Multi-Core Allocation Model for Database Systems

Simone Dominico
Supervised by Eduardo Cunha de Almeida
Federal University of Paraná, Brazil
sdominico@inf.ufpr.br

ABSTRACT

Non-Uniform Memory Access (NUMA) architecture provides a multi-task run in parallel with different access latency between the nodes. The emergence of multi-core hardware offers high processing power for multi-threaded Database Management Systems (DBMS). However, database threads run across multiple NUMA cores without exploring the hardware to its full potential. The impact of data movement between NUMA nodes is a major challenge for multi-threaded DBMS. The goal of our thesis is to find out the efficient distribution of CPU-cores among the NUMA nodes in order to mitigate the data movement. We propose an abstract core allocation mechanism for query processing based on performance metrics. In our preliminary results, our mechanism is able to improve data traffic ratio between nodes in up to 3.87x with increased memory throughput in up to 27%.

1. INTRODUCTION

The burgeoning ingestion of data requires new hardware to allow real-time data analysis. In our thesis we focus on Non-Uniform Memory Access (NUMA) hardware to boost throughput of multi-threaded Database Management Systems (DBMS) in Online Analytical Processing (OLAP). The NUMA architecture is formed by multi-core nodes with processors (or CPUs) attached to a memory bank. The memory access latency varies according to the distance between the node and the memory being accessed.

The varying memory access latency of NUMA impacts the performance of multi-threaded query processing. We observed the impact of NUMA in multi-threaded DBMS executing the Volcano query parallelism model [4], like Microsoft SQLServer, and the materialization model, like MonetDB. For instance, in the Volcano model, the parallelism is encapsulated in "exchange" operators in the query execution: multiples threads execute the query plan and the Operating System (OS) scheduler is in charge of managing data and thread locality. In both models, however, the conventional approach is letting the OS do the mapping of

Proceedings of the VLDB 2018 Ph.D. Workshop, August 27, 2018. Rio de Janeiro, Brazil.

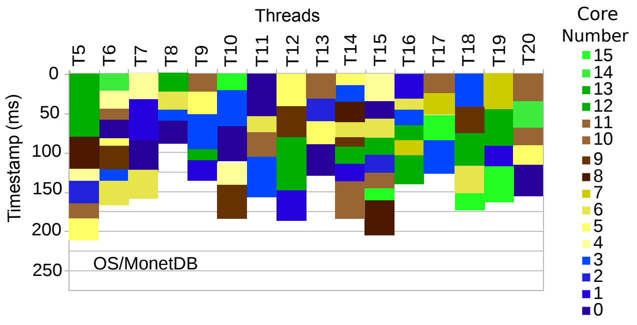

Figure 1: Evaluating the migration between cores of the threads generated by the TPC-H Q6 in a single-client execution in a 4x Quad-Core NUMA machine.

threads to as many nodes and cores as possible to keep load balance, not considering the different access latencies in the NUMA architecture. The result is the constant migration of threads all over the nodes. Figure 1 shows the execution of the TPC-H query 6 (Q6) in MonetDB: a multi-threaded DBMS. Our NUMA architecture consists of 4-nodes with a Quad-Core AMD Opteron 8387 each. The OS tries to keep the load balancing between the cores causing many migration of threads with more data movement between the NUMA nodes. When running OLAP workloads, the current load balancing approach can cause many problems: resource contention, cache conflicts, cache invalidation and inefficient data movement.

Many authors have investigated these problems by trying to control the scheduling of threads and data movement in NUMA nodes [8, 3, 11, 5, 12, 7, 13] from within the DBMS kernel. Our thesis follows a different direction where we manage the allocation of NUMA cores from an abstract model-based mechanism, aiming to accommodate the current workload and mitigate the data movement. Abstract models allow supporting several underlying systems without modifications in the source-code. Besides, an abstract model allows different resources to be monitored, for example: memory throughput and CPU load. Our mechanism in a nutshell: a dynamic mechanism computes the local optimum number of cores with a rule-condition-action pipeline integrated with performance monitoring. This pipeline defines the performance-invariant through performance thresholds. If this performance-invariant holds, the local optimum number of cores is set. The main challenges to present our mechanism are, as follows:

1. Provide an elastic multi-core allocation mechanism for the NUMA architecture.

2. Define an optimum number of cores based on the use of computational resources to meet a workload.

3. Manage the allocation of cores for the execution of mixed workloads.

In this paper, we discuss our approach to tackle these three challenges, the contribution already published and current work in progress.

