Multi-Assignment Single Joins for Parallel Cross-Match of Astronomic Catalogs on Heterogeneous Clusters

Xiaoying Jia, Qiong Luo
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Kowloon, Hong Kong
{xjia, luo}@cse.ust.hk

ABSTRACT

Cross-match is a central operation in astronomic databases to integrate multiple catalogs of celestial objects. With the rapid development of new astronomy projects, large amounts of astronomic catalogs are generated and require fast cross-match with existing databases. In this paper, we propose to adopt a Multi-Assignment Single Join (MASJ) method for cross-match on heterogeneous clusters that consist of both CPUs and GPUs. We chose MASJ for cross-match, because (1) cross-matching records from astronomic catalogs is essentially a spatial distance join on two sets of points, and (2) each reference point is mapped to only a small number of search intervals. As a result, the MASJ cross-match, or MASJ-CM algorithm is feasible and highly efficient in a heterogeneous cluster environment. We have implemented MASJ-CM in two packages: one is an MPI-CUDA implementation, which fully utilizes the multi-core CPUs, GPUs, and InfiniBand communications; the other is on top of the popular distributed computing platform Spark, which greatly simplifies the programming. Our results on a six-node CPU-GPU cluster show that the MPI-CUDA implementation achieved a speedup of 2.69 times over a previous indexed nested-loop join algorithm. The Spark-based implementation was an order of magnitude slower than the MPI-CUDA; nevertheless, it is widely applicable and its source code much simpler.

1. INTRODUCTION

In astronomy, cross-match is crucial for integrating physical attributes of celestial objects from multiple catalogs and distributed archives. The information integrated from observational results at different wavelengths or at different points in time, is essential to picture the entire universe and study the temporal evolution for certain celestial objects. However, these observational results often record slightly different positions for the same object due to different resolutions or calibrations. Thus, in cross-matching two catalogs, the condition to determine a matching pair of observation records is a distance threshold.

As other scientific areas, astronomy is experiencing a data avalanche. The state-of-the-art astronomical missions, which adopt high-resolution detectors and large arrays of cameras, are archiving a vast number of objects on a nightly basis. For example, large catalogs that offer public access, such as SDSS DR12 [3] and USNOB1 [26], contain billions of objects. With the increasing scale of catalogs, the ability to efficiently cross-match large catalogs is crucial for real-time analytics and astronomical breakthroughs. However, most existing studies, which implemented cross-match on a single computer [5, 33, 40, 49] or on a small set of homogeneous CPU servers [21, 27, 28], can only process smaller catalogs.

Given the increase of data volume, processing data on computer clusters is an excellent choice. Modern clusters are composed of multiple computer nodes, each of which may be heterogeneous containing various multi-core CPUs and possibly co-processors such as GPUs, APUs and Intel Xeon Phi processors. In addition to the great computing power, such clusters offer low-latency communication channels, such as the InfiniBand network interface, for data movements between nodes. Furthermore, for RDMA-enabled clusters, data transfer across machines is further accelerated since data is transferred directly between main memory and the network card. On top of the hardware improvements, parallel programming interfaces, such as MPI, OpenMP, CUDA and OpenCL, together with some state-of-the-art distributed computing software systems, such as Spark, facilitate the development of applications.

In this paper, we first propose a cross-match algorithm named MASJ-CM and parallelize it using MPI and CUDA on a multi-node CPU-GPU cluster. Our basic idea is to treat the cross-match of two catalogs as a spatial distance join and to replicate the reference candidate objects for each sample object for matching. Specifically, with the positions of celestial objects recorded as points in a spherical coordinate system, an alternative view of cross-matching one reference point with all sample points is a batch of point-in-circle queries. Thus, cross-match of a reference catalog with a sample catalog is essentially a spatial distance join operation and has high degrees of parallelism in both data and computation.

As a naive nested-loop join incurs a vast number of unnecessary pair-wise comparisons, we proposed an index-based cross-match algorithm (IB-CM) in our previous work [41]. IB-CM loads both catalogs in multiple passes and the same sample set is usually sent in the cluster multiple times. To overcome these issues, we propose MASJ-CM for current

SSDBM '16, July 18-20, 2016, Budapest, Hungary

 ISBN 978-1-4503-4215-5/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2949689.2949705

clusters, where the aggregate memory is sufficient to hold both catalogs and the multi-assignment of the reference point set on each node fits into the node's memory.

Different from IB-CM, both catalogs are loaded and distributed in the cluster at the first step in MASJ-CM. We index the sample catalog and calculate the index search intervals for each reference point. Then, instead of filtering sample points directly by the index search intervals, we replicate each reference point and associate each replica with an index in the search range. Due to the nature of astronomical data and indexing, the number of replicas of each reference point is small. By partitioning the entire task and redistributing the data across the machines, the local indexed join on multi-assigned data is done concurrently on each machine. Our evaluation shows that MASJ-CM outperformed IB-CM on all datasets.

In addition to parallelizing the MASJ-CM using MPI and CUDA on clusters, we implemented our MASJ-CM algorithm on Spark [44] under the map-reduce model. The key components, such as multi-assign and join, were implemented via invoking various RDD (Resilient Distributed Dataset) APIs. As operations performed on the data sets were automatically parallelized in Spark, our Spark implementation was much simpler than our MPI-CUDA implementation. Furthermore, even though the Spark implementation ran much slower than the native MPI-CUDA implementation, it offered greater applicability and scalability on various platforms.

Our contributions are summarized as follows. First, we propose the MASJ-CM algorithm and parallelize it to cross-match large catalogs on heterogeneous clusters. It is suitable for main-stream astronomical indexing methods. Second, our parallel MASJ-CM approach based on MPI and CUDA effectively utilizes the computing and communication resources of clusters composed of different types of heterogeneous nodes, whereas our MASJ-CM implementation on Spark is simple and applicable to various Spark clusters. Third, we have optimized our implementations and evaluated them on real astronomical datasets. Our results show that the self-match on a billion-record catalog can be done on our MPI-CUDA implementation under five minutes on a six-node cluster. To our best knowledge, this cross-match performance is the fastest in the literature for such workloads.

The remainder of this paper is organized as follows: section 2 introduces the background of astronomical cross-match and discusses studies related to ours. Section 3 presents the sequential MASJ-CM algorithm and section 4 introduces the details of parallelizing MASJ-CM on a CPU-GPU cluster. Section 5 presents our MASJ-CM implementation on Spark. Section 6 evaluates our work and section 7 concludes this paper.

2. BACKGROUND AND RELATED WORK

In this section we first briefly introduce the background. Then we discuss related work on cross-match and spatial joins. Notations are listed in Table 1.

2.1 Cross-Match of Astronomical Catalogs

The position of a celestial object in an astronomical catalog is usually recorded as a two-dimensional coordinate (α, δ) in the Equatorial Coordinate System [1]. The ranges of α and δ are $[0°, 360°]$ and $[-90°, 90°]$, respectively. Due to the time variation and differences in telescopes and observational purposes, a celestial object may be recorded with slightly different positions in different catalogs. Thus, a distance threshold based on the calibration errors and other considerations, also called search radius θ, is the condition for testing whether two positions are observed on the same celestial object.

Table 1: Notations

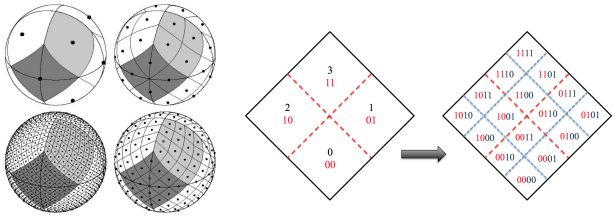
Notation	Meaning
(α, δ)	(Right ascension, Declination) in equatorial coordinate system
θ	Threshold of distance in equatorial coordinate system
$dist_e(O_1, O_2)$	Angular distance of objects O_1 and O_2 in equatorial coordinate system
R, S	Reference, Sample catalog
R_{ma}	Reference catalog after multi-assignment
$C(p, \theta)$	Circle with p as origin and θ as radius
D_{sid}	Diamond cell under HEALPix where sid is the index of this region
$\|D\|$	Total number of diamond cells at current resolution under HEALPix
$d_{(p,\theta)}$	Number of diamond cells that overlap $C_{(p,\theta)}$ under HEALPix
λ	The maximum number of diamond cells that overlap circle $C_{(p,\theta)}$ at a given resolution in HEALPix
b	Total number of bytes of a tuple (α, δ, sid)
N	Number of nodes in a cluster

Given two catalogs R and S, called the *reference* and *sample* catalogs respectively, a search radius θ, and a distance function $dist_e(p, q)$ between two points p and q, the cross-match problem is to find all pairs (p, q) where $p \in R$, $q \in S$ and $dist_e(p, q) \leq \theta$. In database terms, cross-match is a spatial distance join $R \bowtie_\theta S$ on two point datasets R and S.

