# SOJA: A Memory-efficent Small–large Outer Join for MPI

Liang Liang, Guang Yang, Thomas Heinis
Imperial College London

David Taniar
Monash University

## ABSTRACT

The join is a fundamental and widely used operation in data analytics but equally, it is also one of the most expensive ones. Considerable work has been carried out to improve and evaluate join approaches based on popular distributed processing systems such as Spark and Hadoop, however, it has not been widely studied on MPI.

In this paper, we first implement, analyse and compare existing algorithms for the common small-large outer join operation and develop a novel approach, the swap-based outer join algorithm (SOJA). SOJA is designed to minimise the expensive communication between the distributed nodes while also reducing the cost of the local join operations. We demonstrate the benefits of SOJA experimentally, showing that it achieves at worst an execution time similar to its competitors. More importantly, SOJA requires substantially less memory (typically half the memory compared to the best competitor) and that memory usage scales very well.

## KEYWORDS

Outer Joins, Algorithm, Parallel Processing, MPI

## 1 INTRODUCTION

Collecting and storing data has never been as easy and as cheap as today. It comes as no surprise that vast amounts of data are being stored today and it is predicted that all known data worldwide will grow to 250 Zettabytes by 2025. Many real-world applications benefit from the efficient analysis of large amounts of data, be it for medical applications [1, 2, 13], scientific applications [11] or commercial applications such as traffic analysis [7] and others. Analysing this data efficiently and at scale has thus never been more important than today and is also a considerable challenge.

Crucial in the analysis of large amounts of data is the combination of multiple datasets before analysis. One central operation thus is the join operation which combines multiple datasets (or one with itself) by matching tuples with a shared attribute. More specifically, a join on datasets $R$ and $S$ based on equality (or a different relationship) will pair tuples $r \in R$ and $s \in S$ if $r.c_1 = s.c_2$ where $c_1$ and $c_2$ are attributes of the tuples. The operation is frequently used but very costly due to computational overhead but also because of I/O.

In this paper, we develop the swap-based outer join algorithm (SOJA), a new approach to the specific problem of the small-large outer join where a small and a large dataset are joined. We develop SOJA for Message Passing Interface (MPI) on HPC infrastructure as such large-scale parallel infrastructure is one of the few efficient ways to join large datasets [3]. MPI is a message-passing standard which is widely used in high performance applications [10]. The standard defines the syntax and semantics of approximately 250 library routines that allow users to develop a wide variety of communication operations on different types of parallel computing infrastructure[8, 9]. Point to point communication between two MPI processes (ranks) and collective communication among MPI processes are most commonly used. For each of them, MPI also provides multiple communication modes that fall into either blocking or non-blocking communication according to whether constituent operations of the communication complete synchronously. Additionally, MPI supports a derived datatype and a virtual topology which allows users to control data movement among processes efficiently and flexibly. MPI thus is a promising tool to design and implement algorithms to handle and analyze massive amounts of data efficiently.

We use MPI running on HPC infrastructure to efficiently execute a small-large left outer join. The small-large left outer join can be denoted as $R ⟕ S$ with $|R| << |S|$ where $|R|$ and $|S|$ are number of tuples in tables or, more generally, datasets $R$ and $S$. In the query below, $R$ and $S$ represent the left table and right table, and a join is performed between $R$ and $S$ based on join keys $R.a$ and $S.b$:

```
SELECT * FROM R LEFT OUTER JOIN S ON R.a = S.b;
```

For parallelising the left outer join, we assume that $R$ and $S$ are distributed among the $N$ processes in a round-robin fashion. Each partition $R_i$ is assigned to the $i-th$ process, and has the same number of elements $|R_i| = \frac{|R|}{N}$. The same applies to all $S_i$.

The parallel left outer join with partitioned data then has two goals: (1) find all matching tuples from the two tables; (2) find all dangling tuples from the left table and output them with no matching tuple from the right table.

Existing approaches mainly adopt two methods, redistribution and broadcast, to produce the entire join results while guaranteeing data locality. Redistribution refers to redistributing both tables among all processes to make tuples such that the same join keys are placed in the same process. Broadcast, in this context, means the left table in each process is duplicated and sent to all other processes, so that each process holds the complete left table. These two methods either lead to inevitable skewness or duplication. SOJA adopts a novel method, *swap*, to ensures tuples in the left table can join all possible matching tuples in the right table by swapping the left table among processes. Based on the swap method, SOJA can also perform other types of parallel joins such as inner or right outer joins by replacing local join methods.

As our extensive set of experiments shows, SOJA in many cases outperforms its competitors and at worst has an execution time similar to its competitors. Most importantly, however, SOJA requires substantially less memory which in a supercomputing environment is crucial (as data cannot easily be swapped to disk). SOJA typically requires only half the memory and, as our experiments show, scales extremely well.

