Computing manuscript No.
(will be inserted by the editor)

On the performance limits of thread placement for Array Databases in Non-Uniform Memory Architectures

Simone Dominico · Eduardo C. de Almeida · Marco A. Zanata Alves.

the date of receipt and acceptance should be inserted later

Abstract Array Database Management Systems (Array databases) are specialized software to streamline multi-dimensional data processing. Due to the data-hungry nature of multi-dimensional data applications (e.g., images and time series), array databases must ideally provide linear speedup when using a multi-processing system. However, when dealing with Non-Uniform Memory Access (NUMA) machines, array databases may require massive data movement for processing across the NUMA nodes resulting in severe performance impact. This paper investigates the performance impact of five well-known thread pinning strategies running array filtering operations in two different NUMA architectures. To identify the maximum potential performance improvement, we perform an in-width analysis evaluating all possible thread pinning combinations. Our experiments showed execution metrics of two array databases, namely SAVIME and SciDB. We observe a maximum speedup by 2.25× in SAVIME with a reduction in remote memory access by 5×. For SciDB, we observed a speedup of up to 5.83× and a reduction on the remote memory access by 4.1×. Our main finding is that well-known static thread pinning strategies only yield 48% from the potential speedup (and 26% of the energy reduction), opening multiple opportunities for improvements.

Keywords NUMA · Array databases · Thread placement. · Query Processing

Simone Dominico
Department of Informatics, University Federal of Paraná, Brazil
ORCID: 0000-0003-3471-8670
E-mail: sdominico@inf.ufpr.br

Marco A. Zanata Alves
Department of Informatics, University Federal of Paraná, Brazil
ORCID: 0000-0003-2440-2664
E-mail: malves@inf.ufpr.br

Eduardo C. de Almeida
Department of Informatics, University Federal of Paraná, Brazil
ORCID: 0000-0002-6644-956X
E-mail: eduardo@inf.ufpr.br

1 Introduction

Many engineering and scientific applications use data arrays to efficiently store collections of elements and support multi-dimensional data analysis. An array database is specialized software for creating and maintaining data arrays. It supports multi-dimensional operations based on geometry and linear algebra operations to query data arrays (e.g., data slices, array transpose, addition, subtraction, opposite array) and requires multi-processing computing when dealing with vast amount of data.

Ideally, the performance of the array databases should increase linearly using multi-processor systems. In this paper, we study the performance of array databases running on NUMA hardware architecture.

NUMA machines provide shared memory across multi-processor nodes with varying data access latency according to the memory location: low latency whenever accessing local memory, while high latency accessing remote memory. The location of data plays an essential role in the overall performance as the execution of multi-dimensional operations may need to move large amounts of data scattered in local and remote nodes to validate query conditions. Suppose, for instance, moving multiple array cells scattered in different nodes to output a data slice operation. Multi-dimensional query processing needs efficient strategies for pinning query threads across specific processing nodes. With an efficient pinning strategy, the validation of the query conditions occurs mainly locally at each node, consequently, reducing the movement of data.

Multi-dimensional query processing models are similar to those used by Relational Database Management System (RDBMS). For example, SAVIME [23] uses the materialization query model (i.e., full materialization of intermediate array cells between multiple operations), whereas SciDB [4] uses the iterator query model (i.e., producer/consumer of one array cell at a time between multiple operations in a pipelined scheme). The parallel execution of queries is similar to many RDBMS and relies on the Operating System (OS) to map the memory and the query threads to CPU cores. The OS uses load balance strategies to spread the query threads all over the cores without considering specific characteristics of the interaction between operations running in the database and the multi-core processor architecture. Previous work on RDBMS [11] observed that the OS attempt to keep load balance between NUMA nodes generated a negative performance impact with increasing interconnection data traffic between nodes.

In this paper, we study this negative impact that may reach up to 5× more remote access if we let the OS in charge of pinning the multi-dimensional query threads. This paper is an extension of the work presented in the 29th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP'21). We extend our previous work with an in-width analysis of the *subarray* operations with all possible thread pinning combinations. We include an analysis of the execution of our thread pinning implementations in two NUMA machines for studying the influence of different NUMA architectures. **We also updated the analysis of the speedup to better**

understand the impact of the number of chunks. Now, we normalize all results with a fixed size of 100 chunks for the baseline (i.e., the execution of the OS scheduler). As far as we know, we are the first to evaluate such impact on array databases. We have chosen the SAVIME and SciDB state-of-the-art systems for the experiments. They implement two different query processing models and implement the multi-dimensional array data model from scratch (i.e., no adaptations from the relational model).

Our main contributions in this paper are: **Traditional techniques comparison:** We analyze the speedup and energy impact of five different thread pinning strategies for NUMA systems when executing SAVIME and SciDB. Implementing different strategies, we observe a maximum speedup of 2.25× (3.1× less energy) with 5× less remote memory accesses for SAVIME. For SciDB, we observed a speedup of up to 5.83× (1.17× less energy) and a reduction on the remote memory access by 4.1×.
Analysis of all thread pinning combinations: We perform an extensive evaluation of the impact of thread pinning. We show that performance can improve in 3.89× of speedup in SAVIME. Our experiments showed that not all combinations affect performance positively.
NUMA architectures comparison: Using different multiprocessing Intel platforms, we evaluate the impact of two distinct NUMA systems in the execution of our thread pinning implementations.
Maximum performance analysis: We show that traditional techniques for distributing threads across NUMA cores are still far from a perfect point of improvement. Our experiments showed that, on average, 52% performance and 74% energy improvements are still available to be collected by newer and improved techniques.

