A NUMA Aware Spark™ on Many-cores and Large Memory Servers

François Waeselynck and Benoit Pelletier
Atos, 1 rue de Provence, Echirolles, France

Keywords: CloudDBAppliance, Bullsequana S, Scale-up, Spark, NUMA.

Abstract: Within the scope of the CloudDBAppliance project, we investigate how Apache Spark™ can leverage a many cores and large memory platform, with a scale up approach as opposed to the commonly used scale out one: that is, the approach is to deploy a spark cluster to few large servers with many cores (up to several hundreds) and large memory (up to several tera-byte), rather than spreading it on many vanilla servers, and to stack several Spark executor processes per cluster node when running a job. It requires to cope with the non-uniform memory access within such servers, so we inculcate NUMA awareness to Spark, that provides a smart and application transparent placement of executor processes. We experiment it on a BullSequana™ S series platform with the Intel HiBench suite benchmark and compare performance where NUMA awareness is off or on.

1 INTRODUCTION

Within the scope of the CloudDBAppliance project, we investigate how Apache Spark™ can leverage a many cores and large memory platform, with a scale up approach in mind, as opposed to the commonly used scale out one. That is, rather than spreading a Spark cluster on many vanilla servers, the approach is to deploy it on a few BullSequana™ large servers with many cores (up to several hundreds) and large memory (up to several tera-byte). Spark jobs execution enrols several executor processes – several per server, that leverage the many-cores and large memory features of the BullSequana server.

But so large servers are designed with Non Uniform memory Access (NUMA) it is necessary to cope with in order to achieve the best performance. We inculcate NUMA awareness to Spark and experiment it on a BullSequana™ S series platform in a scale up approach with the Intel HiBench suite microbenchmark.

This document exposes the general issue induced by NUMA architectures, then explains how it can be addressed for Spark applications. Then it describes the test protocol and testbed and compares the application performance and efficiency where NUMA awareness is switched on versus where it is switched off.

2 NON UNIFORM MEMORY ACCESS (NUMA)

The target hardware platform of CloudDBAppliance is a BullSequana S server, a highly scalable and flexible server, ranging from 2 to 32 processors (up to 896 cores and 1792 hardware threads), up to 32 GPUs and 48 TB RAM, and 64 TB NVRAM. Our staging platform includes a BullSequana S800 server with :8 Intel® Xeon® Platinium 8158 CPU @ 3 GHz (12 cores each), and 4 TB RAM (512 GB per CPU).

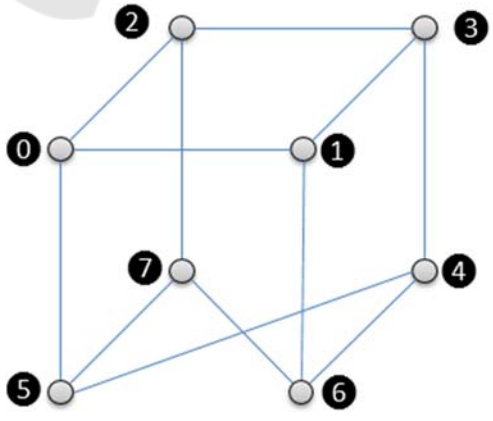

Figure 1: BullSequana S800 NUMA nodes.

BullSequana S has a Non-Uniform Memory Access architecture. Each processor has local

Waeselynck, F. and Pelletier, B.
A NUMA Aware SparkTM on Many-cores and Large Memory Servers.
DOI: 10.5220/0007905506480653
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 648-653
ISBN: 978-989-758-365-0

memory, that forms what is termed a NUMA node. Each NUMA node is linked to other NUMA nodes by UPI links, in such a way that all nodes are transitively linked, so that the whole server memory can be consistently accessed from any node. The BullSequana S800 has eight NUMA nodes numbered from 0 to 7, which are linked so that there are at most two hops between nodes as shown in Figure 1.

