Enabling NUMA-aware Main Memory Spatial Join Processing: An Experimental Study

Suprio Ray, Catherine Higgins, Vaishnavi Anupindi and Saransh Gautam
University of New Brunswick
Fredericton, NB, Canada
{sray, c.higgins, vanupind, sgautam}@unb.ca

ABSTRACT

Spatial join queries are integral to spatial data analytics. As the volume of spatial data grows at a rapid pace, the performance of spatial join queries remains ever more relevant. With the modern hardware trends leading to non-uniform memory access (NUMA) architecture, how to leverage it for efficient spatial join execution becomes an important research question. Although previous researchers explored how to accelerate spatial join performance, to our knowledge, they have not considered NUMA architecture in this context.

In this paper, we investigate how to enable efficient processing of in-memory spatial join on NUMA systems. We implemented five different parallel in-memory spatial join algorithms, that include a tailored NUMA-aware algorithm. We conduct an experimental study of these spatial join algorithms in various settings with seven different spatial join queries involving four spatial data tables from a real-world dataset. Our study reveals a few interesting findings and directions for future research.

1. INTRODUCTION

Large volumes of spatial data are constantly being generated from a multitude of sources. Consequently, the landscape of spatial analytics is rapidly evolving, as the popularity of many spatial data driven applications rise and new applications emerge regularly. Spatial join operations play an important role in many spatial analytics applications and geographical information systems (GIS). For example, the ability to combine two spatial inputs in order to create a new map is an essential operation in many fields, such as cartography, urban planning and landscaping [3]. Spatial join queries provide valuable information on topological relationships between real-life spatial and geographical elements such as landmarks, roads and rivers and how they relate in a two dimensional space [15]. Unlike regular join queries, spatial joins are compute intensive operations that involve processing geometric objects of different types like points, polylines, polygons and complex shapes such as swiss-cheese-polygons. Since, a spatial database system may need to deal with large quantities of data, it is imperative for such a system to perform spatial join queries efficiently.

Due to the importance of spatial queries, most modern commercial and even several open-source database systems support some spatial functionalities. For instance, PostgreSQL supports an extensive set of spatial features with its PostGIS extension. As the current hardware trends move towards multi-processor systems with non-uniform memory access architecture (NUMA), database systems vendors need to consider how to leverage them for spatial query execution. NUMA architecture allows for scalability and presents many hardware advantages, as it is a multi-socket architecture where each socket possesses its own local memory and memory controller and can have many cores. However, how to maximize performance gains on such a system is a fairly new area of research and there is no one-size-fits-all solution. To make matter worse, it has been demonstrated that the default OS settings can result in sub-optimal performance on NUMA systems [8]. As NUMA is becoming the norm in modern hardware, exploring its potential only makes sense.

A common way to improve query performance is by partitioning work to different cores to enable multiple threads to work in parallel. Typically, the dataset is evenly distributed such that each partition or tile occupies approximately the same number of objects. However, the resulting workload may not always be evenly balanced, due to inherent processing and data skew, and hence the longest task determines the overall query execution time. Although improving the efficiency of spatial joins were the focus of a number of research projects [12, 13, 10, 18, 19, 9, 17], none of them explored spatial join in the context of NUMA architecture. In this paper we investigate the performance of spatial join operations on such a system. To the best of our knowledge, it is the first such study. When considering parallel work in NUMA architecture, we need to consider the locality of data in memory. A specific NUMA architecture is usually composed of several NUMA nodes, where each node possesses one or more CPUs with its own local memory. NUMA nodes are connected by high-speed interconnect. Remote memory accesses are slower than local memory accesses, because data must be transferred over one or more interconnect hops. Also, CPUs can generate a high number of memory requests, resulting in congestion in the interconnect links and memory controllers. Task parallelism can also cause performance hindrance in situations where memory is being accessed concurrently by more than

11th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'20) August 31, 2020, Tokyo, Japan.
ISSN xxxx-xxxx.
DOI:

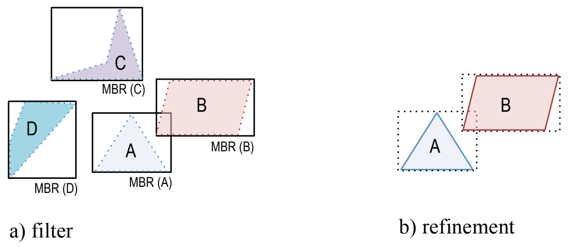

Figure 1: Two step spatial join evaluation

one threads [8]. Due to these factors, applications that are not NUMA-aware can suffer from performance issues on a NUMA system.

This paper focuses on how to enable NUMA-aware spatial join query execution. To that end, we implemented five different parallel in-memory spatial join algorithms: BNLJ, BINLJ, TWNLJ, PBSM-IM and PBSM-IM-NA. Of these, the first four are not NUMA-aware. PBSM-IM is our in-memory parallel implementation of the disk-based Partition Based Spatial-Merge (PBSM) algorithm [12]. We also implemented a tailored NUMA-aware spatial join algorithm PBSM-IM-NA by extending PBSM-IM. We utilize the Linux library *libnuma* [2] that offers fine-grained control over memory placement.

