Fine-grained Dynamic Load Balancing in Spatial Join by Work Stealing on Distributed Memory

JIE YANG AND SATISH PURI, Department of Computer Science
Marquette University, USA
HUI ZHOU, Mathematics and Computer Science Dept.
Argonne National Laboratory, USA, USA

Spatial join is an important operation for combining spatial data. Parallelization is essential for improving spatial join performance. However, load imbalance due to data skew limits the scalability of parallel spatial join. There are many work sharing techniques to address this problem in a parallel environment. One of the techniques is to use data and space partitioning and then scheduling the partitions among threads/processes with the goal of minimizing workload differences across threads/processes. However, load imbalance still exists due to differences in join costs of different pairs of input geometries in the partitions.

For the load imbalance problem, we have designed a work stealing spatial join system (WSSJ-DM) on a distributed memory environment. Work stealing is an approach for dynamic load balancing in which an idle processor steals computational tasks from other processors [5]. This is the first work that uses work stealing concept (instead of work sharing) to parallelize spatial join computation on a large compute cluster. We have evaluated the scalability of the system on shared and distributed memory. Our experimental evaluation shows that work stealing is an effective strategy. We compared WSSJ-DM with work sharing implementations of spatial join on a high performance computing environment using partitioned and un-partitioned datasets. Static and dynamic load balancing approaches were used for comparison. We study the effect of memory affinity in work stealing operations involved in spatial join on a multi-core processor.

WSSJ-DM performed spatial join using *ST_Intersection* on *Lakes* (8.4M polygons) and *Parks* (10M polygons) in 30 seconds using 35 compute nodes on a cluster (1260 CPU cores). A work sharing Master-Worker implementation took 160 seconds in contrast.

CCS Concepts: • **Information systems → Geographic information systems**; • **Computing methodologies → Parallel algorithms**.

Additional Key Words and Phrases: High Performance Computing, Distributed Computing, NUMA, MPI

ACM Reference Format:
Jie Yang and Satish Puri and Hui Zhou. 2022. Fine-grained Dynamic Load Balancing in Spatial Join by Work Stealing on Distributed Memory. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA*. ACM, Seattle, Washington, USA, 19 pages. https://doi.org/10.1145/3557915.3560936

1 INTRODUCTION

In Geographic Information Systems (GIS) and spatial databases, two datasets are combined based on some spatial relationship among geometries in the input datasets. For instance, given two sets of polygons, R and S, find all of the pairs of overlapping polygons between the two sets, that is, for each polygon r in dataset R, find overlapping polygons from dataset S [22].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGSPATIAL '22, November 1–4, 2022, Seattle, WA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9529-8/22/11.
https://doi.org/10.1145/3557915.3560936

High performance computing (HPC) clusters and supercomputers with GPUs are used to analyze geospatial data [11, 15–17, 22]. CyberInfrastructure centers like Polar Geospatial Center, CyberGIS project at National Center for Supercomputing Applications (NCSA), and WiFire project (forest fire) at San Diego Supercomputer Center are involved in running geospatial computations and simulations in a large-scale HPC environment. The distributed memory programming model in these environments is Message Passing Interface (MPI) and the storage layer is a parallel file system like Lustre. MPI is preferred over MapReduce due to fast communication on high performance interconnection network and more flexibility in terms of programming. This is different architecture when compared to MapReduce model and its distributed file system based storage. We have used an HPC compute cluster in our load balancing design and evaluation. MPI-GIS software utilizes message passing interface [17], parallel I/O and partitioning capabilities [16, 22] and GPU acceleration modules [11, 12] for high performance spatial join and map overlay on HPC clusters. The present work is a dynamic load balancing component in MPI-GIS software stack.

The overall load imbalance in spatial join is determined by two factors - 1) size and distribution of geometries in the two input maps that need to be joined together by a process (thread) and 2) number of outputs produced per process (thread). The output-sensitive nature makes load balancing difficult because the number of outputs is not known a priori and can not be estimated easily for complex geometries where approximations result in a large number of false hits [19]. Therefore, input data and intermediate output data partitioning techniques are used to minimize variation of load across partitions [16, 22].

Load balancing approaches can be classified into two categories: 1) work sharing and 2) work stealing. In work sharing approach, a busy processor with excess work sends them to idle processor with less or no work (e.g., master-worker pattern). However, in work stealing approach, an idle thread initiates task migration from the work queue of a busy thread with pending tasks. When a processor finishes all the tasks in its queue, it becomes a thief and tries to steal a task from another processor (victim). This difference can be stated as push (work sharing) vs pull (work stealing) techniques.

Work stealing technique has some advantages. First, it has been shown to improve data locality [1]. In work sharing, a busy processor incurs context switch overhead while sending work to remote processor. Second, idle processors are mostly involved in the work distribution (overhead); busy processors continue spatial join processing. Moreover, when (until) all processors are busy, no load balancing overhead occurs [5].

Dynamically load balancing on a distributed memory system is challenging because load balancing requires serialization and communication of complex geometries by a busy sending process, and deserialization (parsing) of geometries at an idle receiving process. This is a significant computation and communication overhead for large geometries. In a distributed setting, work stealing can be a significant overhead if the execution time of the tasks are in second or millisecond range. However, this overhead is not present in a shared memory queue based implementation [19]. Another challenge is effective flow control among processes participating in pull-based task sharing in spatial join.

Our flow control using MPI Remote Memory Access (RMA) guides the granularity and timing of task sharing to keep the idle processes busy and while minimizing the overheads at busy processes. Our new design is able to leverage multiple compute nodes efficiently to speedup parallel spatial join, in the presence of serialization and work coordination overheads. From a performance perspective, this is an improvement over shared memory spatial join [19] and distributed memory MPI-based spatial join systems [16, 17, 22]. To the best of our knowledge, this is the first demonstration of effectiveness of work stealing on a large scale distributed memory machine with thousand processor cores.

We present the effect of memory affinity in work stealing operations involved in spatial join on a NUMA system. Our results complement existing line of work on spatial join [14, 18].

Contributions of our paper are as follows:

- We present a novel NUMA-optimized Work Stealing Spatial Join system (WSSJ) on shared memory. We extended WSSJ to distributed memory (WSSJ-DM) environment. Source code is publicly available. [1]
- We demonstrate effective mitigation of data skew in a fine-grained manner to avoid stragglers (threads taking much longer than others to finish). Both WSSJ and WSSJ-DM are experimentally shown to be load balanced and efficient.
- Both WSSJ and WSSJ-DM can perform a variety of spatial relationship joins and spatial overlay joins. Our system can effectively handle data skew in spatially partitioned and un-partitioned datasets.

This paper is organized as follows. Section 2 introduces background and related work. Section 3 describes WSSJ on shared memory and WSSJ-DM on distributed memory. Section 4 evaluates the performance of WSSJ and WSSJ-DM. We conclude this paper in Section 5. Appendix section presents experimental evaluation using un-partitioned datasets.

