algorithms were not included here since there were too many computer environmental issues to take care for this project.

## 6 Conclusions and Next Steps

We had a great learning experience in implementing this optimization problem in 3 different programming languages. Although our `Spark` implementation is in pure python expressing the problem in-terms of `Spark RDDs` took us the longest to understand and implement, nevertheless it was educational.

A word of advice for someone starting to implement linear algebra operations with Spark. If you are familiar with Scala the Scala API to spark is a lot more flexible and you should use that. However, if you decide to use the Python API of `Spark` first try to express the algorithm interms of transformations of `RDDs`. We were given this advice (thanks Alex Mara !), but we decided first to experiment with `(Py)Spark` linear algebra library, which costed us a lot of time!

We found the use of `Py(Spark)`'s linear algebra offerings quite limitting, since most often some operation (for example: transpose) can only be performed in one matrix type[12] and there are multiple such matrix types to choose from.

In our experience, we found the Triton computing infrastructure to have a bit of a learning curve, specially when trying to figure out an appropriate workflow for using `Spark` with Triton which is also (conceptually) easy to setup. However, after the initial learning curve the system was quite easy to use. We hope to see more courses make use of the Triton environment !

To conclude, a clear next step to continue this work would be to find a big dataset (where $L_1$ loss minimization makes sense) and test the spark implementation.

[12] https://spark.apache.org/docs/2.0.0/mllib-data-types.html

# 7 Referencies

[Bradley] Bradley, Kyrola, Bickson, Guestrin: Parallel Coordinate Decent for L1-Regularized Loss Minimisation, In the 28th International Conference on Machine Learning, July 2011, Washington, USA

# A Shotgun Algorithm - Spark Implementation

Listing 6: `RDD-impl-1d.py` Spark code implementing the *Shotgun* algorithm using RDDs.

```python
from __future__ import print_function
from __future__ import division

import numpy as np
import scipy.io as sio
import pprint

from pyspark import SparkContext, SparkConf, SQLContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import array, col, explode, struct, lit

def normc(x):
    """
    Normalize such that sum of squares of columns = 1
    """
    return x / np.sqrt(np.square(x).sum(axis=0))

def deltaF(x, AtA, ytA, lamda):
    """
    1st Derivative of LASSO
    """
    retVal = (np.dot(x.transpose(),AtA) - ytA) + lamda;
    return retVal

def F(x,A,y,lamda):
    """
    LASSO function
    """
    return 0.5 * np.square(np.linalg.norm(np.dot(A,x) - y)) + lamda*np.linalg.norm(x,1)

def maxEigenValue(AtA, max_iters=100, eps=10**-2 ):
    """
    Given a symmetric (column major) matrix, returns its
    maximum eigen value using power iteration method.

    params:
    """
    d = AtA.count() # the number of dimensions
    max_eig = None
    v = np.random.random(d)
    for _ in xrange(max_iters):
        # print(v)
        v_new = AtA.map(lambda (AtAi): np.dot(AtAi,v)).collect()
        norm = np.linalg.norm(v_new,2)
        v_new /= norm
        # print(v_new)
        max_eig = norm
        l2_diff = np.linalg.norm(v_new-v,2)
        # print("Debug maxEig : {}, l2_diff : {}".format(max_eig,l2_diff))
        if l2_diff < eps:
            break
        v = v_new

    return max_eig

if __name__ == "__main__":
    data = sio.loadmat("../../data/Mug32_singlepixcam.mat",mat_dtype=True)
    y = data['y']
    A = data['A']

    N,d = A.shape

    # sc = SparkContext(appName='pySparkShotgun', master='local')
    sc = SparkContext(appName='pySparkShotgun')

# In this implementation A fits into memory
    A = normc(A)
    # If A doesn't fit then:
    # 1. Load A as a columnwise (Normalize as you load) RDD
    # 2. Maintain column ids for the next steps.

    # Initialize an empty x
    x = np.zeros(d)
    # Calculating A_newTA_new where A_new = [A, -A]
    # So A_newTA_new is
    # |<-d->
    # | AtA|-AtA |
    # |----+-----|
    # |-AtA| AtA |
    # So we just compute AtA once
    # if column id > d (dimension of AtA)
    # concatenate(-AtA[:,id],AtA[:,id]) columnwise
    # else
    # concatenate(AtA[:,id],-AtA[:,id]) columnwise
    ## aTa = sc.parallelize(xrange(d)).map(lambda cid: np.dot(np.transpose(A[:,cid]),A))

    # Parallelizing column wise
    data = sc.parallelize(xrange(d)).\
            map(lambda cid: (cid,
                            np.dot(np.transpose(A[:,cid]),A),
                            np.dot(np.transpose(y),A[:,cid]))).\
            persist() #here cid is the column index.

    AtA = data.map(lambda (cid, AtAi, ytAi): AtAi)
    max_eig = maxEigenValue(AtA)
    print("Maximum Eigen value : ",max_eig)

    P_opt = int(d/max_eig) # maxeig is the spectral radius (rho) in the paper.
    print("Optimal number of parallel updates : ",P_opt)

    # NOTE: When computing AtA if A doesn't fit into memory we:
    # 1. Compute individual elements of the AtA matrix.
    # 2. Reconstruct/Reduce the elements of the matrix into RDDs of columns.
    # 3. Proceed with the next steps. Remember to maintain the column ids for the next step.

    # Constants
    beta = sc.broadcast(1)
    lamda = sc.broadcast(0.05) # lamda ("lambda") for Mug32_singlepixcam = 0.05

    maxiters = 1000
    print(data.count())

    for i in range(10):
        P_vals=[P_opt]
        # NOTE : YAY !! if P_vals = P_opt/2 (400 iters) then we should take twice the number
        # of iterations as P = P_opt (200 iters). Exactly that happens :D and here P_opt =
            158 (w00t !)
        # and runs fine on my laptop :P
        for P in P_vals:
            x = np.zeros(d)
            for iter in xrange(maxiters):
                randIdxs = np.random.permutation(d)
                randPidxs = randIdxs[0:P]

                # Get the data corresponding to the P indexes.
                data_sub = data.filter(lambda (cid, AtAi, ytAi): cid in randPidxs)
                updates = np.array(data_sub.map(lambda (cid, AtAi, ytAi): (cid,
                    np.dot(x.transpose(), AtAi) - ytAi + lamda.value)).collect())

                idxs = np.array(updates[:,0],int)
                x[idxs] += np.max(np.array([-1*x[idxs], -1*updates[:,1]/beta.value]),axis=0)

                if iter % 100 == 0:
                    print(x[x!=0])
                    # # Following line would change if A doesn't fit in memory
                    print("Iter {} NormVal {}".format(iter, F(np.reshape(x,(d,1)), A, y,
                        lamda.value)))

    # For sanity check otherwise operations don't execute.
    data.collect()
```

