University of Glasgow | School of Computing Science

Exploring the possibility of accelerating Hurricane Simulator using Apache Spark framework

Andrej Hoos

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 19, 2016

Abstract

The bestest abstract

Education Use Consent

Contents

Chapter 1

Introduction

Simulating real world weather system is a difficult task, requiring a lot of computational power. One such simulator is The Large Eddy Simulator for the Study of Urban Boundary-layer Flows (LES) developed by Hiromasa Nakayama and Haruyasu Nagai at the Japan Atomic Energy Agency and Prof. Tetsuya Takemi at the Disaster Prevention Research Institute of Kyoto University[2][3]. It simulates meteorological systems in an urban environment with building resolution. Originally LES is implemented in Fortran 77 with no explicit parallelization.

1.1 Motivation & Aim

As multi core computational devices are now commonplace, parallelisation has become a important consideration when implementing simulations. Therefore LES has been ported to Fortran 95/OpenCL implementation by Wim Vanderbauwhed[7]. Using this implementation, LES can be run parallelised on any OpenCL compatible computational device, which shows significant improvements in speed. While these improvements can reduce the time needed to run single simulation run or increase the size of the spatial domain of the simulation, there are limits to both. One of these limits is number of computational units the device has, limiting the extent of parallelization. The other is memory available on the device.

Using a network cluster of computers can, in theory, help overcome these limits to some extent. However, splitting work onto multiple nodes in a cluster introduces a issue of communicating important information over network. Specifically, in case of LES, data that is at the edge of the spatial domain of each node needs to be exchanged in so called halo exchanges.

Recently, there has been a rise in a new computational frameworks focused on working in distributed environment. One of these frameworks is Apache Spark which is based on the MapReduce parallel execution paradigm.

The aim of this report is to explore the possibility of using the Apache Spark in Java to parallelize LES, or any algorithm relying on halo exchanges in network distributed environment.

1.2 Outline

Further background will be described in chapter 2, including information on LES, Apache Spark and OpenCL implementation used. Implementation of halo exchanges in Apache Spark will be discussed in chapter 3 and chapter 4 will explain the process of porting LES code from Fortran to Java. Evaluation of the achieved performance can be found in chapter 5 and chapter 6 contains final remarks and notes about future work.

Chapter 2

Background

2.1 Large Eddy Simulator

The Large Eddy Simulator for the Study of Urban Boundary-layer Flows developed by Disaster Prevention Research Institute of Kyoto University is a high resolution meteorological simulator designed to model turbulent flows over urban topologies. It achieves this by using mesoscale meteorological simulations and building resolving large-eddy simulations. It also uses The Weather Research and Forecasting Model [1] to compute the wind proile as an input for the simulation. The simulation is split into multiple subroutines, however, the most important one is the Poisson Equation solver. This is the most computationally intensive subroutine and it uses successive over relaxation to solve a linear system of equations. Each time step of the simulation needs to execute following subroutines in a sequence:

velnw	Update velocity for current time step
bondv1	Calculate boundary conditions (initial wind profile, inflow, outflow)
velfg	Calculate the body force
feedbf	Calculation of building effects
les	Calculation of viscosity terms (Smagorinsky model)
adam	Adams-Bashforth time integration
press	Solving of Poisson equation (SOR)

LES runs over spatial domain represent by a 3D grid. Size of 150x150x90 is used for the most of this report, but the size can be varied in x and y direction. By increasing the size of the grid in x and y directions, LES can be run with higher resulution, or it can used to cover larger area. However doing this increases the time needed to complete the simulation and more importantly there is a limit to how big the grid can be so that it fits inside the RAM.

2.1.1 Paralellisation

Using a cluster of computers, we can split the domain such that every node in the cluster runs LES on a small portion of the domain. This allows to run LES on much larger domain than what would be possible on a single computer. To run LES correctly in this scenario, each node needs access to the edge data belonging to each of its eight neighbours (Figure 1).

