Similarities and Differences Between Parallel Systems and Distributed Systems

Pulasthi Wickramasinghe, Geoffrey Fox
School of Informatics and Computing ,Indiana University, Bloomington, IN 47408, USA

In order to identify similarities and differences between parallel systems and distributed systems, comparisons were done on features and functionality of MPI, Charm++, HPX (parallel) and Spark, Flink (distributed) frameworks. This report lists important features and functionality of such frameworks and discusses each of them. This is by no mean a complete list, there may be many other important features that can be compared and discussed.

Fault tolerance

One of the main advantages that distributed frameworks such as Spark and Flink have over parallel frameworks such as MPI is the inherent support for fault tolerance. Most distributed frameworks are designed with fault tolerance as one of the main objectives. Which is why fault tolerance is supported from the very basic level of abstraction. For example RDD’s[7] in Spark which is its basic abstraction layer supports fault tolerance.

On the other hand parallel frameworks like MPI do not provide inherent support for fault tolerance. Many projects have worked on adding Fault tolerance support to parallel frameworks but none of the seem to be as seamless as in the case of distributed frameworks where the algorithm developer does not have to consider anything regarding fault tolerance when implementing the algorithm. Most of the fault tolerance support created around MPI are based on checkpoint restart model, Run through stabilization[1] was another fault tolerance method proposed by the fault tolerance working group. Charm++ has double checkpoint-based fault tolerance support[2] which performs checkpointing in-memory.

Support of collectives

Parallel frameworks provide a more rich set of collective operations than distributed frameworks. Distributed frameworks mainly provide support for broadcast, scatter, gather and reduce operations. Parallel frameworks provide a much more richer set of collectives such as allReduce, allGather. These collectives are implemented in an highly efficient manner in parallel frameworks such as MPI and HPX. Frameworks such as Spark and Flink have very limited communication between worker nodes other than broadcast variables and accumulators. Some other frameworks such as Harp do provide such collectives on top of Hadoop. Because of the absence of such collectives it is not possible to program algorithms in BSP model, which is a very efficient model to use for some algorithm implementations.

Dynamic resources utilizations

Another major area of concern for parallel and distributed frameworks is dynamic resource utilization. It is important that frameworks are able to scale up and scale down the number of resources that are used with the load of work that is done. Distributed systems are well adept for this and are able to dynamically scale with the workload and the amount resources (number of nodes) that are available. Parallel frameworks such as HPX and Charm++ do also support dynamic resource allocation but available implementations in MPI do not provide such functionality.

Communication protocols

Parallel systems such as MPI, HPX and Charm++ support high end communication protocols such as Infiniband and GEMINI in addition to Ethernet. Which gives parallel frameworks the ability to leverage high end performance from the networks. On the other hand distributed systems such as Spark, Flink only support ethernet by default. However there are several research projects that work providing such support to distributed frameworks[1]

Level of abstraction

Another important aspect of distributed and parallel frameworks are the level of abstraction provided to the user (algorithm developer). In this regard distributed systems provide a higher level of abstraction to users. Users can think of data structures such as arrays as distributed arrays. Parallel frameworks such as MPI provide a lower level of abstraction. But frameworks such as Charm++ and HPX do provide some higher level constructs. It is also noteworthy that MPI IO and MPI collectives can be considered to be higher level abstractions.

Higher level and lower level abstraction both have pros and cons. Higher level abstractions allows the user to develop algorithms without having to worry about low level details, which makes algorithm development more easy. With lower level abstractions the user has more control over the system and can obtain more performance from the system.

Performance-Ease of Use tradeoffs in fault tolerance

One of the main selling points of distributed frameworks such as Spark and Flink is the fact that the frameworks handles fault tolerance within the framework. Even though it is quite straightforward to add a fault tolerance model like checkpoint-restart to an application in parallel frameworks like MPI, it does need to be done by the programmer. Thus not having to worry about this aspect is one of the main reasons for the popularity of distributed frameworks . But this automatic fault tolerance does come with hefty price on performance as one can observe in performance differences between MPI, Spark and Flink. Sometimes the difference in performance can be more than an order of magnitude. For complex long running applications this can have a huge impact. For instance if one could write an certain application in MPI and it takes 1 day to generate a solution, implementing the same application in Spark, Flink might end up taking 10 days to reach the same solution. Of course this difference depends on the application and its features, but for some applications domains the difference is significant.

Another reason that frameworks like Spark and Flink are popular is their ease of use, automatic handling of faults is one major contributor to the ease of use, and for some application domains (which we will look into in a following section) the performance numbers achieved through the use of distributed frameworks is more than enough and thus the use of distributed frameworks are clearly justified, but for some application domains a little bit of complexity on the code is clearly justified when looking at the gain of performance and reduction of time to solution.