We executed these algorithms with seven different spatial join queries (see Table 2), involving different spatial join predicates. We also ran experiments in which we configure different NUMA memory placement policies (*Localalloc*, *Interleave* and *Preferred*) for the 4 non-NUMA-aware spatial join approaches using Linux command `numactl`, which is a Linux command and is not as fine-grained as *libnuma*. We conducted an extensive experimental evaluation with a real-world spatial dataset involving four data tables (see Table 1) consisting of spatial objects, such as points, polylines and polygons, in California.

Our study reveals several interesting findings (see Section 5). For instance, the *Interleave* policy does not offer the best performance for spatial join, which was previously shown to be the best performing policy for regular (non-spatial) queries [8]. Furthermore, a purpose-built NUMA-aware spatial join (PBSM-IM-NA) does not necessarily offer the best performance, if it does not consider dynamic load conditions. Also, it is important to consider the organization of the in-memory data store for spatial objects. We believe that these finding are useful for practitioners and researchers and these will open up new avenues for further research.

In summary, the main contributions of this paper are:
- We implement four parallel in-memory spatial join algorithms and a tailored NUMA-aware parallel in-memory spatial join algorithm.
- We conduct extensive experimental evaluation with a real-world spatial dataset and seven spatial join queries.
- Our research reveals several interesting findings.

2. BACKGROUND

In this section we provide some background related to our work.

2.1 Spatial join processing

A spatial join operation involves joining two sets of geometric objects that are associated by a spatial predicate. A spatial predicate identifies topological relationship between two objects, such as, intersects, overlaps, touches, or contains. Spatial join is a computationally expensive task and typically involves a two-step process to reduce the latency. As shown in Figure 1, the first step, *filter*, involves eliminating objects that do not satisfy the spatial predicate by comparing an approximation of the objects from each input using their minimum bounding rectangles (MBR). The objects that pass this step constitute the *candidate set*. The second step, *refinement*, is where each candidate object pairs from the two sets are evaluated to determine if they satisfy the predicate. This step is the more expensive one, as it employs computational geometric algorithms to evaluate the spatial predicate. The filter step helps minimize the time spent in the refinement step by eliminating objects that do not meet the criteria.

2.2 NUMA policies

There are two different ways to configure NUMA policies for a Linux application. The first option is *libnuma* [6], which is a user space shared library in Linux that offers C/C++ APIs to control NUMA policies from application code. It is considered a more user-friendly interface than using the system calls directly, when fine-grained control is required. The second option is a Linux command line tool, called `numactl`, to execute programs with a specific NUMA policy. It does not offer as much control as the *libnuma*. All children processes inherit the policies from parents.

Since local accesses are faster than accesses to remote NUMA nodes, memory placement policies can be used to control the location of memory pages. There are several different policies available. In *Localalloc*, the memory pages are placed on the same NUMA node as the thread performing the allocation [8]. The *Interleave* policy distributes memory pages in a round-robin manner to a set of given nodes. This is often useful when multiple processors may want access to the same memory. *Bind* and *Preferred* policies both assign memory to the specified nodes and use node-local CPU cores. The difference between the two is that when memory cannot be allocated on the specified node, *Preferred* will allocate memory to nearby nodes, whereas *Bind* will fail. Hence, *Bind* policy can result in more frequent swapping, which may slow down performance.

3. APPROACH

In Section 3.1 we outline the spatial join algorithms that we implemented. Then in Section 3.2, we describe the queries and dataset that we used in our experimental study.

3.1 Spatial join algorithms

First, we formally define spatial join and then we describe the spatial join algorithms we utilized in our study. We implemented and evaluated five different parallel in-memory spatial join algorithms: BNLJ, BINLJ, TWNLJ, PBSM-IM and PBSM-IM-NA. They are described in sections 3.1.2 through 3.1.6.

3.1.1 Definition

Typically, a spatial object contains a spatial (geometry) attribute and several non-spatial attributes. Assume that a spatial object takes the form: $o = (id, geom, attrs)$, where id is the object id; $geom$ is the geometry attribute and $attrs$ is a set consisting of the other attributes such that $attrs =$

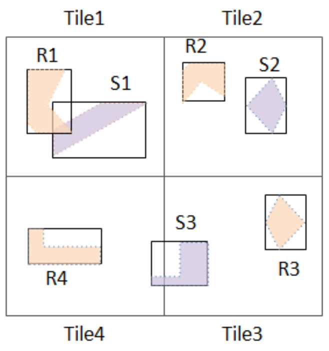

Figure 2: Spatial declustering

Table 1: Dataset details

Table	Abbreviation	Number of records	Geometry
Pointlm	Pl	49837	Point
Edges	Ed	4173498	Polyline
Arealm	Al	5951	Polygon
Areawater	Aw	39334	Polygon

$\{attr_i : i <= n - 2\}$, n is the total number of attributes including id and $geom$.

