



Article

JAMPI: Efficient Matrix Multiplication in Spark Using Barrier Execution Mode

Tamas Foldi ^{1,*} , Chris von Csefalvay ¹ and Nicolas A. Perez ²

¹ Starschema Inc., Arlington, 22066 VA, USA; chris@chrisvoncsefalvay.com

² Google Inc., Seattle, 98103 WA, USA; anicolaspp@google.com

* Correspondence: tfoldi@starschema.net

Received: 10 July 2020; Accepted: 26 October 2020; Published: 5 November 2020



Abstract: The new barrier mode in Apache Spark allows for embedding distributed deep learning training as a Spark stage to simplify the distributed training workflow. In Spark, a task in a stage does not depend on any other tasks in the same stage, and hence it can be scheduled independently. However, several algorithms require more sophisticated inter-task communications, similar to the MPI paradigm. By combining distributed message passing (using asynchronous network IO), OpenJDK's new auto-vectorization and Spark's barrier execution mode, we can add non-map/reduce-based algorithms, such as Cannon's distributed matrix multiplication to Spark. We document an efficient distributed matrix multiplication using Cannon's algorithm, which significantly improves on the performance of the existing MLlib implementation. Used within a barrier task, the algorithm described herein results in an up to 24% performance increase on a $10,000 \times 10,000$ square matrix with a significantly lower memory footprint. Applications of efficient matrix multiplication include, among others, accelerating the training and implementation of deep convolutional neural network-based workloads, and thus such efficient algorithms can play a ground-breaking role in the faster and more efficient execution of even the most complicated machine learning tasks.

Keywords: Apache Spark; distributed computing; distributed matrix algebra; deep learning; matrix primitives

MSC: 68W15

1. Introduction

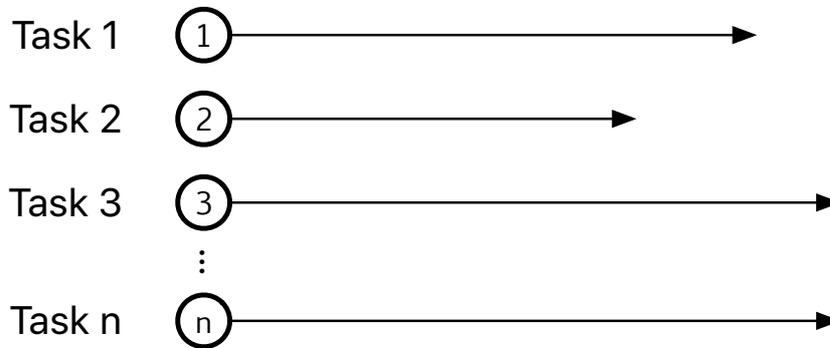
The past decade has seen the emergence of two immensely powerful processes in tandem: the rise of big data handling solutions, such as Apache Spark on one hand, and the apotheosis of deep learning as the tool of choice for demanding computational solutions for machine learning problems on the other hand. Yet, at its essence, big data and deep learning remain not only separate communities but also significantly separate domains of software. Despite deep learning over big data becoming a crucial tool in a range of applications, including in computer vision [1,2], bioinformatics [3–6], natural language processing (NLP) [7–10], clinical medicine [11–16], anomaly detection in cybersecurity and fraud detection [17–19], and collaborative intelligence/recommender systems [20–23], its full potential remains to be harnessed. The primary impediment in this respect is largely a divergence of attitudes and concerns, leading to two divergent paradigms of development:

- The big data paradigm, primarily designed around RDDs and the the DataFrame-based API. This outlook has dominated the development of Apache Spark.
- The DL/ML paradigm, which is primarily focused on efficient linear algebra operations to facilitate machine learning approaches, especially matrix algebra for deep neural networks.

The future of deep learning over big data depends greatly on facilitating the convergence of these two worlds into a single, unified paradigm: the use of well-designed big data management tools, such as Apache Spark, to interoperate with the demands of deep learning. The road towards this convergence depends on the development of efficient matrix primitives that facilitate rapid calculations over distributed networks and large data sets.

The current execution model of Apache Spark is principally focused on independent, embarrassingly parallel, tasks that are run and scaled, but the needs of deep learning are primarily focused on distributed training: the performance of completely communicating and coordinating tasks, optimized for interconnectivity rather than independent parallel running, while also maintaining scalability and efficiency. With the recent introduction of the barrier execution mode in Apache Spark, it has finally become possible to construct a computational approach that allows for such networked execution to take place, facilitating distributed training of deep neural networks (see Figure 1).

Parallel execution (Apache Spark)



Distributed training

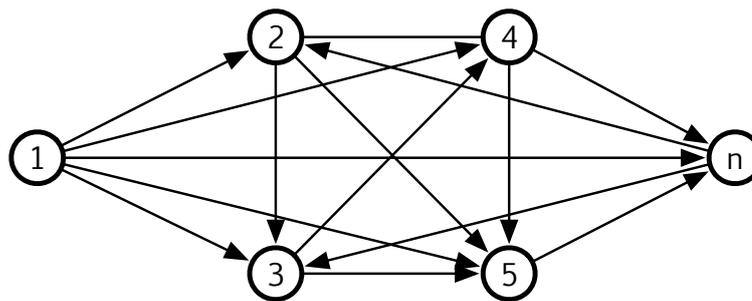


Figure 1. Comparative execution models: Apache Spark versus distributed training for neural networks.