2.2 Astronomical Indexing Schemes

Zones [38], the Hierarchical Triangular Mesh (HTM) [39] and the Hierarchical Equal Area isoLatitude Pixelisation (HEALPix) [13] are three main-stream sky partitioning/indexing schemes for celestial objects. The common idea behind them is similar with the grid file proposed for canonical spatial problems [29]. That is, for a given resolution, the sky is partitioned into a fixed number of regions, and each celestial object is associated with an integer index of the region that contains the object. A property of these indexing schemes is that, celestial objects that are near to each other in the sky are assigned to the same index or indices of nearby regions. Therefore, finding nearby objects can be done via examining certain neighboring regions. Additionally, neighboring celestial objects can be stored closely on disk or in main memory of a compute device for efficient fetching.

Zones, first proposed by Gray in year 2004 [38], maps the celestial sphere into horizontal zones, each of which is a declination stripe of equal height. Different from Zones, both HTM and HEALPix hierarchically partition the sphere into small spherical triangles and diamond-shaped cells, respectively. HEALPix results in a decomposition of the sphere with equal-area diamond cells whereas HTM partitions the sphere in regions of different sizes.



(a) Subdivision Procedure [13] (b) Hierarchical Numbering Scheme

Figure 1: HEALPix

Both HTM and HEALPix organize indices in a hierarchy of resolutions with the entire sky as the root node and regions at the finest resolution as leaves. They label a newly created region by appending a label from a finite set to its parent node's index. Furthermore, in both HTM and HEALPix, the subdivided regions under the same parent region are named in a nested and clockwise manner, respectively.

In this paper, we choose HEALPix as our indexing scheme in the implementation; our techniques are applicable to the other indexing schemes. Figure 1(a) shows a partitioning process to split the sky from resolution level 0 to level 3. Figure 1(b) shows an example of labeling the indices of the newly subdivided pixels at resolution level 1.

There are various versions of HEALPix implementations, including C, C++, F90, IDL, Python and Java, with public access [2]. The number of cells after partitioning is $12 * 4^l$ at resolution level l. For example, at resolution level 13, the entire sky is partitioned into $12 * 4^{13} = 805,306,368$ cells. Two key functions, *getPix* and *queryDisc*, are used in cross-match. getPix returns the HEALPix index of a celestial object (α, δ) at a given resolution level. queryDisc calculates the list of cells that overlap a query circle with the origin (α, δ) and radius θ. We extracted these two functions from the C source code and implemented our own CUDA C and Scala versions as counterparts.

2.3 Spark

The Apache Spark, which extends the MapReduce model and improves the Hadoop implementation, was proposed for fast and general cluster computing [43]. The ability of executing most computations in memory without writing intermediate results to disks offers a high efficiency. Spark has been used as computing platforms in machine learning [44], as well as in scientific areas, such as neuroscience [12], astronomy [48] and genomics [30].

The core construct in Spark is the RDD, which is a collection of elements and operations that is automatically parallelized. A typical Spark application consists of a driver program, which connects to a computing cluster and operates on the distributed datasets. The driver program mostly starts from creating RDDs by referencing data from an external storage, such as HDFS [36], Amazon S3, or shared file systems, e.g., GlusterFS [8] or Lustre [9]. Then, various RDD APIs are used to operate these RDDs. Transformations and actions are two types of RDD operations. A transformation, such as map, flatMap and filter, defines a new RDD based on a previous RDD. An action, such as count and cache, kicks off a job to execute on a cluster and usually returns a value to the driver program. To get a better knowledge of the operations to be performed, transformations are evaluated lazily until they meet actions.

Table 2: Parallel Cross-Match Performance Summary

Indexing Method	Data Set Size	Evaluation Environment	Execution Time
Zones	2m * 2m	8 SQL servers	20 minutes [27, 28]
	3m * 0.03m	One hybrid MySQL server	7 seconds [21]
	2m * 2m	One single GPU	1.1 seconds [40]
	15m * 450m	6 GPUs on a single node	4 minutes [5, 22]
HEALPix	100m * 470m	One single SQL server	32 minutes [49]
	470m * 1b	One quad-core CPU node	30 minutes [33]
	1.2b * 1.2b	Seven-node CPU-GPU cluster	10 minutes [41]

2.4 Related Work

In this section, we revisit the studies related to ours and classify them into two categories.

2.4.1 *Cross-match of astronomical catalogs*

Gray et al. [38] proposed and implemented the zone-based cross-match with some SQL extensions in a single Microsoft SQL server. Nieto-Santisteban et al. [27, 28] implemented a parallel zones algorithm on multiple servers and cross-match of two million-record catalogs was done under 20 minutes with 8 servers. Kumar et al. [21] extended the parallel zone-based cross-match to a hybrid MySQL cluster and the optimized algorithm was able to cross-match two catalogs with sizes of 3 million objects and 30,551 objects respectively in seven seconds. Most recently, Wang et al. [40] and Budavári [5, 22] parallelized the zones algorithm on a single GPU and a single node with multiple GPUs, respectively. With the aid of GPUs, these methods completed cross-match of million-scale catalogs in a few seconds. For cross-matching partially overlapping catalogs, Fan et al. [11] optimized the original zones algorithm by firstly filtering out irrelevant objects with sky coverage information.

In contrast to the zones-based cross-match, Zhao et al. [49] and Pinearo et al. [33] employed HEALPix as the partitioning scheme and performed cross-match of two catalogs in partitioned regions. The work of Zhao et al. on a single SQL server performed cross-match of SDSS DR6 (100 million objects) and 2MASS (470 million objects) in 32 minutes, whereas that of Pinearu et al. [33] on a machine with two hyper-threaded quad-core CPUs finished cross-match of 2MASS (470 million objects) with USNOB1 (1 billion objects) in 30 minutes. Our previous work [41] took an indexed-loop join approach utilizing the HEALPix index and was able to cross-match billion-record catalogs on a seven-node CPU-GPU cluster under 10 minutes. Table 2 summarizes the performance of these parallel cross-match methods.

In addition to cross-match of two catalogs with distance thresholds, Budavári et al. [6] proposed a Bayesian model to cross-match multiple catalogs while taking physical properties, such as colors, redshift, and luminosity, into account. Most recently in year 2015, Fan et al. [10] employed this

model to cross-match radio catalogs, which contain physical properties of celestial objects other than point coordinates.

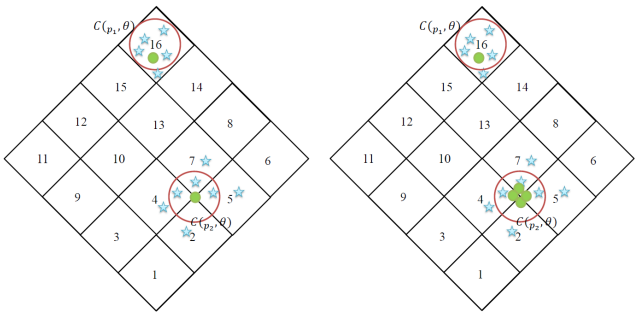
2.4.2 Spatial Joins

Spatial databases usually contain complex spatial objects, such as polygons and polylines. Conventional spatial joins mostly focus on polygon intersection queries. Sequential algorithms are either tree based that require pre-built spatial indices, such as the R-tree [4, 14, 17, 18], R$^+$-tree [23], PMR quadtree [17], or are based on data partitioning [24, 31]. Furthermore, R-tree based methods are essentially single-assignment multi-join (SAMJ) since an object is only assigned to one tree node and a spatial search usually touches multiple tree nodes. In comparison, the R$^+$-tree and grid-file partition based approaches are multi-assignment single-join (MASJ) as spatial objects that span multiple regions are inserted into each of the leaf nodes or replicated into each of the regions. As MASJ incurs object duplication, either duplicate elimination or avoidance mechanism should be adopted.