In the remainder of this paper we first review the state of the art in Section 2, present our approach SOJA in Section 3.1, analyse SOJA experimentally in Section 4 and conclude in Section 5.

## 2 RELATED WORK

In this section, we first describe four related parallel outer join algorithms (Figure 1) and then discuss their limitations and communication implementations on MPI .

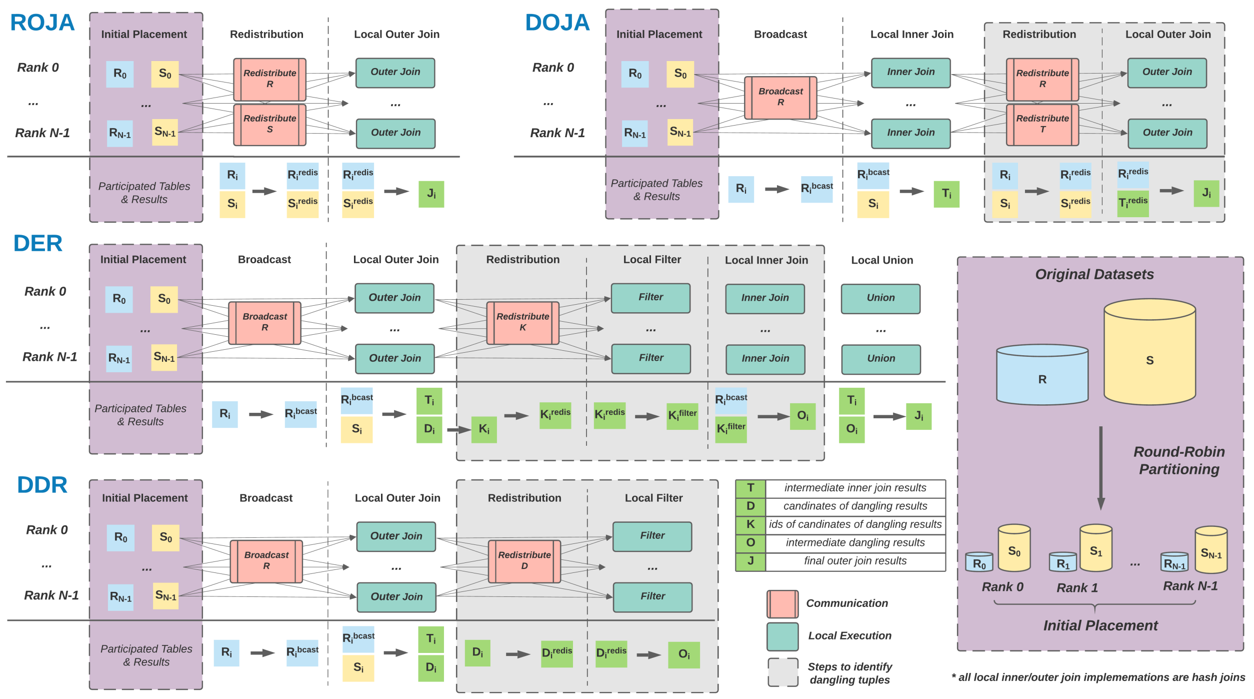

**Figure 1: Illustration of ROJA, DOJA, DER and DDR**

## 2.1 Redistribution Outer Join Algorithm (ROJA)

ROJA uses the redistribution approach to execute the local outer join, it has two steps, redistribution and local outer join, which can be considered as an extension of a typical re-partition join algorithm [4, 12]. The only difference to a typical re-partition join algorithm is that ROJA adopts the local outer join in the second step instead of local inner join. ROJA proceeds as follows:

(1) all tuples of $R_i$ and $S_i$ in each process are redistributed based on the join keys ($R_i.a$ and $S_i.b$).
(2) each process performs a local outer join between reallocated two tables denoted as $R_i^{redis}$ and $S_i^{redis}$.

## 2.2 Duplication Outer Join Algorithm (DOJA)

DOJA is a variation of the broadcast join algorithm [4, 12]. It obtains inner join results by performing inner join after broadcasting the tables similar to broadcast join algorithms. However, DOJA needs to execute two additional steps to identify dangling tuples:

(1) each process broadcasts the smaller left table $R_i$ to all other processes, which results in each process now having an entire table $R$ denoted as $R_i^{bcast}$.
(2) local inner join in each process between $R_i^{bcast}$ and $S_i$ outputs inner join results stored in $T_i$.
(3) $T_i$ and $R_i$ are redistributed based on join keys ($T_i.a$ and $R_i.a$).
(4) after receiving redistributed $T_i^{redis}$ and $R_i^{redis}$, each process executes local outer join to output complete result.