2 BACKGROUND AND RELATED WORK

2.1 Spatial Join

Spatial join involves two spatial datasets R and S. The output contains all pairs of objects satisfying a given relation between the objects. The spatial relationships [2] such as *ST_Within*, *ST_Intersects*, etc are supported. We have also handled overlay computation[3], such as *ST_Intersection*, and *ST_Union*. *ST_Intersects* is used to answer a query - Is there any overlap between the two geometries?

2.2 Load Balancing in Parallel Spatial Join

An existing approach in partition-based spatial join (PBSM) is to create a certain number of grid cells and assign the cells to processors [3, 16, 17]. Some approaches use static round-robin assignment [16, 17] and others use dynamic load balancing [19]. The unit for load balancing in this approach is a set of geometries in a grid cell. Our technique for load balancing is fine-grained because our tasks are at individual geometry level compared to existing approaches that work at grid cell level. Therefore, our task construction enables fine grained load balancing.

Partitioning of map layers into tiles (grid cells) has been used in [22]. The tiles are then assigned to processors in a round-robin fashion. Declustering is proposed as a load balancing strategy in [20]. [21] uses bitmaps to determine the number of spatial objects to perform dynamic load balancing. SPINOJA [19] uses object decomposition based declustering to mitigate data processing skew on shared memory. MapReduce-based spatial join systems first create data partitions using various partitioning techniques and then use dynamic load balancing supported by MapReduce implementations like Hadoop and Spark to join grid partitions [4, 8, 23]. Current message passing based systems do not support work stealing, for example, MPI-based spatial join systems like MPI-GIS and ParADP [2, 16, 22].

2.3 Work Stealing

Work stealing is a dynamic load balancing strategy [5–7, 10, 13]. Work stealing has been used in shared memory and distributed memory [7] load balancing solutions.

[1]https://github.com/satishphd/WorkStealing-Spatial-Join

[2]https://postgis.net/docs/reference.html#Spatial_Relationships

[3]https://postgis.net/docs/reference.html#Overlay_Functions

Chase-Lev's lock-free deque [6] is an important data structure in many shared-memory work stealing designs. The deque uses a *dynamic-cyclic-array*, which allows: 1) the owner to push and pop elements from the top of the deque, 2) others to perform concurrent lock-free steal from the bottom of the deque. Nhat's Work Stealing Queue [10] implementation in C++11 is based on Chase-Lev's lock-free deque and shows a remarkable performance in benchmarks. We use it in our work stealing implementations. For simplicity, we have referred to Work Stealing Queue as queue or deque (double-ended). There are very few work stealing libraries that work on shared nothing architecture - for instance, Charm++. Our implementation WSSJ-DM is geared towards spatial computing workloads that will be integrated into an MPI-based GIS ecosystem. Charm++ is a different programming model compared to message passing framework that MPI-GIS is based upon.

2.4 Non-Uniform Memory Access (NUMA)

In non-uniform memory access, processor cores have access to local memory and remote memory. Remote memory access is costly compared to local access. There has been some earlier work on NUMA-aware algorithms. [18] discusses an experimental study on enabling NUMA-aware main memory spatial join processing. [14] discusses a systematic approach for efficient in-memory query processing on NUMA systems.

Many NUMA policies can be used on current Linux systems. *MPOL_DEFAULT*, *MPOL_INTERLEAVE*, *MPOL _PREFERRED*, and *MPOL_BIND* are typically available.[4] These policies can be set by calling a system function *set_mempolicy*. Our findings on NUMA policies are novel.

3 IMPLEMENTATION OF WORK STEALING SPATIAL JOIN

3.1 Work Stealing Queue

A simple work stealing system for spatial join on shared-memory consists of the following steps:

(1) Create one thread for each processor core and each thread uses a queue to hold tasks to be scheduled.
(2) Each thread pushes its tasks to its own queue from the bottom. And then pops and executes tasks from the queue.
(3) A thread can steal tasks from the top of other threads' queues after all tasks in its own queue are finished.

Based on these steps, we built a work stealing system for spatial join on shared memory (WSSJ). In WSSJ, there are multiple worker threads and each worker thread holds its own queue.

A worker thread generates spatial join refinement tasks after the filter phase and pushes these tasks into its queue. It can pop tasks from its own queue and steal tasks from a victim's queue. The victim can be chosen randomly. In WSSJ, each worker performs the filtering phase and refinement phase independently.

3.2 NUMA Memory Policies

The execution of spatial join computations are impacted by NUMA memory policies. Spatial join algorithms allocate a temporary buffer to carry out intermediate steps of join algorithm on two geometries. The spatial objects are copied to the temporary buffer to carry out Quadtree partitioning of an individual geometry, to order the coordinates, and to populate the intersection matrix.

The default NUMA policy on most Linux systems after boot-up is *MPOL_DEFAULT*, which is "local allocation". Under this policy, Linux will attempt to satisfy memory requests from the nearest NUMA node of the CPU which submits the memory requests. *MPOL_DEFAULT* works fine in many

[4]https://linux.die.net/man/2/mbind

scenarios. However, in terms of work stealing, a thread on one NUMA node can steal a task from another NUMA node. A page in memory corresponding to geometry data structure can be accessed by multiple threads because of many-to-many overlap relationship in spatial join tasks. For spatial join, in all experiments we have conducted so far, the tasks on a few worker threads (usually 1 to 4) take much longer to finish than the rest of the threads. When multiple threads allocate and write to temporary buffers for tasks from remote NUMA nodes, there can be memory requests congestion.

MPOL_BIND and *MPOL_PREFERRED* can mitigate the memory requests congestion issue. Under these two policies, the temporary buffers are on the same NUMA node as the pairs of geometries to be joined. The issue with *MPOL_BIND* is that it is a strict policy; the OS can only utilize the memory on specified NUMA node(s). This can be a problem in case there is more memory required than available on a single NUMA node.

Under *MPOL_INTERLEAVE* mode, the memory allocations are uniformly distributed among all NUMA nodes. The temporary buffers are allocated in an interleaved manner as the pairs of geometries are joined.

The NUMA effects discussed here are due to work stealing inherent in parallel spatial join with higher number of threads. We compared the impacts of different memory policies in Section 4.1.

3.3 Work Stealing Algorithm

Now we present details on how to apply work stealing idea using filter and refine based spatial join. Algorithm 1 describes task generation by a worker thread. These tasks are added to work stealing queue data structure maintained per thread. R and S stand for two spatial datasets to be joined. WSSJ uses spatially partitioned datasets in this algorithm based on our earlier work [22]. For instance, spatial partitioning of R and S into 4 partitions will result in grid cells R1 to R4 and S1 to S4. This creates 4 join tasks, (R1, S1), (R2, S2), (R3, S3), and (R4, S4). Each thread is assigned one or more partition(s) as input. $queues[T]$ are instances of work stealing queue, where T equals the number of worker threads.

Task Construction: A spatial join task consists of a subset of geometries from R and S that spatially overlap using minimum bounding rectangle (MBR) overlap test. We chose one geometry from R and multiple geometries from S as a unit task in our system. For overlap detection using MBR approximation of geometries, we use a search tree (index) for MBR query. Filter phase is done using the standard R-tree index-based nested loop spatial join approach. Assuming, a join on partition pair (R1, S1), where R1 = $\{r_0, r_1, .., r_m\}$ and S1 = $\{s_0, s_1, .., s_n\}$, a unit task is a key-value pair, where key is r_i, $i \in \{0, m\}$ and value is an arbitrary subset from S1.