Parallel spatial join algorithms first partition both relations and then distribute the leaf nodes [19, 20, 35, 16] or partitioned regions [25, 50, 32] across the cluster. Zhou et al. [50] showed that, for parallel spatial join processing,especially for unindexed data, MASJ was superior to the R-tree based SAMJ method. Instead of dynamically inserting objects into the R-tree, the MASJ-based parallel spatial join first statically partitions the space into cells. Spatial objects that span multiple cells are replicated to each of these cells. Second, these cells are merged into partitions, and the partitions are mapped to computer nodes in a parallel machine or a cluster. To handle data skew and balance the workload in the cluster, Zhou et al. [50] used a Z-order merging strategy whereas both Luo et al. [25] and Patel et al. [32] used a hash function to virtually merge cells into partitions. Third, after both relations are partitioned and mapped to computer nodes, the spatial join will be done on each node locally by joining the small cells on the node.

Recently, Zhang et al. [47] parallelized the MASJ-based spatial join on Hadoop. Both relations were split into disjoint regions in the map phase, and each region pair was joined at the reduce stage. You et al. [45, 46] accelerated a Z-order based spatial join on a GPU cluster and an R-tree based indexed spatial join on cloud computing platforms, including Spark and Cloudera Impala [42].

As we focus on parallel cross-match of astronomical catalogs on heterogeneous clusters, our work adopts the MASJ partitioning method but differs from previous work in the following aspects. First, our work deals with celestial objects as points in a spherical coordinate system whereas conventional spatial joins mostly process geospatial datasets in which objects are positioned as polylines and polygons on a quadrant plane. As a result, we do not utilize conventional spatial indexing but rather specialized astronomical partitioning schemes, such as the HEALPix indexing method. Second, instead of replicating both relations, we only replicate the smaller relation, because cross-match is essentially point-in-circle queries. Consequently, the result of MASJ-CM naturally contains no duplicates. Finally, as main memory size grows and celestial objects are represented as two-dimensional coordinates in our system, the replicated coordinates in each partition fit into main memory of each cluster node and the final join phase can be done in-memory and requires no extra communication.



(a) Before Multi-Assignment (b) After Multi-Assignment

Figure 2: Multiple Assignment Example

3. THE MASJ-CM ALGORITHM

In this section, we first present our sequential MASJ-CM algorithm for the cross-match problem. Then, we estimate the workload of the join and introduce our workload-driven task decomposition scheme for parallel processing. MASJ-CM requires both catalogs are loaded once into the cluster to have a global overview of the data distribution. Thus, MASJ-CM has a hard requirement for the scale of cluster for offering enough main memory. This requirement is feasible to satisfy on current clusters, as each node typically contains tens of gigabytes of memory.

We use the HEALPix as the partitioning method. In Figure 2(a), dots represent reference points and stars represent sample points. Circle $C_{(p_1,\theta)}$ is contained in diamond D_{16} whereas circle $C_{(p_2,\theta)}$ overlaps the diamond set $CL[p_2] = \{D_2, D_4, D_5, D_7\}$. Thus, sample points inside $C_{(p_1,\theta)}$ and $C_{(p_2,\theta)}$, i.e., the stars in Figure 2(a), are contained in diamond D_{16} and diamond set $CL[p_2] = \{D_2, D_4, D_5, D_7\}$, respectively. Consequently, if a query point p is assigned to all diamond cells that the query circle $C_{(p,\theta)}$ overlaps, e.g., dots in Figure 2(b), cross-match can be done in all diamond cells concurrently. Moreover, as multiple assigned reference points can be associated with the spatial indices of the overlapping cells, cross-match can be done via joining all points with an equal index and within distance θ.

3.1 Sequential MASJ-CM

Our sequential MASJ-CM algorithm proceeds as follows.

1. Multi-assignment: For each $p = (\alpha, \delta) \in R$, let $CL[p] = \{D_{sid_1}, D_{sid_2}, ...D_{sid_{d_{(p,\theta)}}}\}$ ($|CL[p]| = d_{(p,\theta)}$) denote the list of diamond cells that the query circle $C_{(p,\theta)}$ overlaps. We duplicate p into the $d_{(p,\theta)}$ cells while inserting the spatial index into the original tuple. Specifically, we duplicate p as $p_1 = (\alpha, \delta, sid_1)$, $p_2 = (\alpha, \delta, sid_2)$, ..., $p_{d_{(p,\theta)}} = (\alpha, \delta, sid_{d_{(p,\theta)}})$. Let R_{ma} denote the R set after this multi-assignment step.

2. Indexing sample catalog: For each $(\alpha, \delta) \in S$, insert the spatial index sid_i into the corresponding tuple such that $q = (\alpha, \delta, sid_i)$ where cell D_{sid_i} contains (α, δ).

3. Single-join: Compute $R_{ma} \bowtie_{(sid,\theta)} S$ to find point pairs having the same sid and within a distance threshold θ.

We discuss three issues of MASJ-CM.

- Upper bound of $|R_{ma}|$: Each $p \in R$ will be duplicated through multi-assignment in the first step. Given the maximum number of cells that the query circle $C_{(p,\theta)}$ overlaps, we have $|R_{ma}| \leq \lambda|R|$.

- Memory Cost: Since in-memory cross-match requires both catalogs stored in memory simultaneously at some moment, storing R_{ma} and S simultaneously needs up to $b \cdot (\lambda|R| + |S|)$ memory space where b is the total number of bytes of a tuple (α, δ, sid). In our implementation, b equals 20 as *sid* costs 4 bytes and point tuple (α, δ), which are two double floating numbers, costs 16 bytes.

- Choice of reference catalog: As in other canonical join operations, choosing which catalog as the reference in cross-match will not make any difference in the final join result. In our MASJ-CM algorithm, multi-assignment only occurs on one catalog. So we take the smaller catalog as the reference to reduce the memory, computation and communication cost.

3.2 Workload Estimation in Step 3

To parallelize MASJ-CM, we first estimate the workload of the third step for partitioning. The first two steps have an even workload on each data point, so there is no need to estimate their workload. Since in the first two steps, all points in both R_{ma} and S are indexed, step 3 can be done as a sort-merge or a hash join on the index *sid* followed by the final matching with the distance threshold. As hash join requires extra memory space for computing the histogram, we choose sort-merge to join the two sets on a single machine.

We use the number of angular distance calculations to estimate the workload in the third step, as these calculations are the most computation-intensive component in the algorithm. Since both R_{ma} and S are indexed and sort-merge join is done on the index *sid*, the angular distance calculation occurs only when two points from R_{ma} and S have an equal index. Let $N_R(i)$ and $N_S(i)$ denote the number of points with spatial index sid_i in R_{ma} and S, respectively. The total number of angular distance calculations for MASJ can be expressed in

$$Cost_{MASJ} = \sum_{i=0}^{|D|-1} N_R(i) * N_S(i). \quad (1)$$

The entire task CM_{MASJ} is to compute the cross-match of each region pair.

$$CM_{MASJ} = \cup_{i=0}^{|D|-1} CM(R_i, S_i) \quad (2)$$

3.3 Task Decomposition Scheme in Step 3

A straightforward approach to execute step 3 in parallel on commodity clusters is to assign a task set $CM(i, ...j) = CM(R_i, S_i) \cup ... \cup CM(R_j, S_j)$, $0 \leq i \leq j < |D|$, the union of multiple regional tasks, to a single node and then let all nodes execute the tasks assigned to them concurrently. As all operand data in a regional task, e.g., $CM(R_i, S_i)$, will be held entirely by a node, this task can be executed locally and no extra communication is needed. Thus, this approach is especially suitable for a shared nothing architecture, where different computer nodes can only exchange data via network. In such an architecture, in addition to minimizing communication cost, it is critical to maintain workload balance among multiple nodes. Therefore, we tackle the problem of workload balance in the presence of data skew.

