Removing Bottlenecks in Big Data Processing Platforms

Sobhan Omranian Khorasani
Distributed Systems Group, EEMCS
Delft University of Technology
S.OmranianKhorasani@tudelft.nl

***Abstract*— Emerging Big Data analytics applications require a significant amount of computational power. As the demand for computational resources continues to grow, analytical needs for Big Data can be satisfied with high performance computing models. Additionally, High Performance Computing (HPC) has evolved over the years to meet the ever-increasing demands for processing speed. While some data-intensive applications are intended to be executed on commodity hardware in a "scale out" architecture, there are certain situations in which ultra-fast, high-capacity HPC "scale up" approaches are preferred.**

Current Big Data processing platforms like Hadoop and Spark which use the Java Virtual Machine (JVM) as an execution environment are abstracting away the architecture and low-level properties of modern systems. This can become a potential bottleneck in large-scale applications and prevent many system- and network-related optimizations. Therefore, the aim of my research is to investigate the performance liabilities of existing large scale data processing frameworks compared to HPC and offer system-aware and efficient solutions.

I. State of the Art

The explosive growth in the number of resources from which data can be obtained has resulted in a surge of data available in digital form which is mostly unstructured and complex. Processing these huge volumes of data is both challenging and paramount as it can help organizations to turn hard numbers into insights that can steer decisions and gain competitive advantages. This has led to the design and development of many Big Data processing frameworks such as Hadoop and Spark, which thrive to handle the various complexities (e.g., velocity, variety, volume) of Big Data analytics. As applications become more data intensive, there is also a need for adopting different approaches to meet their computational requirements. A typical approach would be horizontal scaling (i.e., scale out) by simply adding more nodes for processing. Cloud computing has significantly helped with the situation by providing cheap and flexible resources. However, this would not be a feasible strategy for organizations who have invested in their own local hardware and reached the saturation point of their existing infrastructure. Additionally, applications are becoming increasingly demanding, therefore merely adding more machines is not going to be viable in the long term. Another concern for this approach would be power consumption since each node comes with its own power supply, hence adding more resources would proportionally increase the energy footprint.

In order to address the shortcomings of horizontal scaling, alternative solutions which enable more efficient use of existing infrastructure (i.e., scale up) should not be neglected. High Performance Computing (HPC) has always been associated with applications of tremendous computational needs and could be a viable candidate. Traditional HPC applications such as graph processing and machine learning are becoming more data intensive due to the larger input generated from more powerful scientific instruments and more output data as a result of smarter mathematical models and algorithms. Some of the existing approaches in combining HPC and big data include Nimbus [1] and Thrill [2] in which performance increase is achieved through producing highly-optimized native code.

II. Intended Approach

The convergence of HPC and Big Data analytics, High Performance Data Analytics" (HPDA) [3], is an exciting opportunity in which they both can complement the needs of one another. However, the composition of these paradigms is a challenging task involving various aspects such as data management and computing efficiency. Consequently, any existing or new solution need to be carefully analyzed in the context of HPDA.

As an example, current big data processing platforms such as Hadoop and Spark are abstracting away the architecture and low-level properties of the modern systems by using the Java Virtual Machine (JVM) as their execution environment. Notable features of the JVM, while deemed valuable in other contexts, could become a potential bottleneck at the scale of HPC applications. For instance, garbage collection is known to increase the jitter in the behavior of applications, while executing bytecode could incur overhead. Traditionally, in applications where performance matter (e.g., HPC), using a high degree of abstraction like the JVM is prohibitive because it prevents many system and network-related optimizations from being applied. This effect is exacerbated by the emergence of novel hardware where higher performance is achieved through adhering to the low-level hardware design properties. Project Tungsten in Apache Spark [4] is an example of avoiding such abstractions by moving the data out of the garbage-collected heap in order to apply optimizations such as cache-aware computation and code generation to push the performance closer to the limits of modern hardware.

The purpose of this research is two-fold. First, we analyze the shortcomings of existing platforms and try to find remedies

Memory Policy	Description
Default	This is the default behavior of Spark.
Local	All memory accesses are local.
Remote	All memory accesses are remote.

TABLE I

DIFFERENT MEMORY POLICIES USED IN THE EXPERIMENTATION

by modifying the behavior of big data processing systems. E.g., we have conducted experiments regarding how memory architecture of modern machines affect Spark. (Section III)

Secondly, we are planning to use our experience from the experimental results to build a novel system based on a different paradigm that is more system-aware and closer to how an HPC system would be designed.

