Distributed Computing and JAVA VectorAPI for performance optimization for ScalaTion Framework

By

Chandana Marneni (chandana.marneni@uga.edu)

(Under the direction of Dr. John A. Miller)

January 6th, 2019

ABSTARCT

Scalation is open source Scala based framework for Big Data Analytics that offers simulation, optimization and analytics. And as big-data involves handling massive amounts of data, using a single system to process such huge volumes of data is cumbersome and slow in processing. So, in this work we explored adding distributed functionalities to Scalation and also the possibility of using the OpenJDK VectorAPI to write compute intensive applications in Java without Java Native Interface performance overhead. Our goal was to improve the processing time and efficiently utilize the available resources to reduce operation time and cost. In this work, we initially concentrated on the mathematics package of Scalation as it is the basis for the modeling and database modules of ScalaTion. We worked on the matrix multiplication problem as its runtime optimization is still an ongoing research problem. We leveraged Akka, which is open-source toolkit to add the distributed functionalities in the mathematics package of Scalation.

Index words: *Matrix Multiplication, Distributed systems, Cannon's Matrix multiplication, VectorAPI*

I. INTRODUCTION

Matrix Multiplication is still considered as a very important research problem to study when designing optimization solutions. As of recently, the of the order of $n^{2.37}$ with Coppersmith-Winograd algorithm, the processing time quickly escalates when we have to deal with large amounts of data, say for example multiplying two matrices that have 900 rows and columns each or more. And when such operations are included as a necessary calculation step, then the problem's operational cost would be really high and take a lot of time to process. This operational cost becomes even more prominent in machine learning applications which are part of the ScalaTion framework. Instead we can try to reduce this cost by distributing the work to different machines that work independently to solve subparts of the problem and later these sub-solutions can be combined to get the final result. Also, we considered using Java VectorAPI to directly transform vector-vector operations to optimal vector hardware instructions that use the data parallelism of the underlying hardware instruction set thus effectively lowering the runtimes.

In the Scalation mathematics package, matrix multiplication is one of the primary operations and upon which many other operations are depended as the *linalgebra* package of the mathematics module is the basis for analytics and machine learning modules. So, in this work we narrowed our focus on improvising the performance of matrix multiplication operation. First, we implemented *Cannon's distributed matrix multiplication algorithm* [1] to distribute the work of a single machine onto different machines to compute matrix multiplication. Next, we also tried to use the Java VectorAPI which is still in initial stages of development and not yet released into the market. Using VectorAPI showed significant performance improvements compared to distributed approach.

II. BACKGROUND and RELATED WORK

Much work has been done in the area of distributed matrix multiplication. 1D-systolic [2], 2D-systolic [2], Summa [18] and Cannon's [1] are some of the different approaches proposed for matrix multiplication. Cannon's algorithm is a distributed matrix multiplication algorithm that is suitable for computers laid in a N*N mesh and matrices of equal dimensions. The main advantages of using Cannon's is that the given the problem space, the storage requirements remain constant and works independent of the number of processors in use [19]. Summa algorithm [18] is more generalized than Cannons' and it doesn't have the restrictions of working with 2D grid. Also, in optimizing matrix multiplication process, another approach considered is to transpose one of the matrices to lessen the retrieval cost.

III. SCALATION

Scalation is a Big-Data framework [17] that provides analytics, simulation and optimization. Scalation has different modules that include mathematics, statistics, databases and modeling. It also includes some examples of working with this framework.

The *Scalation_mathstat* module is the basic package of Scalation and it offers the support for mathematics and statistics needed for the scalation_modeling package.

The *scalation_database* module includes the database support of Scalation framework. Columnar databases are used in this module for performance efficiency. The database package also includes support for graph and spatial databases. The columns of the database are implemented as Vector data-type of Scala language. Vector in Scala can include different datatypes – *Int, Long, Double, Real, Complex, Rational, String, Time* that can represent real-world data. This module provides support for the analytics operations of the Scalation by including efficient data management techniques.

The *scalation_modeling* module provides support through the modeling techniques like linear regression, neural networks, classifiers etc. for the analytics, simulation and mathematics of Scalation.

The *scalation_models* package includes examples of using the various modules and how the analytics and the models in Scalation can be used.

The current work focuses on adding distributed capabilities to the mathematics package and explores the way of using Cannon's algorithm using Akka toolkit to achieve this as a starting step.

