# Matrix Multiplication Offloading in Distributed Systems

Nachiket Patel
nnpatel5@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

## Abstract

Despite living in an era where GPU's have become a prevalent force when considering any kind of single instruction multiple data (SIMD) operations does not change the fact that there are large number of systems out there that do not have access to a GPU. Especially cloud based virtual machines, generally lack GPU's, even though they are becoming more common, they are still rare and expensive resources. As such CPU utilization times when performing matrix on these types of machines can be time consuming, to this extent a distributed approach can be used to speed up large matrix multiplication computations.

The proposed solution uses a centralized approach to distributing the multiplication to worker nodes, and collecting the results from the workers when the sub-task is completed while having the ability to detect a sub-task failure and re-offloading that specific task to another worker. With large matrices, transporting the data amounts of data in itself is time consuming, to address this a compression library is used to compress the data before communicating between the main node and the worker. To mitigate the time overhead of re-compression in the event of worker node failures. are utilized to To further improve performance, a high level caching mechanism is also utilized to reduce the make re-offloading failed sub-tasks faster.

The overall results show a large time saving, with large matrices of dimension of 8000x8000 a reduction of ~28% was measured. With larger matrices the saving is even larger up ~55% for matrices with size of 16000x16000.

***Keywords:*** Distributed System, Matrix Multiplication, Task Offloading, Task Splitting

## 1 Introduction

Parallelizing matrix multiplication in through the distributed approach shown in this paper can be viewed analogously to the Amdahl's law where the amount of time saved in the limit eventually diminished with increasing number of sub-tasks. The major motivations behind this system is to speed up matrix multiplication on systems without GPU's, especially with an increase in machine learning algorithms, these algorithms perform a lot of matrix multiplication and can take really long to perform on CPU only machines. This system aims to provide a mechanism which can be used to distribute the matrix multiplication and shorten the compute time while overcoming the challenges of using distributed systems such as node failures, limited bandwidth/speed of connections between nodes.

The main objectives are to split matrix multiplication up into equal parts which can be distributed to worker nodes in parallel. Utilize compression to reduce the raw amount of bandwidth consumed when sending data from the main node to the worker nodes, this will in-turn also reduce the amount of time to send data. Furthermore, the granularity of the sub-task is also going to be a critical part of how long much time is saved in the distributed approach compared to computing on a single node. Finally the overall objective is to achieve all the objectives stated above to create a system/framework to perform distributed matrix multiplication which can save time over local computations.

To my experience there is no exact alternative to the system being described in this paper to my knowledge but, other distributed task offloading systems exist which aim to achieve similar objectives in terms of saving time [2], reduce compute time, and allow low power devices to perform large computations without the need to pack powerful hardware [1]. Before implementing the method proposed in the design section, sources such as [3] were used to understand the theoretical implications of the proposed system and understand its potential. The system proposed in this paper shows potential to reduce ~55% of the compute time for large matrices, although this time saving is reduced under situations if worker nodes fail, the specific details are displayed in the experimental evaluation section. The caching mechanism put in place to mitigate overhead of re-offloading under node failure saves ~2s of the overall compute time in the test cases described in the evaluation section.

### 1.1 Outline

The rest of the paper will have the following format; starting with the system architecture, followed by the algorithms, and finally the test bed and the results of the specific tests.

- System Architecture
  - Main Node
  - Worker Node
- Algorithms
  - Multi-threaded worker node handler
  - Compression & Data transfer

- Test-bed
- Evaluations
- Discussion
- Related work
- Future work

## 2 System Architecture

Choosing the language? With so much machine learning being performed with python, and the excellent library support for mathematical operations provided by numpy. It was selected to implement this system on, to provide as easy of a use scenario as the one provided with numpy, where the following function call; *numpy.matmul(Matrix_A, Matrix_B)* can be used to perform matrix multiplication. A similar concept is abstracted for this system where *matmul(Matrix_A, Matrix_B)* can be called to perform the distributed matrix multiplication. With this in mind, the current state of the system is abstracted into a class where an instance of an object has to be obtained to perform these operations, but with small modification it can be set up to behave just like 'numpy.matmul(Matrix_A, Matrix_B)' does for local computes. Even though the python is not the fastest language to use, the numpy library is predominately written in c/c++ underneath for performance reasons so the matrix multiplication which occurs is done with c/c++ code under python. For 64-bit floating point values numpy is optimized to utilize multi-threading by default to utilize all cores on a system when performing large multiplications, this will provide large speed-up in itself on the worker node side when it is computing the sub-tasks.