The *Break_Down_Task()* function splits a large task into a set of smaller tasks by breaking down the value part of the task. We set a $Threshold_{task}$ as the size limit of a task. This step is necessary as a huge geometry r usually returns a large query result in Line 10 of Algorithm 1, which is one reason of load imbalance. Assuming $Threshold_{task}$ to be 10, if the MBR query result returns 30 geometries from S, then this single task will be broken down into 3 sub-tasks. This step makes sure that individual tasks are relatively of same computational cost.

In WSSJ, each thread occupies one CPU core and Algorithm 1 is executed per thread independent of other threads. Work stealing queues store pointers to tasks.

As presented in Algorithm 2, a worker thread t_i first pops tasks from its own queue $queues[i]$, and performs join operations until its queue becomes empty. Then it finds a victim thread. *Get_Victim()* function returns the next available victim's thread id. Thread ids are checked in cyclic order to get a thread with enough pending work, beginning with the current thread's id + 1. This method is simple and robust. Other methods like choosing random thread as victim or choosing threads based on NUMA consideration and data locality are also possible. The thief thread will keep stealing and performing join operations until the victim's queue becomes empty. The granularity of tasks stolen

Algorithm 1 Algorithm for Pushing Tasks into Queues

```
1:  Input: Subsets of spatial objects from R and S.
2:  T is number of threads.
3:  Output: Queue queues[T] populated with tasks.
4:  Assign NUMA memory policy.
5:  Initialize all the queues in queues[T].
6:  for Thread t_i in Threads do
7:      Build an index Index_i using MBRs of R
8:      for Object s_j in S do
9:          Task tasks ← Index_i.query(s_j)
10:         Task subTasks ← Break_Down_Task(tasks)
11:         queues[t_i].push(sub_tasks)
12:     end for
13: end for
14:
```

Algorithm 2 Algorithm for Work Stealing Spatial Join

```
1:  Input: Queue queues[T] populated with tasks.
2:  Output: Spatial Join results.
3:  for Thread t_i in Threads do
4:      while queues[t_i] not empty do
5:          Task task ← queues[t_i].pop()
6:          results_i ← Spatial_Join_OP(task)
7:      end while
8:      while Not all queues empty do
9:          int victim ← Get_Victim()
10:         while queues[victim] not empty do
11:             Task task ← queues[victim].steal()
12:             results_i ← Spatial_Join_OP(task)
13:         end while
14:     end while
15: end for
```

can be configured. All join results generated by t_i are pushed into $results_i$. *OP* stands for type of spatial join operation.

3.4 Overall Framework on Distributed Memory

In our multi-compute node architecture with distributed memory (WSSJ-DM) design, each compute node still uses the shared memory work stealing system WSSJ, plus one coordinator thread. The coordinators are used to communicate with other compute nodes and shuffle tasks, as shown in Figure 1. The tasks (including coordinates of geometries) are serialized by the sending process and deserialized by the receiving process. The task contains the geometry coordinates in the message itself. So, the overall communication of geometries corresponding to the stolen tasks happens in-memory. When needed, a coordinator spawns multiple threads to speedup the send/receive and parsing of geometries (from the message buffer) among compute nodes for load balancing purpose.

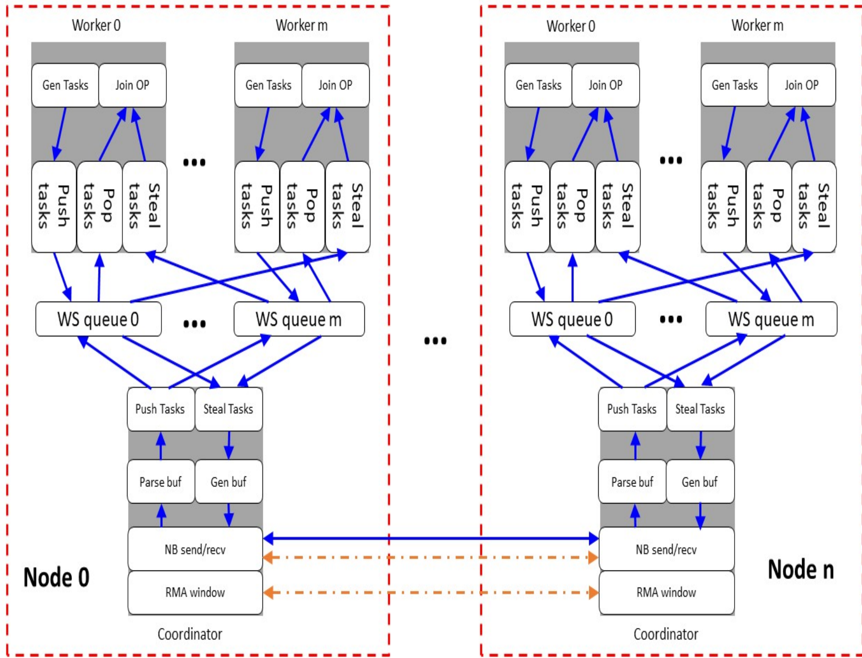


Fig. 1. The Work Stealing Spatial Join system on distributed memory (WSSJ-DM). Two multi-core compute nodes are shown with m threads. “Gen Tasks" represents task generation and OP is join operation. The solid blue arrows show the directions of the flows of spatial join tasks. The dashed orange arrows show the directions of the flows of control messages. “NB send/recv" stands for “Non-blocking send or receive message passing". “Gen buf" stands for "Generate send buffer" and “Parse received buffer".

In Figure 1, each dashed red rectangle stands for a WSSJ-DM node. There are multiple compute nodes. Each node has multiple worker threads to leverage the multiple CPU cores, and one coordinator. Each worker thread has one work stealing deque. The worker threads and deques are same as in WSSJ.

In the beginning, each node takes grid cells of R and S in a Round Robin manner. A worker thread follows the steps as WSSJ: parsing data files, building indices, and pushing tasks into their own work stealing queues. Additionally, after all local tasks are finished, the worker threads in WSSJ-DM wait for tasks from their coordinator, which "steal" tasks from other coordinators. The coordinators are responsible for termination detection. A coordinator also informs its worker threads that all tasks across all nodes are done.

To illustrate how the system works, we present a typical scenario with two coordinators running on two compute nodes. After all the tasks get pushed, a coordinator (say *Coord A*) thread begins to monitor its memory window. If A’s all local queues are empty, *Coord A* begins to seek tasks from other nodes by writing to the memory window of other coordinators using the remote memory access (RMA) functions. Another node (say *Coord B*) notices the change in its memory window because of A’s recent action. *Coord B* resets its window as an acknowledgement (to allow new starving coordinator) and begins to steal tasks from local queues and then sends those tasks to *Coord A*. This is accomplished through non-blocking message passing.

The worker threads mainly focus on performing join operations, and behave similarly to worker threads in WSSJ. In the next two sub-sections, we will discuss about the core module of WSSJ-DM, the coordinators.