As in many spatial problems, data skew is not uncommon in astronomical data. Due to the uneven distributions of celestial objects and the differences in observational purposes of astronomical missions, the number of celestial objects that fall into a region under a certain partitioning scheme, varies from region to region. Thus, the number of celestial objects with equal spatial index in both catalogs, e.g., $N_R(i)$ or $N_S(i)$, varies. Consequently, the scale of each regional task, e.g., $CM(R_i, S_i)$, varies.

To prevent load imbalance, regional tasks of various scales should be merged into a few larger tasks of similar scales. This problem is equivalent to the bin-packing problem, which is NP-hard. As such, we adopt a greedy workload-driven task decomposition scheme that merges regional tasks into larger ones in a bottom-up approach, aiming at balancing the load.

First, we set $COST_D = Cost_{MASJ}/N$ as the default workload of each computer node where N is the number of computer nodes in a cluster. Then, we pack points in R_{ma} and S with equal spatial index into bucket pairs. We further associate each bucket pair with the corresponding index. For example, the i-th bucket pair $(B_R(i), B_S(i))$ is

$$B_R(i) = \{p | p \in R_{ma}, p.sid = sid_i\} \quad (3)$$

$$B_S(i) = \{q | q \in S, q.sid = sid_i\} \quad (4)$$

where $|B_R(i)| = N_R(i)$ and $|B_S(i)| = N_S(i)$.

Then we sort all bucket pairs by their indices. Next, we start to merge as many consecutive bucket pairs in the sorted order into a pair of larger buckets (BR, BS) until the default workload is reached. Thus, for the s-th large pair (BR_s, BS_s), we have

$$BR_s = \cup_{k=i}^{j} B_R(k) \quad (5)$$

$$BS_s = \cup_{k=i}^{j} B_S(k) \quad (6)$$

$$Cost_s = \sum_{k=i}^{j} N_R(k) * N_S(k) \leq COST_D \quad (7)$$

where $Cost_s$ is the estimated workload of the (BR_s, BS_s). If there are any small bucket pairs left after N large bucket pairs are formed, we assign the remaining pairs, one by one, into the large bucket pair with the smallest estimated workload and update its estimated workload. For example, we assign bucket pair $(B_R(i), B_S(i))$ to the j-th large pair and increase $Cost_j$ as

$$Cost_j = Cost_j + N_R(i) * N_S(i) \quad (8)$$

Finally, we assign each large bucket pair as a task set to a computer node and the entire task decomposition is done.

4. THE MPI-CUDA IMPLEMENTATION

In this section, we describe a three-phase framework to execute the MASJ-CM algorithm in parallel in a multi-node cluster. Section 4.1 presents an overview of this framework and the remainders discuss the implementation details.

4.1 Overview

Initially, both catalogs are split into N small files, each of which contains roughly an equal number of celestial objects. Each node loads one reference and one sample file at the beginning of the execution. We assume the data processed by each node can fit into its main memory during the entire processing. Our three-phase framework works as follows.

1. Preprocessing and task decomposition: Each node p_i, $0 \leq i < N$, starts from loading the corresponding reference and sample catalogs into its main memory. Then each node calculates the list of overlapping cells for each reference point, and the spatial index for each sample point, respectively. In the meantime, each node counts the number of reference and sample points that fall into each cell. Finally, one of the computer nodes p_0 gathers the statistics from all the others, merges them and runs the task decomposition algorithm proposed in section 3.3 based on the global statistics. At the end of this phase, p_0 distributes the task decomposition results to all other nodes.

2. Data redistribution: All nodes in the cluster exchange data via the network so that each node loads its assigned partition of data required by the third phase into the node's main memory.

3. Parallel single join: Each node performs an in-memory sort-merge join locally on the index, computes the distance of two points with the same index, and compares with the distance threshold to determine whether it is a match.

In this framework, the entire processing consists of three phases with each phase consisting some sequential steps. Sequential execution has the following problems.

1. Dependence: A step cannot start until the previous step is done even if there is no dependence between them. Moreover, if two steps without dependence are to be executed on the GPU and the CPU concurrently, sequentially executing them will make the GPU or CPU idle.

2. All-to-All communication: Communication among all nodes are required in the second phase for redistributing data. The network latency and the bandwidth limitations affect the performance of this phase. Since the third phase cannot start before all required data is held by each node, the high communication cost may be the bottleneck for achieving a high efficiency when cross-matching large-scale catalogs.

In the following we discuss our optimization strategies, which mainly involve overlapping operations.

4.2 Phase 1: Task Decomposition

Algorithm 1 lists the pseudo-code executed by all nodes in this phase. Each node starts from loading one reference and sample file into its main memory (line 1-2). Then for the in-memory point set R and S, operations performed on them are different. For R, the node calculates the list of overlapping cells for each point based on the search radius (line 6). Then it continues to count the number of points that fall into each cell based on CL (line 7). For S, the node calculates the indices for all points in parallel via OpenMP (line 9-13) and then counts the number of sample points that fall into each cell (line 14). As seen in the algorithm, operations performed on R and S are independent. Furthermore, the most computation-intensive component GPU-ComputeCL (kernel function ComputeCL) is executed on the GPU. Therefore, operations performed on R (line 6-7) and S (line 9-14) can be in parallel. To do this, we first enable the openMP nested parallelism by calling *omp_set_nested()* (line 3). Then, we parallelize these two components by assigning two OpenMP threads to execute them concurrently. After all nodes get the statistics N_R and N_S, we designate a node p_0 to gather the results from all the others via *MPI_Gather()* (line 15-16). Then the designated node runs the task decomposition algorithm and broadcasts the decomposition result $Task\_List$ to all other nodes via *MPI_Bcast()*.

Algorithm 1: MASJ-CM Phase1(R_{file}, S_{file}, θ)

```
   Input: R_file: Reference file
   Input: S_file: Sample file
   Input: θ: Search radius
   Output: ResultList: List of matching pairs
1  R ← R_file ;
2  S ← S_file ;
3  omp_set_nested(1) ;
4  #pragma omp parallel num_threads(2)
5      if omp_get_thread_id() == 0 then
6          GPU-ComputeCL(R, CL, θ) ;
7          count(CL,N_R) ;
8      else
9          #pragma omp parallel for
10             for i ← 0 to |S| do
11                 sid ← getPix(S[i]) ;
12                 insert sid to S[i] ;
13                 i ← i + 1 ;
14         count(S,N_S) ;
15 MPI_Barrier() ;
16 MPI_Gather(N_R, |D|, ..., p0) ;
17 if MASTER_NODE then
18     Task_List ← decompose entire task based on N_R
       and N_S ;
19 MPI_Bcast(Task_List, ..., p0) ;
```

Algorithm 2 lists the pseudo-code for computing the list of overlapping cells for each reference point. We parallelize this phase by invoking all available GPU cards concurrently and assigning GPU threads for processing. We partition R into chunks with each of them fit into the global memory of a single GPU. By invoking CUDA API *cudaGetDeviceCount()*, we get the number of GPUs that are available for execution (line 1). After setting the number of CPU threads as GPU_{num} via *omp_set_num_threads()* (line 2) and choosing a GPU device via *cudaSetDevice()* (line 5), all GPU cards are invoked concurrently to execute the kernel *computeCL*. In ComputeCL, each GPU thread processes multiple reference points. For one point $R[i]$, the thread invokes function *queryDisc* to calculate the list of overlapping cells $CL[i]$ based on the distance threshold θ. $CL[i]$ takes the form $CL[i] = \{D_1, D_2, ...D_n\}$ where each of them specifies the id of a cell that overlaps its query circle $C_{(R[i],\theta)}$.