IV. CANNON'S ALGORITHM FOR DISTRIBUTED MATRIX MULTIPLICATION

In this work, we implemented *Cannon's algorithm* for distributed matrix multiplication. Cannon's algorithm needs working with processes on 2D grid. For multiplying N*N matrices A and B, we need N*N processing nodes with each node getting $(N/\sqrt{p}) * (N/\sqrt{p})$ chunk of data where p is perfect square.

Cannon's algorithm organizes processors into rows and columns of the 2D mesh as shown in Fig 1. Each processor in the mesh gets a chuck of both the matrices A and B which are used to calculate a part of the resultant matrix multiplication. And to fully calculate the complete chunk of the matrix multiplication result, we incrementally move the chunks of matrices A and B among all the processors until the complete result is calculated. Effectively, we make use of ring broadcast algorithm to move parts of A and B matrices to successive processors in each step. At the end, each processor will have a chunk of the final result of multiplying the matrices and these chunks can be either combined together or stored from each processor separately into database.

P(0,0)	P(0,1)	P(0,2)
P(1,0)	P(1,1)	P(1,2)
P(2,0)	P(2,1)	P(2,2)

Fig 1 Description of Processors arranged in a 2D mesh

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

Fig 2. Description of Initial arrangement of matrices A and B on the processor mesh

The Figure 2 shows how matrices A and B are initially arranged on the 2D processor mesh with similar chunks of sub-matrices A and B assigned to the same processor at the beginning of the multiplication. A(i,j) represents the sub-matrix of A that we got by slicing matrix A into chunks of $N/\sqrt{p}$ size each.

To calculate resultant matrix, we multiply the chunks of matrices of A and B, rotate them, multiply the newly obtained chunks together and continue the process until all the sub parts have been calculated.

If we want to calculate C[1,2], we need A[1,0] and B[0,2]; A[1,1] and B[1,2]; A[1,2] and B[2,2] as sets on the same processor at different steps.

$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$

To achieve this configuration, we shift rows of A to the left and columns of B upwards to lineup. For shifting, we move each row i by i columns to the left using send and receive operations. The same is done for matrix B but each column j by j rows upwards. i and j in this shifting process are numbered starting from 0.

After the initial submatrix multiplication, we shift again using the same above-mentioned process and then multiply. Each processor forms the sub-product of the two sub matrices A and B in each step adding to the accumulated sum and finally after the algorithm finishes, each processor will have the chunk of the resultant matrix of size $N/\sqrt{p}$.

Cannon's algorithm shifts chunks of matrices A and B along the rows and columns of the processor mesh. This shifting of data needs a message passing mechanism among the processors to send and receive data. To this effect, we leveraged the use of Akka distributed toolkit for message passing in the processor grid.

V. AKKA DISTRIBUTED TOOLKIT

Instead of developing a distributed mechanism from scratch, we opted to use existing toolkits. We explored Akka toolkit [4] and Apache's Kafka [6], Spark [9], Storm [10], Flink [5] and Samza [8]. Many works on comparing these distributed technologies exist. After studying the features of these technologies, their compatibility with Scalation and ease of use, we decided on using Akka to add distributed capabilities to Scalation.

Akka is free and open-source distributed toolkit developed by Lightbend and includes support for both Java and Scala. Akka was initailly developed by Jonas Boner to simplify the overhead and errors involved in directly using threads and synchronization of the critical parts with locks. Akka was developed from the inspiration of Erlang's highly-concurrent and event-driven approach. The result is that Akka became light-weight and offers transparent remote communication between the actors by passing immutable messages. Thus Akka effectively obviates the burden of dealing with multi- threaded components, thus making it less complex to work with. And the messages passed in Akka are immutable and we need not worry about other components altering the message which is quite common when working with distributed systems.

Akka is actor model system where each component is envisioned as an actor that can send or receive messages. An actor object encapsulates state and behavior and the actors communicate among themselves by exchanging messages which are stored in an actor's recipient mailbox. Actors are contained in *ActorSystem* that manages the resources it is configured to use and runs the actors that it contains. Actor is handled with *actorRef* that is returned upon the creation of actor within the ActorSystem instead of using it directly. Following are the syntaxes for creating ActorSystem, actor and message passing between actors.

ActorSystem creation:

val *actorSystem*: ActorRefFactory = *ActorSystem*(**"ActorSystem_name"**, *config*)
where config represents the configuration of ActorSystem.