2. RELATED WORK

Over the last years, many different approaches were presented to improve the performance of DBMS multi-core parallelism in NUMA architecture. [10] presents a study of the NUMA effect when assigning threads to NUMA nodes in OLTP. The authors present an approach of "hardware islands" to execute different OLTP deployments. They implement a prototype within the Shore-MT storage manager that achieved robust OLTP throughput. However, the "hardware islands" do not scale-out and the optimum size of the island is yet undetermined.

[8] presents a NUMA-aware ring-based algorithm to coordinate the movement of threads to catch up with data. The algorithm uses two rings: an inner ring to represent the data partitions and an outer ring to represent the threads. The outer ring rotates clockwise for all threads to access specific data partitions. The goal is to improve data placement and thread placement. In [3], NUMA cores are allocated one by one to mitigate access to remote memory when the OS tries to keep data locality of MonetDB. Other storage and worker placement approaches are presented to mitigate the NUMA effect in SAP Hana [11] or to build the ERIS storage engine from scratch [5].

All of these contributions show important efforts into making the DBMSs explore multi-core CPUs to their full potential, but they differ from our goal as they allocate NUMA cores statically to perform data and worker placement strategies. Recently, [12] presented an adaptive NUMA-aware data placement mechanism. The mechanism decides between data placement and thread stealing when load imbalance is detected. The number of working threads changes based on the core utilization, which differs from MonetDB [3] and SQLServer [6] that statically define the number of threads based on the number of available cores.

In [7] is presented a database scheduler to control the dispatching of query fragments, called "morsels". The "morsels" are statically pinned in specific cores to take advantage of the data location and avoid data movement between the nodes. However, this scheduler does not take into account the optimum number of cores to tackle the current workload.

In contrast to the related work, we propose a dynamic multi-core allocation mechanism to mitigate the data movement providing to the OS the efficient sub-set of NUMA cores to perform database thread mapping.

3. PROPOSED APPROACH

Our proposal involves an elastic multi-core allocation mechanism to define the optimum number of cores to treat the current workload. We define that the allocation of CPU-cores should be performed dynamically to satisfy the demands of the workload and facilitate the OS thread scheduling with the least possible negative impact on performance. Our key idea is to consider data placement statistics from the OS side to define on which node the cores will be allocated.

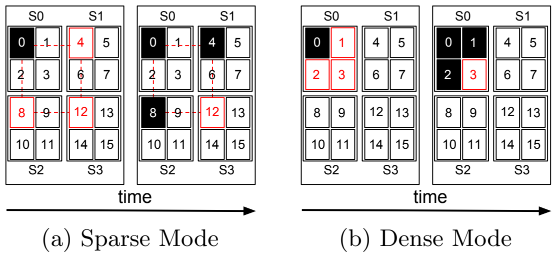

Figure 2: The Sparse and Dense modes over time. Only the black boxes (i.e., cores) can be accessed by the OS. The red boxes are the next cores to allocate.

3.1 Elastic Multi-core Allocation Mechanism

The full description of our multi-core allocation mechanism is presented in [2]. Our mechanism is based on Predicate/Transition PetriNets (PrT). Petri Nets are powerful abstract models to design concurrent and distributed systems. Our mechanism leverages Petri Nets to model general properties of DBMSs up against concurrent OLAP workloads. Using the abstract model we have the flexibility to check any resource usage by the database threads. The mechanism monitors the resource usage of the worker threads on top of OS kernel facilities to decide for the allocation of CPU cores (e.g., cgroups, mpstat, numactl, likwid).

3.2 The Allocation of Cores

The challenge of the mechanism is to prevent both underutilization or overutilization of the system by finding out the local optimum number of cores (LONC) to accommodate a given workload. In this section, we define the LONC and also the allocation modes explored by the mechanism.

3.2.1 The multi-core allocation modes

In the mechanism, we define resource usage thresholds (e.g., CPU or data traffic in the memory controller) that are used to decide when it is necessary to allocate or release cores. An adaptive algorithm decides on which node to allocate/release cores taking into account the accessed memory addresses kept in a priority queue data structure.

Figure 2 shows the allocation of closed (Dense) and far apart cores (Sparse). The adaptive allocation is a combination of both w.r.t. the efficient subset of cores. To find the affinity between threads and data, our approach checks misses in the Translation Lookaside Buffer (TLB). When a TLB miss occurs, the OS maps the thread accessing the missed address to a node keeping this information in a data structure. Over time, new threads requesting the same address range are mapped to the same node where data is allocated. Therefore, we refer to as "the efficient subset of cores" the processing cores in the nodes with the requested address range.