Next, let's discuss the communication that occurs between the main node and the worker nodes. The main node socket server has to be started and then the worker nodes can connect to the main node if they know the server IP and port number. Once the worker node has connected, it terminated the connection because the required information (IP, Port) about the worker have been acquired by the main node. When the main node attempts to distribute a matrix multiplication, each thread attempts to establish a connection to a worker node then follows the sequence of communication described in figure 1.

The steps that occur on the main node can be summarized by the following;

1. Receiving the multiplication task
2. Determining if enough workers are connected to the worker node to split the multiplication and distribute it.
3. Dispatch a thread to handle each worker
4. Each thread compresses the data which needs to be sent to the corresponding worker and saves it to the high level caching mechanism
5. Each thread sends the data to each worker and waits for the results to arrive

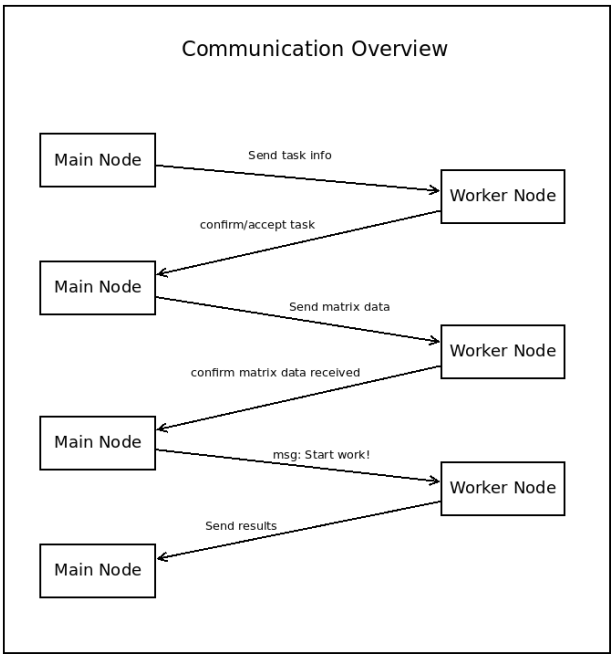


**Figure 1.** Shows the communication mechanism that takes place from making connection to the worker node to receiving the final results back from the worker node.

6. Upon arrival of the results the worker stores the results in the data structure provided to it when the thread was launched
7. Finally the combined results are returned to the user

An example of an execution process can be seen in figure 2. The list of workers are stored in a custom data structure which is implemented utilizing a two sets, one holds the free workers, and the other holds the currently occupied workers. This structure is protected by a mutex lock. The caching mechanism is a hash map, the key is a unique identifier which identifies the portion of the initial matrix which is compressed and stored as the value.

Moving on to the worker node steps which are more simple compared to that of the main node side.

1. The first step is the receiving of the compressed data (the data is decompressed by the send and receive module provided along with the worker code)
2. Compute the results
3. Finally send the results back using the send and receive module

A key difference to note about the worker sending results back to the main node as compared to the main node sending data to the worker is the absence of caching the compressed data. This is because the worker is not designed to recover from a main node failing, because if the main node failed then the whole multiplication is lost due to the centralized approach. A diagram of the steps on the worker node side is