Prior work on efficient matrix algebra operations has primarily focused on spatial separation. Thus, for instance, Bosagh Zadeh et al. (2016) discuss the most frequent algorithmic approach to solving linear algebra problems in Spark: separating matrix and vector operations, retaining the latter with the driver node for single node execution and distributing the former across the cluster [24]. This is a convenient method to allow for running legacy code written for single node devices on large Spark clusters, but does not generalize to a wide range of problems. A more promising approach is Marlin, a more generic set of matrix algebra primitives (including multiplication), proposed by Gu et al. (2015) [25]. Marlin’s performance is similar to that of MLlib, and a somewhat better performance is attained by Stark, a scalable matrix multiplication primitive using Strassen’s

algorithm [26]. However, the time complexity of Strassen’s algorithm is $\mathcal{O}(n^{2.807355})$, which is inherently less advantageous than other approaches, such as Coppersmith–Vinograd (1987) [27]. Neither this nor the somewhat faster algorithm introduced by Williams (2012) leverage a large number of cores as efficiently as Cannon’s algorithm [28].

This algorithm [29] and its generalization (Lee et al., 1997) rely on a toroidal mesh of interconnected nodes [30]. While the use of MPI as the vehicle of connecting the nodes distributed along the toroidal mesh has been considered in the past (e.g., Li et al., 2012), [31] to the best of the authors’ knowledge, this is the first time such a communication interface has been natively implemented in Spark for matrix multiplication.

JAMPI (Java Assisted Matrix Product with Inter-task communication), the framework described in this paper, is an efficient and rapid solution to an aspect of efficient matrix primitives, namely matrix multiplication. By integrating JDK’s new `Vector` API, asynchronous network IO (`nio`) for distributed message passing and Spark’s barrier mode, a pure Scala implementation of Cannon’s 2.5D matrix multiplication algorithm can be devised that is significantly more efficient than `MLlib`’s `BlockMatrix.multiply` function. JAMPI thus avoids reliance on foreign, low level or native code in combination with `JNI`, on the one hand, being a pure Scala implementation. On the other hand, it provides a pre-written framework that integrates with Spark as a native task rather than an external MPI procedure call, and handles inter-task communication directly, yielding performance benefits that would otherwise be associated with a low-level MPI implemented resource negotiation framework.

1.1. Cannon’s Algorithm

Matrix multiplication plays a significant role in a range of practical applications, including (but not limited to) scientific computing, non-linear modeling, agent-based models and the training of deep convolutional neural networks (deep learning). The proliferation of deep learning as the cognitive technology of choice for problems with large source data sets and high-dimensional or high-order multivariate data means that efficiency gains in the underlying linear algebra primitives has the potential to enable significant performance benefits in a wide range of use cases. In particular, constructing primitives that leverage computational capacity through rapid parallel computation and efficient interchange lends itself as an avenue towards these performance gains. While packages comprising efficient matrix primitives already exist [32], these often operate at a low level and do not integrate well with existing and proven solutions to manage large computational loads.

The matrix multiplication operation \star for an $p \times q$ matrix \mathbf{A} and an $q \times r$ matrix \mathbf{B} is defined so that for the resultant matrix $\mathbf{C} = \mathbf{A} \star \mathbf{B}$, each element $c_{i,j}$ is the dot product of the i -th row of \mathbf{A} and the j -th column of \mathbf{B} —i.e.,

$$c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j} \quad (1)$$

The multiplication of square matrices constitutes a special case. For a square matrix of order n —i.e., an $n \times n$ matrix—a special case is obtained, which can be resolved efficiently using Cannon’s algorithm [29].

For a square matrix of order n —i.e., $n \times n$ —Cannon’s algorithm uses a toroidally connected mesh $\mathbf{P}^{n \times n}$ of n^2 processes. Rendered in pseudocode, the algorithm (Algorithm 1) can be expressed as follows for p processors:

Algorithm 1 Cannon's algorithm

```

1: for all  $i = 0 : \sqrt{p} - 1$  do
2:   CShift left  $A[i; :]$  by  $i$ 
3: for all  $j = 0 : \sqrt{p} - 1$  do
4:   CShift up  $B[:, j]$  by  $j$ 
5: for  $k = 0 : \sqrt{p} - 1$  do
6:   for  $i = 0 : \sqrt{p} - 1, j = 0 : \sqrt{p} - 1$  do
7:      $C[i, j] += A[i, j] * B[i, j]$ 
8:     CShift left  $A[i; :]$  by 1
9:     CShift up  $B[:, j]$  by 1

```

Cannon's algorithm is designed to be performed on a virtual square grid \mathbf{P} of p processors (i.e., a $\sqrt{p} \times \sqrt{p}$ matrix). The multiplicand and multiplier matrices \mathbf{A} and \mathbf{B} are laid out on \mathbf{P} , after which the i -th row of \mathbf{A} is circularly shifted by i to the left and the j -th column of \mathbf{B} circularly shifted by j elements up. Then, n times, the two entries mapped onto $p_{i,j}$ are multiplied and added onto the running value of $p_{i,j}$, after which each row of \mathbf{A} is shifted left by one element and each column of \mathbf{B} is shifted up by one element.