## B Shotgun Algorithm - Python + NumPy Implementation

Listing 7: `Shotgun-NumPy.py` Python code implementing the *Shotgun* algorithm using NumPy and SciPy

```
from __future__ import print_function

import numpy as np
import scipy.io as sio
import pprint

# from pyspark import SparkContext
# from pyspark.ml.linalg import DenseVector



def normc(x):
    """
    Normalize such that sum of squares of columns = 1
    """
    return x / np.sqrt(np.square(x).sum(axis=0))

def deltaF(x, AtA, ytA, lamda):
    """
    1st Derivative of LASSO
    """
    retVal = (np.dot(x.transpose(),AtA) - ytA) + lamda * np.sign(x);
    return retVal

def F(x,A,y,lamda):
    """
    LASSO function
    """
    return 0.5 * np.square(np.linalg.norm(np.dot(A,x) - y)) + lamda*np.linalg.norm(x,1)

if __name__ == "__main__":
    data = sio.loadmat("../../data/Mug32_singlepixcam.mat")
    lamda = 0.05 # lamda ("lambda") for Mug32_singlepixcam = 0.05
    y = data[’y’]
    A = data[’A’]

    # sc = SparkContext(appName="Shotgun")
    # y = sc.parallelize(y)
    # A = sc.parallelize(A)

    N,d = A.shape
    x_org = np.zeros((d,1))
    condition = True

    x = x_org
    A = normc(A)
    AtA = np.dot(A.transpose(),A)
    ytA = np.dot(y.transpose(),A)

    # Initialization
    iter = 0
    #P = 20
    # rho = max(eigs(AtA)) # TODO: Implement eigs
    # P_opt = d/rho; # TODO: implement this once we have the rho

    beta = 1 # for LASSO
    x_opt_collection = []
    for i in range(10):
    # for i in range(2):
        # Run shotgun 10 times
        iterations=[]
        # P_Val=[1,2,4,6,8,10,20,30,40,50,60,70,80,90,100,110];
        P_Val=[79];
        print("Nonzero x {}".format(np.nonzero(x)))
        for P in P_Val:
            normVal = []
            iter = 0
            condition = True
            x=np.zeros((d,1))
            print("Nonzero x {}".format(np.nonzero(x_org)))
            while condition:
                iter += 1
                randIdxs = np.random.permutation(d)
                randPidxs = randIdxs[0:P]
                x_subset = x[randPidxs]

                deltaF_vals = deltaF(x,AtA,ytA,lamda)
                # print(deltaF_vals)
                deltaF_subset = deltaF_vals[(0,randPidxs)]
                # print(deltaF_subset)
                # P parallel updates
                # NOTE: Not parallel in this version
                for j in range(P):
                    delta_xj = np.max([-1 * x_subset[j], -1*deltaF_subset[j]/beta])
                    # print("x_subset {} deltaF_subset {} delta_xj {}".format(-1 *
                        x_subset[j],-1*deltaF_subset[j]/beta, delta_xj))
                    x_subset[j] = x_subset[j] + delta_xj

                x[randPidxs] = x_subset

                # normVal.append(F(x,A,y,lamda) - F(x_opt,A,y,lamda))
                condition = iter < 10000
                # normVal.append( (F(x) - F(x_opt2.beta)) /F(x_opt2.beta))
                # TODO: How to calculate x_opt2.beta in this case ?

                if iter % 100 == 0:
                    # print("Iterations {} NormVal {} x-non_zero {}".format(iter, F(x,A,y,lamda), np.nonzero(x.transpose())))
                    print("Iterations {} NormVal {} x_negatives = {} x % nonzeros = {}".format(iter, F(x,A,y,lamda),np.sum(x<0), 100.0*np.sum(x!=0)/len(x)))

            iterations.append({P:(iter,x)})
            print("i {} p {} iter {}".format(i,P,iter))
        x_opt_collection.append({i : iterations})
    pprint.pprint(x_opt_collection)
```