3.5 Coordinator in Send Status

The coordinators are threads within a WSSJ-DM node meant to facilitate communication with other nodes. A coordinator can be in two status based on the number of all tasks in local work stealing queues: 1) send status and 2) receive status.

A coordinator (*Coord A*) maintains a memory window, initially as waiting for task requests. It waits until all local spatial join tasks are enqueued. It then steps into the **Send Status**. *Coord A* checks its memory window periodically. If no change is found, it will update the window with its current remaining tasks and then goes on to sleep until next period to save CPU cycles for the worker threads. If there is information that other coordinators are looking for tasks, *Coord A* will mark those coordinators as starving. It then begins to steal tasks from local queues and send those tasks to other coordinators.

The vertices of geometries corresponding to a task are converted to basic data type arrays to be used by message passing functions. *Coord A* uses non-blocking send function to send those task arrays. It will keep sending tasks to starving coordinators until all local tasks are done or almost done. *Coord A* can fork multiple threads to accelerate the task sending process.

While sending tasks, *Coord A* still checks and updates its memory window periodically, to signal its current status to other starving processes. *Coord A* also uses non-blocking receive function to gather status signals from other coordinators. If it finds that some coordinators have received too much work to finish on time, it sends a temporary stop signal to those coordinators and stops sending tasks to them. When all local tasks are done, *Coord A* sends a stop signal to all starving coordinators in its record.

3.6 Coordinator in Receive Status

After all local tasks are done, *Coord A* enters the **Receive Status**. *Coord A* checks the remote memory windows of other coordinators. If a window indicates that all its owner's tasks were finished, *Coord A* records this information and checks the next available remote memory window. Among all the other coordinators, it will ask for tasks from the one with the most tasks left (say *Coord B*). If a window is written by other starving coordinators, *Coord A* will skip this window.

In case when its task request is put on *Coord B*'s window, *Coord A* will use non-blocking receive function and wait for tasks to arrive. When the data is received, *Coord A* parses the received data to spatial join tasks and pushes those tasks to an empty queue. This task receiving-parsing-pushing progress can be accelerated by using multiple receive threads. After that, *Coord A* marks the queue to allow workers to steal.

Coord A sends the number of its local tasks to *Coord B* after a few invocations of receiving function, also using non-blocking send. *Coord B* uses this number to judge if *Coord A* needs a temporary stop, i.e., *Coord A* has received too many tasks but its worker threads are processing tasks slowly. If *Coord A* receives a temporary stop, it will be on sleep until most received tasks are done by its worker threads. After waiting, it will again seek another coordinator which still has tasks. If *Coord A* receives a stop sign, it will mark *Coord B* as "All Tasks Finished" and seek another coordinator for more tasks. When *Coord A* finds that all other coordinators have no remaining task, it will inform all its worker threads and terminate itself.

3.7 Inter-node Communication

Our distributed memory system uses non-blocking communication functions and remote memory access functions supported by Message Passing Interface standard. Appendix section contains some detail on this topic. The most important feature of non-blocking send and receive in WSSJ-DM is that it allows overlapping communication and computation.

The coordinators in WSSJ-DM use multiple threads to perform *MPI_Isend()* to send spatial join tasks and *MPI_Irecv()* to receive tasks. These *send/receive* threads can perform all send and receive operations concurrently, and then go to sleep. These threads wake up periodically to check if their send and receive operations have finished. Thus, for the most part, send/receive threads are on sleep and yield the CPUs to the worker threads to perform compute-intensive join operations.

Remote Memory Access (RMA) allows access to remote memory. By using the feature, a coordinator in WSSJ-DM nodes can show the node's status in its memory window. It can tell others if current node: 1) has spatial join tasks and the number of tasks, or 2) has no tasks, or 3) is hand shaking with another node. A coordinator can also write a request to another coordinator's window based on the information on that window, and wait for instructions for moving tasks and associated geometry data.

3.8 Theoretical Analysis

We analyze the theoretical performance of WSSJ-DM in this subsection. The benefit to be gained by WSSJ-DM depends on the computational complexity of spatial join operations because there is a tradeoff between doing work locally vs sending the work to a remote node. For instance, spatial overlay join is more compute-intensive than overlap-test based join. This difference will impact the potential speedup made possible by work stealing.

A model is developed here to study the impact of work stealing by remote compute nodes on overall execution time. Even though multiple processes are active in parallel join processing in WSSJ-DM (some in stealing mode and others in victim mode), our model considers one such extreme scenario, to show the scalability bottlenecks because of overheads in work stealing.

Let us assume, among n nodes, only $Node_1$ has tasks which require a total computation denoted by V and the other $n-1$ nodes have no tasks. We denote the local processing rate of $Node_i$ by f_i which means number of computations executed per second corresponding to local tasks.

WSSJ-DM allows a task originally belonging to a source node S to be executed by a remote node for load balancing. This leads to the notion of remote processing rate for stolen tasks. We denote the remote processing rate of ith node by f_{iS} to finish tasks that belongs to source node $Node_S$. For instance, f_{i1} means remote processing rate of ith node for tasks belonging to $Node_1$. Remote processing rate is bounded by local processing rate. This is because of serialization, communication and coordination overheads over the network.

In WSSJ-DM, multiple compute nodes can be leveraged in parallel, so aggregate processing rate increases by using more compute nodes upto a limit. For instance, when $Node_1$ sends tasks to a new node $Node_i$, the aggregate processing rate of $Node_1$ and $Node_i$ is $f_1 + f_{i1}$, minus the processing rate penalty γ due to inefficiency of remote processing. γ is based on the average size of tasks, buffering of geometries, parsing speed, and the network bandwidth. γ denotes per node penalty. γ increases with more idle nodes requesting $Node_1$ for tasks. Formula 1 models the execution time of WSSJ-DM on tasks with computational cost denoted by V before $Node_1$ reaches its limit of communicating tasks. Time is computed by dividing number of computations by processing rate.

$$T = \frac{V}{f_1 + (\sum_2^n f_{i1}) - (n-1) * \gamma} \tag{1}$$

To explain Formula 1, Figure 2 shows the execution time on compute cluster with multiple CPUs. We assume γ is fixed here to 1. We assume: $f_1 = 100$, all $f_{i1} = f_1/3$. The range of V is $[1000, 10000]$. The range of n is $[1, 10]$. We can see that for spatial join with larger computational cost (increasing V), processing time increases. For a given V, the reduction in execution time by using additional CPUs is more significant for spatial join with higher V. For lower V, the benefit is less. WSSJ-DM will be slowed down by using more nodes after reaching its bottleneck.

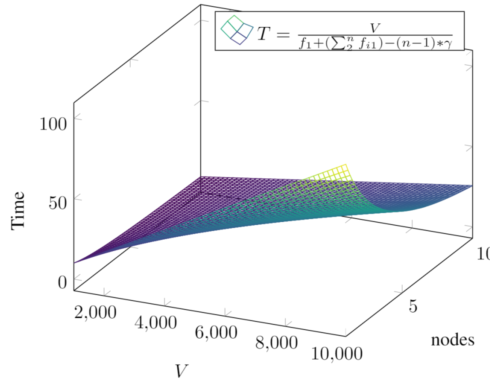


Fig. 2. Theoretical performance modeling of WSSJ-DM before reaching bottleneck.