The paper is organized as follows: Section 2 discusses the n-dimensional array model; Section 3 describes our thread pinning strategies; Section 5 presents a study on the effects of the NUMA architecture on two Array databases; Section 6 discusses related work; Finally, Section 7 concludes the paper.

2 Array Database Systems

This section briefly describes the array data model and the query operator to slice multi-dimensional data that we used in our evaluations.

The array data model represents data using n named dimensions that are contiguously indexed. The multiple dimensions speed up the data access and analysis through different views. In this model, each cell belonging to an array contains attributes with the same data type. Figure 1 illustrates the array data model with three dimensions being respectively *latitude*, *longitude*, and *year*[1]. The values are accessible through a set of indexes. Experts can quickly analyze, in the example of Figure 1, the change in temperature over the years.

Along with the array data model concept, a wide range of Array database has emerged, such as RasdMan[3], ArrayStore[31], SciDB [4], SciQL[34] and

[1] `geoserver.geo-solutions.it/edu/en/multidim/netcdf/netcdf_basics.html`

SAVIME [23]. In this paper, we focus on two full-stack databases, SAVIME [23] and SciDB [4]. They implement the array model from scratch without adaptations in the relational model. In scalable multi-processing machines, this allows the distribution of data chunks on separated machine nodes. A chunk is the smallest physical representation of an array. Both database systems, SAVIME and SciDB, split the arrays into chunks according to the stored data types. The chunk size and format depend on the density of the array. For dense arrays, all the chunks will have the same size. On the other hand, when arrays are sparse, chunks may have different sizes and formats. Non-regular chunks (i.e., sparse arrays) are prone to be non-uniformly distributed, causing load unbalance that reduces system efficiency.

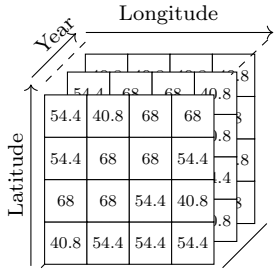

Fig. 1 Model of a multi-dimensional array: temperature on many latitudes and longitudes over the years.

Considering the query processing iterator model implemented by SciDB, chunks are extracted from the arrays using nested function calls, and a processing pipeline emits one array cell at a time through query operators. The data array structure makes faster access to a set of cells using the indexes, and buffer pool management keeps frequently used chunks in memory [12].

The SAVIME and SciDB databases support an Array Functional language (AFL) with a series of operators defined as functions [21, 17]. In this paper, we focus on the operator responsible for slicing an array. The operation has different names in the array databases: called *subset* in SAVIME, and *subarray* in SciDB. From this point, we refer to both operations as *subarray*. The *subarray* operator uses dimension indexes for fast access to data ranges, generating a new chunk according to the specified bounds. The *subarray* works like the selection operation of the relational algebra with range filtering condition. The code snippet below creates a new array with the cells within the three dimension range predicate.

Listing 1 Filtering the 3D temperature array using the subarray SciDB API

```
subarray(temperature,-25.423,-49.267,2000,-25.426,-49.265,2020);
```

Array databases implement *subarray* operators in different ways. SAVIME finds cells between the range using the query filter and generates many chunks with different sizes. SciDB decodes the compressed binary data for the chunks

and redistributes data to produce a new chunking configuration for the results that are within the range of interest. Although the *subarray* is a simple filter operation in the array data model, we may require processing all the chunks according to the query selectivity.

Considering that Array database systems use multiple threads to scale query processing, in the next section, we present the NUMA architecture and thread pinning strategies to benefit from multi-processing.

3 NUMA Architectures

The NUMA architecture is a hardware design to provide high throughput using multiple processing cores with a unified memory view. Ideally, this architecture provide high performance by maximizing computing power and data sharing.

The multiple processing cores are grouped in nodes that share the memory with non-uniform access latency. The NUMA architecture has node-to-node communications links, which provide high bandwidth with separate memory controllers per node. The memory hierarchy is composed by multiple cache levels and memory sharing schemes among the nodes. Each of the referred nodes can access both local memory banks with low latency and remote memory banks from neighboring nodes with higher latency (see Figure 2).

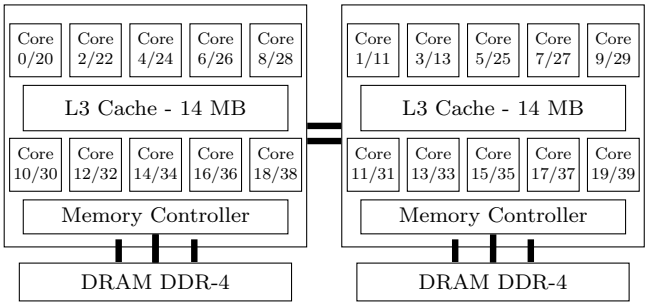

Fig. 2 Example of a NUMA architecture with 2-nodes inspired by the Intel Xeon Silver 4114 Skylake microarchitecture.

Considering that different data and thread allocation within a NUMA system may provide different latencies, it is essential to analyze such allocation for each specific application. Many related-work study thread pinning strategies for specific applications niches, such as numerical modeling and computational fluid dynamics applications, data mining, financial analysis, and media processing [9, 10, 20, 32, 8, 30, 27].