Actor creation:
val *actorRef*: ActorRef = *actorSystem*.actorOf(*Props*[**Actor_Class**], **"Master"**)

Message passing: done using '!'

actorRef ! print("Hello world")

Akka also supports clustering where individual nodes are grouped into a cluster are managed by the Cluster system as a single unit. Many high-level clustering techniques are provided by Akka that support different use cases. For our use in Scalation, we used simple Akka cluster to define the distributed nodes involved in performing the subparts of the same computation as belonging to the same cluster. With the use of Akka clustering, low-level of details involved in remote communication can be effectively handled with less complexity.

VI. JAVA VECTOR API

Vector API is being developed as part of Panama project under Oracle's OpenJDK [15] to give Java the feature of expressing vector computations that compile to optimal vector hardware instructions at run-time. Vector computations make use of data parallelism and thus give far better performance optimization compared to scalar computations. They obviate the need of working directly with Java Native Interface (JNI) thus reducing the performance overhead in converting to native code. It is especially useful for compute intensive applications like machine learning, BigData and artificial intelligence applications in Java.

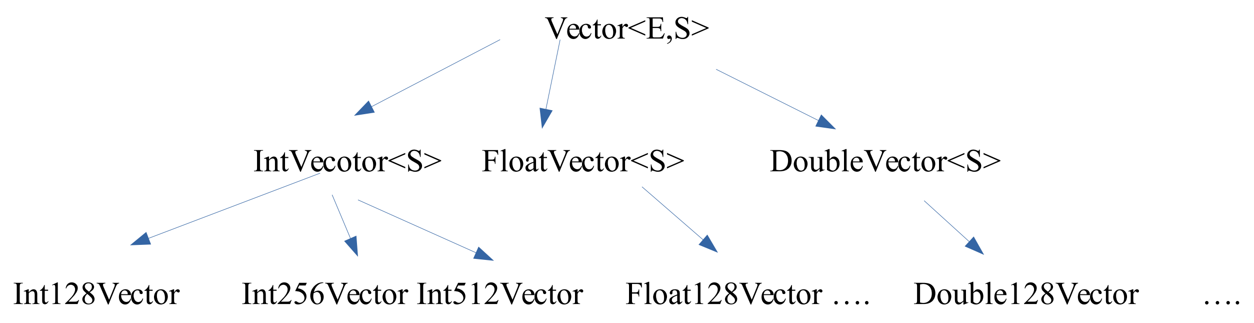

Fig 5 Vector API Interface hierarchy

Throughput can be improved by a combination of executing many instructions per cycle (pipelining) and also by processing multiple data items per instruction (*SIMD*). Intel already provides SIMD instruction sets as the various generations of *SSE* and *AVX*. To improve throughput, SIMD can me maximized. In Single Instruction Multiple Data Items (SIMD), we have multiple processing entities perform the same instructions on multiple data points at the same time thus working quickly on the entire input range in a short time compared to singe instruction working on single data item at any given point of time. *Vectorization* in Java allows throughput to be increased by the use of SIMD instructions. For exampleif we have a vector 256 instruction set, we can simultaneously work on 64 bytes of data at a time. Analytical workloads are particularly suitable for vectorization, especially over columnar data, because they typically involve operations consuming the entire range of a few numerical attributes of a data set.

As per the Panama project VectorAPI, the Vector type is defined as Vector<E,S> where 'E' is the Element type and 'S' is the shape or bitwise length of the vector. Based on the recent development of project Panama supports, Vectors creation of the following Elements and Shapes. Are possible:

Element types: Byte, Short, Integer, Long, Float, and Double
Shape types (bit-size): 128, 256, and 512

Basic Vector-Vector functionalities like addition, multiplication are available for all of the above Vector types. And Vector Masks are used for conditional if-else type operations and these vector masks used together with the normal vector operations act as filters. In the work we did, VectorAPI is shown to improve performance by twofold compared to scalar operations. And the API also provides many convenient high-level methods to convert arrays to vectors and vectors to arrays so that we can read the data into a vector and perform Vector-Vector operations and write back the results in our desired format. The Vector API provides Vector.preferredSpecies instance method in order to query for the appropriate vector size to use. This is especially useful when we don't want to explicitly state the Shape of the Vector to use as it may vary from machine to machine. And instead this enables the size to be dynamically computed depending on the system. The returned species of this method can be used for generically sized vector computations so no concrete shape need to be declared.