[1] `http://www.wrf-model.org`

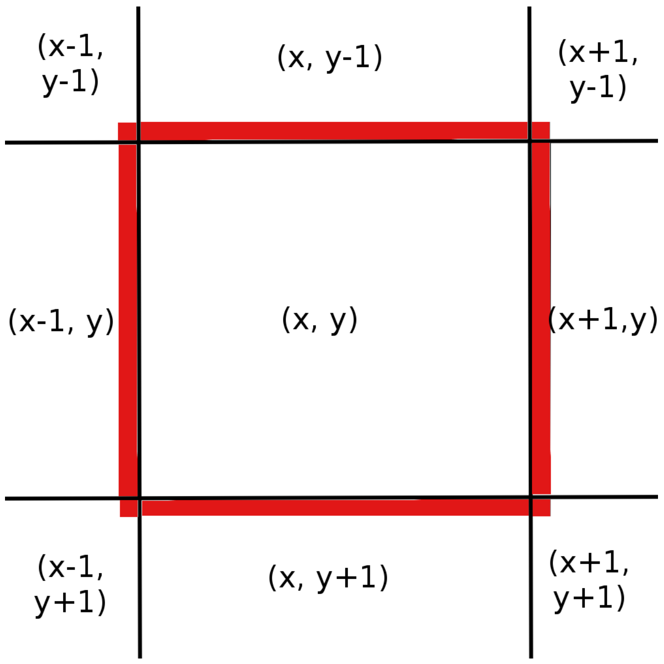

Figure 1: Node (x,y) needs access to the edges of all its neighbours

Message Passing Interface (MPI) [2] is considered to be most commonly used for communication in distributed memory environments. Because of this, LES has already been parallelised using MPI by Gordon Reid as his MSci project at University of Glasgow[5] and others[4]. Although MPI is considered standard and achieves good performance, it isn't easy task to parallelise algorithms using MPI, especially for scientists with minimal experience with Computing Science. Therefore it is important to investigate the possibilty of using frameworks with higher level abstraction and that are easy to deploy onto a standard network cluster.

2.2 MapReduce parallel processing

MapReduce is a programming model used mainly for processing large quantities of data and is intended for network clusters[1]. Program using this model consists of Map and Reduces methods which are automatically paralellised. MapReduce was originally name of a technology by Google, but is now used for the model in general. MapReduce libraries and frameworks are now available in many programming languages.

MapReduce model usually operates on key-value pairs and can be described by three main steps:

- **Map**: **map()** function is applied to all of the pairs, creating a collection of new key-value pairs
- **Shuffle**: the output data is redistributed accross network such that pairs with the same key are located on the same node
- **Reduce**: the output data is processed by applying the **reduce()** function to the output pairs

Advantage of using MapReduce is that only map() and reduce() functions need to be written and the shuffle step is handled automatically by the library or framework and is usually optimised. In addition, most MapReduce frameworks provided more data manipulation methods such as grouping or joining.

[2] `http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm`

2.2.1 Apache Spark

Apache Spark[3] is one of these frameworks and is available Java, Scala, Python and R. It claims to be faster over alternatives such as Apache Hadoop, because it allows MapReduce to run in memory as opposed to reading and writing inputs and outpus solely from disk. This, with combination that is the increasing community and high-level abstraction makes this framework suitable for the purposes of this project.

TODO: Describe main points about how Spark works (master, worker, driver, RDDs...)

2.3 OpenCL

TODO: OpenCL background[4]

2.3.1 Aparapi

TODO: Aparapi background [5] [6]

[3] `http://spark.apache.org/`

[4] `https://www.khronos.org/opencl/`

[5] `http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/`

[6] `https://aparapi.github.io/`

Chapter 3

Halo exchanges in Spark

The main challenge of this project is to construct viable halo exchange algorithm, using data manipulation methods provided by Apache Spark. This has not been done before, although one work[8] has used Spark to solve a stencil problem computation. In stencil problems, the problem is divided into cells, and to compute one cell, information about its neighbours are needed. However, only simple method using broadcast variables (covered later in the chapter) was used in this work.

Multiple possible algorithms were considered and their implementation details, advantages and drawbacks are discussed in this chapter.

3.1 Game of Life as a proof of concept

To test the proposed alorithms, a implemetation of Conway's Game of Life was used. This was because of its simplicity and becasue it resembles the simulator in that in needs data from neighbours to be transfered in order to be parallelised in distributed environment.