For any two datasets R and S, each containing spatial objects, and a given spatial predicate sp, the spatial join $\bowtie_{sp}$ finds pairs of objects, which satisfy the following:

For $r \in R$ and $s \in S$, $sp(r.geom, s.geom) = true$.

3.1.2 *Block Nested Loop Join (BNLJ)*

A nested loop join based approach can be used to evaluate a spatial join, by iterating over the objects in both datasets R and S and evaluating the spatial predicate sp. We parallelize this with a block nested loop join approach, BNLJ. In this, one of the datasets R is range partitioned (non-spatial), whereas S is retained as is. It partitions the dataset R into a set RB of B blocks such that $RB = \{RB_j : 1 \leq j \leq B\}$. Then each block RB_j is processed in parallel by a separate thread. For each record in $r_j \in RB_j$ and for each record $s \in S$, the spatial predicate sp is evaluated that follows a two-step process as described in Section 2.1.

3.1.3 *Block Index Nested Loop Join (BINLJ)*

In an index nested loop join approach for spatial join, a spatial index, I_S, is created on one of the datasets, S. This index could be based on a spatial data structure such as R-tree, quadtree, k-d tree or a variant of them. Then for each object $r \in R$, I_S is searched to find all matching objects M_S and each object $s \in M_S$ and r are used to evaluate sp. We parallelize this approach with a block index nested loop join that we call BINLJ. In BINLJ, the dataset R is range partitioned into a set RB of B blocks such that $RB = \{RB_j : 1 \leq j \leq B\}$ and each RB_j is processed in parallel. For each object $r_j \in RB_j$ the spatial index I_S is searched and the matching objects are used to evaluate sp.

In our algorithm we use a Sort-Tile-Recursive (STR) [7] as the index, which is a variant of R-tree that utilizes bulk-loading for R-tree packing. Our implementation utilized an STR implementation from the GEOS library [5].

3.1.4 *Tile-wise Nested Loop Join (TWNLJ)*

Table 2: Spatial join queries

Query	Description	Geometry objects involved
Pl_wi_Aw	Pointlm within Areawater	Point & Polygon
Ed_cr_Al	Edges cross Arealm	Polyline & Polygon
Aw_ov_Aw	Areawater overlap Areawater (self join)	Polygon & Polygon
Al_ov_Aw	Arealm overlap Areawater	Polygon & Polygon
Al_wi_Aw	Arealm within Areawater	Polygon & Polygon
Al_to_Aw	Arealm touch Areawater	Polygon & Polygon
Al_in_Aw	Arealm intersect Areawater	Polygon & Polygon

In this approach, the spatial domain is decomposed into equal-sized partitions called *tiles*. This process, as shown in Figure 2, is called *spatial declustering*. If an object overlaps multiple tiles, such as $S3$, it is replicated to all such tiles ($Tile3$ and $Tile4$). Each tile can be processed independently by a separate processing element and hence different variants of spatial declustering are often used in parallel spatial join algorithms, such as [12, 13]. For each tile, only those objects whose MBRs overlap with the tile MBR need to be processed and the rest can be ignored. In Figure 2, while processing $Tile1$ only objects $R1$ and $S1$ need to be considered.

In tile-wise nested loop join (TWNLJ), we perform spatial declustering on both the datasets R and S to form the tiles T_R and T_S. Then the tiles are assigned to available threads in a round-robin fashion. The l-th tile (where $1 \leq l \leq L$, L is total number of tiles) is processed by iterating over the objects in $r_l \in T_{Rl}$ and $s_l \in T_{Sl}$ and performing a tile-local nested loop join.

3.1.5 *PBSM In-Memory Parallel (PBSM-IM)*

The original Partition Based Spatial-Merge (PBSM) [12] was designed to be a disk-oriented algorithm. We implemented an in-memory parallel version of this algorithm that we call PBSM-IM. In this approach, the spatial domain is partitioned using spatial declustering, as described in Section 3.1.4. Then the tiles are assigned to the available threads in a round-robin manner and the objects in each tile, T_R and T_S corresponding to the two datasets R and S, are evaluated locally. Although PBSM-IM and TWNLJ use a similar declustering technique, there are differences between them. Unlike in TWNLJ, during the filter step a plane-sweep is performed on all the MBRs of the objects in a tile to determine overlap among them. Those objects that pass the filter step are further processed in the refinement step. In Figure 2, none of the objects in $Tile2$ passes the filter step and so a refinement step is not needed for the objects in this tile.

3.1.6 *PBSM In-Memory Parallel - NUMA Aware (PBSM-IM-NA)*

The PBSM-IM algorithm is not NUMA-aware. We explicitly make PBSM-IM NUMA-aware. To achieve this we first assign the tiles to the available threads in a round-robin manner during the tile assignment, and then bind each thread to a specific NUMA node using the `numa_bind()` function from the Linux library *libnuma* [2]. This makes sure that each tile is processed by a specific NUMA node. We call our tailored NUMA-aware algorithm PBSM-IM-NA.

