A PetriNet Mechanism for OLAP in NUMA

Simone Dominico
UFPR, Brazil
sdominico@inf.ufpr.br

Eduardo Cunha de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

Jorge Augusto Meira
SnT, University of Luxembourg
jorge.meira@uni.lu

ABSTRACT

In the parallel execution of queries in Non-Uniform Memory Access (NUMA), the operating system maps database processes/threads (i.e., workers) to the available cores across the NUMA nodes. However, this mapping results in poor cache activity with many minor page faults and slower query response time when workers and data are allocated in different NUMA nodes. The system needs to move large volumes of data around the NUMA nodes to catch up with the running workers. Our hypothesis is that we mitigate the data movement to boost cache hits and response time if we only hand out to the system the local optimum number of cores instead of all the available ones. In this paper we present a PetriNet mechanism that represents the load of the database workers for dynamically computing and allocating the local optimum number of CPU cores to tackle such load. Preliminary results show that data movement diminishes with the local optimum number of CPU cores.

CCS CONCEPTS

•Computer systems organization →Multicore architectures;
•Information systems →Data management systems;

KEYWORDS

Multi-core CPUs; OLAP; Abstract Model; NUMA

1 INTRODUCTION

The emergence of multi-core hardware combined with the burgeoning ingestion of data sparked the implementation of many new Database Management Systems (DBMS). But, there are still many DBMSs designed for symmetric multiprocessing (SMP) without multi-core requirements in mind [2]. In these DBMSs concurrent workers (threads or processes) can run across multiple cores without exploring the hardware to its full potential. The Operating System (OS) maps the workers to as many cores as possible considering the hardware almost as "SMP on a chip". However, this mapping can be treacherous and eventually muck up performance due to many reasons. Performance degrades for DBMSs executing the *thread per DBMS worker model* [8], like MonetDB and MySQL, where DBMS workers are mapped onto OS threads, because the migration of a thread between chips requires flushing cached data leading to the loss of cache state [4]. Memory access to move data around becomes the bottleneck. In 2009, the cost to hit L3 cache between cores in different Non-Uniform Memory Access (NUMA) nodes went up to 300 CPU cycles in the Intel Xeon 5500 [9] with direct impact on query processing.

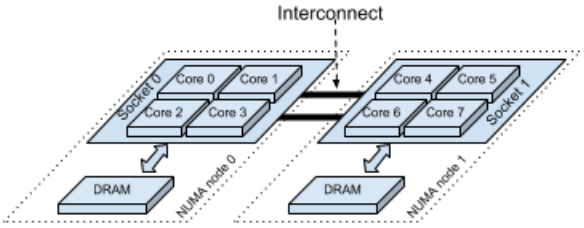


Figure 1: Schematic Diagram of the NUMA Architecture.

In modern DBMS, the workers of complex queries can run in parallel in different CPU-cores across the NUMA nodes. For instance in the NUMA machine depicted by Figure 1, the workers of query Q_i may run in both nodes 0 and 1. However, if data is allocated in a different NUMA node from the running workers, it must be moved around increasing interconnection traffic. Indeed, there is an intrinsic urgency to take advantage of NUMA hardware, and many algorithms for query processing [6, 14] are written from ground up for the new generation of DBMS. In the other hand, the legacy DBMSs are still in production in many systems and revisiting their code is costly and takes time [16].

In this paper, our contribution is a mechanism to allocate CPU cores across NUMA nodes for alleviating interconnection traffic in OLAP. This mechanism is based on PetriNets with the benefit of being lightweight (i.e., implemented as an adjacent matrix) and non-intrusive to the DBMS and the OS. The mechanism represents the load of the database workers for dynamically computing the local optimum number of CPU cores to tackle such load. Preliminary results show less interconnection usage among NUMA nodes when running a microbenchmark of the TPC-H with the local optimum number of CPU cores allocated by our mechanism.

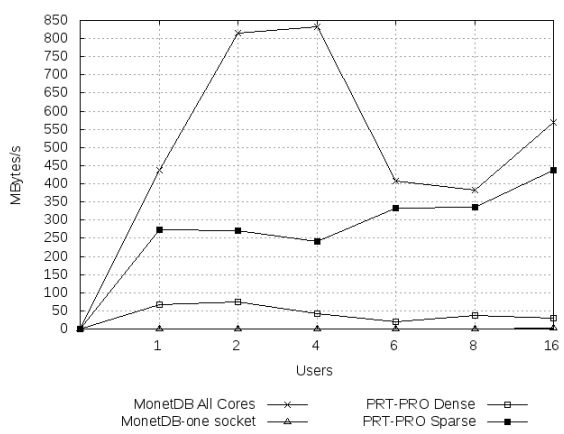
This paper is organized as follows: Section 2 presents the problem of data movement in NUMA. Section 3 presents our PertiNet mechanism to allocate/release cores. Section 4 presents empirical results and Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 PROBLEM: DATA MOVEMENT IN NUMA