Formula 1 considers that all idle processors steal tasks from a single node. Formula 2 generalizes the formula to include a subset of work stealing nodes to model the performance of WSSJ-DM in which $Node_1$ reaches its limit of sending tasks to m nodes, where m can be fixed and $n > m$.

$$T = \frac{V}{f_1 + (\sum_2^m f_{i1}) - (n-1) * \gamma} \tag{2}$$

4 EXPERIMENTAL RESULTS

All of our experiments used five real world datasets: *cemetery*, *sports*, *lakes*, *parks*, and *roads*, which are taken from SpatialHadoop website[5]. The datasets are stored in Well Known Text (WKT) format and the characteristics of these datasets are shown in Table 1.

Name	Type	#Geometries	File size
cemetery	Polygons	193 K	56 MB
sports	Polygons	1.8 M	590 MB
lakes	Polygons	8.4 M	9 GB
parks	Polygons	10 M	9.3 GB
Roads	Polylines	72 M	24 GB

Table 1. Attributes of the datasets

[5]http://spatialhadoop.cs.umn.edu/datasets.html

All of the experiments are done on a HPC cluster named *Bebop*[6] at Argonne National Laboratory. A regular node on *Bebop* has two Intel Xeon E5-2695v4 (36 cores per node), and 128GB DDR4 memory. We used GCC 8.2.0, C++ 17, Intel MPI 3.1, and GEOS[7] 3.9.1 in all of the following experiments.

For comparison, we used implementations based on Asynchronous Dynamic Load Balancing (ADLB) [13], Round Robin scheduling, and shared queue design. Shared queue design was used in SPINOJA [19]. ADLB is a scheduling strategy for dynamic load balancing. This library uses message passing interface, so it can use multiple CPUs in an HPC cluster. Using ADLB, tasks are added to the run-time task data structure using put operation. Using get operations, idle workers can access tasks. ADLB programming model handles the load balancing under the hood. In our ADLB implementation, some servers are in charge of generating and populating tasks for future processing and tasks are shared among the servers.

The idea of using single-master multiple-workers has been widely used in shared memory solutions, such as SPINOJA and MPI-GIS [2]. SPINOJA is only designed for shared memory. We did not have access to SPINOJA code, so we implemented a shared queue based spatial join system. We call this system SQSJ. SQSJ is a parallel spatial join system where candidate tasks are stored in a shared queue for concurrent access by available threads. Our shared queue design was motivated by SPINOJA. Compared to SQSJ, WSSJ uses multiple queues per compute node. To provide a fair comparison, we extended shared queue design to distributed memory using the same distributed framework of WSSJ-DM. We refer to the distributed memory version of SQSJ as SQSJ-DM.

Round Robin scheduling is a widely used technique where each core/node takes parts of partitioned R and S in a cyclic manner, and the cores/nodes finish its work independently [16, 17, 22]. Dense areas get distributed among processors due to Round Robin assignment. Because of static partitioning, the overheads are minimal in this scheduling strategy.

In the following experiments, the value of $Threshold_{task}$ is set to 20. The number of send/receive threads is set to 5 and the number of tasks per send/receive is set to 100.

4.1 Impact of NUMA Policies on WSSJ

Based on Section 3.2, we designed comparison experiments among different NUMA memory policies on WSSJ. The pair of datasets being used is *Sports* and *Cemetery*, which was partitioned using Quadtree into 8192 grid cells. The reason to use *Sports* and *Cemetery* is that both datasets are small and most of the geometries are small in these datasets. *ST_Intersects* is one of the lightest spatial join operations.

In the experiments, we controlled the sizes of R and S by duplicating the original datasets. The duplication coefficient D means that R and S contain n copies of *Sports* and *Cemetery* respectively.

The regular nodes on *Bebops* only have two NUMA nodes, 0 and 1. The policies settings are: *I. MPOL_INTERLEAVE*, node 0 and 1; *II. MPOL_BIND*, node 0; *III. MPOL_PREFERRED*, node 0; *IV. MPOL_DEFAULT*. In all of the tests, threads were evenly distributed on two NUMA nodes. When multiple threads are launched, the core affinity of threads is managed by the OS.

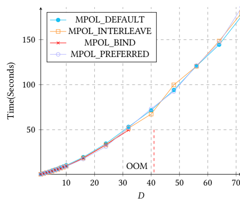
The results are shown in Figure 3. We can see that different policies do not have much difference in spatial join execution time with 4 threads in Figure 3a. With more threads, in Figure 3b, it takes longer using *MPOL_DEFAULT* than the other three policies. As mentioned earlier, more threads may lead to higher memory request congestion between the NUMA domains. In our experiments, this performance difference due to memory policy is noticeable for datasets with small geometries. For datasets with large geometries, the difference is very small.

[6]https://www.lcrc.anl.gov/systems/resources/bebop/
[7]https://trac.osgeo.org/geos

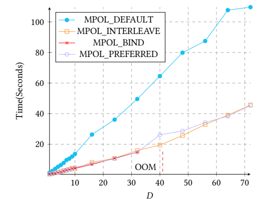
Local allocation policy is the default memory policy. This policy can not necessarily guarantee that all accesses will be local to the NUMA node because it is possible that a page is allocated by one thread, but can be accessed by other work stealing threads. The first thread to touch/write to a memory page will determine its location in terms of the NUMA node. So, first touch policy may violate local NUMA node allocation when a geometry is stolen by a thread on remote NUMA node. In this case, a thread allocated space for a geometry, however, it was accessed (written) by a thread on a remote NUMA node. The default policy gets negatively impacted by resource contention when compared to other policies.

Interleave memory placement works well in WSSJ because thread access pattern is irregular and random due to work stealing among threads. Interleave policy benefits from the load-balancing of memory access requests across available NUMA nodes, even though memory access time is not uniform.

Because *MPOL_BIND* only use one NUMA node, it runs out of memory at *D=40* while others run out of memory at *D=80*. In most cases, using *MPOL_PREFERRED* shows a similar performance with using *MPOL_INTERLEAVE*.



(a) 4 Threads



(b) 36 Threads

Fig. 3. Execution time comparison of different NUMA policies in WSSJ for performing *ST_Intersects* on *Sports* and *Cemetery*. OOM is out of memory for BIND memory policy.

4.2 Tasks Composition of WSSJ

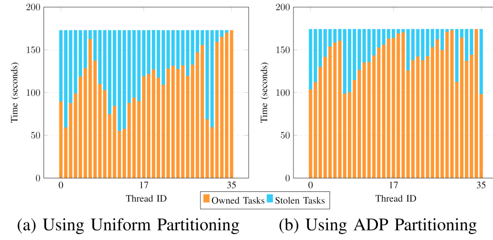
There are two types of tasks for a WSSJ worker thread: *owned tasks* and *stolen tasks*. *Owned tasks* are tasks being assigned to a worker in the beginning. *Stolen tasks* are tasks stolen from the other workers.