3.2 Query and datasets

Table 3: Experimental settings

Parameter	Settings
Spatial join algorithms	BNLJ, BINLJ, TWNLJ, PBSM-IM, PBSM-IM-NA
NUMA policies	***Default***, *Localalloc*, *Interleave*, *Preferred*
Number of threads	1, 2, 4, **8**, 16

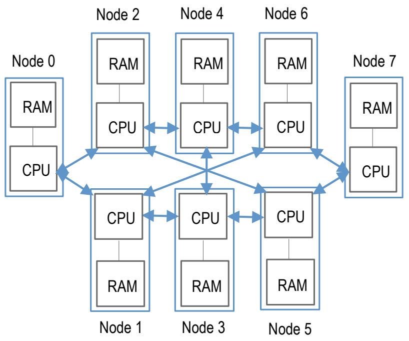

Figure 3: NUMA topology

For our experimental study we used a real-world spatial dataset from the US Census Bureau's TIGER [1] dataset. It consists of four data tables, the details of which are shown in Table 1. They include edges (polylines), landmark points, land area (polygons) and water area (polygons) for all counties in California. We selected a number of queries from the Jackpine [16] spatial database benchmark. They are shown in Table 2. These queries involve several spatial predicates between different combination of pairs of spatial objects from two data tables (or one table, for self join). A spatial predicate specifies the relationship between two spatial objects in terms of topological constraints. For instance, in Figure 2 objects *R1* and *S1* satisfy the spatial predicate *overlap*. A number of formal models were proposed to model these topological relationships. We use the Dimensionally Extended Nine-Intersection Model (DE-9IM) [4] that was adopted by the Open Geospatial Consortium (OGC) [11]. We use seven queries in our study, as shown in Table 2.

4. EXPERIMENTS

In the section we describe our experimental study in detail. We start by outlining the experimental settings. Then we present the results in the following sections.

4.1 Settings

The experiments were conducted on a machine with dual-core AMD Opteron processors with a total of 16 cores. The cores were equally divided among 8 NUMA nodes. The topology of our system is shown in Figure 3. Our system has an aggregate memory of 128 GB, where each NUMA node is associated with 16 GB of memory. Our system ran Ubuntu 16.06 OS, which is a Linux-based OS. We implemented our spatial join algorithms with C++ and compiled using g++ with the -O3 flag. For the spatial declustering based algorithms, we used a 32×32 regular grid (1024 tiles). Except for Section 4.2, we present the results of multi-threaded execution involving 8 threads.

4.2 Multicore scalability

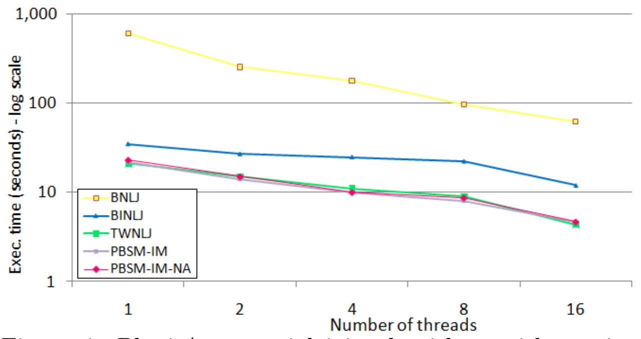

Figure 4: Pl_wi_Aw: spatial join algorithms with varying number of threads

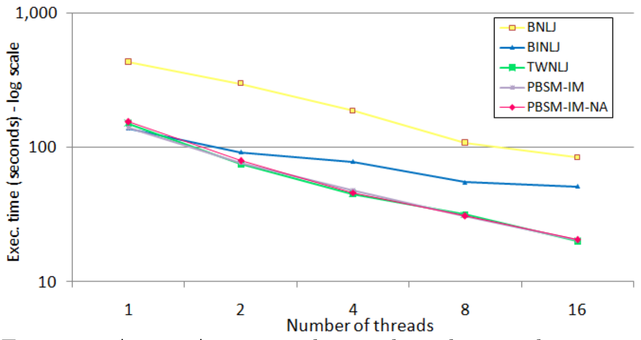

Figure 5: Aw_ov_Aw: spatial join algorithms with varying number of threads

In this section, we evaluate the multicore scalability of the five parallel in-memory spatial join algorithms. We executed the algorithms and varied the number of threads 1, 2, 4, 8 and 16. Due to space constraint, we present results from three queries: Pl_wi_Aw, Aw_ov_Aw and Ed_cr_Al. They cover all four data tables and also include query execution times that can be termed as low, medium and high. Note that, of the five algorithms only PBSM-IM-NA is NUMA-aware. With the other algorithms no NUMA policies were set explicitly, that is, the Linux default settings applied.

The first set of results are for the query Pl_wi_Aw. As shown in Figure 4, all five algorithms achieved improved speedup as the number of threads were increased. However, BNLJ performed the worst in all cases and the performance of BINLJ was the second worst. Among the rest of the algorithms, the performance gaps were not that high. With 16 threads, PBSM-IM took 4.7 seconds to complete, whereas, the NUMA-aware version PBSM-IM-NA also took 4.7 seconds.