At the end of Phase 1, each node has the subset of the

Algorithm 2: GPU-ComputeCL(R, CL, θ)

Input: R: Reference Point Set
Input: CL: List of Overlapping Cells
Input: θ: Search Radius
1 $GPU_{num} \leftarrow$ cudaGetDeviceCount() ;
2 omp_set_num_threads(GPU_{num}) ;
3 **#pragma omp parallel**
4 $GPU_{id} \leftarrow$ omp_get_thread_num() % GPU_{num} ;
5 cudaSetDevice(GPU_{id});
6 cudaDeviceReset() ;
7 **for** $i \leftarrow 0$ *to* $|R|/(GPU_{num}$ * *chunk_size)* **do**
8 $R_d \leftarrow (i + \frac{GPU_{id} * R}{GPU_{num}})$-th chunk ;
9 invoke kernel ComputeCL(R_d, θ, CL) ;

original reference and sample catalog in its main memory. Furthermore, each sample point has its index, and each reference point has its overlapping cell list stored in $CL[i]$.

4.3 Phase 2&3: Redistribution and Single-Join

Phase 2 is the communication phase in which the reference and sample point subset are redistributed to each node according to $Task\_List$. Specifically, each node gathers all reference and sample points whose spatial indices fall into the range assigned to it. In the meantime, each node scatters the data in its main memory to the corresponding destination nodes. After scattering and gathering are done, each node starts to execute the computation phase (phase 3) to perform an in-memory sort-merge join on its gathered reference and sample point sets.

For node p_i $(0 \leq i < N)$, let $S^* = \cup_{j=0}^{N-1} S_j^*$ and $R^* = \cup_{j=0}^{N-1} R_j^*$ denote the final gathered sample and reference point sets whose spatial indices fall into the range assigned to it, respectively. S_j^* (or R_j^*) is the set gathered from node p_j.

We can redistribute the sample point set as the spatial indices are already added to the sample point tuples in phase 1. Moreover, the total amount of data is not increased heavily since for each tuple, we only add an integer to it. To gather the sample point set, each node only needs to allocate a buffer for S^* and gather the required data by invoking *MPI_Recv()* with different sources. To scatter the required sample set, each node packs the sample points that fall into the certain range and scatters them to the corresponding destination node. We do this via an in-place sort operation on S by spatial index first and then send points to a destination node by *MPI_Send()*.

We redistribute the reference point set, two buffers for R^* and R_{ma} are required by each node. B^* is the buffer for gathered points. B_{ma} is the buffer for the storage of multi-assigned R. Firstly, each node follows the first step of MASJ algorithm to produce R_{ma}. For each tuple $p \in R$, we make $d(p)$ duplicates and for each tuple we insert the corresponding spatial index *sid* into it.

The redistribution of reference point set and the single join can be executed as a whole. We consider the following three options.

Option 1 One-Time Redistribution and Single-Join Each node allocates two buffers, one for R^* and the other for R_{ma}, with enough space to gather and scatter the entire data set. Then all nodes calculate $R^* \bowtie_{(sid,\theta)} S^*$ in one pass. This option works well when cross-matching two small catalogs. When it comes to large catalogs, for example, self-matching a billion-record catalog SDSS, dozens of gigabytes of memory are required. Under such situation, some memory space may be allocated as swap memory that incurs high latency in network communication and memory access.

Option 2 Pipelined Redistribution and Single-Join We pipeline the redistribution and single-join in iterations. In one iteration, every two nodes exchange their data and then join the entire set S^* with their partially gathered R^*. This way, the calculation of $R^* \bowtie_{(sid,\theta)} S^*$ is converted into $\cup_{i=0}^{N_M-1} R_i^* \bowtie_{(sid,\theta)} S^*$. In one iteration, each node only needs to allocate two small buffers for gathering partial R^* and scattering partial R_{ma}, respectively. Compared with option 1, option 2 requires less memory space and suits for larger catalogs.

Option 3 Overlapping Redistribution and Single-Join

To reduce the communication overhead, we further optimize option 2 to option 3. The main idea is to allocate extra buffers for overlapping the communication and computation. Our experiments demonstrate that this option can cut the total execution time of phase 2&3 nearly by half.

Algorithm 3: Single-Join(R, S, θ, resultList)

Input: R: Reference Point Set
Input: S: Sample Point Set
Input: θ: Search Radius
Output: resultList: List of matching pairs
1 run tbb:sort_by_key on R by sid;
2 $chunk\_num \leftarrow \frac{|S|}{chunk\_size}$;
3 $chunk\_id \leftarrow 0$;
4 $GPU_{num} \leftarrow$ cudaGetDeviceCount() ;
5 omp_set_num_threads(GPU_{num}) ;
6 **#pragma omp parallel**
7 $GPU_{id} \leftarrow$ omp_get_thread_num() % GPU_{num} ;
8 cudaSetDevice(GPU_{id}) ;
9 cudaDeviceRest() ;
10 **while** $chunk\_id < chunk\_num$ **do**
11 **#pragma omp atomic**
12 $chunk\_id++$;
13 $S_d \leftarrow S_{chunk\_id}$;
14 $R_d \leftarrow$ getMatching(R, S_d) ;
15 invoke kernel sort-merge(S_d, R_d, θ, resultList) ;

4.4 Single Join

The outline of single join is listed in Algorithm 3. The basic idea to join two sets in parallel on multiple GPUs is to perform the sort-merge join on a chunk of S and its matching chunk in R concurrently. Suppose S is split into *chunk_num* chunks (*chunk_num* $= |S|/$*chunk_size*, *chunk_size* is the size of each chunk and $S = \bigcup_{i=1}^{Q} S_i$). Then, the spatial index *sid* of the first and the last tuples of each chunk S_i are used to identify the corresponding matching chunk R_i. Each time, one GPU loads a chunk pair (one sample chunk S_i and its matching chunk R_i) into its device memory and invokes the kernel program *sort-merge* to join this chunk pair (line 13-15). If this chunk pair is too large to be loaded simultaneously, the reference chunk is split into smaller ones and loaded in multiple passes. After this chunk pair is merged, the GPU continues to load a new chunk pair until all chunks are merged. Note that the *chunk_id* is a value shared by all CPU threads. Once a GPU processes a new chunk pair, an

Listing 1: Driver program of MASJ-CM Spark Implementation

```
val sc = new SparkContext(conf)
/* load reference and sample catalog, then preprocess them */
val refCatalog = sc.textFile(reference directory).map(x => x.split(" ")).map(x => (x(0).toDouble, x(1).toDouble))
val samCatalog = sc.textFile(sample directory).map(x => x.split(" ")).map(x => (x(0).toDouble, x(1).toDouble))
/* multi-assign the reference point set */
val refSet = refCatalog.flatMap(x => healpix_func.multiAssign(x,radius))
/* index the sample point set */
val samSet = samCatalog.map(x => (healpix_func.getPix(x._0,x._1),(x._0,x._1))
/* cross-match two sets */
val matchedSet = refSet.join(samSet).filter(x => Geometry.matched(x._2._1, x._2._2,radius))
/* save the result as text file */
matchedSet.saveAsTextFile("result.txt")
```

atomic add operation is performed on *chunk_id* (line 11-12) to make sure each GPU processes a unique chunk pair.

The implementation of the sort-merge kernel follows the approach proposed by He et. al [15]. Specifically, each sample chunk is split into smaller ones, and each thread block is responsible for one of the smaller chunks. Moreover, each thread is responsible for one or more sample points. Then the indices of the first and last tuples of each chunk are used to identify the corresponding reference chunk. Then the reference chunk is loaded into the on-chip shared memory in multiple passes for fast access. In one pass, each thread performs an indexed nested-loop search or binary search to find all matching reference points.