## 2.3 Duplication and Efficient Redistribution (DER)

DER [14] improves DOJA by reducing the size of the redistributed data. DER executes as follows:

(1) this step is same as step 1 for DOJA with the only difference being the addition of ids. DDR needs to add keys to tuples, if there is no global identifier in $R_i$.
(2) a local outer join between $R_i^{bcast}$ and $S_i$ generates inner join results ($T_i$) as well as non-matching tuples.
(3) each process in DER redistributes ids ($K_i$) extracted from non-matching tuples ($D_i$).
(4) after redistribution, $K_i^{redis}$ is filtered so that only keys that appear $N$ times are stored, because a dangling tuple in $R$ will not be matched in all $N$ processes. The result is in $K_i^{filter}$.
(5) truly dangling tuples $O_i$ in the left table can be retrieved by a local inner join between $K_i^{filter}$ and $R_i^{bcast}$.
(6) the union of $O_i$ and $T_i$ in each process is the final results.

## 2.4 Duplication and Direct Redistribution (DDR)

DDR [5] is a broadcast-based outer join algorithm focusing on optimizing the identification of dangling tuples. Its steps are:

(1) the first two steps of DDR are identical to DER.
(2) each process outputs matched join results ($T_i$) and redistributes non-matching results ($D_i$).
(3) similar to step 4 of DER, dangling tuples after redistribution ($D_i^{redis}$) are kept if they occur $N$ times. The results $O^i$ is directly outputted.

## 2.5 Discussion

We will compare the communication cost and local execution cost between the different approaches.

### 2.5.1 *Communication.*

The two main methods for communicating among processes are: (1) redistributing where the tuples are categorized and sent to their respective destinations; (2) broadcasting, where a table is duplicated across all of the processes. Redistribution requires additional calculation of destination, thus, the redistribution cost is higher than broadcast cost ($t^{redis} > t^{trans}$). To compare communication cost of algorithms, we develop a simple cost model.

$$comm \begin{cases} (R+S) \times t^{redis} & \text{(ROJA)} \\ R \times (N-1) \times t^{trans} + (R+T) \times t^{redis} & \text{(DOJA)} \\ R \times (N-1) \times t^{trans} + K \times t^{redis} & \text{(DER)} \\ R \times (N-1) \times t^{trans} + D \times t^{redis} & \text{(DDR)} \end{cases} \tag{1}$$

In small-large joins, based on the above equation, the communication cost of ROJA can be very high, since each process in ROJA has to redistribute the large right table $S_i$. The other three algorithms share a similar broadcast cost. The cost may appear low initially, but with an increase in the number of processes $N$ and the size of left table ($|R|$), the cost can grow significantly. The difference in communication cost between the three broadcast-based algorithms is that they redistribute different intermediate data. DOJA redistributes the left table $R$ and the inner join result $T$ (the size of $T$ depends on the selectivity ratio $\sigma_j$). With an increasing $\sigma_j$, the redistribution cost will increase significantly, thereby dominating the overall performance of DOJA. As a consequence, two improved methods, DDR and DER, were developed to optimize this stage by redistributing dangling results. DER uses ids of dangling tuples ($K$) and DDR uses dangling tuples ($D$). The number of dangling ids ($|K|$) is the same as the number of dangling tuples ($|D|$), however, the storage size of $K$ may be smaller than $D$. Thus, from the perspective of communication, DER can outperform DDR. The overall redistribution cost of DDR and DER is not significant. However, the number of dangling tuples will increase when the number of processes ($N$) increase simply due to the fact that the $R$ is duplicated in more processes. It is worth mentioning that redistributing may potentially result in an unbalanced workload due to skewness. In the presence of data skew, the overall performance is adversely impacted by the redistribution stage as well as subsequent operations. Additionally, both redistribution and broadcast require more memory.

*2.5.2 Local Execution.* The existing algorithm mainly involve two local operations: join (inner join and outer join) and filter. Their time costs are denoted as $t^{join}$ and $t^{filter}$.

$$local \begin{cases} (R_i^{redis} + S_i^{redis}) \times t^{join} & \text{(ROJA)} \\ (R + S_i + R_i^{redis} + T_i^{redis}) \times t^{join} & \text{(DOJA)} \\ (2R + S_i + K_i^{filter}) \times t^{join} + K_i^{redis} \times t^{filter} & \text{(DER)} \\ (R + S_i) \times t^{join} + D_i^{redis} \times t^{filter} & \text{(DDR)} \end{cases} \tag{2}$$