Current array databases have the query threads allocated by the OS, which may not be efficient in the database context. For example, thread mapping of the Linux OS aims load balancing. The OS allocates memory pages next to the node where the first access to the page occurs. During the load balancing,

threads may migrate, leading to an increase in data traffic and affecting the performance of the array database. Thus, our challenge is pinning threads on specific cores to benefit from the local memory and avoid the migration caused by the OS attempt to keep load balancing.

4 Thread Pinning Strategies

We now present five well-known thread pinning strategies that are used in our experiments. Our goal is to analyze the impact of these strategies compared to the baseline (i.e. OS scheduling strategy) and other thread mappings. **We get at runtime the required information to apply the pinning strategies. The strategies work seamlessly in SAVIME and SciDB systems because they use the same premise in their parallel query processing: Multiple threads are defined to process each chunk. SAVIME processes chunk groups with a pre-set number of threads for each group. In SciDB, the number of threads is defined to process the chunks grouped in instances. Notice that the thread placement strategies do not change data mapping policies. By default, the Linux OS uses the first-touch policy and we leave the OS-level features active for NUMA load balancing.**

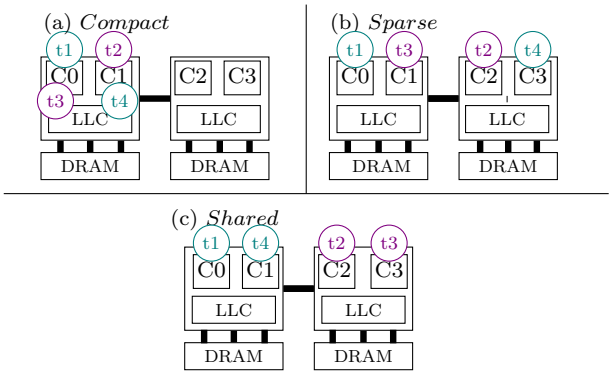

Fig. 3 Compact, Sparse and shared thread pinning strategies on a two NUMA node with four threads.

Baseline: To provide a commonly used baseline, we decided to measure the OS thread pinning performance without user interference. The OS uses a load balancer to distribute threads over all the NUMA nodes to maximize the usage of the computing cores.

Strategy 1 - Compact: In Figure 3(a) we present the *compact* strategy. **The thread pinning follows the order the threads are created and only uses one NUMA node. This strategy can provide workload**

benefits with high data reuse. However, the benefits are limited by the amount of memory available on the node.

Strategy 2 - Sparse: The *sparse* strategy distributes the threads equally among the nodes, one thread per node. For instance, $t1$ is allocated in node 0, $t2$ in node 1, and so on, as we show in Figure 3(b). Hence, our goal is to measure the performance when scattering the threads in the nodes and pinning them in one specific core inside the node.

Strategy 3 - Shared: The *shared* strategy aims to pin sets of threads that work on the same chunk of data to a single NUMA node. **In Savime, this is done through the thread affinity control of OpenMP. In SciDB, we capture the threads that work under the same instance and place them close together.** These threads share the Last-Level Cache (LLC) node, as shown in Figure 3(c). We use this strategy to analyze if data reuse has a positive impact on minimizing the effects of the NUMA memory latency disparity.

Strategy 4 - Petri net: In this strategy, we implement the dynamic core allocation mechanism presented in [11]. This mechanism implements a Petri net abstract model that decides the allocation of computing cores. The allocation decision relies on the performance states of the current database workload, respecting the assumption that a minimum number of cores maintains performance. Monitoring uses the CPU load metric of the execution machine as input to the performance state model. The Petri net controls the group and number of cores that the OS can use to schedule the database threads. This strategy assigns threads to a group of processing cores and does not pin threads to specific cores.

Strategy 5 - Random: In this strategy, the threads are randomly pinned to all the cores in the NUMA nodes. We generated all the possible thread pinning combinations, but avoided symmetrical core pinning, as well as similar allocations from the compact and sparse strategies. This random strategy aims at identifying possible thread pinning combinations that improve performance compared to the other strategies. In our experiments, we only show the best pinning combination concerning this strategy.

5 Experimental Evaluation

We now evaluate the performance of SAVIME version (v.1.0) and SciDB version (v.19.11.5) array databases in two NUMA machines using the previously described strategies of thread pinning. We essentially use the first machine in our experiments, and we point out the use of the second machine when appropriate.

The first NUMA machine (here called NUMA-Skylake) has two nodes, each node with an Intel Xeon Silver 4114 (with Skylake microarchitecture). Each Xeon socket has ten cores with private L1 (I+D) cache (32 KB each core), a private L2 cache (1 MB each core), and a shared L3 cache (14 MB total per node). The two NUMA nodes are interconnected by a Quick Path Interconnect

(QPI) [14] link 4.x with 21.5 GB/S bandwidth. The machine includes 128 GB DDR-4 main memory and 14 TB of disk storage (at 15000 rpm) running the Ubuntu OS in version 18.04.01 LTS for SAVIME and 14.04.6 LTS for SciDB. The different OS versions were used according to the Array database documentation. We run an unmodified Linux kernel, version $4.15.0 - 121 - generic$.