Standard methods of multiplying dense matrices require $O(n^3)$ floating operations for an $n \times n$ matrix. Cannon's algorithm improves on this by reducing it to $O(\frac{n^3}{p})$. In particular, because of the fact that memory is not dependent on the number of processors, it scales dynamically with the number of processors. This makes it an attractive candidate for implementation as a high-performance distributed matrix multiplication primitive.

1.2. Spark's Barrier Mode

Spark's barrier mode is a new mode of execution introduced to Apache Spark as part of Project Hydrogen [33]. Barrier execution features gang scheduling on top of the MapReduce execution model to support distributed deep learning tasks that are executed or embedded as Spark steps. The current implementation ensures that all tasks (limited to mapPartitions) are executed at the same time, and collectively cancels and restarts all tasks in the case of failure events. In addition to true parallel execution, the workers' host names and partition identifiers are accessible inside the tasks, alongside a barrier call, similar to MPI's MPI_Barrier function [34].

While this functionality is sufficient to support the primary use case of Spark's barrier mode—namely, executing embedded MPI or other foreign (i.e., non-Spark and non-JVM, steps within a Spark application)—it does not provide any inter-task communication primitive to implement the same algorithms within JVM/Spark native steps. In fact, the design documentation for Spark's barrier mode clearly defines this as outside the scope of the project, stating that beyond a simple BarrierTaskContext.barrier() call, no intra-communication functionality will be part of the implementation. It is assumed that such functionality would be handled by the user program. It is our view, based on our extensive experience with implementing deep learning solutions on distributed systems, that this is a clear show-stopper: if Spark is to be a force to be reckoned with as the data layer for deep learning applications over big data, it should not force execution outside Spark's boundaries.

2. Methods

2.1. Cannon's Algorithm on MPI

The MPI version of the algorithm described in Section 1.1 relies on MPI's Cartesian topology. After setting up a 2D communication grid of processors with MPI_Cart_create, processors exchange data with their neighbors by calling MPI_Sendrecv_replace. In the main loop, each processor executes a local dot product calculation, then shifts the results horizontally for matrix a and vertically for matrix b. In our benchmarks, we used MPICH version 3.3.2 as the underlying MPI implementation.

To speed up matrix multiplication, we applied -O4 -ftree-vectorize -march=native GNU C compiler flags to ensure vectorized code execution. By vectorization, we refer to using SIMD

(Single Instruction, Multiple Data) CPU features, more precisely Advanced Vector Extensions (AVX-512F) that allows for the faster execution of fused multiply-add (FMAC) operations in local/partial matrix dot product steps. After compiling our code with GCC 7.3.1, we ensured that the disassembled code contains `vfmadd231sd` instruction for vectorized FMAC.

2.2. JAMPI

JAMPI is a de novo native Scala implementation of Cannon’s algorithm, as described in Section 1.1. For message passing, we built a nio-based asynchronous message passing library that mimics MPI’s Cartesian topology and send-receive-replace functionality. To avoid unnecessary memory copies and to optimize performance for both throughput and latency, our `PeerMessage` object allocates fixed 8MB off-heap buffers for both sending and receiving data. Send and receive network operations are executed asynchronously and in parallel.

The matrix multiplication is embedded into a barrier execution task, which is parametrized by the the number of partitions, the local partition ID, the hostnames for the other partitions (address from `BarrierTaskContext.getTaskInfos()`), as well as the local matrix pairs from the RDD.

```
def dotProduct[T : ClassTag](
  partitionId: Integer,
  numOfPartitions: Integer,
  hostMap: Array[String],
  matrixA: Array[T],
  MatrixB: Array[T]): Array[T]
```

JAMPI supports double, float and int Java primitive data types passed as Java Arrays.

2.3. Vectorization Using Panama OpenJDK

In order to achieve performance on par with the optimized MPI implementation for local dot product steps, we used JVM’s native vector intrinsics and super-word optimization capabilities for both JAMPI and MLLib Spark application benchmarks. The most recent and most comprehensive vectorization support in JVM is found in the `Vector API` module, part of OpenJDK’s *Project Panama*. While the `Vector API` module is currently in incubation status, we consider it stable enough to use for both the Spark platform and application code.

For fair benchmarking, we avoided using `Vector<>` objects or advanced methods, such as manual unrolling. While these techniques could potentially further improve performance, our goals were to compare the distributed algorithms’ performances with the same CPU opcodes used in local matrix multiplications. From the JIT compiler outputs, we confirmed that both Spark applications were using `vfmadd231sd`, just as in the GCC compiled MPI version.