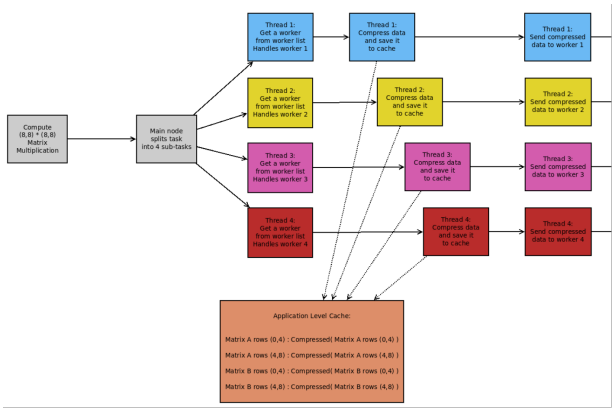


**Figure 2.** Shows the steps that take place on the main node side when a matrix multiplication is split up into sub-tasks.

shown in figure 3 which shows the parallel execution of 4 worker nodes in the 4 sub-task split example mentioned in the main node steps previously.

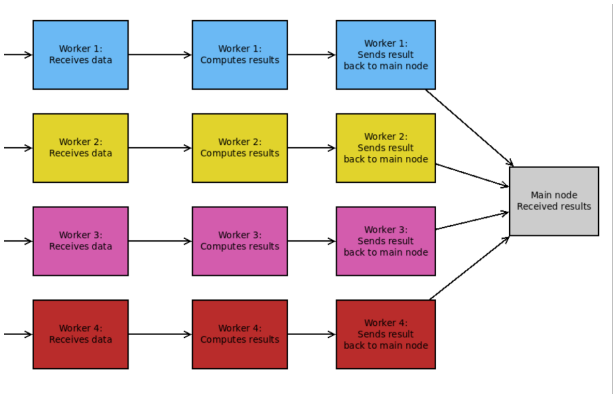


**Figure 3.** Shows the steps that take place on the main node side when a matrix multiplication is split up into sub-tasks.

## 3 Approach/Algorithm

Before looking at the algorithm for the main node, first let's look at the algorithm for sending and receiving matrix data. This is contained withing the 'transportMM.py' file/module which contains two main components. First is how data is sent, the 'send_mm(sendSocket, frame, atol=None, compDataCache=None, key=None)' function call requires two mandatory arguments, the socket object relating to the worker node which is receiving the data, and the 'frame' is the numpy matrix. The optional arguments 'atol' can be used if the user wants to compress the matrix data with a specific precision tolerance. By default the compression is lossless, but if the user specifies they do not need full precision, a tolerance level can be defined. The final two arguments are used to perform the high level caching mechanism., for the main node these are defined if the caching flag is enabled, the worker node never caches the compressed data. After the data is compresses it is wrapped in 'struct' which is used to ensure the receiver knows the size of the package being received. The second major part of the 'transportMM.py' module is receiving the data, this follows on from the send function where, the 'struct' struct which was sent contains the size of hte data being received so, the receiver will try and collect the amount of data specified, or until the buffer is empty and in the event that it does not receive the correct amount it will fail and return 'none' to indicate an error.
The worker node algorithm is simple, initially the worker establishes a connection to the main node, then proceeds to enter an infinite while loop in which it keeps waiting for tasks to be sent. After this it follows the communication protocol described in 1. The main node algorithm is more involved, when an instance of 'DMM' is created a thread is dispatched, whose sole purpose is to connect new worker nodes and save their connection information into the 'workerList'. Once the object has been created, the matmul function can be called to perform the distribute the multiplication, the following pseudocode shows the algorith:

```
waitForWorker(sleepTime, tries, taskSplit, spareWorkers)
split <--- compute how to split based on taskSplit value
# the split is a has two values, how to split the
# number of rows and columns
rSplit <--- split[0]
cSplit <--- split[1]
results <--- matrix of shape (rSplit, cSplit)
wThreads <--- empty list
cache <--- dict()

acquire(mutex)

for i from 0 to rSplit
    for j from 0 to cSplit
        w <--- workerList.getWorker()
        wt <--- Thread(workFunc, args=(w, subTaskID,
                        sub_Mat_A, sub_Mat_B, results,
                        cache, atol, 0))
        wThreads.append(wt)

release(mutex)

for t in wThreads
    join(t)
```