## C Shotgun Algorithm - Matlab Implementation

Listing 8: `Shotgun-Matlab.m` `Matlab` code implementing the *Shotgun* algorithm.

```
% Shotgun : Parallel SCD
clear all;
format long;
load('../../data/Mug32_singlepixcam.mat'); lambda=0.05;
[N,d] = size(A);

x_org = zeros(d,1);
condition = true;

%difference of the lambda that Matlab solver uses and what we use
%Matlab use the division with N for L2 norm part

x=x_org;
%Normalising the data columnwise
A = normc(A);
AtA = A' * A;
ytA = y' * A;

% initialise
iter = 0;
P = 20;
rho=max(eigs(AtA));
P_opt=d/rho;

%One line function call for normalised data
deltaF = @(x) (x' * AtA - ytA) + lambda;
beta = 1;% for LASSO

%weight=sqrt(sum(A.^2))=20.2485 for each column thus instead of vector we
%can use scalar for scaling the result. Otherwise we should normalise to
%the value where L-2 norm of lambda is lambda but weigths are individual
%for the different features now it enoungh to divide the opt_x/20.2485

% Experiment with two optimal values one from matlab lasso implementation
% this is not used because they have a different functional form than
% what is given in our paper.
x_opt = lasso(A,y,'Lambda',lambda);
x_opt2 = solveLasso(y,A,lambda);

% LASSO
F = @(x) 0.5 * (norm(A * x - y).^2) +lambda * norm(x,1);

x_opt_collection = zeros(10, 16);

% run Shotgun 10 times
for index=1:10
    iterations=[];
    % calculate for different number of threads P in each iteration
    % so that when the code finishes we have a set of results
    % which can be plotted. To see if there is a linear decrease in the
    % number of iterations taken to converge.
    P_Val=[1 2 4 6 8 10 20 30 40 50 60 70 80 90 100 110];
    for P=P_Val
           normVal = [];
           iter = 0;
           condition= true;
           % clear x to be zeros(d,1)
           x=x_org;
           while condition
               iter = iter + 1;
               % choose random subset of P indexes from 1..d
               % get the random weights x_i for i in the set P
               randIdxs = randperm(d);
               randPidxs = randIdxs(1:P);
               x_subset = x(randPidxs);

               deltaF_vals = deltaF(x);
               deltaF_subset = deltaF_vals(randPidxs);

               % P parallel updates
               parfor j = 1:P
                   delta_xj = max(-1 * x_subset(j), -1 * deltaF_subset(j) / beta );
                  x_subset(j) = x_subset(j) + delta_xj;
               end
               x(randPidxs) = x_subset;

               % In Fig 2: Y axis is # of iterations taken
               % to reach within 0.5% of true value x*

               normVal = [(F(x) - F(x_opt2.beta))/F(x_opt2.beta) normVal];
               condition = (normVal(1) > 0.005) && iter < 10000;

               % Print value of F(x) every 100 iterations to check for convergence
               % temp = iter;
               if mod(iter,100) == 0
                  fprintf('Iterations Taken = %d NormVal = %f\n',iter,F(x));
               end
           end
       iterations=[iterations iter];
    end
    x_opt_collection(index,:) = iterations;
end
save('x_opt_collection.m', 'x_opt_collection')
```

