Project Summary

We were presented with a ferrocrystal simulation developed in Matlab. The computationally intensive simulation had a highly parallel structure, making it perfect for distribution. Because of the massively parallel structure of the simulation, we attempted to distribute the simulation. Unfortunately, Matlab only allows users to parallelize on a basic license, and caps the number of parallel workers at twelve. In addition, in order to distribute to additional nodes, the user must buy a separate license for several thousand dollars, and still can only use a maximum of 256 workers. As such, we decided to port the simulation into a more open language which allowed for infinitely scalable distribution.

After some research, we decided to use Spark as our distribution framework, leaving us with two languages to code in. After a successful proof-of-concept implementation with Python, we moved to Spark's native language, Scala, hoping to further improve performance. Unfortunately, Scala's poor linear algebra support meant our implementation slowed to a degree that distribution was unable to make up for. In response, we moved back to Python for access to Numpy, a widely used and powerful linear algebra library. Our new Python implementation had single node performance on par with or better than Matlab, and we saw further performance increase from distribution.

Implementation

Our simulation can be broken into three basic parts: the setup of the crystal grains, the calculation of the polycrystal's resting state, and the calculation of the piezo coefficients that describe its response to mechanical stress and electric field variation. The setup portion takes the smallest amount of time, and relies on various input values to determine the mechanical and electrical state of each of the grains of the polycrystal. In our first version of the simulation, the data for each grain was stored relative to its own orientation, and had to be rotated into the orientation of the polycrystal at each later step of the simulation. This version relied heavily on for loops, creating a lot of processing overhead which slowed down our initial implementation in Python (more on that later). In our later versions, the grain data was stored in relation to the orientation of the entire crystal, eliminating the need for rotation and allowing for greater vectorization of necessary calculations. This slightly increased the setup overhead of the simulation, but greatly increased overall performance.

The second part of the simulation repeatedly implements a function, "polyfunc," to determine the resting state of the polycrystal. The polyfunc takes the grain data as input, along with an applied mechanical stress and electric field, and outputs the adjusted grain data along with the induced piezoelectric response of the polycrystal. To obtain this data, the function repeatedly iterates over each grain of the polycrystal, adjusting its state based on the applied stress and field until the grains converge to a stable state, and then averages the state of each grain to obtain the overall response of the polycrystal. This method is described in THE PAPER WE BASED MODEL OFF OF, and uses up most of the computation time of the simulation. However, the second part of the simulation had little opportunity for distribution, as the state of the polycrystal needs to be updated with each iteration of the polyfunc, preventing multiple workers from working simultaneously on the polycrystal data.

Most of the distribution happens in the third part of the simulation, which once again relies on the polyfunc to determine the piezoelectric response of the polycrystal across a variety of stress and electric field vectors. Using on some range of stress and electric field values, the simulation runs polyfunc multiple times on the same poly crystal data, providing a good

opportunity for parallel workflow. First, all possible combinations of stress and electric field values are determined, and Spark creates a separate partition for each of these combinations. Next, the Spark master broadcasts the polycrystal data to each of the workers and dispatches a different combination to every available worker. When all of the workers finish, the data is combined and output as a tensor which contains the piezoelectric response coefficients of the polycrystal.

Process and Issues During Implementation

We initially decided to distribute with Apache Spark as it was a well developed and supported framework for distribution of a wide variety of tasks. After deciding on a framework, we chose to use Python for our initial implementation as it was easy to port the code from Matlab. Our initial Python implementation outperformed Matlab on sixteen nodes, but ran much slower than Matlab without distribution, which we originally blamed on Python status as an interpreted language. However, after our later implementations, we realized it was because of our simulation's lack of vectorization and heavy reliance on for loops for the rotation of the polycrystal grains. Because of our assumption that Python's slowness came from its interpretability, we attempted to port the simulation into Spark's native language, Scala.

Porting the simulation into Scala proved much more difficult, largely because of the lack of a competitive linear algebra library. Our implementation used Nd4j, an attempt at a Numpy-like library for jvm languages. Nd4s (Scala version of Nd4j) purported to have comparable speeds to Numpy, made use of the JNI for back-end matrix math optimization, and supported manipulation of n-dimensional matrices, making it an attractive pick to handle our simulation's needs. Indeed, Nd4s's performance was comparable to Numpy and Matlab for individual matrix math, so we moved forward with a port to Scala. However, we ran into two main issues with the Scala simulation.

The first was a problem with generalizability. Ultimately, we hoped to extend our simulation to a more generalizable framework for distribution of massively parallel scientific simulations using spark, but since Scala is a strongly-typed language, generalization was much harder than with a language like Python. Nd4s required much more specific manipulation of the matrices, and a lot of extraneous data had to be stored and passed between functions to retain functionality of the simulation. While we could maintain the general structure of our python implementation, the functions that were eventually distributed and the structures used to store data had to be clearly declared and were limited to manipulating matrices of numerical data.