We designed experiments to show the tasks composition and execution time breakdown for owned and stolen tasks using WSSJ in Figure 4. We used 36 WSSJ workers (one worker for each core) to perform *ST_Intersection* on *Lakes* and *Parks* which were partitioned into 8192 grid cells using Adaptive (ADP) [22] or Uniform Partitioning. ADP is workload-aware partitioning method which first finds all candidates from the two input layers and then partitions the candidates using quadtree partitioning [22]. ADP method was shown to be more effective than standard quadtree partitioning of individual layers.

From Figure 4, we can see that the tasks compositions vary in all workers. Every worker was able to finish tasks at approximately the same time. WSSJ is not sensitive to different partitioning approaches. Using Uniform Partitioning is even slightly faster (172s) than using ADP (174s), as it has less data duplication (2.38%) than ADP (5.82%).

4.3 Tasks Composition of WSSJ-DM

A WSSJ-DM node can have two types of tasks: *local tasks* and *remote tasks*. *Local tasks* are tasks being assigned in a Round Robin scheme to each node in the beginning. *Remote tasks* are tasks received from other nodes by its coordinator.

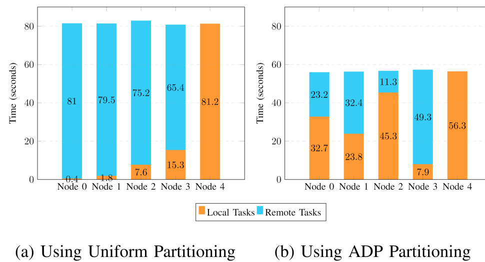


(a) Using Uniform Partitioning (b) Using ADP Partitioning

Fig. 4. Execution time breakdown of tasks at different WSSJ workers. Both cases used 36 workers to perform ST_Intersection on *Lakes* and *Parks*.

We designed experiments to show the tasks composition of every WSSJ-DM node in Figure 5. We used five WSSJ-DM nodes to perform *ST_Intersection* on *Lakes* and *Parks* which were partitioned into 8192 grid cells using ADP or Uniform Partitioning.

From Figure 5, we can see that the tasks compositions vary in all nodes in both cases. In both cases, there is one node that only works on local tasks. WSSJ-DM is able to re-balance the tasks which enabled each node to finish at approximately the same time. We can observe that using a more statically balanced partitioning (ADP) shows a better performance in WSSJ-DM. This is because a task takes more time when performed remotely than locally because of overheads in serialization, communication and coordination. A more balanced initial assignment can reduce the total number of remote tasks. In this example, there is extreme load imbalance at Node 0 because it only takes a fraction of second to finish local tasks. Node 4 does not need to steal tasks in this example.



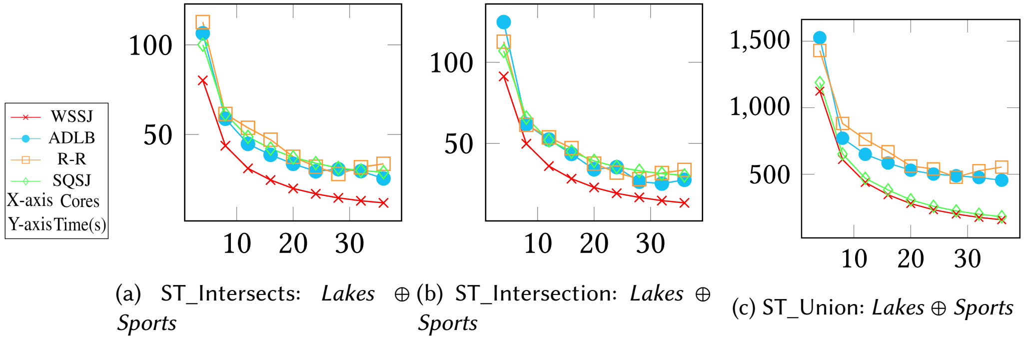
(a) Using Uniform Partitioning (b) Using ADP Partitioning

Fig. 5. Execution time breakdown of tasks at different WSSJ-DM nodes. Both cases used 5 nodes to perform *ST_Intersection* on *Lakes* and *Parks*.

4.4 Comparison Experiments for WSSJ

We designed experiments to compare the performance of WSSJ, Master-Worker, ADLB, single shared queue based spatial join (SQSJ) and Round Robin assignment using different join operations on *Lakes* and *Sports* which were partitioned into 8192 grid cells using ADP partitioning. Round Robin assignment has a better load balancing using ADP partitioning compared with Quadtree or Uniform partitioning [22].

The results are shown in Figure 6. In all cases, a single compute node was used but with different number of cores. *WSSJ* shows the best performance among four implementations in all cases. In these experiments, *WSSJ* has a parallel efficiency between 80% (at 36 cores) and 107% (at 4 cores) with respect to sequential spatial join using R-tree index (as shown in Table 2).



(a) ST_Intersects: *Lakes* ⊕ *Sports* (b) ST_Intersection: *Lakes* ⊕ *Sports* (c) ST_Union: *Lakes* ⊕ *Sports*

Fig. 6. Execution time comparison among WSSJ, ADLB, Single Queue Spatial Join (SQSJ), and Round Robin assignment (R-R) performing spatial joins on *Lakes* and *Sports*, which are partitioned into 8192 grid cells using ADP partitioning.

4.5 Comparison Experiments for WSSJ-DM

We compared WSSJ-DM with Master-Worker, ADLB, single shared queue-based distributed memory extension (SQSJ-DM), and Round Robin assignment using different join operations on different pairs of spatial data in Figure 7. The experiments were using 1 to 10 nodes (36 to 360 CPU cores).

As shown in Figure 7, WSSJ-DM performs better than ADLB, SQSJ-DM, and Round Robin assignment in most tests. WSSJ-DM performs similar with SQSJ-DM in the *ST_Union* test for *Lakes* and *Parks*. For union, using a single shared queue vs multiple work stealing deques did not make much difference. However, for intersects and intersection join, work stealing deques had an advantage. This is because on average *ST_Union* tasks are more compute-intensive than *ST_Intersection* and *ST_Intersects* using GEOS. So, degree of contention on the single shared queue per node will be different for various spatial join operations. Execution time of WSSJ-DM and SQSJ-DM keep decreasing with more CPU cores, while in general WSSJ-DM shows a better performance. The ADLB and Round-Robin implementations reach their bottlenecks quickly because of load imbalance. ADLB works very well in cases where tasks have less memory footprint. However, with geometries, server memory usage was very high leading to performance degradation.