The next set of results are for Aw_ov_Aw, as can be seen in Figure 5. BNLJ again performed the worst in all cases. On the other hand, with 1 thread BINLJ performed the best. However, as the number of threads increased, the performance of BINLJ became worse than that of TWNLJ, PBSM-IM and PBSM-IM-NA. Recall that BINLJ uses a spatial index STR on dataset *S*, which enables it to perform well with 1 thread. However, with more than 1 thread, the declustering based approaches performed better, even without utilizing any index. With 16 threads, TWNLJ performed the best, whereas PBSM-IM and PBSM-IM-NA both came very close to it.

The results for Ed_cr_Al are presented next. As shown in Figure 6, this query is a long running one. For instance, with 1 thread BNLJ takes 3744 seconds to complete. Once again,

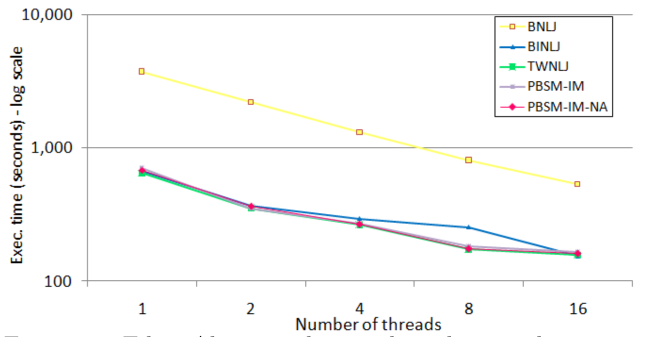

Figure 6: Ed_cr_Al: spatial join algorithms with varying number of threads

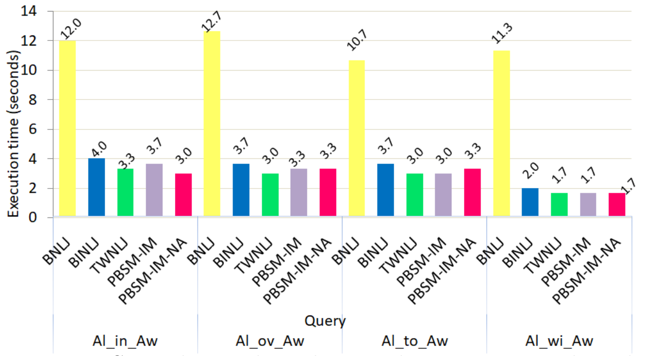

Figure 7: Spatial join algorithms with varying spatial predicates

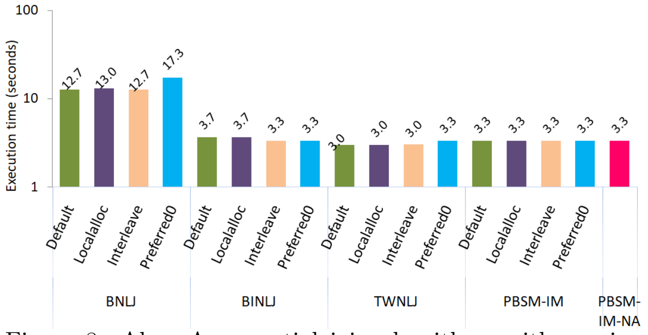

Figure 8: Al_ov_Aw: spatial join algorithms with varying NUMA policies

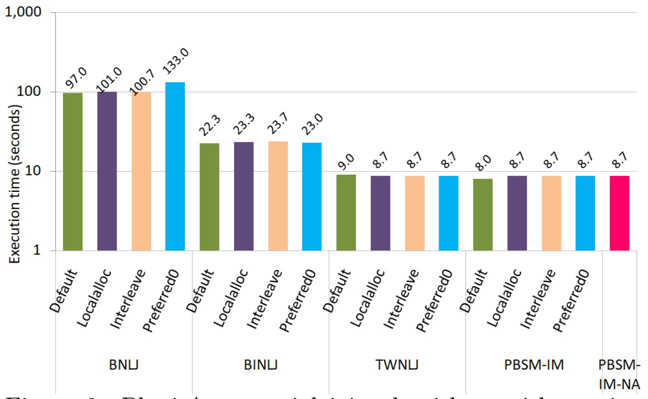

Figure 9: Pl_wi_Aw: spatial join algorithms with varying NUMA policies

BNLJ performed the worst among the algorithms. TWNLJ performed the best in all cases except for the case with 16 threads, in which interestingly, BINLJ did slightly better. The execution times of PBSM-IM and PBSM-IM-NA came quite close.

From the above results, it can be observed that a custom-built NUMA-aware spatial join approach (PBSM-IM-NA) performs well, but it does not necessarily win in all cases. Overall, spatial declustering has a significant impact on the query performance.