The Game of Life is a cellular automaton and is run on a 2D grid of cells. To compute the transition of a single cell, state of each of its eight neigbours are needed. The transitions are defined as:

- Any live cell with fewer than two live neighbours dies.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies.
- Any dead cell with exactly three live neighbours becomes a live cell.

We can group the cells into same size rectangle shaped chunks and compute the transitions of each chunk on a separate node of the cluster. However, now the nearest edges of neighbouring chunks need to be exchanged before each transition, in order for the edge cells to know the state of all of their neighbours. This is the same case in the LES and is therefore suitable as a proof of concept.

3.2 Broadcast variables

The simplest way exchange halos is using broadcast variables. Spark allows to create a broadcast variable in the driver program, which is the shipped to worker nodes if needed. However the limitation is that only the driver program can create broadcast variables (i.e. they cannot be created in the worker code) and therefore the workers would need to send their edge data to the driver program, which would contruct the appropriate halos and create a broadcast variable(TODO: diagram).

This means that the halos would be transferred to their destination always via the node which hosts the driver program, and therefore the algorithm would do twice as much network transfers as the ideal case where the halos are transferred to their destination directly. Therefore this approach was deemed unsuitable.

3.3 MapReduceByKey

Another way is to use a combination of map and reduce steps (TODO: diagram). To represent the chunks, we use Spark Pair Collection of pairs of key and chunk (`JavaPairRDD<String, Chunk>`). The key represents the coordinates of the chunk.

In the first stage we transform this collection into another collection which contains the chunks from the original collection together with all neighbours using map (`flatMapToPair`). The keys of the neighbours key value pairs are the keys of their respective target chunk.

The second step then does reduce on this new collection using `reduceByKey`. The reducer function takes a chunk and a neighbour and reduces them into a chunk with the neighbour added to it, which results in a collection of chunks which are larger by one cell on each side.

The last step is a map step, which computes next transition on each chunk, returning collection of transformed chunk which are the original size.

There are, however, some drawbacks to this approach. The first is, that we cannot control the exact order in which these operations are executed. For example, Spark might decide to execute the first map step on first two chunks (assuming there are more than two) and run the reduce step on the results. Then, the first map step might be run on the rest of the chunks. The second issue is that we cannot control, on which node is each step executed. This two issues result in all of the chunks and not only neighbours are transferred over network, which is unacceptable for the purpouses of this project.

3.4 MapGroupByKey

To resolve one of the issues mentioned above, it is possible to replace the reduce step with group step (TODO: diagram). This uses `groupByKey` to group chunk together with neighbours that share the same key. The grouping function takes the chunk with its neighbours and combines them into a single chunk, such that the result is the same as after the reduce step in previous section.

Using this method the issue of the order of execution is solved, since the group can be executed only after all of first map stage has been executed. The second issue of not being able to control where is each stage executed still remains and therefore the issue of transferring all chunk over network remains as well.

3.5 MapCogroup

Method described in this section was inspired by chapter 4 in *Learning Spark*[9]. It uses custom partitioner and two separate collections for chunks and neighbours to prevent Spark from moving chunks around the network. It uses combination of map and cogroup steps.

The first map operates on the same key value pair collection as above. However, now it transforms the collection into separate key value pair collection of neighbours, instead of creating one mixed one as was done previously (TODO: diagram).

In second step, the `cogroup` transformation is very similar to the grouping transformation from the MapGroupByKey method, with the difference that in this case it operates on two collections instead of one. The result is the same as in previous sections.

In addition, the collection containing chunks is partitioned using custom partitioner. The partitioner takes the key, which represents coordinates and outputs index of node on which the chunk should be physically present. Using this partitioner spark shuffles only the other collection, which contains neighbour data, over network and keeps the chunks collection in place.

3.6 Caching and checkpointing

Spark uses lazy execution when running programs stages. This means that transformation on a collection are saved in its lineage are executed only after a call that materialises the collection on the driver program is called on it. Examples of such calls are `count` that returns the number of items in the collection, `collect` which returns array of all items in the collection or `reduce`, which, as opposed to `reduceByKey` reduces the collection into single item.