Each node has 3 neighbours at one hop, and four nodes at two hops. Obviously, the latency to access a portion of memory from a given node varies with the distance between the accessing node to the accessed node: where there is one hop, the latency is about twice the latency of an access to local memory; where there are two hops, the latency is about three times the latency of a local access, as shown in Figure 2. This may dramatically affect the performance of the application.

node	0	1	2	3	4	5	6	7
0:	10	21	21	31	31	21	31	31
1:	21	10	31	21	31	31	21	31
2:	21	31	10	21	31	31	31	21
3:	31	21	21	10	21	31	31	31
4:	31	31	31	21	10	21	21	31
5:	21	31	31	31	21	10	31	21
6:	31	21	31	31	21	31	10	21
7:	31	31	21	31	31	21	21	10

Figure 2: Distances between NUMA nodes.

NUMA awareness aims at maximizing access to local memory by the threads that run on every core.

3 NUMA AWARE SPARK

A Spark application comprises of:

- A driver process, that controls the whole processing of the application. The application is modelled as a direct acyclic graph (DAG). The driver understands and interprets this model, and, when operating on a given dataset, splits the processing in stages and individual tasks that it schedules and distributes to executors that run in the cluster nodes.
- Executor processes that perform the actual data processing, as instructed by the driver. The executor processes hold data parts in their memory: a task applies to a data part, and each executor receives (and runs) tasks retated to the data parts it holds. Executor processes run on cluster nodes, the so-called worker nodes. There may be one or more executors per worker node. Data processing may involve many executor processus spread on many worker nodes.

A Spark application consumes (large) data from various sources, processes it, then outputs (smaller) results to various sinks. Processing usually transforms the input data so that to build a dataset in the desired form, caches it in memory, then applies algorithms that repeatedly operate on the cached data. Unlike Hadoop - that stores intermediate results on storage (HDFS), Spark retains intermediate results in memory as far as possible, and the more memory it gets, the less pressure on the Input/output system. This makes Spark less sensitive to the input/output system than Hadoop.

Spark will try to perform initial processing close to the input data location to consumes it at the highest rate, but further computing will be more dependent on the way it (re)partitions and retains data in the memory cache of the executors. And as the processing time is usually far higher than the cumulated input and output times, the distribution of data among the executor processes (in their memory) is an important factor to consider.

Spark still writes data to file systems during shuffling operation, when data is rebalanced between executors, e.g. on sort operations or data repartitioning. Shuffling thus may put high pressure on both I/O systems and network, the later incurring CPU consumption as data has to be serialized before being transferred through the network. One way to lower it is to use large memory executors: data analytics often reduce datasets size from a very large input to a far smaller output. The more filtering and transformations that lower the size can be done in a continuous space, the less the number and size of shuffling operations will occur. Large-memory and many-core platforms enable few huge executors as well as more common approaches with several executors, for a single application as well as several ones.

Spark has been designed for scale out and is not natively NUMA aware. We have made extensions to inculcate NUMA awareness to the worker processes of a Spark cluster deployed in *standalone* mode. A worker process manages executor processes in a server of a Spark cluster: placement of executors within the server is a local concern the worker is responsible for. When a worker launches a Spark executor process, it binds it to a NUMA node, so that the threads running the tasks within the executor process access local memory, where the data parts reside. Or, if the executor does not fit to a single NUMA node, it binds it to a set of NUMA nodes close to each other. The worker process manages a table of

the running executors, so that to balance the executor placement. The placement is fully transparent to the application. The placement obey to a configurable policy, that can be set at the cluster level or at submit time.

4 THE TEST PROTOCOL

The test protocol is to run the same benchmark against input datasets of various size onto a single BullSequana S800 server, with the NUMA awareness switched off, then to re-run it with the NUMA awareness switched on, then compare results of both:

a) In terms of performance key indicator(s):
   - duration of the benchmark with NUMA awareness: T_{on}.
   - duration of the benchmark without NUMA awareness: T_{off}.
   - The improvement computed as: $(1 - T_{on} / T_{off})$.
   - The gain computed as: T_{off} / T_{on}.

b) In terms of efficiency, that is, how much resources are consumed to fulfil the same work:
   - Average CPU consumption during the bench, respectively CPU_{on} and CPU_{off}.
   - NUMA awareness CPU efficiency computed as: $CPU_{off} . T_{off} / CPU_{on} . T_{on}$.