The second machine (here called NUMA-SandyBridge) has also two nodes, each node with an Intel Xeon $E5 - 2630$ (with Sandy Bridge microarchitecture). Each Xeon socket has six physical cores with private L1 (I+D) cache (32 KB each core), a private L2 cache (256 KB each core), and a shared L3 cache (15 MB total per node). The two NUMA nodes are interconnected by a QPI [14] link 4.x with 14.4 GB/S bandwidth. The machine includes 48 GB DDR-3 main memory and 869 GB of disk storage (at 7500 rpm) running the CentOS OS in version 7.9.2009.

To measure the hardware performance, we used the Intel Performance Counter Monitor (PCM) [33]. In particular, the Intel PCM tool provides the total power consumption of the main memory. To manage the thread pinning strategies, we used the OpenMP (version 4.5) thread affinity [7] and the *taskset* Linux command. In our experiments, we set the maximum number of threads available for each query execution to 20, which corresponds to the number of available physical cores. The workload has a dense array based on data from the HPC4e BSC seismic benchmark [6] used in work presented in [21]. We present the results considering the average over 10 executions for each thread pinning strategy and the *baseline*. **In our experiments, we found maximum values of coefficient of variation at 0.16 and minimum values at 0.07. These values indicate a low dispersion of data around the mean used to calculate the acceleration.**

We focus our analysis on the *subarray* operator. We chose this operator due to its simple memory access behavior and its considerable amount of applications [21, 26]. This operation has a coalescing memory access pattern due to the inequality predicate that retrieve ranges of array cells and no cache data reuse due to its data streaming behavior. Furthermore, the parallel processing of the *subarray* operation leads to data movement between NUMA nodes with direct performance impact in the array databases. **The performance of the subarray operator depends on how data is chunked and laid out [21], but it is still the most time-consuming query operator, as described in [5]. The execution of the subarray requires traversing the array and materializing the results for further use along the query pipeline.**

As for the results presented in this section, we normalized all results with the baseline that is the execution of the OS scheduler with a fixed size of 100 chunks. In the experiments with selectivity, we normalize the results with the execution of the OS scheduler with exact query filters (EQ). Our main goal is to understand the impact of the variation of the number of chunks.

5.1 Impact of the number of chunks

In this section, we investigate the impact of the number of chunks in an array of 1 GB. We recall that a chunk is the smallest storage unit in an array database. For this setup, whenever we increase the number of chunks for a given array, the size of each chunk decreases, thus keeping the same total storage size. In this experiment, all the configurations and strategies execute the same query operation over the same dataset, generating the same output.

Figure 4 depicts the speedup results for SAVIME and SciDB when varying the number of chunks. Here we used different thread pinning strategies running on the NUMA-Skylake machine. We normalized all the results to the OS scheduler (*baseline*) with 100 chunks. For this specific experiment, the random strategy was chosen among 20 possible random mappings due to time constraints (on Section 5.3 we present the exhaustive search over all random combinations).

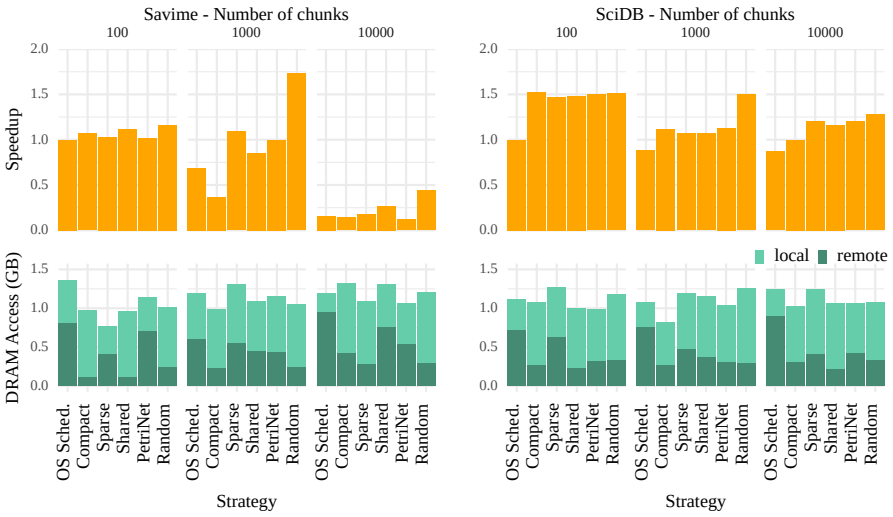

Fig. 4 Speedup and amount of memory accesses comparing different thread pinning strategies varying the number of chunks on a 1 GB array database (NUMA-Skylake machine).

We observe that the *random* strategy produces the most positive impact reaching a maximum acceleration of up to 1.7× in SAVIME and 1.5× in SciDB, both executing with 1000 chunks. The thread pinning strategies benefit from the NUMA architecture using smaller number of chunks (thus, bigger chunk sizes), possibly due to the reduced amount of chunk scheduling to the threads and the lower amount of data scattering among the NUMA nodes. These results show that we can achieve on average 52% performance gains with new thread pinning approaches.

