NUMA-aware Deployments for LeanXcale Database Appliance

Ricardo Jiménez-Peris[1], Francisco Ballesteros[2], Pavlos Kranas[1], Diego Burgos[1]
and Patricio Martínez[1]
[1]*LeanXcale, Campus de Montegancedo, Madrid, Spain*
[2]*Universidad Rey Juan Carlos, Madrid, Spain*

Keywords: Scalable Databases, NUMA Architectures, Database Appliance, Scalable Transactional Management.

Abstract: In this paper we discuss NUMA awareness for the LeanXcale database appliance being developed in cooperation with Bull-Atos in the Bull Sequana in the context of the CloudDBAppliance European project. The Bull Sequana is a large computer than in its maximum version can reach 896 cores and 140 TB of main memory. Scaling up in such a large computer with a deep NUMA hierarchy is very challenging. In this paper we discuss how LeanXcale database can be deployed in NUMA architectures such as the one of the Bull Sequana and what aspects have been taking into account to maximize efficiency and to introduce the necessary flexibility in the deployment infrastructure.

1 INTRODUCTION

In this paper we discuss NUMA awareness for the LeanXcale database appliance being developed in cooperation with Bull-Atos in the Bull Sequana in the context of the CloudDBAppliance European project (CloudDBAppliance, 2019). The Bull Sequana is a large computer than in its maximum version can reach 896 cores and 140 TB of main memory. Scaling up in such a large computer with a deep NUMA hierarchy is very challenging. In this paper we discuss how LeanXcale database can be deployed in NUMA architectures such as the one of the Bull Sequana and what aspects have been taking into account to maximize efficiency and to introduce the necessary flexibility in the deployment infrastructure.

2 NUMA ARCHITECTURES

2.1 Background

NUMA (Non-Uniform Memory Access) architectures are motivated due to the bottleneck introduced by previous UMA (Uniform Memory Access) architectures. In the past, CPUs ran slower than the main memory, so all CPUs were connected to a global pool of main memory (see Figure 1).

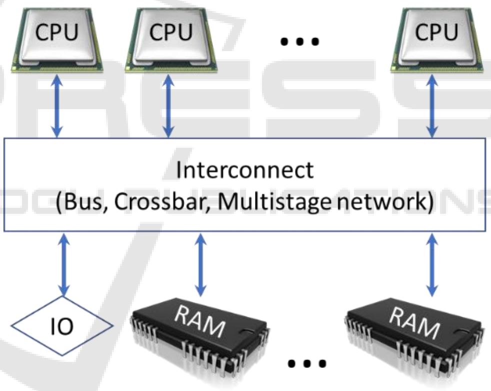


Figure 1: UMA Architecture.

However, CPUs have become over time faster than memory and at some point, they found a limitation on the speed improvements, so multi-core and multi-CPU architectures started to proliferate even in commodity servers and desktops. At some point the interconnect between CPUs and memory could not provide the necessary bandwidth. NUMA architectures then became the solution. Basically, each NUMA unit consists of a CPU and a local pool of memory. Memory is still globally accessible by all other cores at other NUMA units, however, now the access speed and bandwidth are not uniform anymore. The local memory at the NUMA unit is the one that has highest bandwidth and can be accessed faster.

Jiménez-Peris, R., Ballesteros, F., Kranas, P., Burgos, D. and Martínez, P.
NUMA-aware Deployments for LeanXcale Database Appliance.
DOI: 10.5220/0007905806660671
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 666-671
ISBN: 978-989-758-365-0