As we mentioned before, the skewness in the data can result in an unbalanced workload in subsequent operations and the performance of ROJA is highly affected by data skew (as data in some processes after redistribution ($R_i^{redis}$, $S_i^{redis}$) may be significantly bigger than the initial placement ($S_i$ , $R_i$)). DOJA not only needs to perform a local join after the broadcast to compute all inner join results but also needs to identify non-matching tuples in the left table by joining the redistributed left table ($R_i^{redis}$) and inner join results ($T_i^{redis}$). The cost of a second join strongly depends on the inner join ratio ($\sigma_j$) and skewness degree ($\theta$). For DER and DER, a filter step is required after the local join. DER needs to filter keys which may have a smaller size in bytes than the tuples filtered by DDR. Thus, in terms of cost of the filter stage, DER has an advantage. However, after the filter stage, DDR can directly output dangling tuples while DER has to retrieve dangling tuples by performing an additional join. In addition, DER has extra local update costs ($R_i \times t^{update}$) if no ids are assigned to the left table $R_i$.

## 2.6 Broadcast and Redistribution on MPI

Both broadcast and redistribution are collective communication in which all processes are participating. For broadcasting in DOJA, DER and DDR, all processes need to send their participating partitions to all other processes. From the point of view of the receiver, all processes gather data from all other processes, which can be achieved by the MPI routine `MPI_Allgather`. Different processes may hold data of different size, we implement the broadcast by using `MPI_Allgatherv` which allows each process to contribute different amounts of data. As for redistribution, all processes could be the destination of the redistributed data in all processes. Such a communication pattern can be supported by `MPI_Alltoall` and `MPI_Alltoallv` which send data from all processes to all processes. To implement hash redistribution, before calling these routines, the data needs to be reorganized based on its corresponding destinations which is determined by the hash function.

## 3 SWAP-BASED OUTER JOIN ALGORITHM (SOJA)

### 3.1 Approach & Design of SOJA

Compared to existing algorithms that mainly uses redistribution and broadcasting, SOJA adopts a novel *swap* approach using a ring topology to perform the parallel outer join. Additionally, SOJA decomposes the traditional local outer hash join operations to hash (create hash table from right table) and lookup (loop over tuples in $R_i$ to look up the hash table). Hashing the right table allows the dangling tuples in the left table to be determined with a single loop for a left outer join. The steps for SOJA are:

(1) each process creates a hash table ($HS_i$) for the right table ($S_i$). It then loops over the tuples in the left table ($R_i$) to query the hash table $HS_i$, meanwhile marking non-matching tuples in the $R_i$ as candidates for dangling tuples. Finally, the intermediate inner join results will be directly returned.
(2) each process sends the updated $R_i$ to the next process $(i+1)$ in the ring topology. Note, the $(N-1)-th$ process will send $R_{N-1}$ to process 0.
(3) a lookup would be carried out on the new set of $R_i$ and $HS_i$, and the intermediate results would be outputted. The non-matching mark will be removed if the candidate tuple is able to find a match.
(4) repeat steps 2 & 3 $N-1$ times, and output the tuples in each of the $R_i$ with marks as dangling tuples in the last iteration.

The cost model of the iterative operations is the following:

$$SOJA \begin{cases} R \times (N-1) \times t^{trans} & \text{(comm)} \\ (R + S_i) \times t^{join} + R \times t^{update} & \text{(local)} \end{cases} \tag{3}$$

From perspective of cost model, the communication cost is the same as the broadcast operation, the total join cost of SOJA is also the same as outer join after the broadcast in DER and DDR. However, no redistribution, filter, or additional joins are required by SOJA. It identifies non-matching tuples in the left table and updates the left table $R$ while looping. Additionally, the cost of $t^{update}$ is not significant, due to the left table being small.

Further, compared to existing methods, SOJA saves memory for the following reasons: (1) directly outputs of the intermediate results after each iteration, (2) no duplication of tables, and (3) no temporary dangling data as the size of temporary data can increase with the number of processes as well as with the selectivity ratio in the left table $\sigma_R$. In addition, SOJA can easily be extended to perform other types of joins such as inner join or right outer joins. For example, inner join is executed by performing a local lookup without marking and updating dangling tuples in the left table. For right outer joins, SOJA simply flips the operation by creating a hash table for the left table and updating the marks on the right table.

Although, there is no redistribution in SOJA, its performance is still sensitive to an imbalanced workload among processes, especially due to imbalanced initial placement as there are synchronization costs for communication. Therefore, for SOJA, initially placing partitions of equal size is a prerequisite for achieving ideal performance.