4.3 Spatial predicates

In this section, we evaluate the effects of spatial predicates on spatial join performance. To do a fair comparison, we evaluated four queries that involve the same two data tables, Arealm and Areawater, while we varied the spatial predicates. The queries that we evaluated are: Al_in_Aw, Al_ov_Aw, Al_to_Aw and Al_wi_Aw. As shown in Figure 7, BNLJ was the worst performer in all the four queries, while BINLJ came next, although there was a significant performance gap between these two. Among the rest of the algorithms, the performance gap was less prominent. There was no clear winner in all the fours queries, with PBSM-IM-NA performed the best in Al_in_Aw, while TWNLJ did the best in Al_ov_Aw. In Al_to_Aw and Al_wi_Aw, there were ties for the top spots. Across different queries, the execution time for a particular algorithm varied and it was primarily dependent on the candidate set size for that spatial predicate. Recall, that a candidate set contains the objects that pass the filter step and are processed further in the refinement step.

4.4 NUMA policies

In the previous experiments described in sections 4.2 and 4.3, the tailored NUMA-aware PBSM-IM-NA was evaluated against four non-NUMA-aware spatial join algorithms. They were run without setting any NUMA memory placement policies explicitly. In this section we conduct experiments in which we specify NUMA memory placement policies for the non-NUMA-aware approaches with `numactl` command. We specify the following NUMA memory placement policies: *Interleave* (`--interleave=all`), *Localalloc* (`--localalloc`) and *Preferred* (`--preferred=0`). The *Default* policy means we do not provide any policy specification with `numactl`. We do not evaluate *Bind* policy, as it may slow down application performance. Results are presented for four queries, in which we vary the NUMA memory placement policies for BNLJ, BINLJ, TWNLJ and PBSM-IM. Alongside, we compare the results observed for PBSM-IM-NA. All queries were run with 8 threads, in an attempt to limit the intra-node interactions.

We present the results of the four queries in increasing order of the length of the average execution time, with Al_ov_Aw being the fastest query, followed by Pl_wi_Aw, and then Aw_ov_Aw. The Ed_cr_Al was the longest running query. The overall trend of BNLJ being the worst performing algorithm, regardless of the NUMA policy, holds in all cases, with BNLJ - *Preferred* combination showing the worst execution time overall. Similarly, BINLJ was the second worst in terms of execution time, regardless of the NUMA policy. Next, we discuss individual query results.

The results corresponding to query Al_ov_Aw are shown in Figure 8. TWNLJ performed the best, with three different NUMA policies tied for the top spot. Figure 9 shows the results for the query Pl_wi_Aw. In this case, there was no

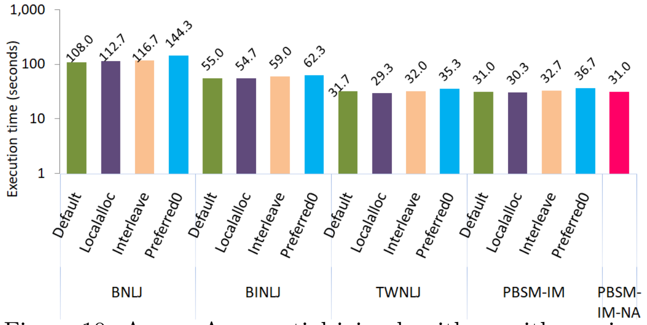

Figure 10: Aw_ov_Aw: spatial join algorithms with varying NUMA policies

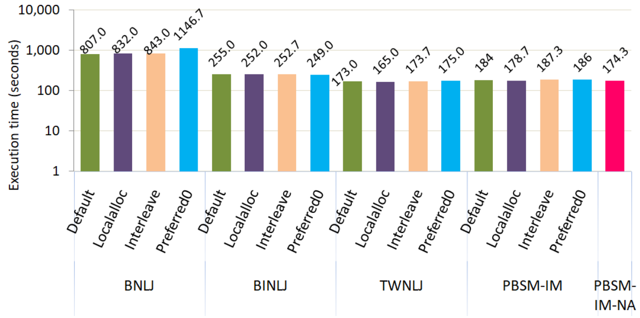

Figure 12: Ed_cr_Al: spatial join algorithms with varying NUMA policies

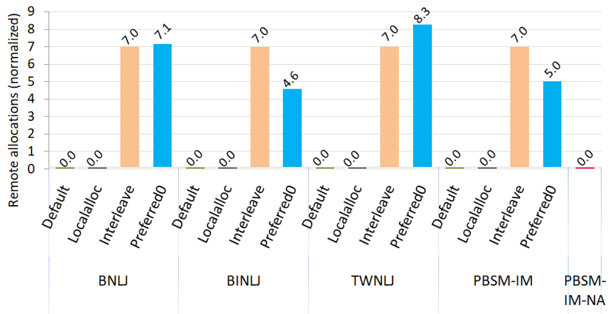

Figure 11: Aw_ov_Aw: remote allocations (normalized)