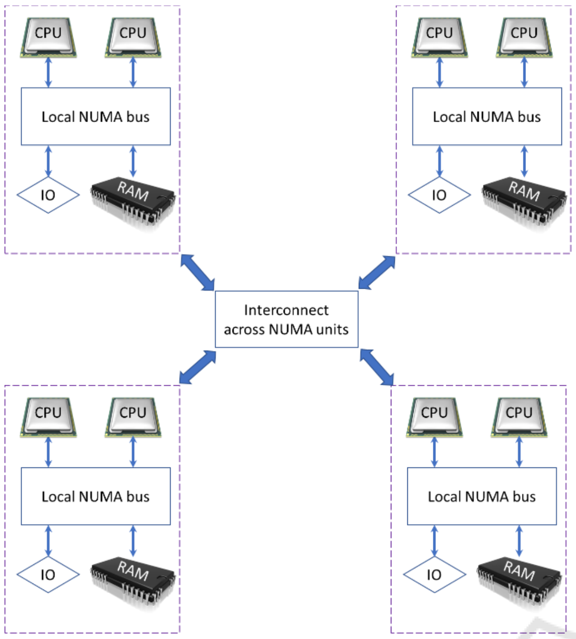
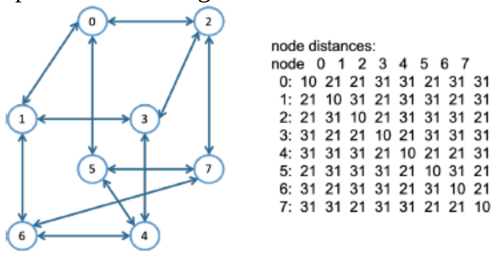


Figure 2: Two-level NUMA Architecture.

Depending on how many units this hierarchy can become deeper. Computers with 2 or 4 processors might have a 2-level hierarchy (see Figure 2), but computers like the Bull Sequana has a 3-level hierarchy (see Figure 3). Actually, with the new Cascade Lake-AP Intel processor, the hierarchy becomes 4-level since each CPU (NUMA unit) is internally split into two halves with result in a new NUMA hierarchy (called sub-NUMA clustering by INTEL). As can be seen in 1 memory latency is 3.1 times higher in the farthest node with respect local memory.