5. THE SPARK IMPLEMENTATION

We implemented the MASJ-CM algorithm as a map-reduce program using Spark RDD APIs. The equivalent operations of the three steps in MASJ-CM are flatMap, map, join followed by filter, respectively. Listing 1 lists the code skeleton of the driver program of the MASJ-CM Spark implementation. For readability purpose, the RDD APIs provided by the Spark library are in bold. At the beginning, the driver program creates a value *sc*, which is a SparkContext object and represents a connection to a computing cluster. Then, the driver program builds two RDDs, refCatalog and samCatalog, by loading reference and sample catalogs through sc. In the meantime, it converts each element in the RDDs as an (ra, dec) coordinate pair using two map operations. After the preprocessing step, the driver program multi-assigns each reference point. Specifically, it passes a user-defined function, healpix_func.multiAssign, to the flatMap operation and transforms the refCatalog to a pair RDD where each element takes the form $(sid, (ra, dec))$. Similarly, the driver program indexes the sample point set by transforming samCatalog with a map operation. Finally, the cross-match is done by performing an inner join between two RDDs and filtering the matched pairs with a filter operation on the distance threshold.

Compared with parallelizing the MASJ-CM algorithm using MPI, OpenMP and CUDA on a CPU-GPU cluster, the equivalent Spark implementation is quite concise. Furthermore, the MASJ-CM Spark program is able to run on various platforms, including Hadoop, EC2, Mesos, standalone or in the cloud. As parallelizing and distributing collection methods and datasets to multiple nodes can be done automatically, Spark makes the parallelization effortless to a certain extent and offers fault-tolerance and scalability at the same time. For example, the source code of our Spark and MPI-CUDA based implementations is 498 and 2005 lines, respectively. Furthermore, in the Spark implementation, 94% of the source code are HEALPix related whereas this percentage is only 41.4% in the MPI-CUDA approach.

However, there are two shortcomings that greatly slow down the Spark-based implementation's performance. First, the two key functions used in the driver program, getPix and queryDisc, are implemented in Scala, which are less efficient than their equivalent C implementations. Second, the Spark implementation does not utilize the GPU, which processes most computation intensive tasks in the MASJ-CM MPI-CUDA implementation. Overcoming these problems is interesting future work.

6. EVALUATION

In this section, we first describe the experimental setup and then present our experimental results.

6.1 Experimental Setup

We carried out our experiments on a heterogeneous cluster maintained at our university. The entire cluster consists of 22 nodes of which 16 are equipped with multiple GPUs. Considering the purpose of our evaluation and available resources, we employed a 7-machine cluster (Cluster-IB) and 6-machine cluster (Cluster-MASJ) to evaluate the IB-CM and MASJ-CM algorithm, respectively. The difference between Cluster-IB and Cluster-MASJ is that Cluster-IB has a CPU-only node as the coordinating master node whereas Cluster-MASJ contains no CPU-only node. All nodes, except the CPU-ONLY node, are equipped with multi-core CPUs and multiple GPUs. Furthermore, all GPUs are the NVIDIA Tesla cards with compute capability 3.5 while the device memory varies from card to card. Table 5 shows the specifications of all nodes and the configuration of two clusters used in our evaluation.

The operating system that ran on the entire system was CentOS 6.5 (64-bit version). CUDA 6.5 was used to compile all CUDA programs. The -O3 optimization option was enabled in compilation. Intel MPI 5.0.3 library was used for MPI execution. *tbb_parallel_sort* provided by the Intel(R) Threading Building Blocks library [34] was invoked by the CPU to perform all sort operations. *gettimeofday()* was invoked by the CPU to measure the elapsed time. Spark 1.6.0 was used in our Spark implementation.

We evaluated our implementation on three real-world datasets. All of them are point source catalogs observed by optical telescopes. The detailed description of each dataset is listed in Table 3. 2MASS (short for Two Micron All Sky Survey) [37] and WISE (short for Wide-field Infrared Survey Explorer) [7] are two million-record catalogs from scanning

Table 3: Data Sets

Data Set	Description	File Size	# of Objects
2MASS	$\alpha \in [0°, 360°]$ $\delta \in [-89.9928°, 89.9901°]$	$7.02GB$	$470,992,970$
WISE	$\alpha \in [0°, 360°]$ $\delta \in [-89.9946°, 89.9983°]$	$11.14GB$	$747,634,026$
SDSS	$\alpha \in [0°, 360°]$ $\delta \in [-17.7573°, 84.9799°]$	$18.34GB$	$1,231,051,050$

Table 4: Comparison of Source Code Size in Two Implementations (Measured in lines)

Component	MPI-CUDA	Spark
Main/Driver program	1128	38
getPix	54	63
queryDisc	207	138
HEALPix Initialization	616	259
Total	2005	498

the entire sky. SDSS, which is short for Sloan Digital Sky Survey, is a survey that covers more than one third of the entire sky. The twelfth data release of SDSS [3] used in our evaluation, a billion-record catalog, is one of the largest astronomical catalogs that are publicly accessible.

Table 4 lists the details about source code size in two implementations. The HEALPix related source code has a percentage of 43.7% and 92.3% in the MPI-CUDA and Spark implementation, respectively. Due to the lack of support to element collections in CUDA, such as queue and stack, the HEALPix related source code in CUDA was almost twice the size of that in Spark. The portion of the driver program, including loading data set, MPI communication, kernel invocation and calculation, was up to 56.3% in the MPI-CUDA implementation. In comparison, in Spark implementation, the driver program was 38 lines and the portion 7.7%. Consequently, Spark greatly simplified the implementation, especially when the indexing related APIs were provided.

Table 6 lists an overview of the nine test cases designed for our evaluation. Three of them (T1-T3) are self-matches on the same catalog. The remainders (T4-T9) are cross-matches on two different catalogs. The cases are described in the form of "Reference catalog * Sample catalog" in all figures. In all cases, the number of matching sample objects for each reference object depends on the distance threshold. The distance threshold was set to $0.0056°$, a value suggested by the astronomer. The resolution level of HEALPix partitioning scheme was set to 13, the maximum supported resolution level with 32 bits. The average number of $d(p)$, the number of cells overlapping a query circle of radius $0.0056°$, was roughly 7.76 under resolution level 13. We ran each experiment five times and report the best run. The variation of all runs was low ($< 7\%$). The estimated peak size of main memory consumed during the execution of MASJ-CM MPI-CUDA is also reported. All matching results were checked by the astronomer to guarantee the correctness.

6.2 Performance Results

We show the overall performance in Table 6. First, the MPI-CUDA implementation, MASJ-CM outperforms IB-CM in all cases and achieves a speedup of $1.33 - 2.69$. However, the equivalent MASJ-CM Spark implementation is 16.x slower than the MPI-CUDA implementation. Second, when cross-matching with a large catalog(T6-T9), taking the smaller catalog as the reference catalog had a better performance in both implementations.

To study the impact of the data volume, we extracted small data sets from the SDSS with fixed number of objects and executed self-matches on the extracted data sets. Figure 3 shows that the execution time increases linearly with the increase of data volume in each catalog in the MPI-CUDA implementation. The MASJ-CM spark performance degraded when the number of objects was greater than 960 million.

Next, we continue to investigate the MASJ-CM's time performance in the MPI-CUDA implementation. As shown in Figure 5, the time taken in phase 1 in each case was $24.28\% - 35.53\%$ and the merged phase 2&3 dominated the performance in all cases. In three self-matches T1 to T3, the overall execution time increases linearly with the increases of data volume. When cross-matching two different catalogs, T4, T6 and T8, which took the smaller catalog as the reference catalog, outperformed T5, T7 and T9, which took the larger catalog as the reference, respectively.

Finally, we study the time breakdown of IB-CM in the MPI-CUDA implementation in Figure 4 to understand the performance advantage of MASJ-CM. The time cost of iteration 1, in which the master node loaded the sample catalog from hard disk, was $7.68\% - 17.37\%$. The time taken in the remaining iterations, in which the master node sent sample catalog in the cluster multiple times, was $82.63\% - 92.32\%$ and up to 2.42 times longer than the overall execution time of MASJ-CM. Thus, eliminating sending sample catalog repeatedly can greatly improve the cross-match performance.