This introduces some issues. First issue is that often, when resolving stages that do network shuffle, Spark executes the parent stages more than once. This is considered standard Spark behaviour and is solved by caching the parent collection using `cache` method, which also marks the collection to be explicitly reused when executing, child stages. However, it was found that Spark keeps more cached collections than necessary, and therefore workaround was devised. Using this workaround, first, new collection is created, then cached and materialised using `count` and then `unpersist` is called on the old collection to remove it from the cache.

Another issue is lineage length. In iterative algorithms, such as the ones described in this report, the lineage that Spark keeps in order to be able to restore execution in the case of failure can get quite long. This results in significant slowdown of Spark task resolution and eventually leads to stack overflow errors. One possible solution is to increase JVM maximum stack memory, but the standard practice is to `chekpoint` collections. This serializes the collection to disk and forgets its lineage.

Chapter 4

LES in Java

The second challenge this chapter deals with is reimplementing LES in Java. To do this, the OpenCL host code written in Fortran needs to be rewritten Java. This code consists mainly of a `switch` construct, which executes different routines in the OpenCL kernel. Next methods for initialising variables, constructing and deconstructing halos need to be created. Finally, the newly written Java OpenCL host code needs to be integrated with Spark.

4.1 JNA with Fortran

LES works with about 40 different variables that store states of diferent systems, wind velocity, pressure, halos and grid constants. These variables need to be initialised before the ex ecution of the simulation. This process is covered in a substantial part of the Fortran source code.

Since accessing the existing Fortran subroutines is a viable strategy [1], it was decided to write a simple Fortran subroutine, which groups all subroutines that initialise the needed variables. The Fortran code is compiled into a shared library, and can be accessed from Java using Java Native Access [2].

4.2 Aparapi for Unconventional Cores

To write the OpenCL host code Aparapi for Unconventional Cores (Aparapi-ucores) [6] [3] is used as an underlying library that facilitates communication with the OpenCL kernel. It is a fork of Aparapi library that allows running OpenCL code on unconventional cores such as FPGAs. In addition it allows running OpenCL kernels from precompiled binaries.

Aparapi allows to write Java code which is then translated into OpenCL kernel code and executed. However, since the LES kernel code is already written, this is counterproductive. It is impossible to disable this behaviour and therefore a workaround is neccessary. Although, with binary flow, which is provided by Aparapi-ucores it is possible to execute compiled binary of the provided kernel instead of kernel generated by Aparapi, Aparapi will ecpect the same method signature as the one the generated kernel has. Therefore it is needed to provide Aparapi with dummy Java code, that translates into correct OpenCL method signature. This can be achieved by using all parameters in correct order, the parameters need to have correct datatype and parameters that are supposed to be

[1] `http://www.javaforge.com/wiki/66061`
[2] `https://github.com/java-native-access/jna`
[3] `https://gitlab.com/mora/aparapi-ucores`

written to by kernel need to be marked as such by assigning a value into them. Example dummy code is in the listing below:

```java
float[] p2;
float[] uvw;
float[] uvwsum;
float[] fgh;

@Override
public void run() {
  float sum = p2[0] + uvw[0] + uvwsum[0] + fgh[0];
  uvw[0] = sum;
  uvwsum[0] = sum;
}
```

The corresponding kernel method signature as generated by Aparapi:

```c
__kernel void run(
   __global float *p2,
   __global float *uvw,
   __global float *uvwsum,
   __global float *fgh,
   int passid
)
```

In the example above, four kernel parameters are used. All are arrays of floats. Note that the order in which they are listed in the sum in the first line of the `run` method, is the same as the order of the paramters in the signature. In addition the `uvw` and `uvwsum` variables were assigned into, which marks them as kernel-writable inside Aparapi. Also, important to note, is final parameter `passid`, which is being automatically inserted by Aparapi, and contains the id of the current kernel execution.

Due to the nature, there are some modification that need to be done to the OpenCL kernel, before compiling into a binary. First, the `passid` needs to be added to the parameter list. Second, Aparapi does not support passing non-constant scalar fields into the kernel as paramter. As a workaround to this, scalar fields can be represented as simple single item 1D arrays. Finally the name of the kernel method is restricted to the name `run`. As of writing of this report, however, this was found not to be true and it is possible to use different names, but the written code does not reflect this.

By default, Aparapi copies all parameters to and from kernel before and after every kernel execution. This is due to large number of parameters in LES undesirable. Therefore, for iterative applications, it is recommend to enable explicit Aparapi mode, which then requires the programmer to call `put` and `get` methods to explicitely write and read parameters to and from the kernel.