The system under test comprises of:

- A BullSequana S800 server with 8x12 cores processors and 4 terabytes (TB) RAM memory. That is a total of 96 cores and 192 hardware threads, as hyperthreading is activated.
- a Spark 2.4 platform setup in standalone cluster mode, with NUMA awareness extension.
- Input datasets are stored in a Hadoop Distributed File System (HDFS). Four dataset input profiles of various size have been defined, as shown by Table 1.
- The used microbenchmark workload is HiBench/Kmeans (https://github.com/intel-hadoop/HiBench).
- HiBench/Kmeans is run with a parallelism of 128 tasks – where a task is run by a thread, spread on 8 executor processes with each 16 parallel tasks, and a 128 GB Heap size.
- System metrics are collected by means of a *sar* command, for off Line analysis.
- Spark metrics are collected by means of the Spark history server for off line analysis.

Table 1: Input dataset profiles.

Dataset profile	Size
gigantic	37,4 GB
halfbigdata	112,2 GB
bigdata	224,4 GB
gargantua	401,6 GB

5 ANALYSIS AND RESULTS

5.1 What a HiBench/Kmeans Run Looks like

Whatever the input dataset size, a HiBench/Kmeans run exhibits the following behaviour: it runs *N* jobs in sequence, numbered from 0 to *N-1*. Job #0 triggers the reading of the input dataset and loads it in the spark cache, i.e. the executor memory. Further jobs from 1 to *N-1* mainly operates on the data loaded in the cache. Figure 3 – extracted from the Spark web console, shows a timeline run of Hibench/Kmeans on a *gargantua* (401,6 GB) input dataset, with NUMA awareness on.

The upper pane shows the launch of the executors at the beginning of the run. The lower pane shows the 19 jobs it runs sequentially, numbered from 0 to 18, some of which have been labelled for the sake of readability. Each job includes around 3300 tasks, distributed to the 8 executor processes, each being configured to potentially run 16 tasks in parallel, so that there is a potential of 128 parallel running tasks. Analysis of the benchmark trace logged by the Spark history server, allows to compute the effective number of parallel tasks all along the run, as illustrated by Figure 4. Jobs appear clearly on the graph, with a ramp up, a steady plateau, then a fall to zero (the number of task is always zero between two jobs). The profile shows that full parallelism is effective: for each job, the desired level (128 parallel tasks) is reached after a short and steep ramp up, then stays at this level during the job, then quickly and steeply falls to zero at the end of the job.

This profile denotes well synchronized executor processes, where there are no stragglers, that is the best case to expect.

5.2 Comparing Kmeans Runs

Figure 6 gives the variation of the benchmark dura-

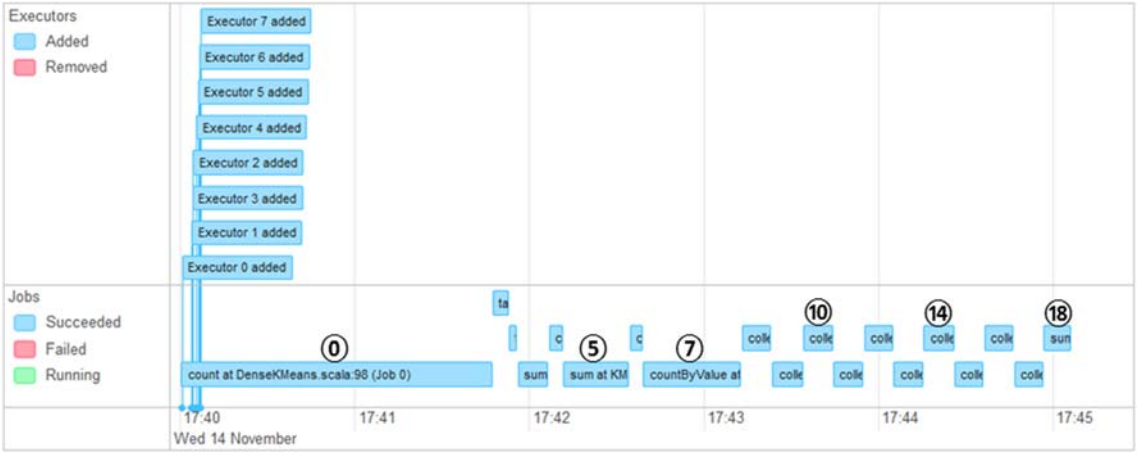

Figure 3: Hibench:Kmeans timeline view.