2.2 Implications of NUMA

In an UMA architecture, it does not matter on which core or CPU is running a thread. The cost to access memory will be the same. However, in a NUMA architecture this is not true anymore. Accessing a data item in the memory of the local NUMA unit is much cheaper than accessing it in a remote NUMA unit.



```
node distances:
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 3: Memory latency factors in Bull Sequana NUMA architecture.

This fact basically means that to be efficient, server software has to exercise as much memory locality as possible. However, this is hard since server software is multi-threaded and threads share memory. This results in being allocated to any core and accessing memory across all NUMA units, resulting in high inefficiency due to higher latency in memory accesses and bottlenecks in memory bandwidth to remote NUMA units.

3 LEANXCALE ARCHITECTURE

3.1 What is LeanXcale Database

LeanXcale (LeanXcale 2019) is an ultra-scalable operational Full SQL Full ACID distributed database (Ozsu and Valduriez, 2014) with analytical capabilities. The database system consists of three subsystems (see Figure 4: LeanXcale Subsystems):

1. KiVi Storage Engine.
2. Transactional Engine.
3. SQL query Engine.

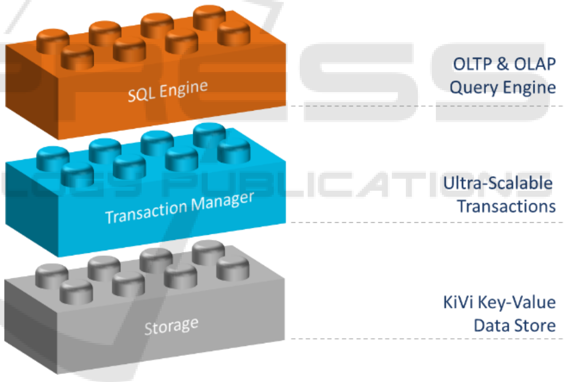


Figure 4: LeanXcale Subsystems.

3.2 LeanXcale Subsystems

The operational database is a quite complex system in terms of different kinds of components. The operational database consists of a set of subsystems namely: the Query Engine (QE), the Transactional Manager (TM), the Storage Engine (SE) and the Manager (MG). Some subsystems are homogeneous and other heterogeneous. Homogeneous subsystems have all instances of the same kind of role. Heterogenous subsystems have different roles. Each role can have a single instance or multiple instances. The transactional manager has the following roles: Commit Sequencer (CS), Snapshot Server (SS), Conflict Managers (CMs) and Loggers (LGs). The former two are mono-instance, whilst the latter two are multi-instance. The Storage Engine has two roles data server (DM) and meta-data server (MS), both multiple instances. The query engine is homogeneous and multi-instance. There is a manager (MNG) that is single instance and single-threaded. Many of these components can be replicated to provide high availability, but their nature does not change. Since replication it is an orthogonal topic, we do not mention anymore.

4 FACTORS TO BE CONSIDERED

Leveraging the full potential of multi-core and NUMA shared memory architectures implies the understanding of three key concepts namely: processor affinity, data placement and the notion and relation of physical and virtual cores.

4.1 Processor Affinity

It is the capability to map a given processing unit to the execution of a given task. Usually, the selection of a given CPU is governed by a scheduler that takes into consideration the systems state and several other policies in order to load balance tasks to the number of available processors. When only one core is available, processes or threads are instructed to start and halt their execution in order to grant permission to other threads, ensuring that the resource is shared among the interested parts. When several CPUs are available, the scheduler splits the thread’s work among the available instances and may decide to halt and reallocate task execution among processors to achieve load balancing or to comply with other policies. Under this scenario, NUMA architectures become problematic since processors and their respective memory blocks become disassociated.

Processor affinity relies in a modified scheduler to systematically associate a given task with a given processor, despite of the available resources. During the lifetime of a given process or thread, the scheduler monitors the relevant metrics to ensure that memory allocation remains local to the process or thread that is using a given processor. This technique by itself may significantly harm performance by preventing the task scheduler to spread load among the available instances. This implies that this technique should be accompanied by a smart observing and allocation scheduler in order to exploit use case specificities that would leverage this technique. On its own, this technique may be negative for most use cases as the scheduler implicitly may provide hints about affinity without preventing a given processing core to execute jobs outside of its strict affinity allowance.

4.2 Data Placement

Data placement is the capability to place data close to the processing cores that are responsible to execute a given task. Regardless if data placement is achieved implicitly, where memory blocks are assigned to specific processing units or, explicitly, where the application is hardware aware and requests ranges of memory to be handled by specific processing units; the more often data can be placed close to the processing unit that is preforming the computation, more performance is expected through the overall reduction in access time. Data placement is thus keen to translate traditional scheduling policies in order to favour fast storage mediums that present smaller access costs over remote memory accesses.

4.3 Virtual Cores/HW Threads Vs Physical Cores Vs Sockets

A computer might have one or more sockets also known as NUMA units. Each socket is basically a CPU that can have one or more cores and has allocated a memory module. Sockets are structured in a NUMA hierarchy that can be from 2 levels to an arbitrary number of levels. In the new Bullion the NUMA hierarchy is pretty deep. Understanding the relationship between physical cores and sockets and the NUMA distance across physical cores is crucial to minimize the cost of communication across components. Components that interact frequently will communicate more efficiently if they are running on closer cores in the NUMA hierarchy.