We also observe that pinning each thread to a specific core is beneficial for the system's performance. It avoids thread migration between the NUMA nodes that happens when the OS tries to keep the load balancing. **This result shows that we need a static thread pinning strategy as the thread pinning does not dynamically change.** Figure 4 also presents the amount of remote and local memory accesses of the *subarray* operation. The results indirectly indicate how well the scheduler pinned the threads. Besides, the total access to DRAM memory indicates how well the cache memories were utilized. The variation of DRAM accesses shows that the strategies benefited differently from the cache memory. The *random* strategy shows a reduction in remote access by 4.9× for SAVIME and 3.5× for SciDB. However, we cannot find a direct link between the DRAM accesses and the final speedup, which indicates that more metrics are required to fully explain the results. Subsection 5.3 makes a broader evaluation to better understand the influence of cache and memory accesses on the final performance

5.2 Impact of Selectivity

We now evaluate the impact of different query selectivity using a 50 GB dataset. Selectivity indicates the percentage of data that needs to be filtered to materialize the *subarray* output. High selectivity (**H** - in this experiment 70%) indicates that we filter out more data, and consequently, less data is materialized to the output in DRAM. Low selectivity (**L** - in this experiment 20%) indicates the opposite. We also varied the number of chunks as they need to be transferred through the memory hierarchy to validate the query filters: Single Chunk (**SC**), Few Chunks (**FC**, here 20% of the chunks) and Many Chunks (**MC**, here 100% of the chunks). Besides, we present the Exact Query (**EQ**), which entirely selects a single chunk. We used a total number of 210 chunks to reproduce the workload executed by the SAVIME testbed [22] and store the 50 GB dataset in three-dimension chunks (250×250×500) of double-value attributes.

Figure 5 presents the speedup and brings the number of remote/local memory access varying the selectivity. First, we can notice that both Array databases performs efficient operations. They only access the memory relative to the requested chunk using their coordinates (observe the reduced amount of memory accesses for LSC and HSC). High selectivity scenarios perform better comparing the three scenarios with their low selectivity pair. This result is expected as high selectivity filters out more data, reducing thus cache memory usage to store the results also reducing pressure in the DRAM memory.

We can observe that neither the number of DRAM accesses, remote or local, seems to correlate to the final speedup directly. Nevertheless, we can observe that for almost every situation exists a different mapping that presents better performance, with 2.25× and 5.83× speedup in SAVIME and SciDB, respectively. In particular, the random mappings seem to have a trend of

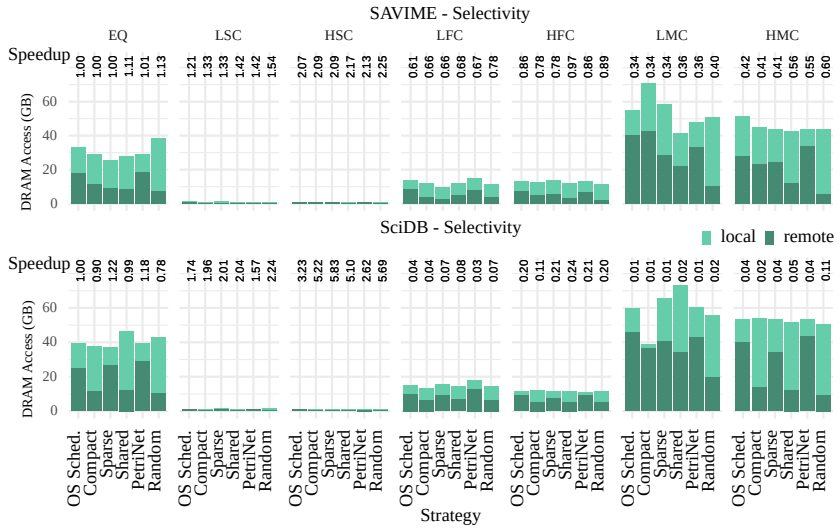

Fig. 5 Speedup and amount of memory accesses comparing different thread pinning strategies varying the operator selectivity on a 50 GB array database (NUMA-Skylake machine). Topmost X-axis number is the speedup for each strategy.

presenting low remote accesses, with an average reduction of 5× in SAVIME and 4.1× in SciDB.

We observed that the best thread pinning strategy is the shared strategy. The threads that work on the same chunk are placed in close cores maximizing data locality. This result shows that selectivity does not influence the choice of the thread pinning strategy. Instead, it indicates that the data locality influences the choice of the best thread pinning strategy.

Figure 6 shows the results of energy consumption in the DRAM memory (normalized by the OS scheduler executing EQ). We observe that the presented behavior is similar to the speedup results. The *random* strategy reduced on average the DRAM energy by 68% for SAVIME and 16% for SciDB. In both databases, the *compact* strategy of pinning threads to cores in only one NUMA node increased memory traffic due to high memory contention that resulted in more energy consumption in the main memory.

Overall, energy consumption was significantly reduced, due to decreases in remote memory access and data movement. The data movement accounts for most of the memory energy consumption.