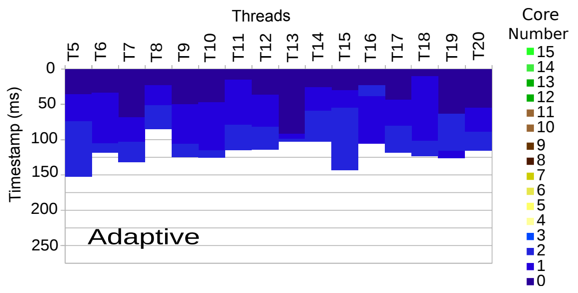

Figure 3: The migration of the threads spawn by a single-client submitting the TPC-H Q6 supported by the adaptive mode.

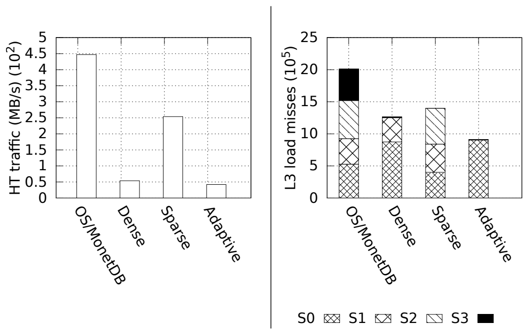

Figure 4: Performance metrics of Q6 with a single client in 1 GB database in MonetDB.

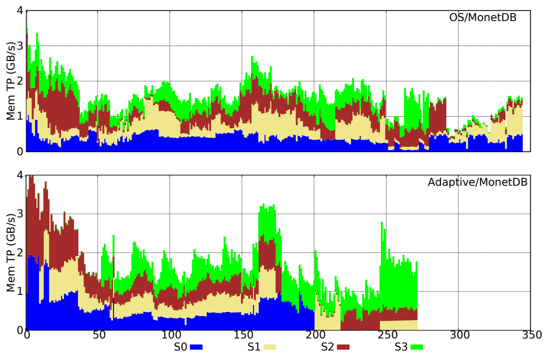

Figure 5: Execution time (secs) and Memory Throughput (GB/s) of the TPC-H execution with 256 concurrent clients in 1 GB database.

In our adaptive mode, each entry of the priority queue keeps the PIDs of the active threads with their address spaces and the number of pages per NUMA node. The cores are allocated on nodes with more accessed pages and cores with the least number of accessed pages are released. In the implementation of the affinity between threads and data, the data structure stores the node IDs used to pin threads and allocate their address space.

3.2.2 The local optimum number of cores

The number of allocated cores takes into account the arithmetic CPU-load average of the active database threads. Formally, we define how to compute the number of cores, as follows:

$$\forall\, w\ \exists\, n_{alloc} | (th_{min} < u < th_{max}) \wedge p(n_{alloc}) \geq p(n_{total}) \quad (1)$$

To any OLAP workload w, there is a certain number of CPU cores n_{alloc} such that the load of each core are between the minimum and maximum *thresholds*, in which the database performance $p(n_{alloc})$ is equal or better than the performance $p(n_{total})$ with all the CPU cores available in the hardware. The performance function $p(x)$ relies on system counters provided by the OS and the database.

3.3 Mixed Workload

In our initial efforts in this thesis, we focused on OLAP due to the pressure to transfer and compute large volumes of data scattered across multiple NUMA nodes [1, 2]. However, the elastic multi-core mechanism can designate underused cores or NUMA nodes to serve different types of workloads. Our next direction is to investigate how to designate underused cores to perform different workloads on different NUMA nodes. We expect the abstract nature of our mechanism and the information of memory usage to facilitate the choice of the optimal node to a different workload.

4. EXPERIMENTS

In a preliminary study, we ran the experiments with OLAP workload on a NUMA machine formed by 4-node with a QuadCore AMD Opteron 8387 each. Nodes are interconnected by Hyper-Transport (HT) link 3.x achieving 41.6 GB/s maximum aggregate bandwidth. We implemented our prototype in C language and we compare our mechanism to the Linux Debian 8 “Jessie” OS scheduling of threads spawn by the MonetDB (v11.25.5) DBMS. We let all the 16 cores available to the DBMS when running without the support of our mechanism. Inside the mechanism, we coded the thresholds to $th_{min} = 10$ and $th_{max} = 70$ following the rules of thumb in the literature [9] and they are kept in all the experiments. We experimented different thresholds, but decreasing th_{min} lets too many cores in idle state, while increasing th_{max} leads to contention with too many busy cores.

4.1 Preliminary Results

Figure 3 shows the lifespan of threads of the TPC-H query Q6 with a single client execution in a 1GB database. As expected, our mechanism limited a subset of cores required to execute the query. The threads were executed in a single NUMA node, while the OS expanded the MonetDB threads on all nodes.