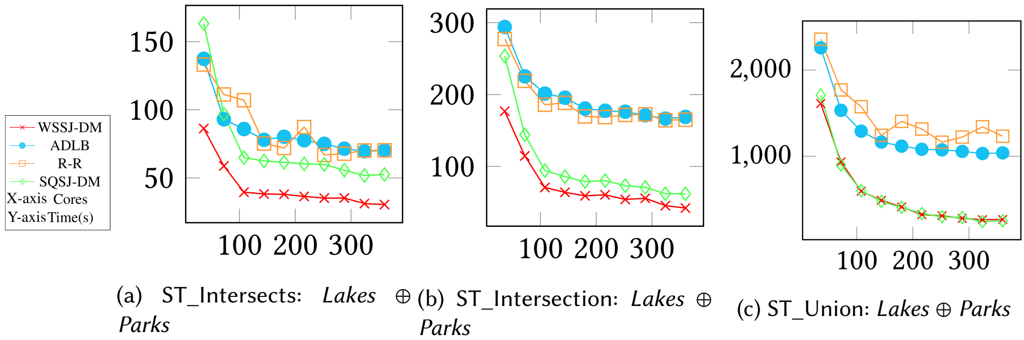
WSSJ-DM shows a more significant decrease in time for compute-intensive spatial join operations. In general, *Union* operation is computationally more expensive than *Intersection. Intersection* operation is more expensive than *Intersects.* This is reflected in the experimental results and our model also predicted the observed performance difference in Section 3.8.

4.6 Strong Scaling for WSSJ-DM

We designed strong scaling experiments for WSSJ-DM. WSSJ-DM was used to perform *ST_Intersection* on *Lakes* and *Parks* partitioned by different methods. By using different number of nodes (36 cores/node), we show the results in Figure 8. The corresponding speedups are plotted in Figure 8b.

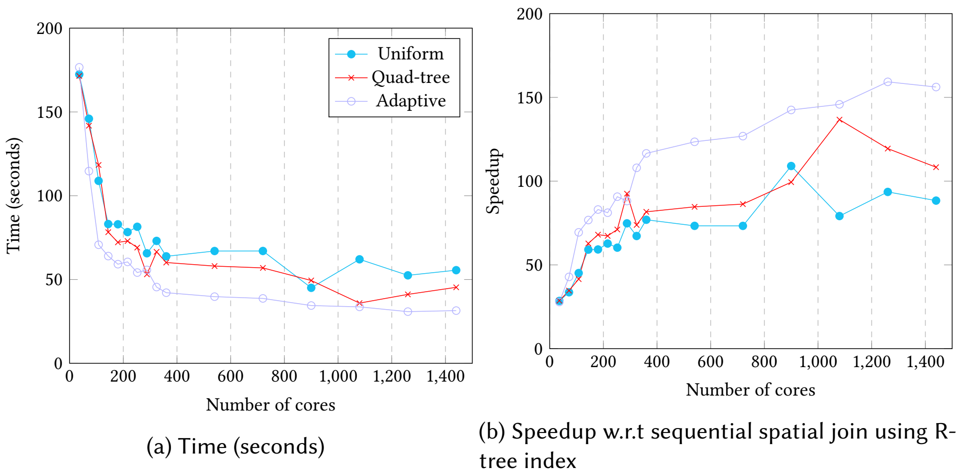
The results also follow our model that we presented in Section 3.8. Due to variation of load across different regions of the input, the performance of WSSJ-DM may fluctuate with different number of nodes. But the general trend is that WSSJ-DM can finish spatial join on *Lakes* and *Parks* faster with more cores before reaching the bottleneck.

WSSJ-DM using ADP partitioning shows the best performance, as ADP is able to provide a better static load balancing than Quadtree or Uniform partitioning [22], which means WSSJ-DM nodes can spend more time on local tasks.



(a) ST_Intersects: *Lakes* ⊕ *Parks*
(b) ST_Intersection: *Lakes* ⊕ *Parks*
(c) ST_Union: *Lakes* ⊕ *Parks*

Fig. 7. Execution time comparison among WSSJ-DM, ADLB, SQSJ-DM, and Round Robin assignment (R-R) performing spatial joins on *Lakes* and *Parks*, which are spatially partitioned into 8192 grid cells using ADP partitioning.



(a) Time (seconds)
(b) Speedup w.r.t sequential spatial join using R-tree index

Fig. 8. Execution time and speedup plot of WSSJ-DM w.r.t sequential join. For comparison, *ST_INTERSECTION* was used on *Lakes* and *Parks*. Input data was partitioned into 8192 grid cells using different approaches.

5 CONCLUSION

In this paper, we proposed fine-grained dynamic load balancing system. To our knowledge, we introduced the first Work Stealing system for Spatial Join on distributed memory (WSSJ-DM). We showed that WSSJ takes advantage of NUMA memory policies for datasets with small geometries.

We have presented experiments on various real-world datasets and evaluated the performance between WSSJ and other parallel spatial join methods based on dynamic load balancing on shared and distributed memory. Various experiments were conducted on WSSJ-DM. WSSJ-DM shows performance improvement and efficient load balancing in an HPC environment with a thousand CPU cores. The results of WSSJ-DM follow the theoretical model we presented.

6 ACKNOWLEDGEMENT

This work is partly supported by the National Science Foundation CAREER Grant No. 2145403 and Grant No. 1756000.

REFERENCES

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. 2000. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. 1–12.

[2] Dinesh Agarwal, Satish Puri, Xi He, and Sushil K Prasad. 2012. A system for GIS polygonal overlay computation on linux cluster-an experience and performance report. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1433–1439.

[3] Kouichi Araki and Taiki Shimbo. 2016. An MPI-based Framework for Proessing Spatial Vector Data on Heterogeneous Distributed Systems. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 554–558.

[4] Furqan Baig, Hoang Vo, Tahsin Kurc, Joel Saltz, and Fusheng Wang. 2017. Sparkgis: Resource aware efficient in-memory spatial query processing. In *Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems*. 1–10.

[5] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

[6] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 21–28.

[7] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–11.

[8] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[9] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press.

[10] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices* 48, 8 (2013), 69–80.

[11] Yiming Liu and Satish Puri. 2020. Efficient Filters for Geometric Intersection Computations Using GPU. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems* (Seattle, WA, USA) *(SIGSPATIAL '20)*. Association for Computing Machinery, New York, NY, USA, 487–496. https://doi.org/10.1145/3397536.3422264

[12] Yiming Liu, Jie Yang, and Satish Puri. 2019. Hierarchical Filter and Refinement System Over Large Polygonal Datasets on CPU-GPU. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 141–151.

[13] Ewing L Lusk, Steve C Pieper, Ralph M Butler, et al. 2010. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review* 17, 1 (2010), 30–37.

[14] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. 2020. The Art of Efficient In-memory Query Processing on NUMA Systems: a Systematic Approach. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 781–792.

[15] Anmol Paudel and Satish Puri. 2022. Accelerating Spatial Autocorrelation Computation with Parallelization, Vectorization and Memory Access Optimization. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 544–554. https://doi.org/10.1109/CCGrid54584.2022.00064

[16] Satish Puri, Anmol Paudel, and Sushil K Prasad. 2018. MPI-Vector-IO: Parallel I/O and partitioning for geospatial vector data. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP*. 13.

[17] Satish Puri and Sushil K Prasad. 2015. A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using mpi. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 576–585.

[18] Suprio Ray, Catherine Higgins, Vaishnavi Anupindi, and Saransh Gautam. 2020. Enabling NUMA-aware Main Memory Spatial Join Processing: An Experimental Study. *ADMS@ VLDB* (2020).