5.3 Exhaustive strategy evaluation

In this section, we performed an extensive evaluation of thread pinning combinations. **The exhaustive execution covers many simple strategies,**

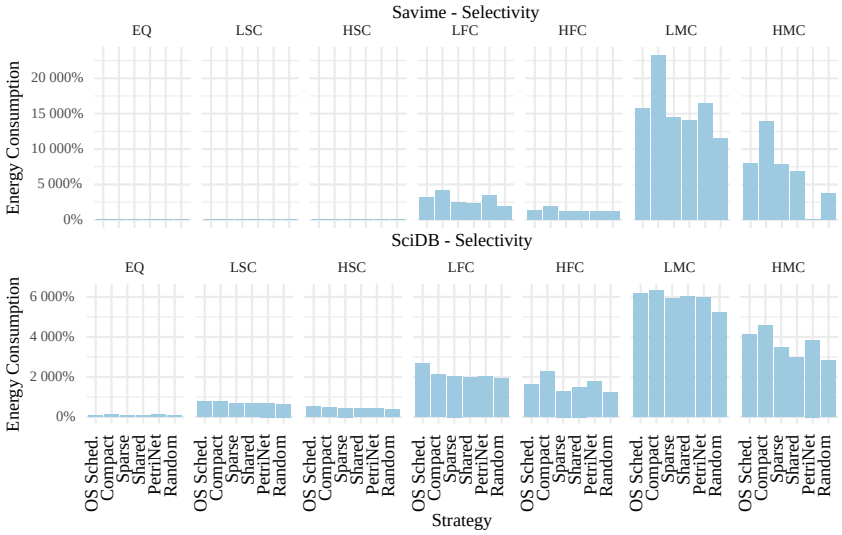

Fig. 6 Energy consumption in DRAM normalized with the OS scheduler executing EQ on 50 GB database using different selectivity (NUMA-Skylake).

for example, pinning threads, in the same order these threads are created, to cores as close as possible to benefit from data locality. We used the NUMA-SandyBridge due to its reduced amount of cores, which allows us to evaluate all 462 possible combinations without repeating analogues. We do not evaluate all combinations in the NUMA-Skylake machine due to a higher number of processing cores (20 cores). Such an exhaustive experiment would be impractical requiring $92,378$ combinations.

When setting the total number of *random* thread pinning combinations, we consider the machine memory hierarchy and the number of threads used by SAVIME. We executed SAVIME with 12 threads varying the number of chunks on a 1 GB array. Looking at these metrics, we use the combinations without repetition $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. We have n threads, and we want to choose r threads out of n to pin in one NUMA node of the architecture that shares cache memory. In the NUMA-SandyBridge, we found a total of 924 combinations running 12 threads. However, considering that we have two identical NUMA nodes, with 6 cores in each node, we reduced to a total of 462 different thread pinning possibilities.

Figure 7 shows the speedup results with three different number of chunks and also reports the local and remote memory access. We observed that the experiments with 100 and $1,000$ chunks present more pronounced variations in the speedup. We observe the clear impact of remote accesses on combinations between 0-100, with green dots higher than blue ones. Whenever the mapping enables a reduction on the amount of remote accesses (combinations between 100-200), we see the improvement in performance. Finally, we also observe

improvement in performance whenever the cache memory is better utilized, reducing thus the total amount of DRAM memory accesses (combinations between 200-462). While it is difficult to distinguish the benefits of better cache usage and less remote access, we found that both metrics affect SAVIME performance.

Fig. 7 SAVIME: Performance and DRAM accesses for all thread pinning combinations on 1 GB database (NUMA-SandyBridge). Results are ordered by speedup compared to the OS scheduler.

We notice the speedup decreases in 7% of the total number of experiments. That is because in some *random* combinations, the threads that work in the same chunk end up in different nodes, forcing remote accesses and 2.5× more memory accesses.

Figure 8 shows that the cache miss ratio follows the same pattern observed in memory access. We observe that performance increases as the last level cache miss ratio reduces. These results corroborate the findings present in the last plotted results. Furthermore, we observe that using fewer chunks decreases cache misses, possibly because the cells of the small chunks fit into the caches avoiding miss penalties.

5.4 Performance comparison on NUMA architectures

This last experiment focuses on the performance comparison between the two NUMA machines (NUMA-Skylake and NUMA-SandyBridge) running SAVIME. **Our main goal is to evaluate whether the scheduling policies**

Fig. 8 L3 Cache miss ratio in a 1 GB database with different number of chunks varying the total number of thread pinning combinations (NUMA-SandyBridge).

would behave equally in terms of performance for different machines. We varied the number of chunks of an array of 50 GB. The results are normalized to the OS scheduler results with 100 chunks from each machine. The three main differences between the machines are a number of cores, L3 cache size and difference between local and remote memory latency (NUMA factor). NUMA-Skylake have 2x 10 cores sharing 14 MB of L3 and NUMA factor of 1.36 and NUMA-SandyBridge have 2x 6 cores sharing 15 MB of L3 and NUMA factor of 1.32.

In Figure 9, we observe that well-known scheduling policies provide different performance results depending on the machine, which indicates that scheduling policies must evolve together with newer architectures. For example, the sparse strategy increases the access latency cost, which fits better to the NUMA-Skylake architecture due to its lower latency and higher bandwidth.

5.5 Conclusions of NUMA effect on Array database

We conclude in this section that well-known thread pinning policies provide moderate speedup and are still far from the maximum performance achievable for array databases. When looking at the selectivity, we could observe that both database systems are impacted with variations in the selectivity, where performance degrades starting from higher to lower selectivity.