The second was a problem with the flexibility and usefulness of Nd4s. The most glaring issue with Nd4s was its lack of splicing capabilities. In Matlab and Numpy, accessing a portion of an array is simple, as is adding or removing additional data to the middle of an array. In Nd4s, in order to access splices of data, a new array must be initialized and the desired data copied over. Obviously, this led to a significant performance decrease in terms of memory efficiency and speed. Furthermore, by this time we had moved away from our initial implementation that required constant rotation of the grains, and vectorized most of our calculations. Matlab and Numpy both allow for optimized matrix-wise multiplication of n-dimensional arrays (implemented in Matlab with the mtimesx library and in Numpy with the einsum function), whereas Nd4s required us to iterate through the array and multiply each matrix individually. While the

performance for each individual matrix was comparable to Matlab, when multiplying n-dimensional matrices Nd4s was more than eighty times slower.

Because of a lack of comprehensive linear algebra libraries in Scala, we realized that any gains from distribution would be lost because of the slowness of Scala's computation. As such, we decided to move our vectorized simulation back to Python, to see if we could outperform Matlab using Numpy's more powerful linear algebra methods. Porting the simulation to Python was much simpler than Scala, and even python's interface with Spark was easier to use. Much of the syntax of Numpy is extremely similar to that of Matlab, allowing for simple translation of the code. Since Python is easy to read and write, a generalized model for distribution using Python would be accessible for scientists who want to distribute a simulation with a powerful framework like Spark.

Spark, as a distribution framework, was also exceedingly easy to use. The cluster we ran off of used Slurm as its own dispatcher, but Spark allowed us to run a virtual server using its own framework on top of the Slurm scheduler. This was achieved by starting a Spark master on a single node, and a number of workers on multiple other nodes, and then submitting jobes directly to the master to complete. Spark's ability to run as a virtual framework on top of existing schedulers and distribution frameworks allows it to be generalizable to many different cluster frameworks.

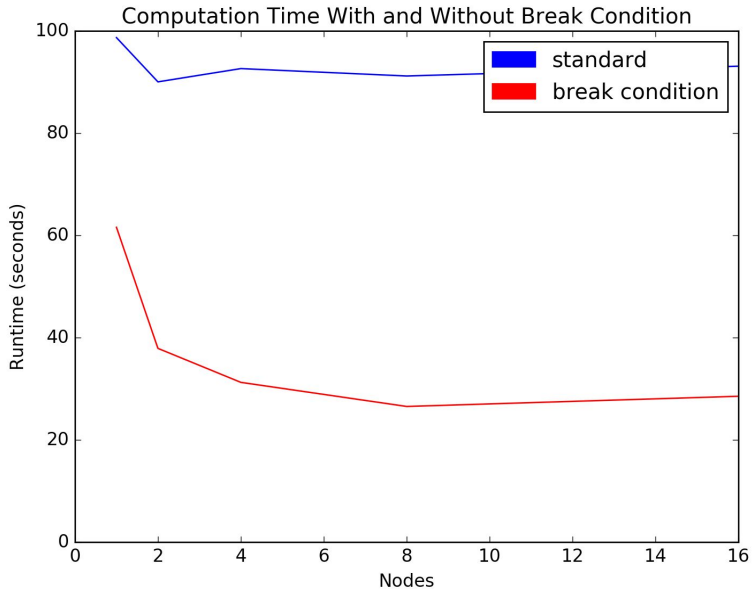
In conclusion, Spark with Python is a clear winner in terms of generalizability and ease of use. In contrast to Scala, Python allows for easy translation of code from common scientific computing languages like matlab, along with competitive performance and linear algebra support. In addition, Python's interpreted nature allows for a more generalizable structure when combined with Spark. Python's ease of use along with Sparks portability to many different systems would allow scientists to port their simulations to any number of cloud distribution services or clusters.

Final Implementation Structure and Results

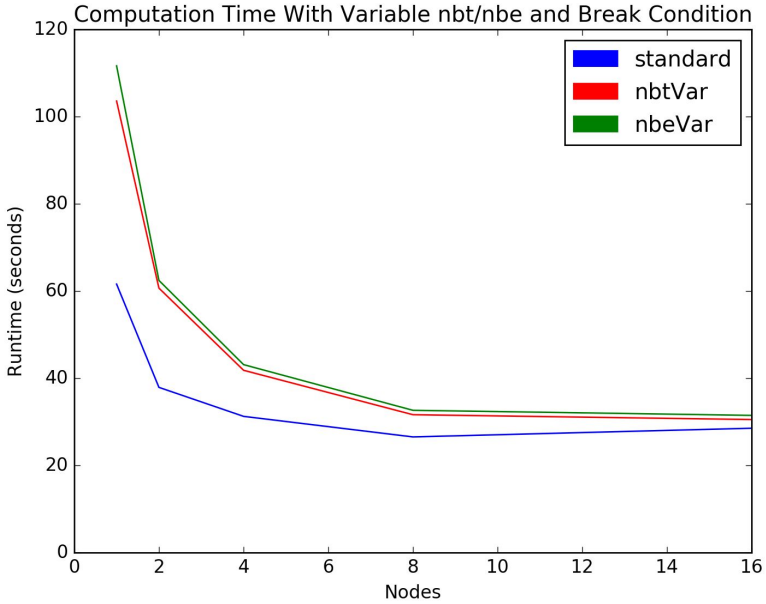
Our final Python implementation uses a structure that allows for easy generalization to other simulations. Similar to a lot of scientific computing, the simulation relies on repeated calls to the same function, in our case the polyfunc. As such, our implementation was segmented into a main function, setup, initial polycrystal calculation, and piezo coefficient calculation. The main function sets up all interaction with Spark, determines tasks, and is dispatched to the master node. Because of the lack of parallelizability of the setup and polycrystal calculation, the master node performs these calculations, but the code split them into different functions to maintain generalizability. Finally, the main function prepares the tasks that need to be done, and submits them to Spark, along with the polyfunc, for distribution. Spark distributes the tasks to workers which repeatedly iterate over the polycrystal data with the polyfunc, and return the results to the master node, which outputs the final data.