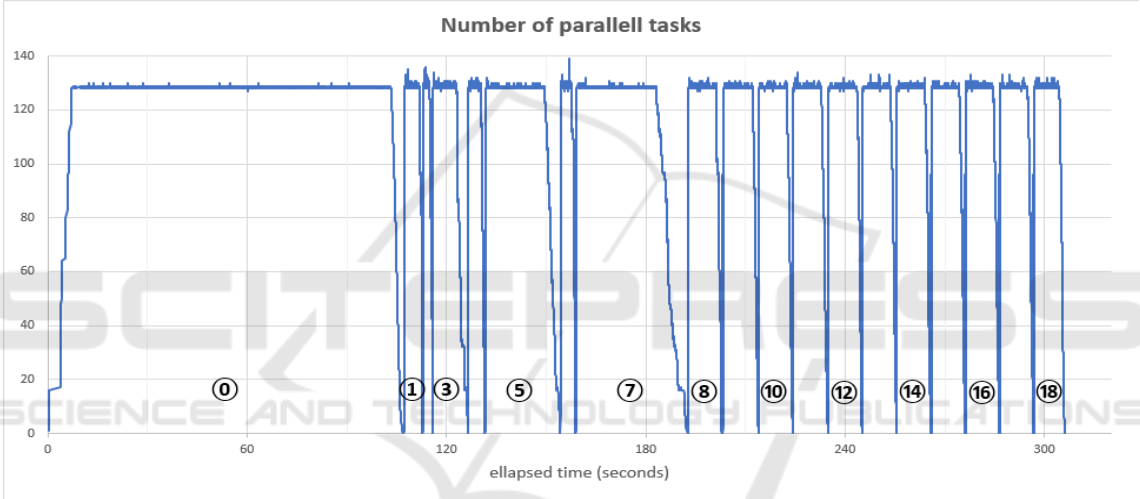

Figure 4: Hibench:Kmeans number of tasks in parallel.

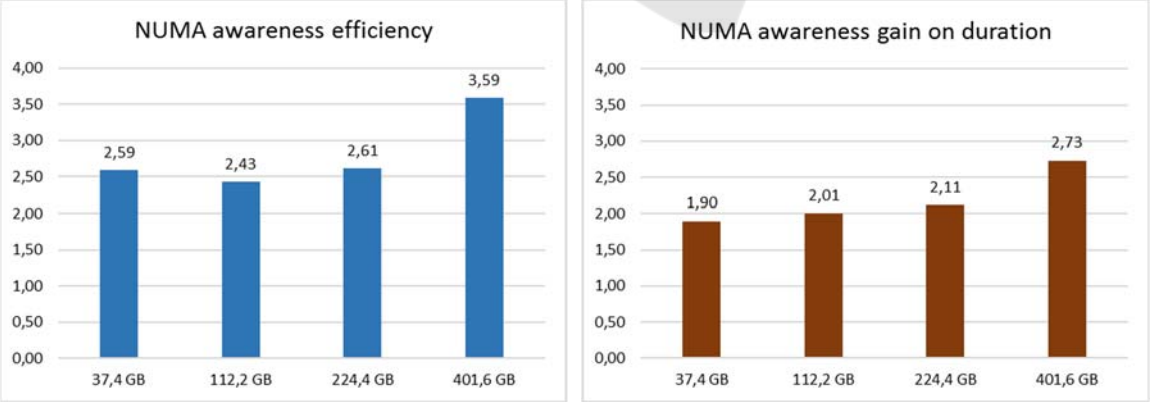

Figure 5: Hibench:Kmeans gain on duration and efficiency of NUMA awareness.

tion (in seconds), function of the input dataset size: *gigantic* (37,4 GB), *halfbigdata* (112,2 GB), *bigdata* (224,4 GB), *gargantua* (401,6 GB).