The performance degradation in distributed frameworks is largely due to the model they follow which allows fault tolerance to be handled automatically. Spark and Flink use a data flow model where the data objects are immutable. This allows lost partitions to be recalculated based on existing data since the data is not modified in place, which in turn allows the framework to be developed in an fault tolerant manner. The major downside of this model is that we need to create new copies of data when the data is processed since we cannot alter the original data. This means higher memory and CPU consumption. The rationale behind this decision is that achieving fault tolerance at the cost of memory and CPU time is justified since the latter is cheaper. This may very well be true for some application domains, but for some this is not. Below we take a quick look what feature of distributed frameworks affect memory consumption and CPU usage

Memory

Since data objects are immutable in Spark and Flink, each transformation creates a new set of data objects which consumes more memory than doing the transformations in place. For some applications memory can be a limiting factor and having to create new copies of data would be troublesome. For examples if the application involves large matrix operations having to keep more than one copy of the matrix would be problematic.

CPU

The main reason for higher CPU usage is overheads that are inherent in the distributed frameworks. Because tasks (workers) are not able to communicate with one another, Spark and Flink do not provide all-to-all operations such as AllReduce.This means for iterative applications which require data to be exchanged between parallel tasks at each iterations data needs to be gathered at the master process and redistributed back. This also means for each iterations a new set of tasks need to be started. This adds several overheads such as task serialization task deserialisation.

I/O

Because all-to-all operations are not available I/O overheads are also increased. There are two factors that contribute to I/O overheads. The first is because performing an AllReduce operation (as in MPI) is much more efficient than performing a reduce and then a broadcast. Secondly more overhead is added because tasks need to be distributed at each iteration.

Which applications are suited for which model.

Most popular distributed frameworks such as Spark and Flink are based on the MapReduce model, even though they extend the capability of the plain MapReduce model by providing iterative MapReduce, In-Memory processing, caching, etc., these frameworks are built around the core mapreduce model. The MapReduce model enforces an application to be consistent of Map and Reduce phases, since it is not possible to reduce all applications in this manner some applications cannot be built with these frameworks. On the other hand parallel frameworks such as MPI are much more flexible, this allows a wide variety of applications to be build with such frameworks. Although it is hard to clearly define when to use distributed frameworks and when to use parallel frameworks there are several key points that can be used to help with the decision.

Time to develop
The time to develop a solution (application) is generally lower with distributed frameworks such as Spark and Flink and requires a lower level of expertise. This is because the programing interface and API's are normally more simpler than MPI and are much easier to grasp for a beginner. Developers do not have to worry about some aspects of the program such as fault tolerance. So if the main goal is to get a solution as soon as possible at the cost of lower performance frameworks such as Spark and Flink are favorable. But it is important to note that the target application must be broken down to map and reduce phases.

Time to Solution
If the main goal of the application is time taken to get an result for a given input then MPI is the best choice. MPI perform efficiently because it has very little inherent overheads that would increase the time to solution. On the other hand distributed frameworks have several inherent overheads that are caused due to the models that they have adopted which were discussed above. This is in addition to limitations that would be caused by the MapReduce model, specially for complex applications which require a large amount of interprocess communication.

Resource Utilization
If the usage and requirement of hardware resources such as RAM and CPU need to be kept at a minimum for the application execution MPI would be the best choice. This is clear since distributed frameworks require more CPU time and RAM as discussed above.

What Applications Suit Distributed Frameworks

There are however a range of applications that work well with distributed frameworks. Below we look at several such application types. This by no mean a complete list, therefore there maybe other applications and applications types that work well in distributed frameworks that are not covered by the areas mentioned below.

Pleasingly Parallel (Map Only)

Map only applications that have little to no communication between processes are well suited for distributed frameworks. Since tasks can be run in parallel without having to do any reduction step the overheads caused are minimal. And for applications with large amounts of data distributed frameworks can be scaled up and down as needed easily. Since a failure in a single node does not affect the other tasks automatic fault tolerance in systems like Spark and Flink can recompute the lost values without any affect to currently running tasks. Examples for such applications are Protein docking, bio-imagery that involves local analytics[4].

Classic MapReduce

Classical MapReduce applications involve a single reduce operation. These are also well suited for distributed frameworks because of the same reasons as map only tasks. The ability to automatically recompute failed tasks allow distributed frameworks to be run on large scale commodity clusters which are prone to faults. Searching, Indexing and Querying are some examples for classical mapreduce applications.