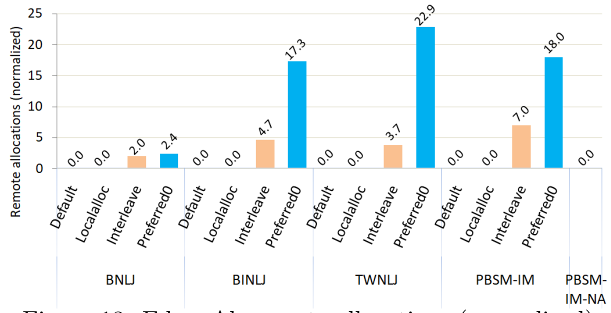

Figure 13: Ed_cr_Al: remote allocations (normalized)

clear winner, with several NUMA policies corresponding to TWNLJ, PBSM-IM and PBSM-IM-NA were tied. In Figure 10, the results for Aw_ov_Aw are presented. TWNLJ - *Localalloc* had the best result, with PBSM-IM - *Localalloc* coming as the second best. Finally, the results of the long running query Ed_cr_Al are in Figure 12. Here, TWNLJ - *Localalloc* was also the winner, with TWNLJ - *Default* and TWNLJ - *Interleave* taking the second and third best spots.

5. DISCUSSION

From the results that we have presented, we observe that the tailored NUMA-aware approach PBSM-IM-NA was quite competitive, however, it was not the best performing algorithm. In general, TWNLJ performed the best, particularly when combined with *Localalloc* policy. The *Interleave* policy did not perform as well with in-memory spatial join. This is somewhat surprising, because previously researchers [8] have demonstrated that *Interleave* was the best performing memory placement policy for regular (non-spatial) queries. To better understand the results, we use the *numastat* tool [6], which provides several node memory allocation related statistics. If the resulting page requested by a process is located on the same node as the process, *local_node* counter is incremented; whereas if the page is located on a different node than the process, *other_node* counter is incremented. We calculate *remote allocations (normalized)* statistic by dividing the difference of the *other_node* by the difference of the *local_node*. The difference of a counter, for example *local_node*, is calculated by subtracting the value reported before (the execution of a query) from the value reported after. Figures 11 and 13 show the *remote allocations (normalized)* for the queries Aw_ov_Aw and Ed_cr_Al respectively. As can be seen, in both figures the values of the statistic are positive for *Interleave* and *Preferred*, which is expected. However, for the other cases, this is very small or 0. In all of the spatial join algorithms that we evaluated, some form of static partitioning is utilized where each partition gets mapped to a thread, which in turn may operate on a NUMA node. Even, in the case of BNLJ, one of the datasets is range partitioned and each partition is processed by a thread. The static data partitioning tends to work favorably with local allocation policies, such as *Localalloc*. The *remote allocations (normalized)* statistic does not completely explain the contribution of the NUMA policies on the query execution times. The *numastat* tool does not provide sufficient information to determine the actual latency incurred due to page allocation and inter-node data transfer. A future work is to investigate this further by utilizing other profiling tools.

Spatial join queries tend to be significantly more computation intensive and involve more data skew than regular join (non-spatial) queries. Therefore, techniques that work well for regular join queries may need to be revisited for spatial join queries, particularly in the context of NUMA systems. Our tailored NUMA-aware algorithm PBSM-IM-NA extends PBSM-IM, which performs a static assignment of spatial declustering. This does not take into account the dynamic load conditions. Furthermore, although PBSM-IM addresses data skew to some degree, more could be done to improve skew-resilience in order to reduce latency, such as, employ dynamic load balancing strategies [17]. However, local NUMA allocation policies will not work very well with such an approach. Hence, a future research direction is to develop a NUMA-aware spatial join algorithm that takes dynamic load conditions into account and handles skew better. In our present study, we utilized a global in-memory column-store table to store the spatial objects corresponding to the data tables. This kind of data store may entail many remote accesses to retrieve the spatial objects from the table. An adaptive data placement capable spatial table may address this issue better. An example of such data placement is the approach [14] in which a table may be moved or re-partitioned across sockets when a load imbalance at the socket level is detected. A future research direction is to investigate how to develop such an adaptive data store for spatial objects, which can better handle NUMA-aware spatial query execution.

6. RELATED WORK

In this section, we describe previous work related to our research. First we describe work related to spatial join and then we outline existing research on in-memory spatial join.

6.1 Spatial Join

The simplest way to execute a spatial join is *nested loop join* that loops through all records in one input table for each record in the other table. A common technique to improve this is to employ a spatial index like R-tree on one or both inputs in the filter step. Such algorithms that use spatial indexes are *index nested loop joins*.

There are times, however, when using an index construction is not possible. For example, if both inputs are intermediate results from a query or when inputs are dynamically redistributed in a parallel environment [12]. In such cases, a partitioning technique such as the Partition Based Spatial-Merge (PBSM) algorithm is a suitable algorithm to use. PBSM works by partitioning data in the filter step into smaller chunks and then it performs a join on each pairwise partition using a plane-sweeping technique. PBSM decomposes the spatial domain into equal-sized tiles and maps each tile to a partition in a round-robin or hashing fashion. The filter step starts by reading tuples $< MBR, OID >$ from each input and appending them to a temporary relation on disk. All key-elements are read from disk for each partition at a time followed by performing an MBR-join. A set of pairs of OIDs from each input are returned. The candidate set is then sorted to remove duplicates and is fetched sequentially from disk for further investigation.