## 4 Experimental Evaluation

All the testing performed was performed on google cloud vm with 4 virtual cores @2GHz and 16GB of ram. This includes the main node and all the worker nodes. This was done to ensure results obtained can easily be compared. Whenever a test case shows a local compute time, it means it was performed on the same spec google cloud VM with 4 virtual cores @ 2GHz and 16GB of ram.
Some of the tests conducted on the system include testing the sub-task granularity to understand the optimal split size to get the best performance from distributing the matrix multiplication. Another test performed was to to understand the effect of node failures on the overall distributed compute time, and further evaluate different stages of worker node failures. To analyze the impact of the caching mechanism and evaluate its effectiveness, it was tested with a forced

node failure. Finally to see how effective the system is in saving compute time with different matrix sizes to understand at what size this distributed approach becomes useful.

### 4.1 Sub-Task Granularity

To see the at what stage the effectiveness of splitting up a task does not yield any benefits a sample test of multiplying two (10000, 10000) matrices was performed at various number of sub-tasks. An symptomatic trend was observed where beyond 8 sub-tasks, no time additional time was saved, and potentially with larger number sub-tasks the time saved could decrease due to the overhead of having to make so many splits. To see if this trend was related to the size of the matrix a larger multiplication of two (12000,12000) matrices was performed with the same number of sub-tasks. The trend seemed to stay the same as shown in figure 5, this leads to the conclusion that the trend seems to be emerging from the limited number of cores on the main node, due to the multi-threaded nature of handling each worker as the number of threads increases beyond the number of cores, each thread can no longer have its own core and will have to share with another thread.

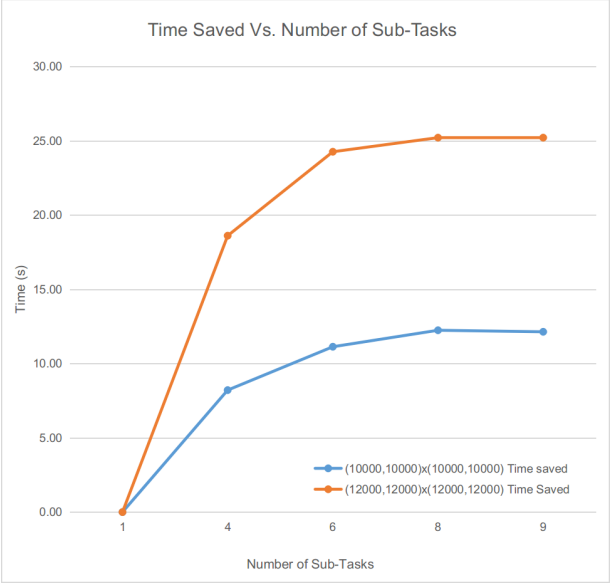


**Figure 4.** Shows the amount of time saved when splitting the task of multiplying two (10000,10000) and two (12000,12000) matrices with varying number of sub-tasks compared to the same tasks computed locally on a single node.

### 4.2 Node Failures

For this test two (10000,10000) matrices were multiplied together with a task split size of 6. This test was performed to evaluate the worker node failure recover implementation to observe to what extent are worker node failures feasible before the distributed approach compute time exceeds local compute times. This test was evaluated at three different stages of the worker node. The first (orange) is if the worker failed after receiving the matrix data, the second (yellow) is if the worker node fails after computing the results, and finally (red) is if the worker node fails at the very end when it is sending the results back to the main node. The results for these three scenarios can be seen in figure **??**. Looking at the results it is evident that a single node failure in the worst case is enough to diminish the time saved to a very small amount such that the benefit of the speed up is not enough of a motivation to expend so much bandwidth and compute resources.


**Figure 5.** The three different stages at which worker node failures were evaluated from, the best case which is after receiving the matrix data, the middle case is after computing the result of the multiplication, and finally the worst case where the failure happens at the end of the sending the results back to the main node.