## 3.2 Swap of SOJA on MPI

We use Cartesian topology routines `MPI_Cart_create` with 1-dimension to construct SOJA's ring topology in which each process is logically connected to two other processes ($left$ and $right$ process) in a circle. In the ring topology, point to point communication routines (`MPI_Send` and `MPI_Recv`) can be used to swap data between adjacent processes. To reduce the synchronization costs, we use non-blocking mode (`MPI_Isend` and `MPI_Irecv`) to transfer data that can overlap other local operations, such as result output or other operations based on intermediate results. The swap stage of SOJA is shown in Algorithm 1.

**Algorithm 1:** Swap in SOJA

```
1:  Require: $R_i$, $N$, left and right neighbour $left$ and $right$
2:  $recv\_buffer = R_i$
3:  for $i = 1$ to $i$ = N-1 do
4:      $send\_buffer$ = lookup_and_update($recv\_buffer$)
5:      MPI_Isend($send\_buffer$, dest = $left$, $send\_Request$)
6:      MPI_Irecv($recv\_buffer$, source = $right$, $recv\_Request$)
7:      overlapping: output result
8:      MPI_Wait($send\_Request$)
9:      MPI_Wait($recv\_Request$)
10: end for
11: $result$ = lookup_and_update($recv\_buffer$)
```

## 4 EXPERIMENTAL EVALUATION

In the following section, we analyse SOJA using experiments with changing the size of left table ($|R|$), selectivity ratio of left table ($\sigma_R$), and skewness degree ($\theta$). We focus on execution time and memory usage.

## 4.1 Setup

*4.1.1 Platform.* We use an HPC cluster with Intel E5-2680 v3 @ 2.50GHz running Centos 8. The MPI version is 3.3.2.

*4.1.2 Datasets.* The experiments are based on `customer` and `supplier` tables from the TPC-H benchmark [6]. For simplicity and consistency of presentation, we use $R$ and $S$ to represent `customer` and `supplier` respectively. To test the algorithms with different scenarios, we modify the data in two ways: first, we scale up or scale down the table size ($|R|$ and $|S|$) by sampling data from the two initial tables (uniform with replacement). Second, we vary the selectivity ratios, i.e., the join selectivity ratio ($\sigma_j$) and the selectivity ratio in the left table ($\sigma_R$). The selectivity ratio affects the number of join results ($|T|$) and can be roughly controlled by the sample population of join key $n$ with uniform distribution, which can be explained using Equations 4 to 6.

The definition of $\sigma_j$ is:

$$\sigma_j = \frac{|T|}{|R| \times |S|} \tag{4}$$

If we sample the join key from a population that contains $n$ values and denote the probability of choosing $i-th$ item as $p_i$, then:

$$\sigma_j = \frac{\sum_{i=1}^{n} p_i \times |R| \times p_i \times |S|}{|R| \times |S|} = \sum_{i=1}^{n} p_i^2 \tag{5}$$

If $p_i$ is a uniform distribution, $p_i = \frac{1}{n}$, then:

$$\sigma_j = \frac{1}{n} \tag{6}$$

The selectivity ratio in the left table ($\sigma_R$) determines the number of dangling tuples in the left table. With the involvement of $\sigma_R$, the number of results can be estimated by Equation 7.

$$|T| = \underbrace{|R| \times \sigma_R \times |S| \times \sigma_j}_{\textit{inner join results}} + \underbrace{(1-\sigma_R) \times |R|}_{\textit{dangling results}} \tag{7}$$

If $\sigma_R$ equals to 0, this means no tuples in table $R$ are selected to perform the join with table $S$. Therefore, table $R$ will be the result. As $\sigma_R$ increases, the number of matching results increases whereas the non-matching results decrease until all tuples in $R$ are involved in the join when $\sigma_R$ equals to 1. We assign a negative join key to number of tuples in table $R$ based on 1 - $\sigma_R$ [5]. Therefore, any tuple with a negative key in the left table ($R$) will not have any matching results. Both $\sigma_j$ and $\sigma_R$ can influence $|T|$, but in outer joins, $\sigma_R$ will provide more insights (dangling tuples), so our experiments focus on $\sigma_R$ to study changes of $|T|$ and only use $\sigma_j$ to keep the number of potential inner join results (when $\sigma_R = 1$) unchanged when varying the input data size.

To achieve the parallel IO, we split the data into $N$ partitions and processes load their partition from disk in parallel.

## 4.2 Impact of the Size of the Left Table

To examine how the performance of approaches changes when the size of left table increases, we conduct the experiment in which we vary $|R|$ and keep all other parameters such as $|S|$, $\sigma_R$, fixed. All five algorithms are executed on 32 cores and 64 cores and we measure execution time and memory usage.