The VectorAPI operations are available in the jdk.incubator.vector of the Panama project *develeopment* branch. We can read data between arrays and Vectors using *'fromArray' and 'toArray'* methods. At each iteration, depending on the size of the vector data equivalent to the vector length gets operated on parallely with the same instruction thus working on multiple data points at the same time.

VII. CANNON'S ALGORITHM IMPLEMENTATION USING AKKA

We implemented Cannon's distributed matrix multiplication algorithm using Akka in the mathematics package of Scalation. This is available as *dist* package in the *LinearAlgebra(linalgebra)* of the scalation mathematics package. Specifically, we worked with MatriD matrix type of the module.

We implemented the distributed matrix multiplication using four systems that form the four nodes in the cluster. So, the processing of matrix multiplication is distributed evenly onto these four machines with one of the machines working as the Master that distributes the sub-matrices across the four nodes. The architecture is designed with single Master and four worker Nodes.

The Master node's main function is message coordination among the individual worker nodes and to achieve synchronization in the process. Synchronizing events is important as we have to ensure that sub-matrix multiplication is completed successfully on all the nodes and then only the sub-matrices can be exchanged between the nodes in the next step.

The worker nodes main functionality is to perform matrix multiplication, update their progress to Master node and upon completion of the process, they send the resultant chunks of the multiplication of matrices to the Master node which builds the final output matrix.

Initially the Master node is tasked with fetching the matrices from the database, divides the matrices A and B into chunks of size equal to that $N/\sqrt{p}$ where N is the dimension of the matrices A and B. Both A and B are assumed to be square matrices of the same order. As each node gets chunks of both A and B matrices, they calculate part of the result of the sub-matrix multiplication and inform the Master node. And after receiving a message from Master node to start shifting when all worker nodes have completed calculating part of the resultant matrix multiplication chunk on those nodes, worker nodes start shifting their chunks of A and B matrices to their successive processors in order according to the Cannon's algorithm.

After each processor has successfully computed the chunk of the resultant matrix, these chunks are transferred to the Master node which forms the final matrix that would be the result of multiplying both A and B matrices.

While working with Akka, we noticed that the overall time to complete the matrix multiplication is higher than what we looked for. This is due to the inbuilt Java serialization mechanism used by Akka. This time to exchange data between nodes increases even more when the nodes are set to run on different machines when the data has to transfer through network. To overcome this, we looked at some binary serialization libraries like Google Protocol Buffers [12], Kryo serialization [7] and tried to use Kryo binary serialization that provided support for Scala and easy to use. But since we are working with MatrixD that has custom datatypes wrapped as a single entity, we found that serialization-deserialization process wasn't straight forward. We observed an issue with deserialization being done out of order in comparison to the serialized messages. So, we couldn't include the Kryo serialization library to improve the communication costs.

VIII. PERFORMACE EVALUATIONS

Matrix multiplication results for matrices of size 900*900 using Vector API in ms:

Method	Time (ms)
VectorAPI	407
Optimized (single node)	836
Cannon Distributed (four node)	892

Table 1 Performance evaluations

The above results show that VectorAPI provides the best possible performance as it makes use of data parallelism of the underlying hardware instructions while also avoiding the nativity conversion overhead involved with JNI. Optimized matrix multiplication and Cannon's both use the optimized version of matrix multiplication. However, Cannon's algorithm is run on four different nodes whereas the single node optimized version uses only a single node. They both have almost similar performance results. But with the use of right serialization approach instead of the slower java serialization used inherently in Akka, Cannon's algorithm may provide even better results than optimized version running on single node.

The following is the breakdown of the individual times taken to perform the Cannon's matrix multiplication at each step starting from the first sub-matrix multiplication excluding the time taken for data exchange:

Node	Before shifting (ms)	After shifting (ms)	Total (ms)
Worker	265	275	540

Node	Merging of matrices (ms)
Master	35

Time taken to shift the sub-matrices among the individual nodes: 224.64ms
And similar times for initial transfer of sub-matrices to each of the nodes from the Master node.

Therefore, total time spent on the processor doing useful work is around 540ms on the worker nodes and even lesser at the Master node. In our setup, we had the Master node and one of the worker nodes running on the same machine and since Master node doesn't do much computations when the worker node is busy and only starts merging matrices after all the worker nodes complete sending their resultant sub-matrices, a single machine can effectively accommodate both Master and worker node. The above times show that by distributing the work onto different machines, we are improving the performance at each node end. If we take out the latency involved in the exchange of data among the nodes, Cannon's algorithm performance is similar to that of VectorAPI. So, by applying effective serialization mechanisms, this latency overhead can be brought down thus improving the overall performance of the algorithm.