Figure 4 shows performance metrics to understand the impact of our mechanism in the migration of threads. The result shows 2× more L3 cache misses and 9× more HT traffic in the OS scheduling than our adaptive mode. With less cores available for thread scheduling, the OS made good scheduling choices resulting in less remote access, less interconnection traffic and improved memory throughput.

Figure 5 shows the result of the 22 queries of the TPC-H with 256 concurrent clients executing the queries in the sequence of the TPC-H in 1GB database. We present the impact of our mechanism in the core allocation when the data access pattern changes. In the memory throughput results, our adaptive mechanism was 41% faster than the OS/MonetDB. We observe that the system does not use all the nodes all the time. For instance, initially the system

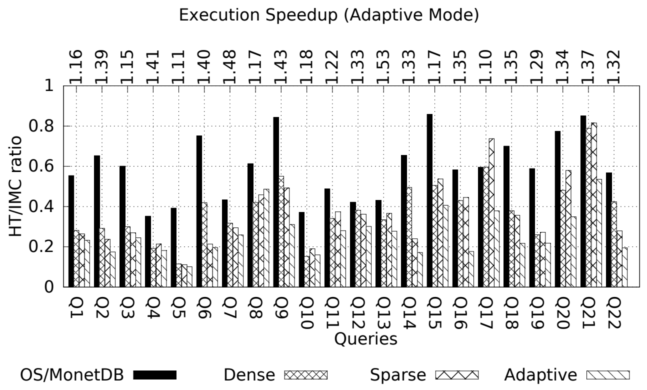

Figure 6: Performance results for 1 GB TPC-H queries with 256 concurrent clients. On the left, the ratio HT/IMC shows how NUMA-friendly is the system (the smaller, the better). On the top is the performance speedup for adaptive mode.

used only the nodes $S0$ and $S2$.

Figure 6 shows the results of interconnection traffic in relation with traffic HT/IMC ratio. In this experiment the 256 concurrent clients execute the 22 queries in random order. The results show the reduction in the local/remote per-query data traffic ratio of up $3.87x$ ($2.47x$ on average). Overall, the adaptive mode presented the best results. For instance, we observed 2x smaller HT/IMC ratio than the OS/MonetDB for queries such as Q9 that have the largest number of joins operations. With less cores available, threads locate data in the local node more often than the current approach of letting all the cores available to the OS scheduler (i.e., OS/MonetDB).

5. CURRENT STATUS & FUTURE WORK

This paper describes the motivation and challenges to our approach in the allocation of NUMA CPU-cores for parallel execution of OLAP. In our first evaluations, we discuss the impact of data movement between NUMA nodes for multi-threaded DBMS that hand over the mapping of query threads to the OS. We show the difficulty faced by the OS scheduler to keep load balance, generating a vast amount of data movement, interconnection traffic and cache invalidation. In the initial contribution of this thesis [2], we present an abstract model-based mechanism to support the thread scheduling and data allocation across NUMA sockets. The mechanism is the first part of our approach to mitigate the data movement in NUMA nodes. The preliminary results showed performance improvements when our mechanism offered to the OS only the local optimum CPU-cores, instead of the traditional approach of making all the cores visible to the OS all the time, like in current multi-threaded DBMSs: MonetDB, SQL Server, SAP Hana and Hyper.

As future directions, we plan to explore mixed workloads as we observed that OLAP not always need the entire setup of CPU-cores. Therefore, our agenda includes redesigning our abstract model to accommodate concurrent OLTP and OLAP. In particular, we plan to study the multi-core allocation in cloud computing environments that can particularly benefit from our model.

6. ACKNOWLEDGMENTS

This work was partly funded by CAPES and CNPQ.

7. REFERENCES

[1] S. Dominico, E. C. de Almeida, and J. A. Meira. A petrinet mechanism for OLAP in NUMA. In *DaMoN*, 2017.

[2] S. Dominico, E. C. de Almeida, J. A. Meira, and M. A. Z. Alves. An elastic multi-core allocation mechanism for database systems. In *ICDE*, 2018.

[3] M. Gawade and M. L. Kersten. NUMA obliviousness through memory mapping. In *DaMoN*, 2015.

[4] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD.*, pages 102–111, 1990.

[5] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A NUMA-aware in-memory storage engine for analytical workload. In *ADMS*, 2014.

[6] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[7] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[8] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[9] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic scale-out for partition-based database systems. ICDEW12.

[10] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *PVLDB*, (11), 2012.

[11] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive NUMA-aware data and task placement. *PVLDB*, (12), 2015.

[12] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, (2), 2016.

[13] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.