We could also observe that finding a combination of thread pinning that improves performance is challenging. Such a thread pinning combination must

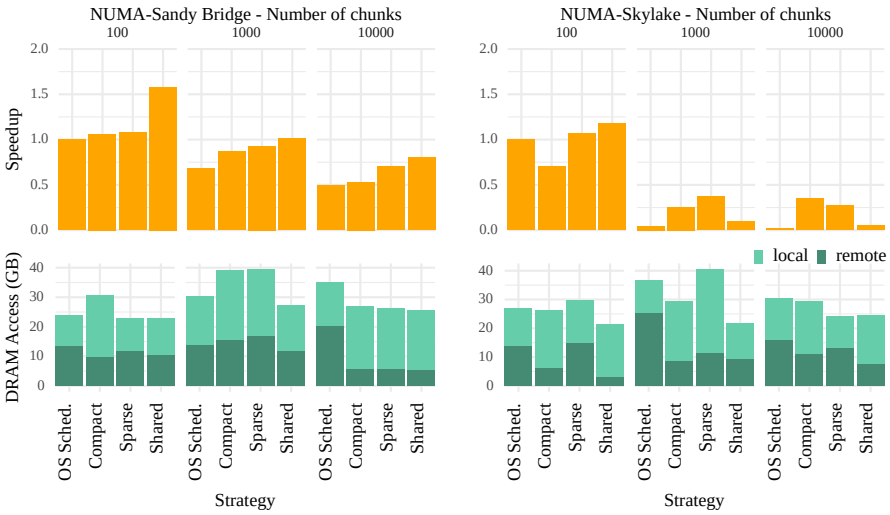

Fig. 9 Performance comparison of *subarray* operator in 50 GB database using different numbers of chunks in SAVIME Array database in NUMA-Skylake and NUMA-SandyBridge. **Our results were normalized with the results of the OS scheduler with 100 chunks from each machine.**

at the same time reduce remote memory access and improve the utilization of the cache memory hierarchy. Results show that different thread pinning combinations, even with similar metric of amount of DRAM accesses, may lead to very different speedups (combinations 100-200 in Figure 7). We observed that using only one metric is insufficient to decide the best mapping and a multivariate analysis is required in future work. Interestingly, results showed that 56% of the possible combinations (combinations 200-462 in Figure 7) leads to the most efficient performance observed in our tests. This result motivates our next steps in developing a specific thread scheduler for array databases.

6 Related Work

The performance impact of the NUMA machines has motivated several recent works in different areas. In computer architecture, these researches employ different thread mapping techniques to minimize the NUMA effect. For instance, the works presented in [9, 20, 10, 32, 8, 30, 16] focus on thread placement techniques based on memory access patterns and communication cost between nodes.

In database systems, researches that mitigate the effects of the NUMA systems in query processing gain momentum. These researches have focused on the relational storage model running specific query operations or thread/data placement strategies. In [1, 2], the authors present techniques to improve the performance of the SQL join operation by pinning the operation threads into

specific NUMA nodes. In [28] Online Transaction Processing (OLTP) threads are pinned in NUMA islands grouping different computing nodes. In [18, 13, 25] the data is partitioned in specific NUMA nodes, and querying threads are statically pinned to the data location. The database scheduler of the HyPer database uses a similar technique to control the dispatching of query fragments, called "morsels" [19]. The "morsels" are statically pinned in specific cores to take advantage of the data location and avoid data movement between nodes.

In [11], the authors present a multi-core allocation technique reducing the number of cores that the OS could use to allocate the query threads. A similar strategy presented by [13] uses OS policies to designate the number of resources needed and creates a communication between the OS and the RDBMS in execution. Recent work present in [24] analyzes the impact of different memory allocation and thread placement at the kernel level running analytical workload in relational databases. The authors discuss the improvements achieved with dynamic memory allocation and OS thread placement policies. The work present in [29] proposes a NUMA-aware algorithm for spatial join and analyzes the behavior of data placement OS policies.

We observed that all of these works focus on the relational model. In contrast, our research investigates the impact of the NUMA architecture on array processing operations that differs from traditional relational operations. We analyzed whether thread placement improves the performance of the array query processing.

7 Conclusions and Future Work

Based on the fact that relational databases and array databases adopt similar strategies when using multi-thread parallelism, only the former has been extensively studied in terms of performance behavior when using NUMA systems. As far as we know, we present the first study of the speedup and energy consumption impact of array query processing in NUMA machines.

By implementing different thread pinning strategies in two array databases, we showed how each strategy behaved. Our results support that the NUMA severely affects the performance of the *subarray* operation. The *subarray* operation is based on inequality conditions and requires moving ranges of array cells across computing nodes for validating these conditions. The baseline thread pinning strategy based on OS thread mapping does not acknowledge the relationship between query operators in the query execution plan. Thus, threads are pinned for load balancing far from efficient query processing performance. We also observed that different NUMA implies distinct performance gains.

Our next steps include understanding the NUMA effects in other query array operators with different memory access patterns, as presented in [15] for relational query operators. Furthermore, we intend to study cost models searching for a good trade-off between perfor-

mance and energy consumption considering the pinning strategies presented in this paper.

Acknowledgements We would like to thank the UFRGS, some experiments in this work used the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.

Declarations

Funding: This work was supported by Serrapilheira Institute (grant number Serra-1709-16621) and CAPES.
Conflicts of interest/Competing interests: Not applicable.
Availability of data and material: The data, queries, and other information used in our experiments are available at `https://github.com/Simone-Dominico/array_database_teste`.
Code availability: Not applicable.