In reference to the work done by Santosh Uttam Babode [16] on the performance evaluation of Summa [18], which is a universal scalable matrix multiplication, the results are shown as 1796ms for 900*900 matrix multiplication for 2-dimensional matrices where the operations are done serially and 167ms for parallel approach. It is to be noted that the machines used to test Summa's performance in Santosh's work and Cannon's and VectorAPI in this work are different. Both Cannon's and VectorAPI perform better compared to Summa 2DS approach.

IX. FUTURE WORK

This work only involves implementation of distributed matrix multiplication of MatriD type matrices using Akka. There are many other components included in the mathematics package to which distributed capabilities can be added. And in the future, we can extend all the Scalation modules incrementally to implement the distributed functionalities and provide both monolithic and distributed support.

And here we used only single instance of Master and slave nodes. The case of node failures has not been handled in this work. Also, we assumed a 2*2 grid with 4 machines. Some changes would be needed to generalize the number of machines used in the cluster. So, in order to obtain resiliency and scalability, maybe we can look into node replication and cluster management systems like Docker, Kubernetes [11] when this system with the distributed functionality get mature enough to use containers.

Akka uses Java serialization to serialize messages that are sent remotely. And Java serialization is slow thus effecting the overall performance. To achieve higher speed-ups, we can implement alternative serialization techniques that provide Scala support like ScalaPB[13], Twitter's Chill[14].

Java VectorAPI is still in the nascent stages development and is rapidly changing. But when this API becomes finally available in the market, this will benefit many compute-intensive applications like machine learning and analytics applications.

X. CONCLUSION

In this work, we present Cannon's distributed matrix multiplication implementation of MatrixD from the mathematics module of Scalation. We used Akka distributed toolkit to add the distributed functionalities as for message passing as it is light-weight, easy to use and provides all the necessary features with support for Scala. This work can be considered as the initial step towards making Scalation framework distributed. With the use of proper serialization mechanism for communication optimization, Akka can be effectively used to obtain the distributed functionalities. We also showed how the Java VectorAPI boosts the performance by twofold with its usage of data parallelism at the hardware level.

REFERENCES

[1] Cannon, L.E., A Cel lular Computer to Implement the Kalman Filter Algorithm, Ph.D. Thesis (1969), Montana State University [Online] Docker – container orchestration. https://www.docker.com/

[2] Golub, G. H. , and C. F. Van Loan, Matrix Computations, Johns Hopkins University Press, 2nd ed., 1989.

[3] [Online] http://cseweb.ucsd.edu/classes/fa12/cse260-b/Lectures/Lec13.pdf

[4] [Online] Akka distributed toolkit documentation. https://akka.io/docs/

[5] [Online] Apache Flink. https://flink.apache.org/

[6] [Online] Apache Kafka – distributed streaming platform. https://kafka.apache.org/

[7] [Online] Akka Kryo serialization – Kryo based serialization for Akka and Scala https://github.com/romix/akka-kryo-serialization

[8] [Online] Apache Samza – distributed stream processing framework. http://samza.apache.org/

[9] [Online] Apache Spark – analytics engine. https://spark.apache.org/

[10] [Online] Apache Storm. http://storm.apache.org/

[11] [Online] Kubernetes – container orchestration. https://kubernetes.io/

[12] [Online] Protocol Buffers – Google for serialization.
https://developers.google.com/protocol- buffers/

[13] [Online] ScalaPB – Protocol buffer plugin for Scala. https://scalapb.github.io/

[14] [Online] Twitter's Chill for Kryo serialization. https://github.com/twitter/chill

[15] [Online] OpenJDK Panama Vector API Incubator project http://openjdk.java.net/jeps/338

[16] [Online] https://getd.libs.uga.edu/pdfs/bobade_santosh_u_201808_ms.pdf

[17] Scalation analytics framework. http://cobweb.cs.uga.edu/~jam/scalation_1.5/README.html

[18] Summa matrix multiplication. http://www.netlib.org/lapack/lawnspdf/lawn96.pdf
[19] Wikipedia contributors. Cannon's algorithm – Wikipedia, the free encyclopedia, 2018
https://en.wikipedia.org/wiki/Cannon%27s_algorithm