4.2.1 Device specific OpenCL binaries

Since OpenCL is supposed to be compatible with wide range of devices, the same OpenCL code can be executed on any OpenCL device. This is not true, however, for the compiled binaries, which are device specific. This means that new binaries need to be created whenever the simulation is to be run on a new device. There is no tool available that allows easily compile kernel into binaries, but OpenCL specification describes a method that allows to retrieve the compiled binary code [4]. Therefore it was decide to create a simple compilation tool [5] which allows the user to specify a `.cl` kernel file and device with which to compile the kernel file.

[4] `https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clGetProgramInfo.html`
[5] `https://github.com/adikus/cl-compile`

4.2.2 Memory alignment

Aparapi-ucores contains functionality which copies the contents of the parameters into a memory aligned location. This is done because some devices such as FPGAs do not interact well with unaligned memory locations, and locations in JVM heap might not be aligned. This functionality is currently experimental, and allocates new memory for every kernel-writeable parameter every kernel execution. In addition it copies the content of parameters from the original JVM memory location to this new location and back every kernel execution for every parameter. This results in a significant slowdown in execution, which is undesirable.

This functionality can be turned of, since in theory, using unaligned parameters should not pose any issues when runnning the kernel on CPUs and GPUs. However, after switching this functionality it was found, that the kernel no longer runs properly on Intel CPU. The reason remains, as of writing of this report, unknown, and therefore it was decided to modify the underlying Aparapi-ucores C++ source code to create a workaroad that deals with the excesive memory copies. The workaround allocates the aligned memory location only once and copies the memory from or to it only when the `get` or `put` Aparapi methods are called.

4.3 Halo construction and deconstruction

Since LES has already been parallelised using MPI[5], functionality for reading and writing halos is already implemented inside the OpenCL kernel code. This format is a single 1D for each halo and therefore needs to be deconstructed before sending and constructed again after receiving.

Since the spatial domain in this project is considered to be wrapped such that bottom connects to top and left connects to right, this halo construction and deconstruction can be demostrated on a single node (TODO: diagram).

4.4 LES in Apache Spark

As a final part of the project implementaion, working LES Aparapi implementation needs to be integrated into Spark. MapCogroup described in chapter 3 is used as the halo exchange algorithm.

Each time step of the simulation is divided into 11 steps, with each step executing OpenCL kernel once. The 8th step (`PRESS_SOR`) is a exception. In this step the successive over relaxation is run, which requres multiple iterations of kernel executions. Normally the over relaxation is run until the simulation state converges, but in this project it was decided to use constant number of 50 iterations.

Each kernel execution needs to be accompanied be a halo write beforehand and halo read afterwards, both of which are an extra kernel execution. After each group of halo write, simulation step and halo read the appropriate halos should be exchanged over network. In addition some stages do a reduction over the whole domain, which means that a reduction over the nodes needs to be done using Spark.

Table 1 contains the summary of each stage.

#	Stage	Halo exchange, reduction
1	VELNW__BONDV1_INIT_UVW	p, uvw and uvwum halo exchange
2	BONDV1_CALC_UOUT	uvw halo exchange
3	BONDV1_CALC_UVW	uvw halo exchange
4	VELFG__FEEDBF__LES_CALC_SM	uvw, uvwsum, fgh, diu and sm halo exchange
5	LES_BOUND_SM	sm halo exhange
6	LES_CALC_VISC__ADAM	fgh and fgh_old halo exchange
7	PRESS_RHSAV	rhs and fgh halo exchange + reduction
8	PRESS_SOR	p halo exchange, there are three kernel executions for each SOR iteration, which means three halo exchanges per iteration + one reduction per iteration to determine the convergence value
9	PRESS_PAV	p halo exchange + reduction
10	PRESS_ADJ	p halo exchange
11	PRESS_BOUNDP	p halo exchange

Table 1: Summary of halo exchanges and reductions for each LES stage in one time step

Spark collection that contains the kernel hosts consists of key value pairs. The key is represented by an integer which has value between 0 and X*Y-1, where X and Y are the size of the node grid. For example, if X is 2 and Y is 3, it means we intend to run the simulation on cluster with 6 worker nodes, and each node is running a simulation on a 150x150x90 size domain. The value in the key value pair is represented by a subclass of Aparapi Kernel class.