### 4.3 Caching Vs. No Caching

In this test scenario the high level caching mechanism in place was disabled and enabled while computing the multiplication of two (10000,10000) matrices with a task split size of 6. The results can be observed in figure 6 along with a diagram indicating the point at which the worker node was intentionally failed to trigger re-offloading so the benefits of

the caching mechanism can be observed. The results show a 2 second benefit of enabling the caching mechanism, although this does incur a small amount of space overhead keeping track of the compressed data until the whole distributed matrix multiplication is completed.

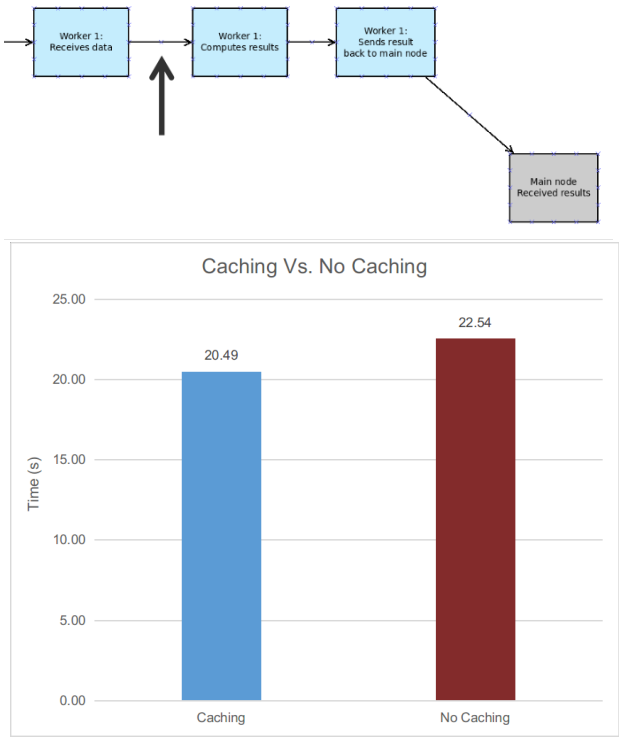


**Figure 6.** The top diagram shows indicates the stage at which the worker node was terminated to simulate a failure, the graph shows the difference in compute time of multiplying two (10000,10000) matrices together with a sub-task size of 6.

### 4.4 Time Saved

Finally the main purpose of this whole system was evaluated where size of the matrix was varied while keeping the number of sub-tasks constant at 6. This was done to evaluate the matrix size at which the distributed system shows a meaningful benefit compared to local compute times. The results were once again asymptotic as observed in 5, which was not surprising because as the matrix sizes get larger the size of the the data increases quadratically due to the square nature of matrices. It seems logical that at small sizes the overhead of splitting matrices into sub-tasks is greater then the time saved from the distributing the task. Figure 7.

Figure 8 shows the amount of time saved compared to performing the computation locally. The observed trend seems to not to be coming from Amdahl's law but instead it might be the a result of the way matrices get larger when they are doubled. More specifically take the example of a 4x4 matrix

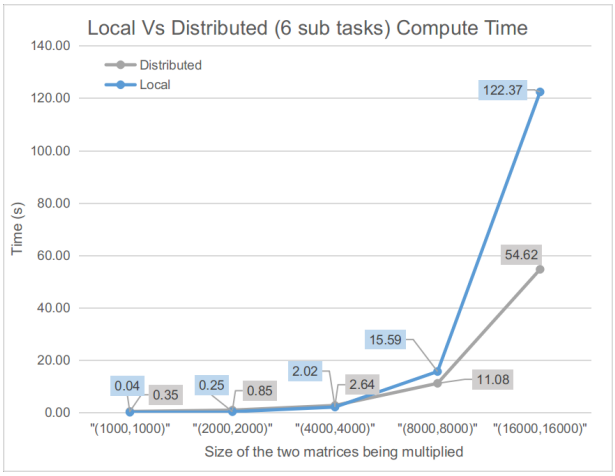


**Figure 7.** This shows the amount of time taken to compute various matrix multiplication locally and in the distributed fashion.