III. EVALUATION PLAN

As an initial step toward analyzing the potential bottlenecks in existing platforms, we wanted to see how the memory layout of modern servers affect big data analytics.

Modern computer architecture is increasingly moving toward turning individual machines into small-scale networks. These systems consist of several nodes (i.e., sockets), each containing a subset of the system's CPU cores and a portion of its RAM. If a core accesses memory from within the same node, it is called a local access. Similarly, an access to a different node is called a remote access. Remote accesses have longer latencies than local ones, because they must traverse one or more interconnect links. The latency of memory-access times is hence non-uniform, and such systems are referred to as NUMA (non-uniform memory access).

Ideally, in the absence of contention, all memory accesses should be local as they are faster. However in case of Spark, there is no guarantee that a thread which allocates memory would be on the same node as the thread accessing that memory. This would incur a longer latency by issuing a remote memory access. Therefore, the aim of this experiment is to see how NUMA-architectured systems affect the performance of Spark. The experiments were conducted on the DAS-5 with up to 4 machines. Each machine consists of two NUMA sockets, each with 8 physical cores (16 logical cores) and 32GB of RAM. The selected workload for this experiment is Terasort and the input size is set to 30GB and the evaluation criteria is the total application execution time.

In order to have access to all the cores and memory, two Spark executors were launched on every node. Each executor uses 16 (logical) cores and 24GB of RAM. Then, the *numactl* library in Linux was used to control the CPU and memory placement for each executor's threads.

To understand the performance penalty of NUMA in Spark, the experiments were ran with 3 different memory policies shown in Table I. The experiment results are shown in Figure 1. As expected, forcing all memory accesses to be remote has the worst performance. Compared to the default version, the execution time is increased by 54.07% in the worst case (1 machine). More importantly, in the Local version where all memory accesses are local, the execution time is reduced by 15.82% in the best case compared to the default version.

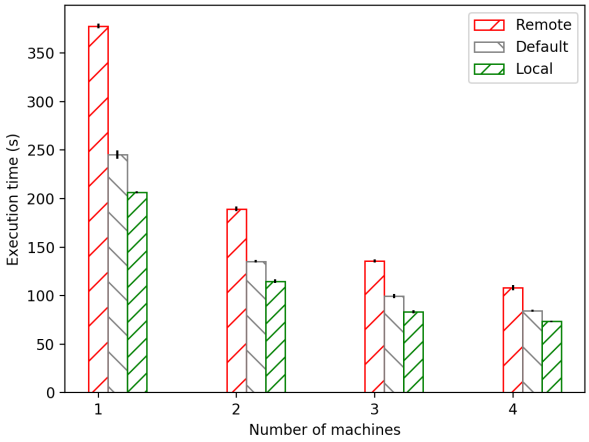

Fig. 1. The effect of NUMA for terasort application in Spark

This experiment is an example of how respecting the design properties of modern hardware is important to achieving better performance. Specifically in case of Spark, where by default one executor is launched on each machine, we found out that it is more efficient on larger machines to partition the system in such a way that there is one executor instance on each socket.

As the next step, we plan to conduct additional experiments on emerging hardware platforms to investigate other important aspects such as memory, compute, storage and networking.

IV. CONCLUSIONS

The experiment shown in Section III is an example of how existing solutions need to adapt to various features of modern hardware in order to be more efficient. Namely, NUMA-aware scheduling can improve the performance of Spark. In the future, we plan to conduct more experiments in order to improve big data processing platforms as well as the JVM in large scale applications. That would give us proper insight into designing a novel framework that respects the underlying features of the system for efficient big data analytics.

V. ACKNOWLEDGMENTS

This research is supervised by Jan Rellermeyer and Dick Epema at TU Delft.

REFERENCES

[1] O. Mashayekhi, H. Qu, C. Shah, and P. Levis, "Execution templates: Caching control plane decisions for strong scaling of data analytics," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, (Berkeley, CA, USA), pp. 513–526, USENIX Association, 2017.

[2] T. Bingmann, M. Axtmann, E. Jobstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders, "Thrill: High-performance algorithmic distributed batch data processing with c++," in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, dec 2016.

[3] B. Schmidt and A. Hildebrandt, "Next-generation sequencing: big data meets high performance computing," *Drug Discovery Today*, vol. 22, pp. 712–717, apr 2017.

[4] "Project tungsten: Bringing apache spark closer to bare metal." https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. Accessed: 2018-09-20.