The neighbours are represent by a Neighbour class which encapsulates the underlying data array and the direction from which it came (N,S,W,E,NW,NE,SW,SE).

Finally, helper methods have been created for executing stage of simulation, exchaning halos and reductions. This means that a execution of a single simalation stage can be written similarly to:

```
// 7th stage - PRESS_RHSAV
kernels = executeKernelStep(kernels, States.PRESS_RHSAV);
kernels = HaloExchanger.exchangeHalos(kernels, "rhs,fgh", ip, jp, kp, X, Y);
kernels = divisionReduction(kernels);
```

4.4.1 Lineage & Checkpointing

The fact that single timestep of the simulation needs to be represented by large number of jobs (10 stages * 2 [execution + halo echange] + 50 SOR iterations * 3 * 2 = 320; this is just an approximation) means that lineage grows very quickly. With the recommended rate of chekpointing, which is once per 100 jobs, three checkpoints are created every time step of the simulation. Since one time step should not take longer than couple of seconds, this means that large quantity of data is serialized to disk over the run of program.

This can create problems when trying to excute the simulation in environment with restricted disk space. Therefore another workaround was devised, which all previous checkpoints before creating a new one. Checkpoints in Spark are used to recover from a crash of a node or a similar scenario. However in our case recovering is not as simple as deserializing a class from disk. Further work would need to be made to initialise the OpenCL kernel and load the recoverd state of parameters into it, and was deemed to be out of the scope of this project. Therefore the checkpoints are completely useless and can be deleted without worry.

Chapter 5

Evaluation

TODO: describe the setup, architecture, tests done, results, one or two bar charts..., easy stuff, skipping for now

5.1 Architecture

[1]

5.2 Baseline tests

5.3 Spark tests - increasing domain size

[1]`http://ark.intel.com/products/75267/Intel-Xeon-Processor-E5-2640-v2-20M-Cache-2_00-GHz`

Chapter 6

Conclusion

6.1 Future Work

6.2 Summary

Appendices

Appendix A

Running the simulation

The project files that are required to run the parallelised version of LES can be found on GitHub. [1]

TODO: Proper instructions, should be mirrored on GH

[1] `https://github.com/adikus/hurricaneProject`

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. `http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf`, 2004. [Online; accessed 13/03/2016].

[2] Hiromasa Nakayama, Tetsuya Takemi, and Haruyasu Nagai. Les analysis of the aerodynamic surface properties for turbulent flows over building arrays with various geometries. *Journal of Applied Meteorology and Climatology*, 50(8):1692–1712, 2011.

[3] Hiromasa Nakayama, Tetsuya Takemi, and Haruyasu Nagai. Large-eddy simulation of urban boundary-layer flows by generating turbulent inflows from mesoscale meteorological simulations. *Atmospheric Science Letters*, 13(3):180–186, 2012.

[4] Siegfried Raasch and Michael Schrter. PALM A large-eddy simulation model performing on massively parallel computers. *Meteorologische Zeitschrift*, 10(5):363–372, 2001.

[5] Gordon Reid. Exploring the parallelisation of the Large Eddy Simulation using MPI and the Glasgow Model Coupling Framework, 2015.

[6] O. Segal, P. Colangelo, N. Nasiri, Zhuo Qian, and M. Margala. Aparapi-ucores: A high level programming framework for unconventional cores. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6, Sept 2015.

[7] Wim Vanderbauwhede. Model Coupling between the Weather Research and Forecasting Model and the DPRI Large Eddy Simulator for Urban Flows on GPU accelerated Multicore Systems. `http://arxiv.org/abs/1504.02264`, April 2015. [Online; accessed 12/03/2016].

[8] Yuzhong Yan, Lei Huang, and Liqi Yi. Is apache spark scalable to seismic data analytics and computations? In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2036–2045, Oct 2015.

[9] Matei Zaharia, Patrick Wendell, Andy Konwinski, and Holden Karau. *Learning Spark*. O'Reilly Media, Inc., February 2015. [Online; accessed 18/03/2016; `https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/`].