Map Collective (Iterative MapReduce)

Unlike the previous two areas for iterative mapreduce choosing distributed frameworks over parallel frameworks depend on other factors like time to solution, time to develop, data communication patterns, etc. This is because most iterative mapreduce applications can be implemented much more efficiently with all-to-all operations such as AllReduce that are available in parallel frameworks. Most machine learning applications such as KMeans fall under
this category.

What Applications Suit Parallel Frameworks

Parallel frameworks such as MPI, Charm++ are very flexible and can be used to implement almost any application domain effectively. Even the areas that are listed above can be implemented with MPI however the developer needs to handle fault tolerance at the application level.

Most applications and algorithms which involve inter-process communication (halo exchange) are well suited to be implemented with parallel frameworks. Examples for such applications span a wide area including scientific applications and complex algorithms. These algorithms can leverage highly from all-to-all operations such as AllReduce and other optimizations that are available in parallel frameworks.

Summary

Table 1 summaries the findings in the report.

	Distributed Frameworks (Spark, Flink)	***Parallel Frameworks (MPI, HPX, Charm++)***
Fault tolerance	Built in, programmer does not need to write additional code to handle.	Generally left to be handled by the programmer. Needs additional code.
Support of collectives	Limited, all-to-all collectives are generally not supported.	Fully Support, and highly optimized.
Dynamic resources utilizations	Generally Available in most frameworks.	Available in some frameworks (ex. HPX, Charm++), But not supported currently in MPI implementations.
Communication protocols	Generally only supports ethernet.	Supports a wide variety of protocols (ex. Infiniband, GEMINI, Ethernet).
Level of abstraction	High level of abstraction. The programmers generally do not need to think about low level details.	Both Low level and high level depending on implementation. But generally at a lower level than distributed frameworks.
Memory Usage	Comparatively higher. Constraints such as data immutability are main reasons for higher memory usage.	Comparatively lower. In place changes to data structures allow for lesser memory usage.
CPU Usage	Comparatively higher. Because of overheads caused by framework model. Especially for fault tolerance	Comparatively lower. Very little overheads and optimized collective functions allow for lower CPU usage.
I/O Usage	Comparatively higher. The inability to perform all-to-all collective operations is one main factor contributing to higher I/O usage.	Comparatively lower. Highly optimized collective operations allow for lower I/O usage.
Time to develop	Comparatively Lower. Can be generally developed faster because of simplified API's.	Comparatively Higher. Generally a little more complex to code.

Level of Expertise required	Low. Can be used to develop applications with lower level of expertise because of simplified API's	High. Programmers need a good understanding of parallel computing concepts and data structures to develop applications.
Time to Solution (Run time)	Comparatively Higher. Because of the inherent overheads in the frameworks, time to solution tends to be higher	Comparatively Lower. The low amount of overheads and optimized collective functions allow lower time to solution. Also easy to do low level optimizations.

Table 1: Summary

References

[1] Hursey, Joshua, et al. "Run-through stabilization: An MPI proposal for process fault tolerance." European MPI Users' Group Meeting. Springer Berlin Heidelberg, 2011.

[2] Parallel Programming Laboratory: Fault Tolerance Support, http://charm.cs.uiuc.edu/research/ft, accessed Nov 20 2016.

[3] High-Performance Big Data (HiBD), http://hibd.cse.ohio-state.edu/, accessed Nov 14 2016.

[4] Fox, Geoffrey C., et al. "Towards an understanding of facets and exemplars of big data applications." Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday. ACM, 2014.

[5] Tikotekar, Anand, Chokchai Leangsuksun, and Stephen L. Scott. "On the survivability of standard MPI applications." LCI International Conference on Linux Clusters: The HPC Revolution. 2006.

[6] Bronevetsky, Greg, et al. "Automated application-level checkpointing of MPI programs." ACM Sigplan Notices. Vol. 38. No. 10. ACM, 2003.

[7] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[8] Elnozahy, Elmootazbellah Nabil, et al. "A survey of rollback-recovery protocols in message-passing systems." ACM Computing Surveys (CSUR)34.3 (2002): 375-408.

[9] Carbone, Paris, et al. "Apache flink: Stream and batch processing in a single engine." Data Engineering (2015): 28.

[10] Hursey, Joshua, et al. "Run-through stabilization: An MPI proposal for process fault tolerance." European MPI Users' Group Meeting. Springer Berlin Heidelberg, 2011.

[11] Reyes-Ortiz, Jorge L., Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." Procedia Computer Science 53 (2015): 121-130.

[12] Jha, Shantenu, et al. "Introducing distributed dynamic data-intensive (d3) science: Understanding applications and infrastructure." (2014).

[13] Jha, Shantenu, et al. "Distributed computing practice for large‐scale science and engineering applications." Concurrency and Computation: Practice and Experience 25.11 (2013): 1559-1585.