To use the new vector intrinsics’ features, we built a custom OpenJDK package from the tip of the `panama/dev` branch (`dev-442a69af7bad`). The applied JVM flags were `-add-modules jdk.incubator.vector` and `-XX:TypeProfileLevel=121` for both JAMPI and MLLib applications.

2.4. Apache Spark MLLib

We used Apache Spark MLLib’s built-in `BlockMatrix.multiply()` as a baseline to compare with JAMPI’s speed and resource usage (see Figure 2). It is known that MLLib’s implementation is often faster if the number of partitions exceeds that of worker cores (typically by a factor of 2–4 at least), a scenario known as *over-partitioning*. To ensure that this is adequately reflected, we performed two test runs—a “normal” test run, where partitions are set to equal the number of worker cores, and an “over-partitioned” test run, where partitions equal four times the number of worker cores (see Table 1).

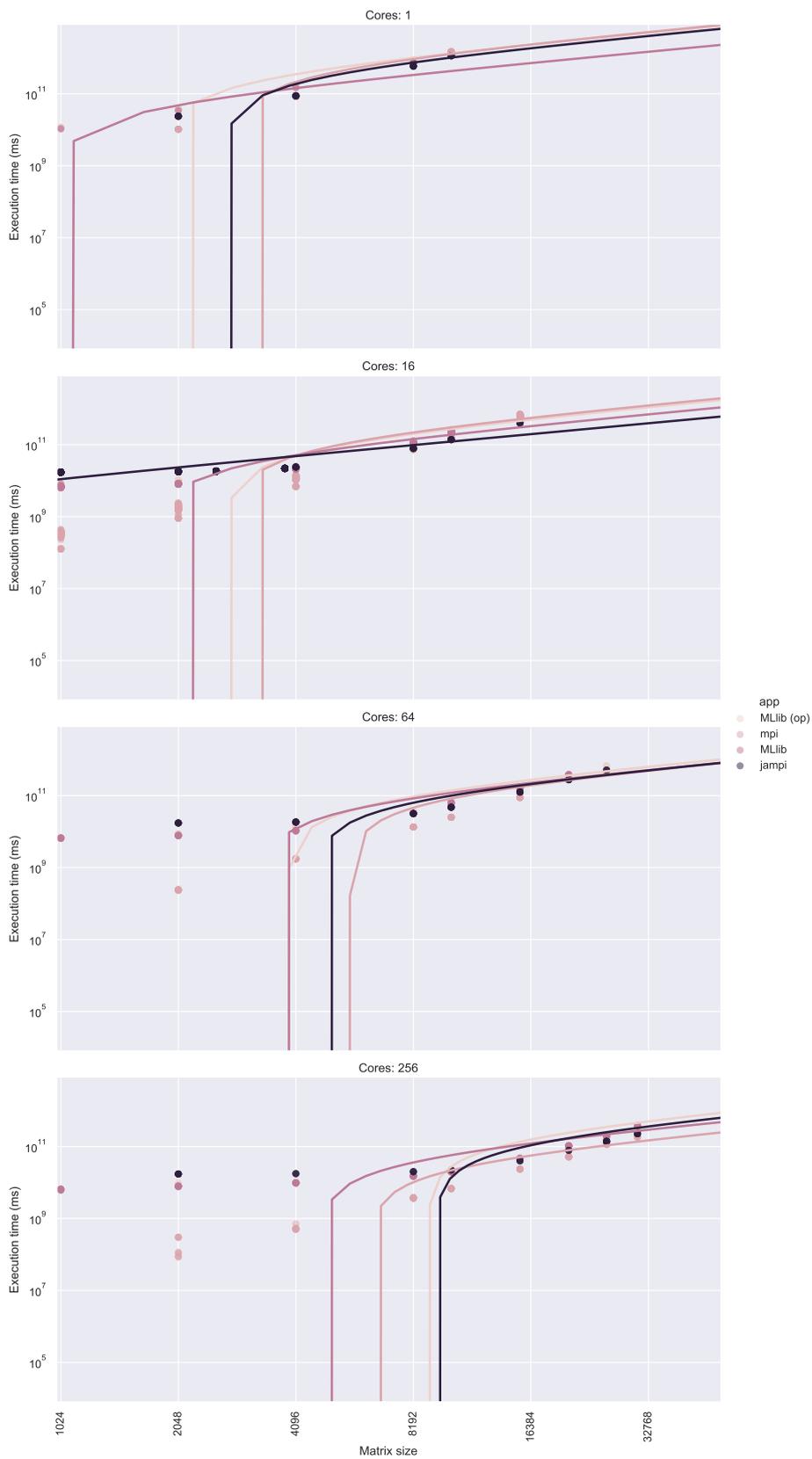


Figure 2. Comparative performance of JAMPI, native MPI and MLib on random matrices of various dimensions, on 1, 16, 64 and 256 cores.

Table 1. Instance configurations by worker cores, nodes and partitions.