In this experiment, we set $|S|$ to $5 \times 10^7$, $\sigma_R$ to 0.5, and increase $|R|$ with a coarser granularity (10×) from $10^3$ to $10^7$. Figure 2 shows that all algorithms maintain a stable performance until $|R|$ reaches $10^6$. We further explored changes of the performance with a fine-grained increase of $|R|$ (increment of $10^6$ tuples) in the interval between $10^6$ and $10^7$. The result from 32 processes and 64 processes (Figure 2 A & C) shows that the execution time of ROJA is steady whereas execution time of all other algorithms has a clear upward trend. Figures 2 B & D indicate that SOJA has an advantage in terms of memory usage across all experiments. Although ROJA has the highest memory usage when $|R|$ is small, the memory usage level stays quite constant with increasing $|R|$. The other three broadcast-based algorithms show that their memory usage increases significantly, particularly DDR. It is worth pointing out that when $|R|$ is relatively large, the performance of DDR on 64 processes is worse than itself on 32 processes.

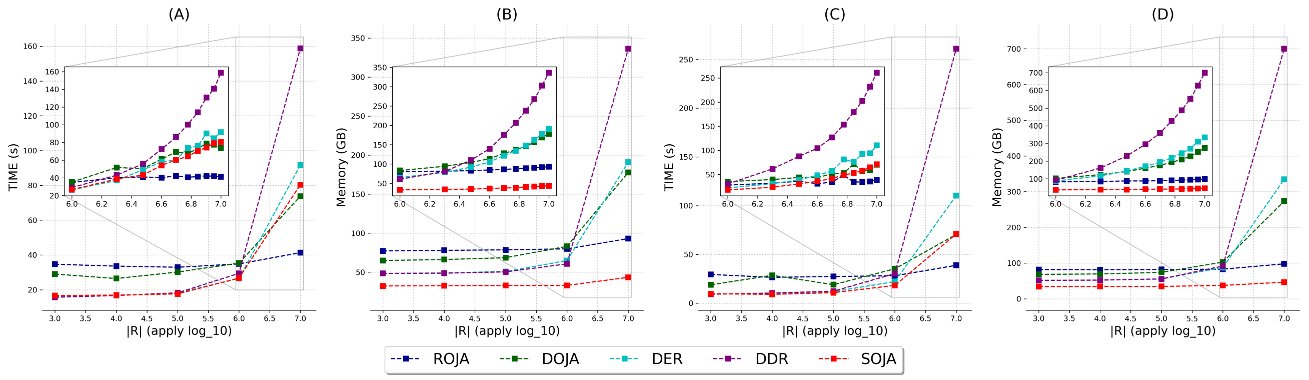

**Figure 2: Execution time and memory usage against $|R|$, wide ranges $|R|$ from $10^3$ to $10^7$, narrow is from $10^6$ to $10^7$. (A) time vs $N$ for 32; (B) memory usage vs $N$ for 32; (C) time vs $N$ for 64; (B) memory usage vs $N$ for 64;**

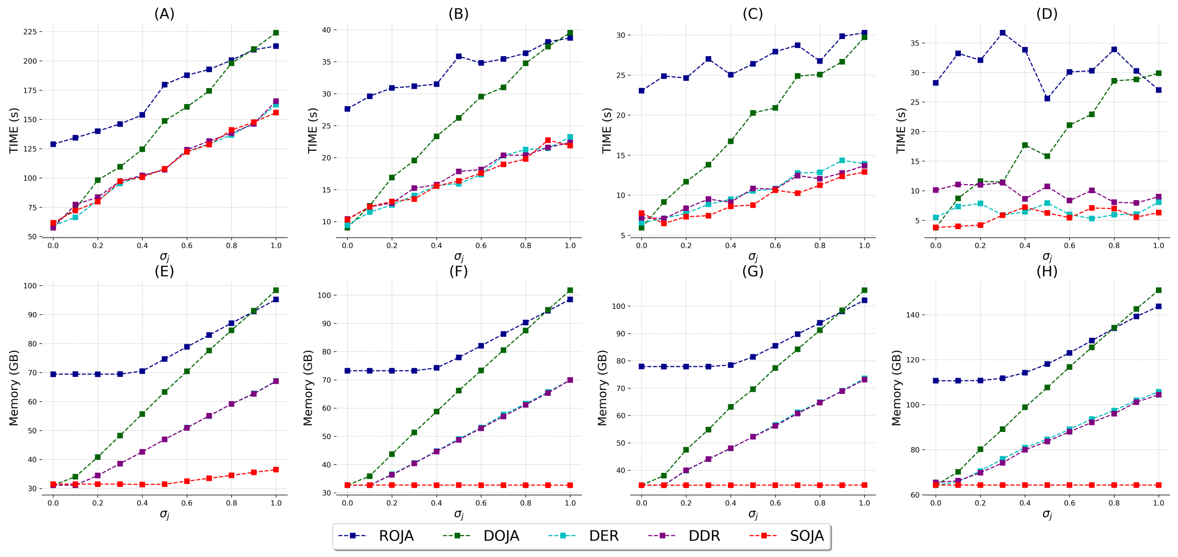

**Figure 3: Execution time against selectivity ratio of left table over varying number of processes: (A - D): execution time for $N$ = 4, 32, 64, 512; (E - H): memory usage for $N$ = 4, 32, 64, 512;**