6.2.1 Phase 1 Performance

Having the performance overview, we next investigate the performance of the first phase. Figure 6 show the elapsed time of this phase. The I/O cost took $13.48\% - 31.24\%$ of this phase and $4.23\% - 8.0\%$ of the overall execution procedure. Thus, loading large volumes of data did not significantly increase the total execution time. The index computation, which processes the loaded reference and sample catalogs concurrently, is the major time consumer of this phase. Different from the I/O cost and index computation, the time cost for task decomposition remained constant in all cases since it only depends on the resolution level and the number of nodes in the cluster. Furthermore, the group (T4,T6,T8) that took the smaller catalog as the reference catalog achieved a better performance than their equivalents that used the larger catalog as the reference in this phase, mainly because of the less execution time of the step *GPU-ComputeCL*.

6.2.2 Phase 2&3 Performance

After investigating the performance of phase 1, we next evaluate the performance of phases 2&3.

Redistributing and sorting sample point set Phase 2&3 starts from redistributing the sample point set in the entire cluster. Then each node performs a sort operation on its sample point set for the following sort-merge join. Figure 7 shows the time cost for redistributing and sorting sample point set. In self-matches (T1-T3), the time cost increased with the increasing volume of sample point set.

Table 5: Multi-GPU Cluster Configuration

Node Name	Configuration	Processor Name	Clock (Hz)	# of Cores	Cache/Shared Memory	Main/Device Memory
CPU-ONLY	4 * CPU	Intel E5-2650 v3	2.30G	4 * 10	640KB(L1) 2.5MB(L2) 25MB(L3)	64GB
2×K20	2 * CPU	Intel E5-2650 v2	2.20G	2 * 10	512KB(L1) 2MB(L2) 20MB(L3)	64GB
	2 * GPU	Tesla K20	706M	2 * 2496	48KB	2 * 5GB
2×K20x	2 * CPU	Intel E5-2650 v2	2.20G	2 * 10	512KB(L1) 2MB(L2) 20MB(L3)	64GB
	2 * GPU	Tesla K20x	732M	2 * 2688	48KB	2 * 6GB
2×K40	2 * CPU	Intel E5-2650 v2	2.20G	2 * 10	512KB(L1) 2MB(L2) 20MB(L3)	64GB
	2 * GPU	Tesla K40	745M	2 * 2880	48KB	2 * 12GB

Cluster Name	Node	Connection
Cluster-IB	Master: CPU-ONLY, Worker Nodes: Three 2×K20 nodes, two 2×K20x nodes, one 2×K40 node	4×QDR InfiniBand 40Gbit/s
Cluster-MASJ	Three 2×K20 nodes, two 2×K20x nodes, one 2×K40 node	

Table 6: Cross-Match Overall Performance
(Time in minutes, memory in gigabytes)

Test Case	Reference Catalog	Sample Catalog	Average Matched Points (For Each Reference Point)	IB-CM MPI-CUDA	MASJ-CM MPI-CUDA		MASJ-CM Spark
				Execution Time	Execution Time	Estimated Peak Memory Consumption Per Node	Execution Time
T1	2MASS	2MASS	5.07	1.9	1.62	11.4	16.94
T2	WISE	WISE	2.51	4.4	2.53	18.1	32.52
T3	SDSS	SDSS	30.21	12.8	4.77	29.81	80.99
T4	2MASS	WISE	2.46	2.96	1.95	12.44	18.75
T5	WISE	2MASS	1.55	3.22	2.27	17.07	30.14
T6	2MASS	SDSS	1.93	5.83	2.88	14.24	47.52
T7	SDSS	2MASS	0.86	4.82	4.07	26.98	68.38
T8	WISE	SDSS	4.95	8.1	3.82	19.90	64.83
T9	SDSS	WISE	1.94	6.2	4.67	28.91	78.41

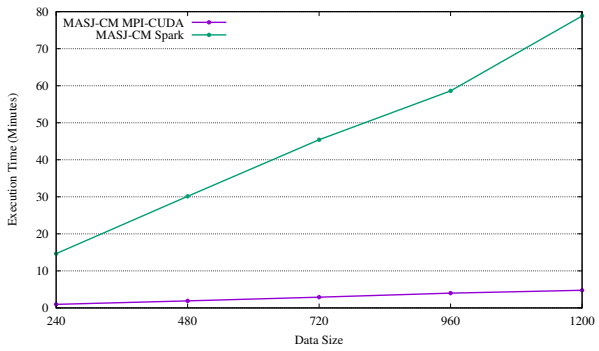


Figure 3: Performance with different number of objects in each catalog (in million).

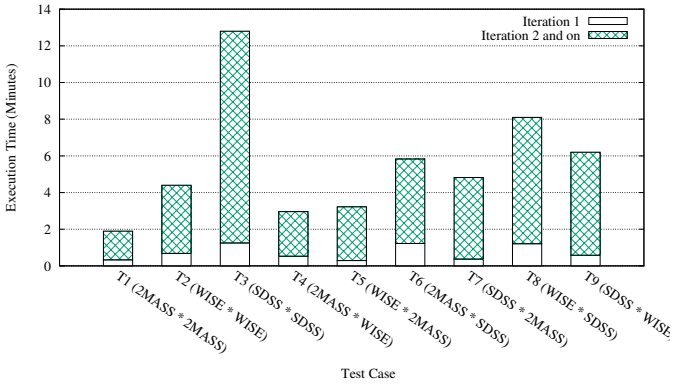


Figure 4: Time breakdown in IB-CM MPI-CUDA.

Cross-matches that took the larger catalog as the sample (T4, T6, T8) cost a few seconds more than the opposite cases (T5, T7, T9).

Redistributing reference point set and single join We now examine the performance of these last two steps. Figure 8 shows the overall speedups of our optimization strategies. The comparison of option 1 and 2 shows that pipelining the communication and computation achieved 0.96x-1.34x speedups. Then the comparison of option 2 and 3 shows that the overlapping strategy brings 1.24x-1.75x speedups. Consequently, option 3 results in the least execution time in all test cases.

7. CONCLUSION

We presented a multi-assignment single-join cross-match algorithm and its parallelization on heterogeneous clusters for cross-matching billion-object astronomical catalogs. The MASJ-CM outperformed our previous work IB-CM in matching speed but required both catalogs to fit into the aggregated memory of the cluster and the multi-assigned reference point set of each node to fit into the main memory of the node. Fortunately, due to the characteristics of astronomical data and the indexing scheme, these memory requirements are readily satisfied on current clusters. Our results shown that, the MPI-CUDA implementation of MASJ-CM achieved a speedup of 2.69 times over IB-CM when self-matching SDSS on a six-node CPU-GPU cluster. Furthermore, self-matching a billion-record catalog completed under 5 minutes with the MASJ-CM on the six-node cluster. To

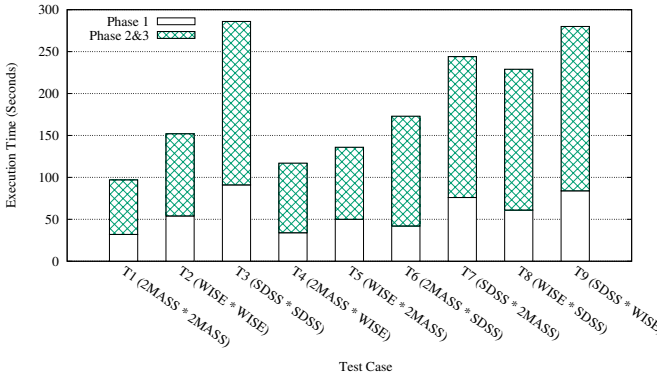


Figure 5: Time breakdown in MASJ-CM MPI-CUDA.

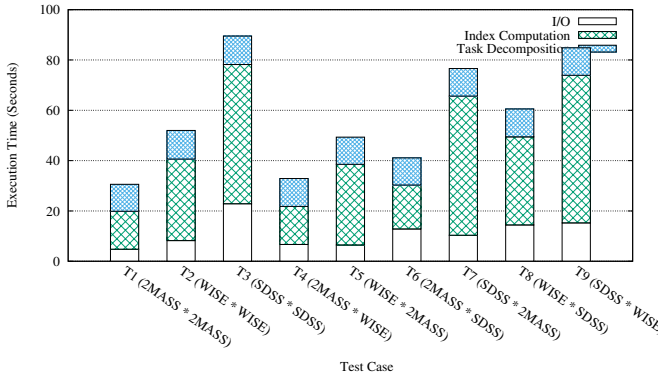


Figure 6: Phase 1 time breakdown in MASJ-CM MPI-CUDA.