Total Worker Cores	Instance Type	Nodes	Partitions
1	m5.large	1	1
16	m5.xlarge	4	4
64	m5.2xlarge	8	8
256	m5.2xlarge	32	8

2.5. Test Protocols

All tests were performed on Amazon Web Services EC2 instances using m5 instance types with Intel® Xeon® Platinum 8175M CPUs and 4GB RAM per core. Tests were conducted on Apache Spark 3.0.0-preview2 with a separate master node. The driver process was initiated from the master node, and its resource consumption is not included in the results. For single core tests, 2-core CPUs were used, with the second CPU core having been manually disabled in the VM. For each permutation of matrix size and number of cores, 100 runs were performed and the resulting runtimes were statistically analysed.

Applications reported only the dot product execution time. A single one-value reducer (avg) was included to trigger RDD reduction/collection on Spark without moving substantial amounts of data to the driver process. Timings thus exclude the MPI and Spark application startup times, but included the time required to establish a barrier task step during the RDD reduction step. For testing, random matrices composed of uniformly distributed 64-bit floating point elements were used. Test scenarios were performed ten times, capturing execution time, CPU and memory consumption, and mean values are reported. Test execution relied on the same set of matrices for every implementation, thereby ensuring a fair comparison. Test scenarios, as well as the original JAMPI source code, are available online [35].

2.6. Scalability Analysis

An important aspect of any distributed algorithm is its ability to scale up as the problem size increases. This is crucial for proving the value of an algorithmic solution, since it demonstrates its ability to solve increasingly complex instances of the same fundamental problem effectively. There are intrinsic issues when scaling distributed multi-processor algorithms. It is known, for instance, that the memory requirement for each processor increases as we add processors to a computation. Therefore, we must analyze the effect of problem size on the memory requirements per processor.

For Cannon's algorithm, multiplying two square matrices of size $n \times n$, the problem size W is on the order of n^2 —i.e.,

$$W = \mathcal{O}(n^2) \quad (2)$$

The sequential time, that is when $p = 1$, is

$$T_1(n) = \mathcal{O}(n^3) \quad (3)$$

For p processors, the execution time for a matrix of size $n \times n$ is given as $T_p(n)$. It follows that the parallelization of the problem yields a speed-up, calculated as $\frac{W}{T_p(n)}$.

In addition, the parallel execution of an $n \times n$ problem size over p processors will incur a performance overhead of $T_o(n, p)$, including all communication costs.

It is known that the communication cost D , which describes the amount of data being shifted across the p processors, can be calculated as

$$D = \mathcal{O}\left(\frac{n^2}{\sqrt{p}}\right) \quad (4)$$

Using the following iso-efficiency relationship of parallel systems,

$$T_1(n) \geq c T_o(n, p) \tag{5}$$

Substituting Equation (3) in Equation (5), it follows that

$$n^3 \geq c \sqrt{p} n^2 \implies n \geq c \sqrt{p} \tag{6}$$

It thus follows from Equation (6) and the definition of W in Equation (2) that

$$\frac{M(c \sqrt{p})}{p} = \frac{c^2 p}{p} = c^2 \tag{7}$$

More generally, it holds that, for a problem size W and p processors, Cannon’s Algorithm memory requirements increase by a constant factor c^2 that are independent of the number of processors p involved in the computation. Since the memory requirements per processor increase linearly, without a direct relationship to p , it can be said that Cannon’s algorithm is extremely scalable.

Figure 3 illustrates this scaling behavior comparatively between JAMPI, a pure MPI implementation and MLib. JAMPI, as well as the MPI algorithm test case, are both direct implementations of Cannon’s algorithm, thus having the same scalability behavior.