Listing 9: `solveLasso.m` `Matlab` code for solving the LASSO. Copyright of the original authors (see comment in code).

```
function z = solveLasso( y, X, lambda )
%==========================================================================
%
%             AUTHOR: GAUTAM V. PENDSE
%             DATE: 11 March 2011
%
%==========================================================================
%
%==========================================================================
%
%             PURPOSE:
%
%  Algorithm for solving the Lasso problem:
%
%         0.5 * (y - X*beta)'*(y - X*beta) + lambda * ||beta||_1
%
%  where ||beta||_1 is the L_1 norm i.e., ||beta||_1 = sum(abs( beta ))
%
%  We use the method proposed by Fu et. al based on single co-ordinate
%  descent. For more details see GP's notes or the following paper:
%
%  Penalized Regressions: The Bridge Versus the Lasso
%  Wenjiang J. FU, Journal of Computational and Graphical Statistics,
%  Volume 7, Number 3, Pages 397?416, 1998
%
%==========================================================================
%
%==========================================================================
%
%             USAGE:
%
%             z = solveLasso( y, X, lambda )
%
%==========================================================================
%
%==========================================================================
%
%             INPUTS:
%
%      =>     y = n by 1 response vector
%
%      =>     X = n by p design matrix
%
%      => lambda = regularization parameter for L1 penalty
%
%==========================================================================
%
%==========================================================================
%
%             OUTPUTS:
%
%      =>     z.X = supplied design matrix
%
%      =>     z.y = supplied response vector
%
%      => z.lambda = supplied regularization parameter for L1 penalty
%
%      =>  z.beta = computed L1 regularized solution
```

```
%
%========================================================================
%
%========================================================================
%
%      Copyright 2011 : Gautam V. Pendse
%
%      E-mail : gautam.pendse@gmail.com
%
%      URL : http://www.gautampendse.com
%
%========================================================================

%========================================================================
%             check input args

if ( nargin ~= 3 )
       disp(’Usage: z = solveLasso( y, X, lambda )’);
       z = [];
       return;
end

% check size of y
[n1, p1] = size(y);



% is y a column vector?
if ( p1 ~= 1 )
   disp(’y must be a n by 1 vector!!’);
   z = [];
   return;
end

% check size of X
[n2,p2] = size(X);

% does X have the same number of rows as y?
if ( n2 ~= n1 )
   disp(’X must have the same number of rows as y!!!’);
   z = [];
   return;
end

% make sure lambda > 0
if ( lambda < 0 )
   disp(’lambda must be >= 0!’);
   z = [];
   return;
end

% get size of X
[n, p] = size(X);
%========================================================================

%========================================================================
%             initialize the Lasso solution
% This assumes that the penalty is lambda * beta’*beta instead of lambda * ||beta||_1
beta = (X’*X + 2*lambda) \ (X’*y); %this gives problems
%beta = zeros(p,1);
%========================================================================

%==========================================================================
%             start while loop
% convergence flag
found = 0;
% convergence tolerance
TOL = 1e-6;
while( found == 0 )
    % save current beta
    beta_old = beta;
    % optimize elements of beta one by one
    for i = 1:p
        % optimize element i of beta
        % get ith col of X
        xi = X(:,i);
        % get residual excluding ith col
        yi = (y - X*beta) + xi*beta(i);
        % calulate xi'*yi and see where it falls
        deltai = (xi'*yi); % 1 by 1 scalar
        if ( deltai < -lambda )
            beta(i) = ( deltai + lambda )/(xi'*xi);
        elseif ( deltai > lambda )
            beta(i) = ( deltai - lambda )/(xi'*xi);
        else
            beta(i) = 0;
        end
    end
    % check difference between beta and beta_old
    if ( max(abs(beta - beta_old)) <= TOL )
        found = 1;
    end
end
%==========================================================================

%==========================================================================
%   save outputs
z.X = X;
z.y = y;
z.lambda = lambda;
z.beta = beta;
%==========================================================================
end
```