Many CPUs have the concept of physical and virtual cores. Many INTEL CPUs exhibit hyperthreading that lies in providing two hardware threads per physical core. They have a superscalar architecture in which a subset of the instructions can operate on separate data in parallel. Additionally, when one thread blocks the other can still run guaranteeing that the physical core is actually performing work.

The operating system actually reports the hardware threads as virtual cores. AMD virtual cores are more complete than INTEL hyperthreads since each has a full set of registers, so threads running on different cores actually do not require to save the thread registers as it happens with hyperthreading. The operating system always reports about virtual cores (hardware threads for INTEL, actual virtual cores for AMD). When setting processor affinity, it is the id of the virtual cores what is actually provided.

It is very important to understand the relationship between virtual cores and physical cores, since some servers should run a single instance per physical core, such as the storage engine. Running any other server on a virtual core over the same physical core would result in serious performance drop. However, there is no straightforward way of getting this association. The way we have found to extract this information is through the thread_sibling_list in the CPU topology meta-information of the operating system: /sys/devices/system/cpu/cpu0/topology/thread_siblings_list.

When the first number is the same as CPU id, then the CPU is a physical core. When the CPU id appears at other position other than first, it means that it is a virtual core.

5 CHARACTERIZING SERVERS

5.1 Taxonomy Dimensions

The previously introduced approaches present alternatives to either move data close to the processing units or the opposite, where the processing is moved closer to the nodes where data is placed. In a distributed deployment, where a multitude of components coexist and interact among each other, it becomes imperative to reason about the underlying specific use case. A system model allows to bound both the expectations and requirements of each component that builds the distributed system has according to several distinct factors, namely:

a. Threading model: single threaded server vs. multi-threaded server;
b. Architecture (Single instance vs Multiple instance components);
c. Resource boundness (IO/CPU/Memory bound)

The above set of properties allows to determine important deployment features that might limit the action of the smart placement, such as the need for component co-location or the required memory thresholds.

5.2 Taxonomy of Servers for NUMA Awareness

The manager is single instance. Table 1 summarizes the main properties of each subsystem/role.

Table 1: LeanXcale Servers and their Taxonomy.

Subsystem	Role	Instances	Multi-threading	Resource Boundness
QE+LTM	QE+ LTM	Multiple	Yes	CPU Memory
TM	CS	Single	No	CPU
TM	SS	Single	No	CPU
TM	CM	Multiple	No	CPU
TM	LG	Multiple	No	IO
DS	MS	Single	No	CPU
DS	DM	Multiple	No	IO/CPU/ Memory
MNG		Single	No	CPU

6 DEPLOYMENT MODELS

6.1 Collocation

The first question is whether is good to collocate servers and if yes, which ones and how. Collocating servers can be good if they communicate a lot, since the communication can be done via local memory. Another reason why it can be good to collocate is when servers are complementary in terms of resource usage. For instance, there are servers that are CPU bound while others are IO bound. In this case, they match well due to their collocation can result in a balanced usage of resources.

Looking closely to the Bull Sequana, IO devices are distributed evenly across NUMA units. This means that it becomes important to be able to distribute the IO activity across all NUMA units.

6.2 Model of Individual Deployment

The other question is how to deploy each individual server, across how many cores or NUMA units. Let us look at the individual features of each server.

6.2.1 Query Engine+LTM

The query engine is a Java application that runs on the JVM. It is a multi-threaded server, in which each client session is served by an independent thread. This means that if a query engine instance has 100 clients (the JDBC drivers running collocated with the client application), it will have 100 threads managing each client session.

The query engine is stateless, it just keeps session state. In terms of memory, it depends on the kinds of queries it runs. Many queries are not memory intensive, just CPU intensive. Analytical queries with multi-way joins are both memory and CPU intensive.

The LTM handles the interaction with all the transaction manager components and it is used as a library by the data storage client side that is used as a library by the query engine. The LTM and client side of the storage engine are CPU bound. LTM and client side of the storage engine code runs as part of invoking query engine Java threads. The client side of the storage engine has also its own threading to deal with the communication with the storage engine servers.

The JVM is known to not scale up well in terms of memory size due to the garbage collection becomes more intrusive when the number of objects grows what happens when the memory used is larger. Running the JVM across NUMA units results in remote NUMA memory accesses and since it does not scale up also results in being less efficient.