Used memory across all workers for multiplying 15360x15360 matrices

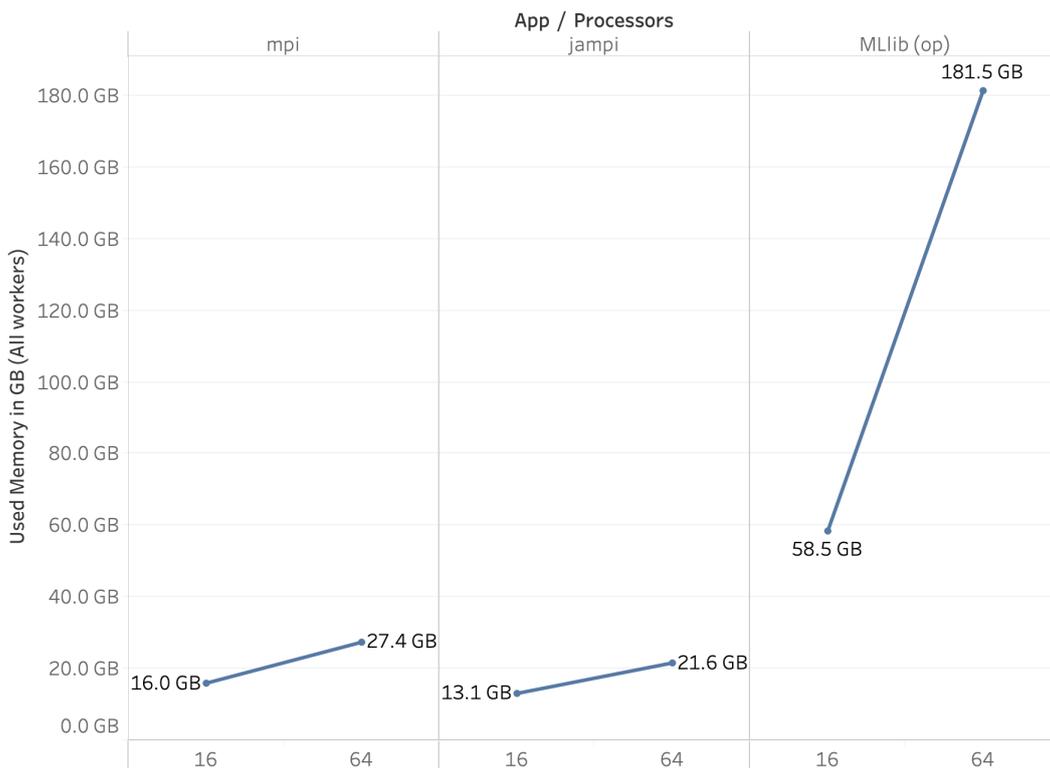


Figure 3. Comparative memory usage between JAMPI, MPI and MLib.

It is evident from Figure 3 that MLib’s memory requirement increases quite fast, suggesting that its scalability factor is larger than that of Cannon’s algorithm (i.e., it is less scalable). This is a key limitation of MLib and Spark when compared to MPI and JAMPI alike, which scale better. Indeed, in some test scenarios, we have been unable to scale MLib beyond a certain problem size, indicating that, in addition to its poor performance compared to MPI and JAMPI, it is also limited in

the maximum problem size it can accommodate with a set level of resources. Neither JAMPI nor the native MPI implementation is so limited.

3. Results

Comparative analysis of runtimes over a range of matrix sizes reveals that JAMPI is significantly superior to MLib, even when over-partitioned (see Figure 2; over-partitioning is denoted by op). When normalized against JAMPI's execution times over 16 and 64 cores, execution time is slower for smaller matrices (under 4096×4096 elements) due to the need to establish and run the barrier execution task. However, beyond a trivial problem size, JAMPI and the MPI implementation rapidly become significantly more efficient, regardless of the number of cores. Notably, plain MLib (i.e., without over-partitioning) was unable to accommodate a problem size beyond $10,240 \times 10,240$ (for 16 cores) or $20,480 \times 20,480$ (for 64 cores).

3.1. Memory Usage

Memory usage has been a documented limiting factor, with pure MLib reaching execution limits at relatively trivial matrix dimensions per processor (Table 2). While over-partitioning slightly increases the maximum matrix size, MLib suffers from not only lower performance but also a memory consumption upper bound that limits its ability to scale to larger problem sizes.

Our research indicates that, for a $10,240 \times 10,240$ element standard matrix, JAMPI and MPI perform approximately equally (4889 MB vs. 5108 MB, respectively, for 256 cores), while both over-partitioned and regular MLib execution create a marginally larger memory footprint (6049 and 6423 MB, respectively, for 256 cores). However, with increasing problem size, differences become vastly apparent: for a $30,720 \times 30,720$ element matrix, MPI and JAMPI continue to require a constant memory footprint (5572 and 6084 MB, respectively), while the same problem size requires 24,525 MB with over-partitioning and 29,445 MB without. In other words, JAMPI and MPI memory burden increases constantly, regardless of the number of cores, while MLib's memory consumption increases rapidly, as Figure 3 indicates. For instance, when processing a $30,720 \times 30,720$ matrix size, MLib requires a 4.03 (with over-partitioning) to 4.84 (without over-partitioning) times larger memory allocation.

Comparative analysis of memory usage (see Figure 3) shows that JAMPI is generally on par (within 30%) of the pure MPI implementation, while MLib typically requires approximately four times the amount of memory allocation that the MPI based approaches demand, with regular MLib requiring typically 15% to 50% more memory than over-partitioned implementations.

Table 2. Out-of-memory boundary sizes for MLib, in normal (MLib) and over-partitioned (MLib (op)) mode.

Cores	MLib	MLib (op)
1	4096	10,240
16	10,240	15,360
64	20,480	25,600
256	30,720	51,200

3.2. Performance

Comparing performance in terms of execution time shows a similar picture in all multi-core environments. MLib, both with and without over-partitioning, presents a lower execution time compared to JAMPI in trivial-sized matrices (4096×4096 for 16- and 64-core environments, $10,240 \times 10,240$ for 256-core environments).

However, MLib execution times rapidly increase. At the largest matrix sizes with 256 cores, for instance, JAMPI consistently outperforms over-partitioned MLib (see Table 3) and while somewhat slower than a pure MPI implementation on a $30,720 \times 30,720$ element matrix, it does not have to

contend with MPI's out-of-memory limitation. As Figure 2 shows, a pure MPI implementation is somewhat faster than JAMPI, but JAMPI greatly outpaces (by as much as 25%) the runtime of an over-partitioned MLLib implementation over the same matrix. This demonstrates a clear benefit for large matrix operations in particular, given that this additional performance increase does not come at the cost of additional development time burden, since JAMPI is implemented in native Spark. Detailed statistics for each permutation of application, core count and matrix size are enclosed as Supplement 1.