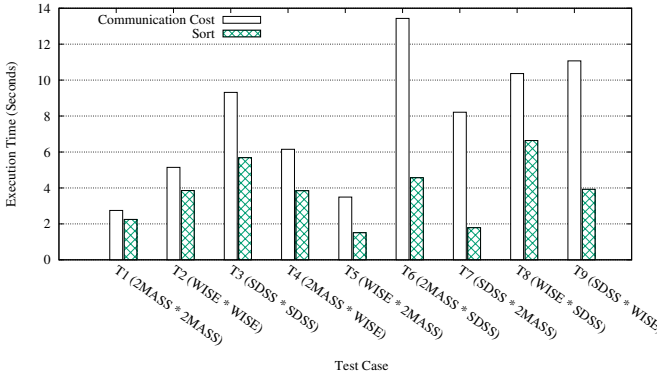


Figure 7: Time cost for sample catalog in phase 2 in MASJ-CM MPI-CUDA.

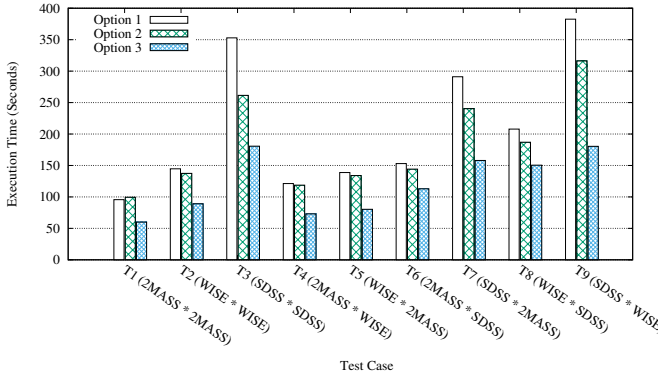


Figure 8: Comparison of three options in phase 2&3 in MASJ-CM MPI-CUDA.

our best knowledge, this work is the first in the literature to achieve such performance for such billion-record workloads. Additionally, the Spark-based implementation of MASJ-CM was less efficient than the MPI-CUDA implementation, but it greatly simplified the programming and reduced the code size by 75%.

8. ACKNOWLEDGEMENT

The authors would like to thank Dr. Dongwei Fan and China-VO for access to their data sets and sharing astronomic domain knowledge. This work was supported by grants 616012 and 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft SQL Server China R&D.

9. REFERENCES

[1] Equatorial coordinate system. https://en.wikipedia.org/wiki/Equatorial-coordinate-system.

[2] Healpix home page. http://healpix.sourceforge.net.

[3] Sdss dr12. http://www.sdss.org/dr12/scope/.

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using r-trees. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, 1996.

[5] T. Budavari and M. A. Lee. Xmatch: Gpu enhanced astronomic catalog cross-matching. *Astrophysics Source Code Library*, 1:03021, 2013.

[6] T. Budavári and A. S. Szalay. Probabilistic cross-identification of astronomical sources. *The Astrophysical Journal*, 679(1):301, 2008.

[7] R. Cutri et al. Vizier online data catalog: Wise all-sky data release (cutri+ 2012). *VizieR Online Data Catalog*, 2311:0, 2012.

[8] A. Davies and A. Orsaria. Scale out with glusterfs. *Linux Journal*, 2013(235):1, 2013.

[9] S. Donovan, G. Huizenga, A. Hutton, C. Ross, M. Petersen, and P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[10] D. Fan, T. Budavári, R. P. Norris, and A. M. Hopkins. Matching radio catalogues with realistic geometry: application to swire and atlas. *Monthly Notices of the Royal Astronomical Society*, 451:1299–1305, 2015.

[11] D. Fan, T. Budavári, A. S. Szalay, C. Cui, and Y. Zhao. Efficient catalog matching with dropout detection. *Publications of the Astronomical Society of the Pacific*, 125(924):218–223, 2013.

[12] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature methods*, 11(9):941–950, 2014.

[13] K. M. Gorski, E. Hivon, A. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann.

Healpix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

[14] O. Günther. Efficient computation of spatial joins. In *ICDE*, 1993.

[15] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[16] E. G. Hoel and H. Samet. Performance of data-parallel spatial operations. In *VLDB*, 1994.

[17] E. G. Hoel, H. Samet, et al. Benchmarking spatial join operations with spatial output. In *VLDB*, 1995.

[18] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *VLDB*, 1997.

[19] I. Kamel and C. Faloutsos. Parallel r-trees. In *SIGMOD*, 1992.

[20] N. Koudas, C. Faloutsos, and I. Kamel. *Declustering spatial databases on a multi-computer architecture*. Springer, 1996.

[21] V. S. Kumar, T. Kurc, J. Saltz, G. Abdulla, S. R. Kohn, and C. Matarazzo. Architectural implications for spatial object association algorithms. In *IPDPS*, 2009.

[22] M. Lee and T. Budavári. Cross-identification of astronomical catalogs on multiple gpus. In *Astronomical Data Analysis Software and Systems XXII*, 2013.

[23] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, 1994.

[24] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, 1996.

[25] G. Luo, J. F. Naughton, and C. J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, 2002.

[26] D. G. Monet et al. The usno-b catalog. *The Astronomical Journal*, 125(2):984, 2003.

[27] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay. Cross-matching very large datasets. In *National Science and Technology Council (NSTC) NASA Conference*, 2007.

[28] M. A. Nieto-Santisteban, A. R. Thakar, A. S. Szalay, and J. Gray. Large-scale query and xmatch, entering the parallel zone. In *Astronomical Data Analysis Software and Systems XV*, 2006.

[29] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *TODS*, (1):38–71, 1984.

[30] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, et al. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*.

[31] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, 1996.

[32] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *GIS*, 2000.

[33] F.-X. Pineau, T. Boch, and S. Derriere. Efficient and scalable cross-matching of (very) large catalogs. In *Astronomical Data Analysis Software and Systems XX*, 2011.

[34] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[35] S. Shekhar, D. Chubb, and G. Turner. Declustering and load-balancing methods for parallelizing geographic information systems. *TKDE*, (4):632–655, 1998.

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[37] M. Skrutskie, R. Cutri, R. Stiening, M. Weinberg, S. Schneider, J. Carpenter, C. Beichman, R. Capps, T. Chester, J. Elias, et al. The two micron all sky survey (2mass). *The Astronomical Journal*, 131(2):1163, 2006.

[38] A. S. Szalay, G. Fekete, W. O'Mullane, M. A. Nieto-Santisteban, A. R. Thakar, G. Heber, and A. H. Rots. There goes the neighborhood: Relational algebra for spatial data search. *MSR-TR-2004-32*, 2004.

[39] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. *arXiv preprint cs/0701164*, 2007.

[40] S. Wang, Y. Zhao, Q. Luo, C. Wu, and Y. Xv. Accelerating in-memory cross match of astronomical catalogs. In *eScience*, 2013.

[41] D. F. Xiaoying Jia, Qiong Luo. Cross-matching large astronomical catalogs on heterogeneous clusters. In *ICPADS*, 2015.

[42] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *ICDEW*, 2015.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[45] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *SIGSPATIAL*, 2012.

[46] J. Zhang, S. You, and L. Gruenwald. Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus. *Information Systems*, pages 134–154, 2014.

[47] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, 2009.

[48] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *Big Data*, 2015.

[49] Q. Zhao, J. Sun, C. Yu, C. Cui, L. Lv, and J. Xiao. A paralleled large-scale astronomical cross-matching function. In *Algorithms and Architectures for Parallel Processing*, pages 604–614. 2009.

[50] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2(2):175–204, 1998.