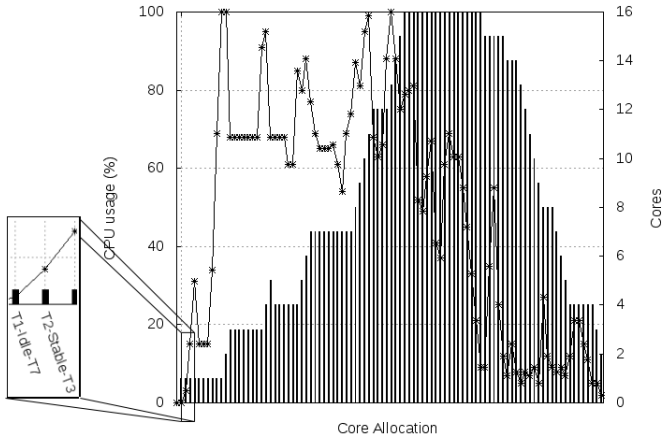
We ran a microbenchmark to understand the problem of moving data around NUMA nodes to catch up with the running workers. We evaluated the usage of interconnection bandwidth and the CPU load by running the TPC-H query Q6 on the MonetDB DBMS in 1 GB scale factor upon different numbers of concurrent clients. Query Q6 has important data access locality pattern because the data generator keeps the same selection propagation on date type attributes across the database. The restrictions on the o_orderdate

DaMoN'17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3076113.3076121



(a) Average Bandwidth Usage of Interconnection with data in-memory for TPC-H Query 6 with Scale Factor 1 GB.



(b) *Dense* mode in the TPC-H Query-mix [14] with 5 concurrent clients in Scale Factor of 100 GB.

Figure 2: Understanding the Interconnection Problem and Self-Adjusting the Number of Cores to a Given Workload.

attribute propagate the selection to the largest table LINEITEM up to 1 to 36 selection fraction [3].

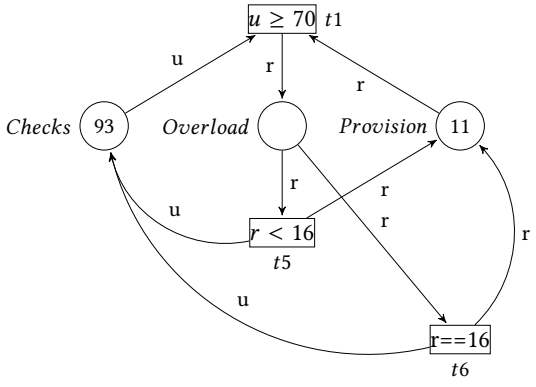
We ran the experiments on a 4-Node Quad-Core AMD Opteron 8387 2.8 GHz (4∗64KB L1, 4∗512KB L2, 6MB L3) with the nodes interconnected by HyperTransport link (HT) of 14.4GB/s. This machine includes 256GB RAM and 1.8 TB of Disk running Debian 8. We let all the 16 cores available to MonetDB to measure what happens in the current DBMS/OS setup. We also let the system with a single core to establish the baseline metrics (e.g., response time, cache faults). All the figures present an average of 10 runs.

MonetDB does a good job exploiting data correlations from selections and join instructions. It keeps the interesting tuples from these operations in a mapping to reduce the volume of intermediary results. However, the OS still allocates the MonetDB workers across the nodes with direct impact on interconnection usage.

We can see this NUMA effect even with only one client generating almost 450MB/s of interconnection traffic (see Figure 2(a)). When increasing the number of concurrent clients, the traffic also increases up to a point when the workers started to share the same memory bank with some decrease in the interconnection traffic.

3 THE PRT-PRO IN A NUTSHELL

Our mechanism, called the PetriNet for Core Provisioning (PrT-PRO), implements a PetriNet to abstract the load behaviour of the DBMS up against parallel execution of queries. With this abstraction, the PrT-PRO does not require any modifications on the DBMS or the OS. It monitors the resource consumption of the DBMS workers (e.g., CPU load) and interacts with the OS to allocate or release the CPU cores (e.g., via the *cgroups* Linux facility). The PrT-PRO is a graph implemented as an adjacent matrix with its edges representing the current load of the workers. The nodes represent the DBMS performance states and also the state transitions to allocate/release the CPU cores. Our aim is to dynamically adapt the number of cores to the given workload without letting all the cores available to the OS, as depicted by Figure 2(b). For instance, when



$$\begin{array}{l} Post \\ Checks \\ Over \\ Provision \end{array} \begin{array}{c} \begin{array}{ccc} t_1 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 0 & 93 & 93 \\ 11 & 0 & 0 \\ 0 & 11 & 11 \end{pmatrix} \end{array}$$