The comparative analysis of performance indicators shows that, while a pure MPI implementation is somewhat faster than JAMPI, this difference is significantly smaller than the difference between the MLLib implementation and JAMPI, proving that JAMPI is an efficient and fast alternative to pure MPI applications without a significant performance overhead.

Table 3. Large matrix runtime comparison (runtime in 10^{11} seconds) between MLLib, overpartitioned (op) MLLib, JAMPI and pure MPI.

Matrix Size	MLlib (SD)	MLlib (op) (SD)	MPI	JAMPI (SD)	% JAMPI vs. MLLib (op) Speed-Up
30,720	3.782 (0.057)	2.868 (0.0124)	1.837 (0.008)	2.284 (0.009)	25.569%
51,200	–	11.554 (0.022)	–	9.749 (0.015)	18.514%

4. Discussion

Cannon's algorithm can be implemented quite conveniently using a barrier task within Spark, providing a native interpretation of this highly efficient distributed linear algebra primitive. By using barrier tasks to reimplement matrix primitives with Panama's built-in efficient vectorization and asynchronous communication (as provided by `nio` in this case), very significant performance gains can be effected on frequently used tasks. The proposed implementation of Cannon's algorithm, for instance, has yielded an almost 25% decrease in execution time, and has been superior to the MLLib implementation on all core sizes above trivial matrix sizes. While this algorithm is limited to square matrices, the general effectiveness gains are indicative of a strong theoretical and practical benefit of further research in ways efficient matrix primitives can be integrated with big data solutions such as Apache Spark. Further research in this field is required to create a coherent stack of matrix primitives in order to allow for modern deep learning applications, relying greatly on such building blocks to leverage the performance benefits of big data solutions in storing and managing data as a layer of an integrated framework of large-scale machine learning.

Supplementary Materials: The following are available at <http://www.mdpi.com/2504-2289/4/4/32/s1>.

Author Contributions: Conceptualization, T.F. and N.A.P.; methodology, T.F. and C.v.C.; software, T.F.; validation, T.F. and N.A.P.; formal analysis, T.F. and N.A.P.; writing—original draft preparation, T.F., N.A.P. and C.v.C.; writing—review and editing, C.v.C.; visualization, T.F. and C.v.C.; supervision, C.v.C.; project administration, C.v.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Guo, Y.; Liu, Y.; Oerlemans, A.; Lao, S.; Wu, S.; Lew, M.S. Deep learning for visual understanding: A review. *Neurocomputing* **2016**, *187*, 27–48.
- Voulodimos, A.; Doulamis, N.; Doulamis, A.; Protopapadakis, E. Deep learning for computer vision: A brief review. *Comput. Intell. Neurosci.* **2018**, *2018*, doi:10.1155/2018/7068349.
- Spencer, M.; Eickholt, J.; Cheng, J. A deep learning network approach to ab initio protein secondary structure prediction. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2014**, *12*, 103–112.
- Alipanahi, B.; Delong, A.; Weirauch, M.T.; Frey, B.J. Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. *Nat. Biotechnol.* **2015**, *33*, 831–838.