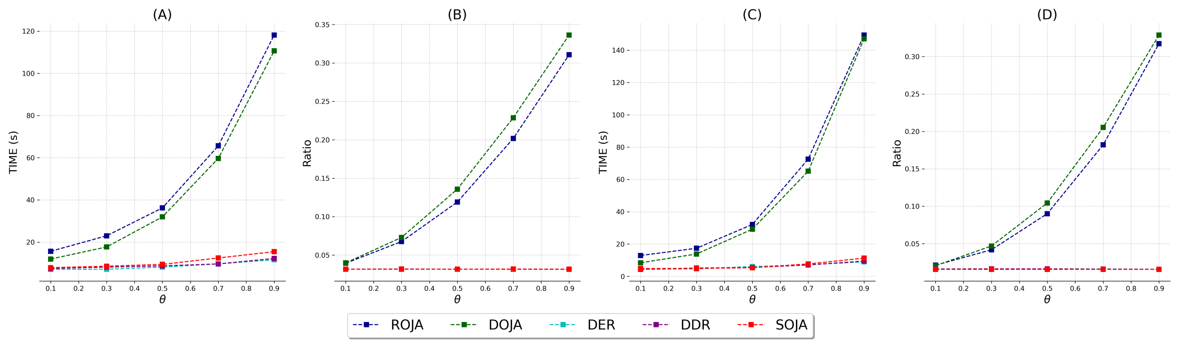

**Figure 4: Execution time and memory usage ratio against skewness degree: (A) execution time for $N$ = 32; (B) memory usage ratio $N$ = 32; (C) execution time for $N$ = 64; (D) memory usage ratio for $N$ = 64;**

Both execution time and memory usage in ROJA are not considerably affected by changes in $|R|$, because the cost of ROJA is dominated by the large table (although $R$ increases, it is still at least 5 times smaller than $|S|$). The broadcast cost and subsequent local join between $R$ and $S_i$ in DOJA, DER and DDR increases with the growth of $R$, increasing both execution time and memory usage. Furthermore, dangling tuples candidates after the first local outer join will also increase as $R$ increases. Therefore, DER and DDR have to redistribute and filter more data to identify non-matching tuples. The number of candidates is roughly

$|R| \times (1-\sigma_R) \times N$. Unlike DER which use ids, DDR directly works with tuples; therefore, its execution time and memory usage increases dramatically with an increase of $|R|$ and $N$. As for SOJA, the swap step becomes more expensive with the growth of $|R|$. Overall, we can see that when $|R|$ is 10 times less than $|S|$, DER, DDR and SOJA outperform ROJA in both time and memory.

## 4.3 Effect of Selectivity Ratio of Left Table

These experiments examine the algorithms' performance with different $\sigma_R$ from 0.0 to 1.0 using a varying number of processes (4, 32, 64, 512). We set $|R|$ to $10^5$ and $|S|$ as $5 \times 10^7$, and the $\sigma_j$ to $10^{-5}$.

The results in Figure 3 (A - D) show that the overall execution time has an upward trend with an increase of $\sigma_R$ as a high $\sigma_R$ leads to an increase of the result size and thereby increasing IO costs. The execution time of DOJA increases dramatically since DOJA has to redistribute and perform a second local join based on increasing inner join results. Overall, DER, DDR and SOJA share a similar performance which outperform the two conventional algorithms. The upward trend stabilizes with increasing number of processes due to reduction of local IO costs. Figure 3 (E - H) shows the change of memory usage with increase of $\sigma_R$. When $\sigma_R$ is small, the redistribution cost in ROJA takes a considerable amount of memory, and it require even more memory to hold join results as $\sigma_R$ increases. DOJA uses more memory compared to the other two broadcast algorithms because it redistributes the inner join results rather than the dangling tuples.