which has 16 elements, and if we double the shape to 8x8 which has 64 elements we do not observe a doubling of the number of elements but instead observe quadrupling of the number of elements. I think this is the reason for the asymptotic trend. Despite both the granularity and thins showing the trend they seem to be emerging from different reasons. The granularity asymptotic trend is definitely Amdahl's law but the time saving percentage is not.

$$Time\ Saved = \frac{Local\ Time - Distributed\ Time}{Local\ Time} * 100$$

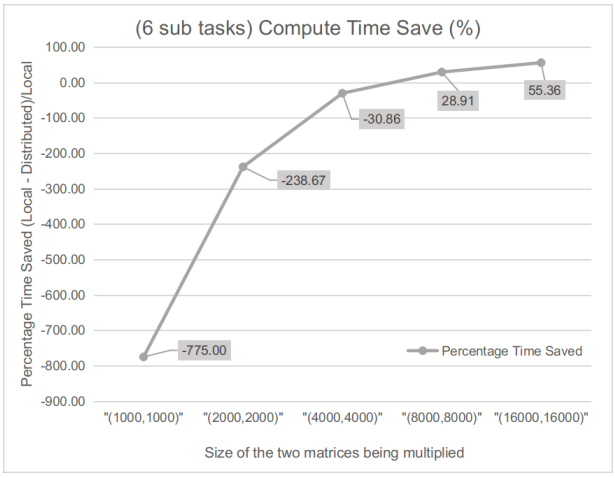


**Figure 8.** Shows the amount of time saved in terms of percentage relative to the local compute time.

## 5 Conclusion

To conclude the testing shows that for large matrix sizes the amount of time saved can be as high as ~55% and overall this

could be beneficial in clusters without GPU's. With all the testing being performed on Google cloud VM's within the same network, even though half the workers were in South Carolina and the other half in Virginia the connection speed between the two was relatively high compared to average network connections. This means to observe the same results as the ones in the testing, one would have to deploy the system in a similar environment with high bandwidth. The large amount of data that has to be transferred between the main node and worker limits the type of network on which this system can be deployed with the same types of benefits as the ones observed in this paper. The node failure testing revealed another potential limitation which is the number of node failures before the speed up is counteracted by the overhead is only one, although in the grand scheme of things if this system is used to compute hundreds or thousands of matrix multiplication then having multiple failures should not have an adverse impact on the time save from the distributed approach.

Another limitation could be the centralized nature of the system but with a decentralized approach the main concern was complexity and difficulty of determining the potential overhead of decentralizing. This approach would not yield desired results because the results show that there is already a significant amount overhead with the centralized approach that a decentralized approach would not provide better compute times, which is the main objective of this project. Various metrics were evaluated when testing the system; time saving, sub-task granularity, overhead of node failures, and the effect of an application level caching mechanism.

Finally, the main objective of providing a framework which reduced the compute time of large matrix multiplication with the use of distributed approach is successful based on the time saving presented in the evaluations section.

## References

[1] Shigeru Imai. 2012. Task Offloading Between Smartphone and Distributed Computational Resources. *Faculty of Rensselaer Polytechnic Institute* (2012), 52. http://wcl.cs.rpi.edu/theses/imai-master.pdf

[2] Z. Liu, Y. Yang, K. Wang, Z. Shao, and J. Zhang. 2020. POST: Parallel Offloading of Splittable Tasks in Heterogeneous Fog Networks. *IEEE Internet of Things Journal* 7, 4 (2020), 3170–3183. https://doi.org/10.1109/JIOT.2020.2965566

[3] Qian Yu, Mohammad Ali Maddah-Ali, and A. Salman Avestimehr. 2018. Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding. In *2018 IEEE International Symposium on Information Theory (ISIT)*. 2022–2026. https://doi.org/10.1109/ISIT.2018.8437563