5. Zhang, S.; Zhou, J.; Hu, H.; Gong, H.; Chen, L.; Cheng, C.; Zeng, J. A deep learning framework for modeling structural features of RNA-binding protein targets. *Nucleic Acids Res.* **2016**, *44*, e32–e32.
6. Wei, L.; Ding, Y.; Su, R.; Tang, J.; Zou, Q. Prediction of human protein subcellular localization using deep learning. *J. Parallel Distrib. Comput.* **2018**, *117*, 212–217.
7. Deselaers, T.; Hasan, S.; Bender, O.; Ney, H. A deep learning approach to machine transliteration. In Proceedings of the Fourth Workshop on Statistical Machine Translation, Association for Computational Linguistics, Athens, Greece, 30–31 March 2009; pp. 233–241.
8. Socher, R.; Bengio, Y.; Manning, C. Deep learning for NLP. *Tutor. Abstr. ACL* **2012**. Available online: <https://nlp.stanford.edu/courses/NAACL2013/> (accessed on 2 November 2020).
9. Young, T.; Hazarika, D.; Poria, S.; Cambria, E. Recent trends in deep learning based Natural Language Processing. *IEEE Comput. Intell. Mag.* **2018**, *13*, 55–75.
10. Otter, D.W.; Medina, J.R.; Kalita, J.K. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Trans. Neural Netw. Learn. Syst.* **2020**. doi:10.1109/TNNLS.2020.2979670
11. Bar, Y.; Diamant, I.; Wolf, L.; Lieberman, S.; Konen, E.; Greenspan, H. Chest pathology detection using deep learning with non-medical training. In Proceedings of the 2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI), New York, NY, USA, 16–19 April 2015; pp. 294–297.
12. Havaei, M.; Guizard, N.; Larochelle, H.; Jodoin, P.M. Deep learning trends for focal brain pathology segmentation in MRI. In *Machine Learning for Health Informatics*; Springer: New York, NY, USA, 2016; pp. 125–148.
13. Liu, Y.; Gadepalli, K.; Norouzi, M.; Dahl, G.E.; Kohlberger, T.; Boyko, A.; Venugopalan, S.; Timofeev, A.; Nelson, P.Q.; Corrado, G.S.; et al. Detecting cancer metastases on gigapixel pathology images. *arXiv* **2017**, arXiv:1703.02442.
14. Stead, W.W. Clinical implications and challenges of artificial intelligence and deep learning. *JAMA* **2018**, *320*, 1107–1108.
15. Campanella, G.; Hanna, M.G.; Geneslaw, L.; Miraflor, A.; Silva, V.W.K.; Busam, K.J.; Brogi, E.; Reuter, V.E.; Klimstra, D.S.; Fuchs, T.J. Clinical-grade computational pathology using weakly supervised deep learning on whole slide images. *Nat. Med.* **2019**, *25*, 1301–1309.
16. Lehman, C.D.; Yala, A.; Schuster, T.; Dontchos, B.; Bahl, M.; Swanson, K.; Barzilay, R. Mammographic breast density assessment using deep learning: Clinical implementation. *Radiology* **2019**, *290*, 52–58.
17. Du, M.; Li, F.; Zheng, G.; Srikumar, V. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 1285–1298.
18. Shone, N.; Ngoc, T.N.; Phai, V.D.; Shi, Q. A deep learning approach to network intrusion detection. *IEEE Trans. Emerg. Top. Comput. Intell.* **2018**, *2*, 41–50.
19. Chalapathy, R.; Chawla, S. Deep learning for anomaly detection: A survey. *arXiv* **2019**, arXiv:1901.03407.
20. Wang, H.; Wang, N.; Yeung, D.Y. Collaborative deep learning for recommender systems. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Denver, CO, USA, 12–16 October 2015; pp. 1235–1244.
21. Deng, S.; Huang, L.; Xu, G.; Wu, X.; Wu, Z. On deep learning for trust-aware recommendations in social networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2016**, *28*, 1164–1177.
22. Karatzoglou, A.; Hidasi, B. Deep learning for recommender systems. In Proceedings of the Eleventh ACM Conference on Recommender Systems, Como, Italy, 27–31 August 2017; pp. 396–397.
23. Batmaz, Z.; Yurekli, A.; Bilge, A.; Kaleli, C. A review on deep learning for recommender systems: Challenges and remedies. *Artif. Intell. Rev.* **2019**, *52*, 1–37.
24. Zadeh, R.B.; Meng, X.; Ulanov, A.; Yavuz, B.; Pu, L.; Venkataraman, S.; Sparks, E.; Staple, A.; Zaharia, M. Matrix Computations and Optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*; ACM: New York, NY, USA, 2016. doi:10.1145/2939672.2939675.
25. Gu, R.; Tang, Y.; Wang, Z.; Wang, S.; Yin, X.; Yuan, C.; Huang, Y. Efficient large scale distributed matrix computation with spark. In Proceedings of the 2015 IEEE International Conference on Big Data, Santa Clara, CA, USA, 29 October–1 November 2015; pp. 2327–2336.

26. Misra, C.; Bhattacharya, S.; Ghosh, S.K. Stark: Fast and Scalable Strassen's Matrix Multiplication using Apache Spark. *IEEE Trans. Big Data* **2020**, doi:10.1109/TBDATA.2020.2977326.
27. Coppersmith, D.; Winograd, S. Matrix multiplication via arithmetic progressions. In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 31 May–3 June 1987; pp. 1–6.
28. Williams, V.V. Multiplying matrices faster than Coppersmith-Winograd. In Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 20–22 May 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 887–898.
29. Cannon, L.E. A Cellular Computer to Implement the Kalman Filter Algorithm. Ph.D. Thesis, Montana State University-Bozeman, College of Engineering, Bozeman, MT, USA, 1969.
30. Lee, H.J.; Robertson, J.P.; Fortes, J.A. Generalized Cannon's algorithm for parallel matrix multiplication. In Proceedings of the 11th International Conference on Supercomputing, Vienna, Austria, 7–11 July 1997; pp. 44–51.
31. Li, Y.; Li, H. Optimization of parallel I/O for cannon's algorithm based on lustre. In Proceedings of the 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science, Guilin, China, 19–22 October 2012; pp. 31–35.
32. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. Cudnn: Efficient primitives for deep learning. *arXiv* **2014**, arXiv:1410.0759.
33. Jules Damji. *Bay Area Apache Spark Meetup Summary at Databricks, HQ in San Francisco HQ*; Databricks Inc.: San Francisco, CA, USA, 19 July 2018.
34. Meng, X. Project Hydrogen: State-of-the-Art Deep Learning on Apache Spark. 2018. Available online: <https://www.slideshare.net/databricks/project-hydrogen-stateoftheart-deep-learning-on-apache-spark> (accessed on 2 November 2020).
35. Foldi, T.; von Csefalvay, C.; Perez, N.A. JAMPI: Efficient matrix multiplication in Spark using Barrier Execution Mode. *arXiv* **2020**, doi:10.5281/zenodo.3908556.

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).