References

1. Albutiu MC, Kemper A, Neumann T (2012) Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB 5(10):1064–1075
2. Balkesen C, Alonso G, Teubner J, Özsu MT (2013) Multi-core, main-memory joins: Sort vs. hash revisited. PVLDB 7(1):85–96
3. Baumann P, Furtado P, Ritsch R, Widmann N (1997) The rasdaman approach to multidimensional database management. In: SAC '97, p 166–173
4. Brown PG (2010) Overview of scidb: Large scale array storage, processing and analysis. In: SIGMOD, p 963–968
5. Camara G, Egenhofer MJ, Ferreira K, Andrade P, Queiroz G, Sanchez A, Jones J, Vinhas L (2014) Fields as a generic data type for big spatial data. In: International Conference on Geographic Information Science, Springer, pp 159–172
6. Center BS (2016) New hpc4e seismic test suite to increase the pace of development of new modelling and imaging technologies. `https://www.bsc.es/news/bsc-news/new-hpc4e-seismic-test-suite-increase-the-pace-development-new-modelling-and-imaging-technologies`
7. Chandra R, Dagum L, Kohr D, Menon R, Maydan D, McDonald J (2001) Parallel programming in OpenMP. Morgan kaufmann
8. Chasparis GC, Rossbory M, Janjic V (2017) Efficient dynamic pinning of parallelized applications by reinforcement learning with applications. In: Euro-Par: Parallel Processing, pp 164–176
9. Cruz EHM, Alves MAZ, Carissimi A, Navaux POA, Ribeiro CP, Méhaut JF (2012) Memory-aware thread and data mapping for hierarchical multi-core platforms. Int Journal of Networking and Computing 2(1):97–116

10. Cruz EHM, Diener M, Pilla LL, Navaux POA (2016) Hardware-assisted thread and data mapping in hierarchical multicore architectures. ACM Trans Archit Code Optim 13(3):1–28
11. Dominico S, de Almeida EC, Meira JA, Alves MAZ (2018) An elastic multi-core allocation mechanism for database systems. In: ICDE, pp 473–484
12. Gerhardt L, Faham C, Yao Y (2015) Accelerating scientific analysis with scidb. In: Journal of Physics: Conference Series, IOP Publishing, vol 664, p 072019
13. Giceva J, Alonso G, Roscoe T, Harris T (2014) Deployment of query plans on multicores. PVLDB 8(3):233–244
14. Intel (2019) Maximizing multicore processor performance. `https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html`
15. Kepe TR (2019) The design and implementation of query execution in modern processing-in-memory hardware. PhD thesis, UFPR - Federal University of Paraná, Curitiba - Brazil, 115 pages.
16. Khaleghzadeh H, Manumachu RR, Lastovetsky A (2018) A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms. TPDS 29(10):2176–2190
17. Kim S, Sohn SG, Kim T, Yu J, Kim B, Moon B (2016) Selective scan for filter operator of scidb. In: SSDBM '16, pp 1–4
18. Kissinger T, Kiefer T, Schlegel B, Habich D, Molka D, Lehner W (2014) Eris: A numa-aware in-memory storage engine for analytical workloads. In: ADMS@ VLDB, pp 1–12
19. Leis V, Boncz P, Kemper A, Neumann T (2014) Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In: SIGMOD, p 743–754
20. Lepers B, Quéma V, Fedorova A (2015) Thread and memory placement on numa systems: Asymmetry matters. In: USENIX ATC '15, pp 277–289
21. Lustosa H, Porto F (2019) SAVIME: A multidimensional system for the analysis and visualization of simulation data. CoRR abs/1903.02949
22. Lustosa H, Porto F, Blanco P, Valduriez P (2016) Database system support of simulation data. PVLDB 9(13):1329–1340
23. Lustosa H, Lemus N, Porto F, Valduriez P (2017) Tars: An array model with rich semantics for multidimensional data. In: ER FORUM, pp 114–127
24. Memarzia P, Ray S, Bhavsar VC (2020) The art of efficient in-memory query processing on NUMA systems: a systematic approach. In: ICDE, pp 781–792
25. Ozturk O, Orhan U, Ding W, Yedlapalli P, Kandemir MT (2017) Cache hierarchy-aware query mapping on emerging multicore architectures. IEEE Computer Society, USA, vol 66, p 403–415
26. Papadopoulos S, Datta K, Madden S, Mattson T (2016) The tiledb array data storage manager. PVLDB 10(4):349–360

27. Popov M, Jimborean A, Black-Schaffer D (2019) Efficient thread/page/-parallelism autotuning for numa systems. In: ICS '19, pp 342–353
28. Porobic D, Pandis I, Branco M, Tözün P, Ailamaki A (2012) Oltp on hardware islands. PVLDB 5(11):1447–1458
29. Ray S, Higgins C, Anupindi V, Gautam S (2020) Enabling numa-aware main memory spatial join processing: An experimental study. In: ADMS@ VLDB
30. Sánchez Barrera I, Moretó M, Ayguadé E, Labarta J, Valero M, Casas M (2018) Reducing data movement on large shared memory systems by exploiting computation dependencies. In: ICS '18, pp 207–217
31. Soroush E, Balazinska M, Wang D (2011) Arraystore: a storage manager for complex parallel array processing. In: SIGMOD, pp 253–264
32. Virouleau P, Broquedis F, Gautier T, Rastello F (2016) Using data dependencies to improve task-based scheduling strategies on numa architectures. In: Euro-Par 2016, pp 531–544
33. Willhalm Thomas FP Dementiev Roman (2012) Intel performance counter monitor. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`
34. Zhang Y, Kersten M, Manegold S (2013) Sciql: array data processing inside an rdbms. In: SIGMOD, pp 1049–1052