Figure 3: Tokensthrough transition in the Overload sub-net to allocate one core with $u = 93\%$, $r = 11$ out of $rd = 16$ cores and $th_{max} = 70\%$.

the load of the workers goes up (represented by lines), the PetriNet reacts to the load and allocates cores for the system (represented by bars). Otherwise, if the load of the workers goes down, the PetriNet releases idle cores from the system.

3.1 The Local Optimum Number of Cores

Formally, the PrT-PRO is a bipartite graph representing the flow of tokens from one place downstream to another place around the net topology. A token represents the number of CPU cores or their load. For instance, Figure 3 depicts a subset of the PrT-PRO

representing the behavior of an overloaded DBMS. In this particular state, the system runs with CPU load of $u = 93\%$ with $r == 11$ cores allocated out of $rd == 16$ in the NUMA nodes. According to maximum th_{max} and minimum th_{min} load constraints defined in the PrT-PRO (i.e., average of the running cores), if the system is considered overloaded, then the PrT-PRO allocates more cores to stimulate the system returning to a stable state. The PrT-PRO defines the different performance states of the DBMS (e.g., *Stable*, *Idle*, *Overload*) and two additional states (*Checks* and *Provision*) to validate the current load of the workers and take action to allocate cores.

Our goal is to keep the system with the local optimum number of cores, because it leads the DBMS to the *Stable* state and avoids letting the OS freely allocating cores all over the NUMA nodes. We define the local optimum number of cores, as follows:

$$\forall\ w\ \exists\ r | (th_{min} < u < th_{max}) \wedge p(r) \geq p(rd) \quad (1)$$

In our definition w is the workload, $p(x)$ is the performance function and x assumes the number of running r or available cores rd. In the implementation level, the tokens through and state transitions update the PrT-PRO matrix. Performance overhead to update this matrix is negligible.

3.2 The Core Allocation Modes

In this section, we present two different allocation modes: sparse and dense for allocating cores in different or in the same NUMA node respectively. The definition of the allocation mode function is straightforward. Our function maps a NUMA node with index i to its j^{th} core and is defined by:

$$core(i, j) = d.i + j, where 1 \leq j \leq d. \quad (2)$$

In our function, d is a constant to represent a d-ary node machine (e.g., $d = 4$ to represent our 4-Node Quad-Core AMD Opteron machine). The *Sparse* mode iterates over i, j to allocate one core at a time in a different NUMA node. For instance in our AMD Opteron machine, we allocate cores sparsely, as follows: $\{0, 4, 8, 12\}$. The *Dense* mode iterates over j, i to allocate one core at time in the same NUMA node. In our example, we allocate cores densely, as follows $\{0, 1, 2, 3\}$.

4 RESULTS

We show preliminary results of our mechanism handing out cores for the incoming workload to analyze whether the interconnection bandwidth diminishes and the CPU load goes back to a stable state.

The PrT-PRO ran with the CPU thresholds set to $th_{min} = 10$ and $th_{min} = 70$ following the rules of thumb [11]. These thresholds showed to be the most effective ones, once decreasing th_{min} lets too many idle cores, while increasing th_{max} leads to contention with too many busy cores.

We focus our analysis on a query-mix of simple and complex operations with different degree of parallelism (DOP) [6]. In particular, we analyze the interconnection bandwidth following the access patterns of the query mix [3]. In the dense mode, Q6 executed entirely in the same NUMA node drawing the lowest interconnection usage. With 5 clients, we observed 92% less minor page faults and 20.91 IPC compared to 8.68 IPC of MonetDB vanilla (Figure2(a)). With smaller data movement, the response time speedups of Q6 improved from 1.1x (sparse) to 1.39x (dense).

We also ran a TPC-H query-mix [14] with 100GB database. In this scenario, MonetDB with the PrT-PRO support required all the 16 cores only for a short time in the test machine and achieved speedups from 1.06x (sparse) to 1.6x (dense) (Figure2(b)). This shows an opportunity for designing DBMS schedulers for NUMA.