For all aforementioned reasons, a given query engine instance is deployed within one NUMA unit. When more instances are needed, they are run on different NUMA units. A query engine since it is multi-threaded it can be deployed across multiple cores.

6.2.2 Commit Sequencer, Snapshot Server and Manager

All these servers are single threaded. They are CPU bound. Although in large deployments it is worth to run them separately, in small and medium deployments they are run as a single entity with three different threads.

6.2.3 Logger

Loggers are IO intensive and single threaded. Their IO activity is based on forced writes. Thus, they have to run on devices without any other IO since otherwise their performance is heavily affected.

Therefore, loggers should run on individual cores spread across different NUMA units and always on devices that they use in isolation, with no other IO activity.

6.2.4 Data Storage

The data storage has two kinds of servers the meta-data server and the data server. The former is mostly CPU bound. The latter can be CPU or IO-bound. It is always memory bound.

The data server is single-threaded and CPU and IO intensive. Thus, it should be deployed on individual cores and without sharing the IO device with other components.

6.3 Collocation Considerations

There are two main considerations to be taken into account that affect collocation. Servers that interact a lot between them is good they are on the same NUMA unit so they can communicate through local memory. Servers that consume different kind of resources also benefit from collocation since the collocation results in a more evenly balanced resource consumption.

What servers interact with a lot of data? The heaviest interaction happens between query engines and data storage servers. Especially with analytical queries. With analytical queries the query engine runs in parallel mode what means that all query engines process a fraction of the query. In analytical queries a lot of information can flow from the data storage engine into the query engine. Since query plans are projected so the first stages are purely local between data storage engine and query engine, it is very beneficial to collocate query engine with data storage servers.

If there are enough IO devices on each NUMA unit for data storage servers and a logger, having a collocated logger with the LTM (that is a library of the query engine) is also very beneficial since all logging activity becomes purely local.

Conflict managers are CPU-bound. Each instance is global so there is no special benefit on being collocated with any particular kind of server. Since they are CPU-bound they can benefit from a collocation with IO bound servers such as loggers or storage engine servers. The number of conflict managers typically required is much smaller than query engines, data storage servers, and loggers. So, it is fine to run them in an arbitrary NUMA unit with spare CPU capacity.

6.4 Deployment Strategy

For the Bull Sequana, we have adopted a uniform deployment strategy for the multi-instance servers. Basically, we deploy for each NUMA unit:

- A query engine instance running on several cores.
- Several storage server instances, each running on a separate core and each of them with an IO device used in isolation.
- A logger running on a separate core and with an exclusive IO device.
- A conflict manager that runs on the JVM of the query engine and share cores with it.

The benefits of this strategy are:

- The intensive communication between data storage instances and query engine instances is purely local in the first stages of the query plan.
- The intensive communication between LTM and logger is purely local.
- By allocating different set of cores with processor affinity the different services attain high caching efficiency.
- By running on a single NUMA unit all memory accesses are NUMA local.
- By distributing the IO intensive services uniformly across all NUMA units the IO load is evenly balanced across NUMA units and devices being able to use all the available IO bandwidth.

The single-instance servers run on its own NUMA unit. In this NUMA unit it is run:

- The meta-data server of the storage engine.
- The commit sequencer.
- The snapshot server.
- The LeanXcale manager.
- The loggers used by all the above servers.

By running these meta servers on a different NUMA unit isolation from the worker servers is attained. This performance isolation is crucial since all these services if they get stalled, they stop the whole database.

7 CONCLUSIONS

Supercomputers such as the Bull Sequana are challenging because existing server software, especially databases are not designed to run efficiently on a large NUMA architecture such as the one of the Bull Sequana. LeanXcale database thanks to its modular design enables to adapt its configuration to use with optimal efficiently the Bull Sequana NUMA architecture. LeanXcale attains high locality of interactions within NUMA units, with no interactions across NUMA units except for the meta servers. This results in using the large NUMA architecture as a distributed system and maximizing the performance. Additionally, the IO usage is guaranteed to be uniform across NUMA units resulting in a very efficient usage of the super computer.

ACKNOWLEDGEMENTS

This work has been partially funded by the European Commission under the H2020 project: CloudDBAppliance – European Cloud In-Memory Database Appliance with Predictable Performance for Critical Applications. Project number: 732051.

REFERENCES

Özsu, T., P. Valduriez.
Distributed and Parallel Database Systems. Computing Handbook, 3rd ed. 2014.
LeanXcale. http://leanxcale.com. 2019
CloudDBAppliance. https://clouddb.eu/ 2019