HiBench/Kmeans against *gigantic*, *halfbigdata and bigdata* have been setup with 8 executors of 16 cores and 128 Gb heap each. For *gargantua*, it is not

possible to keep the same settings. It is necessary to increase the heap size to 160 GB (where NUMA awareness is on) and even 256 GB (where NUMA awareness is off) and to tune the garbage collector: indeed, with the original 128 GB heap, there is too much pressure on the garbage collector, that causes a high CPU overhead saturating the server, so that less CPU is given to the application itself, that dramatically increases the benchmark duration to about 36 minutes.

CPU consumption is far more intensive where NUMA awareness is off and lasts more than twice as long. Figure 7 superposes the CPU consumption of two runs, one with NUMA awareness on, the other one with NUMA aware off. Moreover, the total heap is increased from 1,25 TB (NUMA on) to 2 TB (NUMA off). NUMA awareness thus increases dramatically the efficiency, as the Spark cluster will be able to process more job submissions in a given time interval.

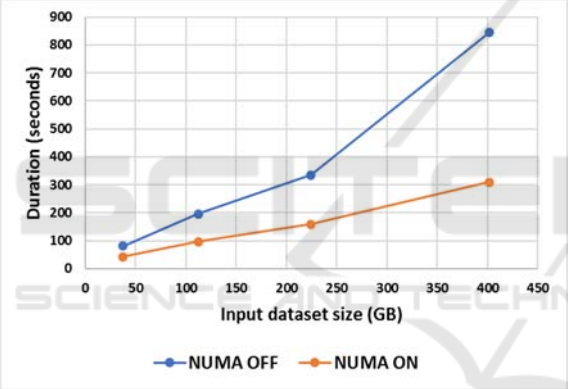

Figure 6: Hibench:Kmeans duration function of input dataset size.

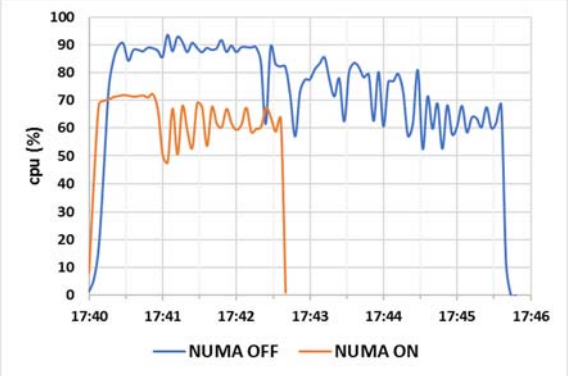

Figure 7: Hibench:Kmeans CPU consumption.

NUMA awareness shortens the benchmark duration by a factor of 2.0 (approximately) or even more. Moreover, it dramatically increases efficiency by dividing by 2.5 or even more the CPU consumption to complete a run. Figure 5 show efficiency and gain on duration for the various input dataset.

These results apply to HiBench/Kmeans only. Other workloads may have a different sensitivity to NUMA awareness.

6 CONCLUSIONS

Spark leverages High-End servers such as BullSequana S, each able to support hundreds to thousands Spark parallel tasks and up to several terabytes of in memory data, thus enabling to process large datasets within a single system or a few systems, thanks to NUMA aware scale up placement of spark executor processes.

Beyond the benefits brought by the many-cores and large-memory features, the support of Apache Pass non-volatile RAM (NVRAM) and GPUs opens new perspectives the CloudDBAppliance project is yet exploring:

- Though Spark intensively uses memory, it still writes data to file systems during shuffling operations, when data is rebalanced between executors, e.g. on sort operations or data repartitioning. Another case is for some cache management policies. The use of Apache Pass is expected to relax the high pressure it may put on the input/output system, as it exhibits higher throughput and lower latency than SSDs or NVMe.
- More and more machine learning and deep learning frameworks and libraries support off-loading to GPU.

ACKNOWLEDGEMENTS

The CloudDBAppliance project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732051.

REFERENCES

Intel Hibench suite: https://github.com/intel-hadoop/HiBench

BullSequana S series technical specification: https://atos.net/wp-content/uploads/2017/11/FS_Bull Sequana_S200-800_specifications.pdf
CloudDBAppliance: https://clouddb.eu/
Spark: http://spark.apache.org