Parallelism is commonly employed when dealing with large volumes of data. One way to achieve parallelism is by partitioning data across multiple nodes in a cluster, allowing work to be done concurrently at each node. Two partitioning algorithms that extend PBSM are Clone Join and Shadow Join, and they use two different declustering strategies to realize parallelism [13]. In Clone join, objects that overlap multiple tiles are replicated at each node. This declustering technique is referred to as D-W ("decluster using whole tuple replication"). Shadow join stores the entire tuple at one node and replicates only part of the MBR that is present in that tile and the OID to all other nodes. The partial replicate of the MBR is called a *fragment box*. This technique is referred to as "partial spatial surrogate" (D-PSS). Both algorithms employ PBSM in the join phase.

6.2 In-memory Spatial Join

With the growing memory capacity in modern machines, in-memory spatial join has received a lot of attention. The previously described approaches (PBSM, Shadow join, and Clone join) are disk-based and their in-memory implementations are not always the most efficient. Nobari et al. [9] proposed an in-memory spatial join approach called TOUCH that uses hierarchical data-oriented space partitioning approach. TOUCH generally performed better than other in-memory spatial join implementations. However, in some cases in-memory PBSM outperformed TOUCH, for example when the data set is clustered or when the join order is switched.

Partitioning-based algorithms, such as Clone Join, Shadow Join and PBSM Join use static work assignment, which consists of distributing work in a round-robin fashion. In a static assignment workload imbalances can arise, since objects may vary in size and point density. In contrast, SPINOJA [17] uses dynamic work assignment and a job scheduler to support better load-balancing. Adaptive Partitioning (ADP) [19] is another load-balanced spatial join that partitions work based on quadtree declustering and employs its own duplication avoidance strategy. None of these in-memory spatial join approaches addressed NUMA-aware processing.

7. CONCLUSIONS

The performance of spatial join continues to be an important factor in spatial analytics applications, particularly in the modern hardware landscape. NUMA architecture is poised to become a norm and it is important to re-examine spatial join in the context of NUMA.

In this paper, we investigated in-memory spatial join performance on NUMA systems. We implemented five different parallel in-memory spatial join algorithms that include a tailored NUMA-aware algorithm. We conducted an extensive experimental study involving four real-world spatial data tables, with seven different spatial join queries that were run with these five spatial join algorithms. We also evaluated several NUMA memory placement policies. Our results reveal some interesting findings that, we hope, will guide researchers in developing efficient spatial analytics solutions.

8. ACKNOWLEDGEMENTS

This research was supported in part by Natural Sciences and Engineering Research (NSERC) Discovery Grant RGPIN-2016-03787 and NBIF Start-Up Grant (S. Ray). V. Anupindi was involved in the project while at UNB.

9. REFERENCES

[1] TIGER/Line and TIGER-Related Products. [Online; accessed 2020].

[2] Numa policy library, 2020. [Online; accessed 2020].

[3] J. Albrecht. *Key Concepts and Techniques in GIS*. 2007.

[4] E. Clementini and P. D. Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, 90(1):121 – 136, 1996.

[5] GEOS - Geometry Engine, Open Source. https://trac.osgeo.org/geos/.

[6] A. Kleen. A NUMA API for Linux. *Technical Linux Whitepaper, Novel Inc*, 2005.

[7] S. T. Leutenegger, J. M. Edgington, and M. A. López. STR: a simple and efficient algorithm for R-tree packing. *International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

[8] P. Memarzia, S. Ray, and V. C. Bhavsar. The Art of Efficient In-memory Query Processing on NUMA Systems: a Systematic Approach. In *2020 IEEE 36th*

International Conference on Data Engineering (ICDE), pages 781–792, 04 2020.

[9] S. Nobari, Q. Qu, and C. S. Jensen. In-memory spatial join: The data matters! In *EDBT*, pages 462–465, 2017.

[10] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[11] Open Geospatial Consortium. Simple Feature Access - Part 2: SQL Option. http://www.opengeospatial.org/standards/sfs.

[12] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[13] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *SIGSPATIAL*, pages 54–61, 2000.

[14] 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. *Proc. VLDB Endow.*, 10(2):37–48, Oct. 2016.

[15] S. Ray. *High Performance Spatial and Spatio-temporal Data Processing*. PhD thesis, University of Toronto, 2015.

[16] S. Ray, B. Simion, and A. D. Brown. Jackpine: A benchmark to evaluate spatial database performance. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1139–1150, 2011.

[17] S. Ray, B. Simion, A. D. Brown, and R. Johnson. Skew-resistant parallel in-memory spatial join. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2014.

[18] D. Tsitsigkos, P. Bouros, N. Mamoulis, and M. Terrovitis. Parallel in-memory evaluation of spatial joins. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 516–519, 2019.

[19] J. Yang and S. Puri. Efficient parallel and adaptive partitioning for load-balancing in spatial join. *34th IEEE International Parallel Distributed Processing Symposium*, May 2020.