After our initial distribution, The results were unsatisfying. We gained little to no performance increase with even sixteen nodes. Even though the data for the tasks was perfectly parallelizable, the computation was not. One of the 256 tasks that we ran was taking up a vast majority of the computation time. This meant that even with distributing to sixteen nodes, the simulation was being held back by one node running one task. In response, we added a break

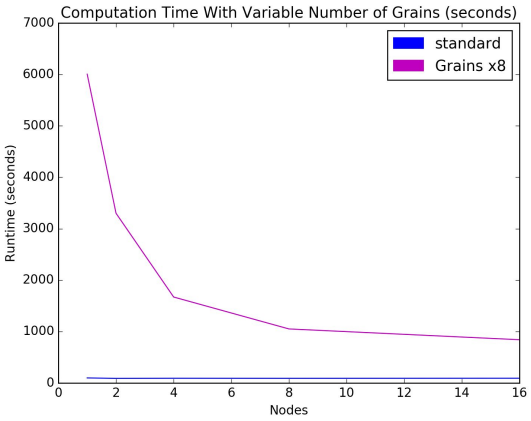
condition to limit the maximum number of iterations that could be run on one task. As a result, the gains from distribution improved dramatically. Below is a graph of the computation time with and without the break condition. The single task held back the simulation immensely, and after adding a break condition distribution worked much more smoothly.

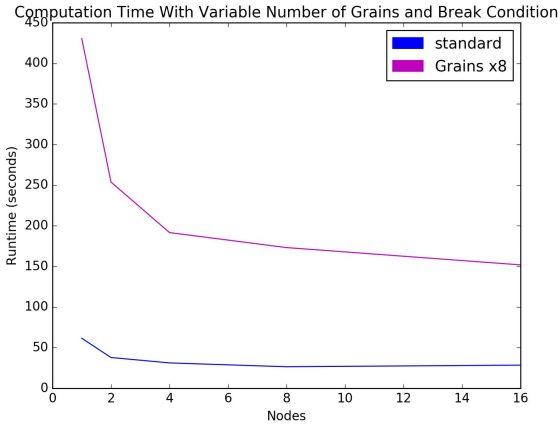


Unfortunately, there is some start-up time for a Spark worker to reach efficient execution, so at 16 nodes, where our number of tasks matched the total number of cores, we saw a slight slowdown due to the overhead of starting tasks on 256 workers. However, when we increased the number of tasks (by varying the number of stress and electric field values we checked), the setup time was less of a factor and the simulation was once again limited by the slowest computation (which depends on the break condition).



Finally, we varied the number of grains in the simulation to test its scalability. Ideally, the simulation would be run on thousands or millions of grains to obtain an accurate measurement of polycrystal behavior, but we tested ours on 546 grains to speed up computation. As a test, we used eight times as many grains to see how the simulation scaled. Without the break condition, the results were poor. The simulation was almost 60 times slower. However, with the break condition, the performance was closer to expected, with eight times the grains running for about eight times as long. Below are graphs showing the effect of increased grains with and without a break condition.





Discussion

What at first seemed an easily distributable model proved to be a little bit harder in practice. Without a break condition to limit the max running time of an individual task, the performance of the whole simulation was limited by its slowest task. We assumed that the perfectly parallel structure of the model would easily translate to distributability, but this wasn't necessarily the case. Instead, we had to perform specific tuning to get the simulation to run as desired. With this in mind, when generalizing the distribution of scientific computation, it is important to adjust the simulation to fit the needs of distribution. This means making sure that all parts of computation are parallelizable and that a single piece doesn't limit the performance of the whole.

In terms of our architecture, Python and Spark was an easy to use framework for distribution. The code was simple to transfer from Matlab to Python, and most of the linear algebra functionality was retained. In contrast, attempting to use Scala for scientific computing was a disappointing experience due to the lack of strong linear algebra libraries. For future distributed scientific computing, it is important that the language be easy to understand for ease of use by scientists, and have strong scientific libraries like Python. With a language like Scala, the distribution may be more developed, but that does little to offset a poorly implemented or optimized simulation.