## 4.4 Effect of Data Skewness

To test the effect of skewness, we use the Zipf distribution model and the number of tuples for both tables after redistribution in the $i-th$ process ($|R_i^{redis}|$ and $|S_i^{redis}|$) is the following:

$$|R_i^{redis}| = \frac{|R|}{i^\theta * \sum_{j=1}^{N} \frac{1}{j^\theta}} \tag{8}$$

To generate data with skew, we sample multiple datasets with the join key using a Zipfian distribution with different degrees $\theta$ from 0.1 to 0.9. We set $|R|$ as $10^5$, $|S|$ as $2 \times 10^7$, and $\sigma_R$ as 0.5. In this experiment, we examine how execution time as well as memory usage changes on 32 and 64 processes with an increase of $\theta$. In case of skewness, the execution time is determined by the process with the biggest workload. To make the memory usage results meaningful, we use a memory usage ratio, which is the maximum memory usage among all processes over the total memory usage, instead of total memory usage.

Experiments on both 32 and 64 processes (Figure 4) show that the degree of skewness has a significant impact on execution time and memory usage ratio of ROJA and DOJA due to unbalanced workload caused by the redistribution step. Although DER and DDR also feature a redistribution operation, their performance remains stable as $\theta$ changes. This is because, in this experiment, the number of dangling tuples/ids to redistribute is relatively small due to the small $|R|$. The memory ratio in Figure 4 (B & D) also reflects that both DER and DDR have a relatively balanced memory usage. Since this skewness is about data skew in the join key rather than the initial placement, both memory and execution performance of SOJA are not much affected by $\theta$.

## 4.5 Discussion

ROJA, as a general-purpose outer join algorithm, does not have an outstanding performance when the left table's size is very small. Although ROJA is less affected by changes in the selectivity ratio in the left table, its performance with high data skew drops dramatically. Another conventional algorithm, DOJA, has no outstanding performance in most cases because it has to broadcast the entire left table and redistribute the entire inner results, which is particularly bad when the join selectivity ratio is large.

In small-large outer joins, DER shows an outstanding performance over most cases as it optimizes the redistribution by focusing on ids to identify dangling tuples. Using ids of dangling tuples makes DER less affected by the selectivity ratio and skewness. As a DER competitor, DDR adopts a simpler procedure, directly redistributing and filtering based on the dangling tuples itself. DER and DDR share the core methodology and their performance in both time and memory aspects is also similar in most cases. However, when the number of the tuples in the left table increases, the performance of DDR becomes the worst since the size of candidates of dangling tuple in bytes increases both computation and memory costs.

SOJA achieves a similar (sometimes better) performance than DER and DDR in terms of time costs in most cases and outperforms all competitors in terms of memory usage. Additionally, SOJA is less affected by data skewness and selectivity ratio since there is no redistribution operation in SOJA.

## 5 CONCLUSIONS

In this paper, we implemented four existing parallel outer join algorithms in MPI and proposed a new algorithm SOJA. SOJA does not simply optimize one specific part of existing algorithms; it provides an entirely novel approach, *swap*, to perform outer joins. The experiment based on HPC infrastructure shows that the performance of SOJA is outstanding, especially in memory usage. Further, we will investigate whether SOJA can be used in other joins, such as inner join, with the same benefits. Additionally, the feasibility of the swap approach in parallel spatial joins will be explored in future works.

## REFERENCES

[1] Ablimit Aji and Fusheng Wang. 2012. High Performance Spatial Query Processing for Large Scale Scientific Data. In *SIGMOD/PODS '12 PhD Symposium.*

[2] Ablimit Aji, Fusheng Wang, and Joel H. Saltz. 2012. Towards Building a High Performance Spatial Query System for Large Scale Medical Imaging Data. In *SIGSPATIAL '12.*

[3] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *PVLDB* 10, 5 (2017).

[4] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD '10.*

[5] Long Cheng, Ilias Tachmazidis, Spyros Kotoulas, and Grigoris Antoniou. 2017. Design and evaluation of small–large outer joins in cloud computing environments. *J. Parallel and Distrib. Comput.* 110 (2017), 2–15.

[6] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.

[7] TLC Trip Record Data. 2020. https://www1.nyc.gov/tlc.

[8] Victor Eijkhout. 2017. *Parallel Programming in MPI and OpenMP.* Lulu. com.

[9] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. *Using MPI: portable parallel programming with the message-passing interface.* Vol. 1. MIT press.

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

[11] Henry Markram. 2006. The Blue Brain Project. *Nature Reviews Neuroscience* 7, 2 (2006), 153–160.

[12] David Taniar, Clement HC Leung, Wenny Rahayu, and Sushant Goel. 2008. *High-performance Parallel Database Processing and Grid Databases.* Vol. 67.

[13] Fusheng Wang, Ablimit Aji, and Hoang Vo. 2014. High Performance Spatial Queries for Spatial Big Data: From Medical Imaging to GIS. *SIGSPATIAL Special* 6, 3 (2014).

[14] Yu Xu and Pekka Kostamaa. 2010. A new algorithm for small-large table outer joins in parallel DBMS. In *ICDE 2010.*