5 RELATED WORK

There are considerable research on multi-core hardware for query processing in the literature. In [1], the authors present a thorough investigation to understand the CPU consumption of DBMS workers from a black-box perspective in closed-source DBMS. To cover the lack of access to the source code, the authors investigate cache data misses, cache instruction misses and CPU stalls. In our work, we share the same black-box perspective, but for a different reason. We understand the current efforts to build new DBMS for multi-core (e.g., MonetDB[1] and Peloton[2]), but the legacy ones are still in production for a lot of systems and revisiting their code is costly and takes time [7, 16]. Therefore, a non-intrusive approach of an abstract model can be of great interest to database architects. A non-intrusive approach is used in a different context by the Greenplum Database optimizer to interact to external DBMS and process queries over shared nothing machines [13].

A technique to map incoming queries to specific cores is presented by the CARIC-DA tool [15]. This mapping happens when a query has been already processed to leverage cached data. The difference from our mechanism is that CARIC-DA still uses all the cores to dispatch queries, which leads to large data movement.

Also targeting cache optimization, but more specifically the last level cache (LLC), [10] presents a method called MCC-DB (*Minimizing Cache Conflicts in Multi-core Processors for Databases*) with performance improvements of 33% when executing OLAP in PostgreSQL. MCC-DB classifies queries in cache-sensitive and cache-insensitive to feed the query execution scheduler. Similar to our mechanism, in [5] NUMA cores are allocated one by one to mitigate access to memory banks in distant nodes when the OS tries to keep data locality. However, these approaches are intrusive requiring modifications in the source-code of the DBMS (e.g., MonetDB and PostgreSQL).

Recently, [12] presented an adaptive NUMA-aware data placement scheduling mechanism taking into account database table information. Although they describe their scheduling mechanism is black-box, they still require information about the underlying database to allocate the requested tables in specific NUMA nodes. Another difference from our mechanism, they do not attempt to figure out the number of cores to tackle a given workload, instead they use the complete NUMA setup with direct impact on the interconnection traffic.

6 CONCLUSION

In this paper, we presented a non-intrusive multi-core allocation mechanism called PrT-PRO to diminish interconnection usage

[1] https://www.monetdb.org/Documentation
[2] http://pelotondb.io/

among NUMA nodes in OLAP. Preliminary results showed performance improvements when the PrT-PRO hands out to the OS the local optimum number of cores, instead of the current approach of handing out all the CPU-cores of the hardware all the time. To hand out the local optimum number of cores, our mechanism keeps track of the performance states of the DBMS on top of monitoring facilities.

The OS improved data locality when the PrT-PRO gradually indicates the available cores, but the different data access patterns required different allocation modes. In particular, the *Dense mode* presented the best core allocation mode for MonetDB due to its worker model. When multiple-workers of a query (i.e., threads) are closed allocated, they share data within the same node and required less data moved across multiple nodes.

Future work includes assessing whether the PrT-PRO support is up to the task of indicating to the OS the fit core allocation for different DBMS architectures: thread per worker (e.g., MonetDB) or process per worker (e.g., PostgreSQL). Moreover, we plan to implement an adaptive version of our approach that combines the dense and sparse modes with different heuristics.

ACKNOWLEDGMENTS

This work was partly funded by the National Counsel of Technological and Scientific Development (CNPq), grant 441944/2014-0.

REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*. 266–277.

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. 37–48.

[3] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*. 61–76.

[4] Jeffery A. Brown, Leo Porter, and Dean M. Tullsen. 2011. Fast thread migration via cache working set prediction. In *HPCA-17*. 193–204.

[5] Mrunal Gawade and Martin L. Kersten. 2015. NUMA obliviousness through memory mapping. In *DaMoN*. 4:1–4:7.

[6] Mrunal Gawade and Martin L. Kersten. 2016. Adaptive query parallelization in multi-core column stores. In *EDBT*. 353–364.

[7] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*. 981–992.

[8] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends databases* 1, 2 (2007), 141–259.

[9] David Levinthal. 2009. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors.* Technical Report. Intel Corporation.

[10] Raymond R.-F. Liao and Andrew T. Campbell. 2001. Dynamic Core Provisioning for Quantitative Differentiated Service. In *IWQoS '01*. 9–26.

[11] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. 2012. Elastic Scale-out for Partition-Based Database Systems. In *SMDB*.

[12] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB* 10, 2 (2016), 37–48.

[13] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD*. 337–348.

[14] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: the architecture and design of a database processing unit. In *ASPLOS*. 255–268.

[15] Fang Xi, Takeshi Mishima, editor="Bhowmick-Sourav S. Yokota, Haruo", Curtis E. Dyreson, Christian S. Jensen, Mong Li Lee, Agus Muliantara, and Bernhard Thalheim. 2014. CARIC-DA: Core Affinity with a Range Index for Cache-Conscious Data Access in a Multicore Environment. In *DASFAA*. 282–296.

[16] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.