[19] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. 2014. Skew-resistant parallel in-memory spatial join. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. ACM, 6.

[20] Shashi Shekhar, Sivakumar Ravada, Vipin Kumar, Douglas Chubb, and Greg Turner. 1995. Load-balancing in high performance GIS: Declustering polygonal maps. In *International Symposium on Spatial Databases*. Springer, 196–215.

[21] Sameh Shohdy, Yu Su, and Gagan Agrawal. 2015. Load balancing and accelerating parallel spatial join operations using bitmap indexing. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 396–405.

[22] Jie Yang and Satish Puri. 2020. Efficient Parallel and Adaptive Partitioning for Load-balancing in Spatial Join. In *34th IEEE International Parallel & Distributed Processing Symposium*.

[23] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 1–4.

7 APPENDIX

7.1 Spatial Join using un-partitioned data

Generally, parallel spatial join implementations use spatially partitioned datasets. Partitioned datasets are useful to reduce data skew in tasks and make it possible to process datasets larger than available memory. On the other hand, spatial dataset partitioning requires extra time and extra storage space. As WSSJ can share tasks among threads, it is feasible to use spatially un-partitioned datasets (smaller than memory limit) directly.

Let each worker in WSSJ take a part of R and a part of S as its input. The subsets of R and S can be randomly distributed, as long as the mapping relations of all subsets can be assembled back to the same relations mapping R to S, as shown in Formula 3. As there is no need to consider the spatial localities of geometries in R, this step can be done at run-time with no additional cost compared with using partitioned datasets.

In Formula 3, R and S are randomly distributed into n and m parts respectively. $\oplus$ stands for a spatial join operation. We can get the same join results of R and S by performing join operations on all pairs of R_i and S_j.

$$
\begin{aligned}
R &= R_0 + R_1 + ...R_n \\
S &= S_0 + S_1 + ...S_m \\
R \oplus S &= \sum_{i=0}^{n} \sum_{j=0}^{m} R_i \oplus S_j
\end{aligned}
\quad (3)
$$

WSSJ using un-partitioned datasets takes slightly longer to finish when compared to partitioned datasets. The benefit of using un-partitioned data is that no data pre-processing is required, which needs extra computing resources and storage space.

To demonstrate that our system performs well with un-partitioned datasets as well, we used Sequential Spatial Join with Index, WSSJ, and WSSJ-DM to perform *ST_Intersects*, *ST_Intersection*, and *ST_Union* on several pairs of spatially un-partitioned datasets, and the results are shown in Table 2. WSSJ was using 1 node (36 cores) and WSSJ-DM was using 25 nodes (900 cores).

We can see that both WSSJ and WSSJ-DM can be helpful in saving time compared with sequential cases, especially with large datasets. For instance, performing *ST_Union* on *Roads* and *Lakes* took sequential join 53.45 hours, while WSSJ finished in 1.89 hours and WSSJ-DM finished in 7.26 minutes. *ST_Union* and *ST_Intersection* are slow in GEOS library because these operations do not internally invoke quadtree indexing for a geometry overlapping with multiple geometries. *ST_Intersects* is optimized using PreparedGeometry class provided by GEOS library.

7.2 Duplicate avoidance for spatially partitioned data

Spatial partitioning of geometries in a single map layer leads to duplication of geometry across cell boundaries. This can result in duplicate (redundant) spatial join output pairs while doing parallel processing of spatial join across cells. We refer to this method as a single layer partitioning based spatial join. We do not use single layer partitioning based method. So, the partitioning scheme used in this work is different. The spatial partitioning method has been described in our prior work (ParADP [22]) on workload-aware spatial join partitioning. We refer to this as output-sensitive duplication avoidance method where we partition the intermediate output of filter-and-refine based spatial join. In short, ParADP only partitions the center points corresponding to output candidate pairs (overlapping MBRs) generated by R-tree indexing and querying of MBRs of geometries (filter phase). Our technique is an extension of reference point method for duplicate avoidance. The duplication avoidance happens before stealing in memory. Please refer to [22] for more details.

Dataset R⊕S	Join OP	Sequential	WSSJ	WSSJ-DM
Sports⊕Cemetery	Intersects	3.39	0.59	0.14
Parks⊕Sports		165.76	10.80	1.78
Lakes⊕Sports		344.71	16.47	2.90
Lakes⊕Parks		2,401.74	119.25	20.95
Roads⊕Lakes		600.60	118.97	20.32
Sports⊕Cemetery	Intersection	3.92	0.61	0.14
Parks⊕Sports		339.32	16.14	2.89
Lakes⊕Sports		389.61	17.546	3.07
Lakes⊕Parks		4,912.32	196.24	29.92
Roads⊕Lakes		14,391.57	520.10	35.29
Sports⊕Cemetery	Union	4.38	0.68	0.13
Parks⊕Sports		1,908.46	71.82	8.60
Lakes⊕Sports		4,550.04	179.66	15.49
Lakes⊕Parks		43,236.40	1,834.39	146.25
Roads⊕Lakes		192,450.86	6,820.24	435.41

Table 2. Execution time (in sec) for Sequential Indexed Spatial Join, WSSJ (36 cores), WSSJ-DM (25 compute nodes) performing spatial join on different pairs of **un-partitioned** datasets.

7.3 Handling other spatial join algorithms

In this paper, we showed work stealing based spatial join on partitioned and unpartitioned data based on filter and refine phases. Filter and refine is implemented using indexed nested-loop spatial join algorithm. However, the proposed work stealing technique can be used with other spatial join algorithms as well. For instance, when spatial join is implemented using plane sweep approach, then the intermediate output produced by plane sweep of MBRs of input geometries can be stored in the work stealing queue. Once the tasks are stored in queues, the system will start load balancing. Similarly, when spatial join is implemented by hierarchical traversal (synchronized traversals) of R-trees, the tree nodes with overlapping ranges will produce intermediate output which can be stored in work stealing queues for further refinement processing. These alternative spatial join implementations can be part of future work.

7.4 Fine-grained load balancing

Most of the work on spatial join considers a grid cell generated from spatial partitioning as a unit task for assignment to a CPU thread and for the purpose of load balancing. A grid cell can have an arbitrary number of geometries contained in it. This is a coarse-grained task in our view. We consider a geometry from a dataset R overlapping with a small number (like 10) of geometries from S as a unit task for assignment to a CPU thread and for the purpose of work stealing. This is a fine-grained task for the purpose of parallelization and load balancing in our view.

7.5 Remote Memory Access (RMA) and Non-blocking Communication

We have used one-sided (put/get) Message Passing Interface (MPI) functions for task coordination between any two processes. One-sided programming model is referred to as Remote Memory Access (RMA) in MPI. It is suitable for expressing irregular communication patterns that arise while coordinating tasks among processes in distributed memory [9]. One-side communication is used for exchanging control messages. However, non-blocking send/receive functions are used for actual data transfers because of programming simplicity.

RMA uses the concept of memory window which is the memory in a process that can be accessed by another remote process through the use of RMA put/get functions.