2015 IEEE 21st International Conference on Parallel and Distributed Systems

Cross-Matching Large Astronomical Catalogs on Heterogeneous Clusters

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

Dongwei Fan
National Astronomical Observatories
Chinese Academy of Sciences
Email: fandongwei@nao.cas.cn

***Abstract*—Cross-matching astronomical catalogs is a central operation in astronomical data integration and analysis. As current commodity clusters typically consist of heterogeneous processors including both multi-core CPUs and GPUs, we study how to efficiently cross-match large astronomical catalogs on such clusters. Specifically, we develop a three-phase common algorithm for parallel cross-match, and optimize it for a single GPU, multiple GPUs on a node, and a heterogeneous cluster of multiple nodes, respectively. Furthermore, we study the performance impact of data chunk size and that of inter-node communication mechanisms in the cluster. Our results show that, with suitable design choices and optimizations, cross-matching billion-record catalogs was completed under 10 minutes on a seven-node CPU-GPU cluster.**

***Index Terms*—Cross-match, Astronomical catalogs, GPU, Heterogeneous cluster**

I. Introduction

In observational astronomy, after the raw data are produced from telescopes, they are processed to extract celestial objects, or called *sources*. The information, e.g., positions and light intensity of these sources, are stored in a tabular format to facilitate further analysis. One central operation on these astronomical catalogs is cross-match, generating matches of the same object from multiple catalogs. Through this operation, astronomers can integrate catalogs from various instruments observed at different points in time to examine a unioned set of properties or to study the temporal evolution for each object.

With the development of high-resolution detectors and large arrays of cameras, the state-of-the-art sky survey projects produce and archive astronomical catalogs of a vast number of objects. For instance, current large catalogs, such as SDSS Dr10 [1] and USNOB1 [2], each contain more than one billion objects. Cross-matching these billion-scale catalogs are both data- and computation-intensive, yet a fast matching time is desirable to astronomical studies and is mandatory to real-time astronomical missions. So far, there have been only a few studies on cross-matching smaller catalogs on a single computer [3] [4] [5] [6] [7] [8] or on a small set of homogeneous CPU servers [9] [10] [11].

In this paper, we study how to efficiently cross-match billion-scale astronomical catalogs on commodity clusters that consist of heterogeneous nodes, including both multi-core CPUs and GPUs. Such clusters are common in practice, since organizations and service providers usually expand their computing resources over time and acquire different types of nodes that suit their computational needs and are available at time. As a result, these clusters are cost-effective in serving a diversity of computational tasks; however, it is challenging to achieve a high efficiency on each individual task, especially due to the heterogeneity of the computer nodes.

Cross-matching astronomical objects, typically by their celestial coordinates, is essentially a spatial join operation and is highly data-parallel. Since a naive nested-loop based match on each catalog requires a large number of unnecessary pair-wise comparisons, prior work has proposed several spatial indices that are suitable for celestial objects, including HEALPix [12], HTM [13], and Zones [14] [15]. As a first step, we choose HEALPix as the indexing method in this work for its high efficiency and wide acceptance among astronomers.

After the catalogs are indexed, we can then perform the cross-match in three phases: (1) partition the index files into chunks and load the chunks into the cluster; (2) for each object in one of the catalogs, called the *reference*, compute the search intervals based on the distance threshold; and (3) for each reference object, find all objects in the other catalog, called the *sample*, whose index keys are within the search intervals of the reference object; for each such sample object within a search interval, compute its distance from the reference object and compare with the distance threshold to finally determine whether it is a match.

A number of research questions arise in this three-phase algorithm, the answers to which mostly depend on the cluster configurations and capabilities of individual nodes. First of all, what is the suitable chunk size for each of the sample and reference files? Second, how do we pipeline the three phases on each node and what communication patterns do we use between nodes? Third, how do we parallelize each phase on a multi-core CPU, a single GPU, and multiple GPUs on a single node respectively?

We answer these questions from bottom up: First, we propose an efficient sequential cross-match algorithm for data that fit into the main memory. Then, we parallelize this sequential algorithm for a multi-core CPU by adding parallel directives to loops in phases 2 and 3. Next, we develop an optimized single-GPU parallel algorithm, which performs phase 2 and 3 in data-parallel GPU kernel programs. Furthermore, it fully utilizes the fast on-chip shared memory of the GPU as well

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.83

TABLE I: Notations

Notation	Meaning
(α,δ)	(Right ascension, Declination) in equatorial coordinate system
(x, y, z)	Cartesian coordinates
θ	Threshold of distance in equatorial coordinate system
γ	Threshold of distance in Cartesian coordinate system
$D(O_1, O_2)$	Distance of objects O_1 and O_2
R	Reference catalog
S	Sample catalog
N_R	# of objects in reference catalog
N_S	# of objects in sample catalog

as the GPU multi-stream technique to overlap data transfer and computation. When there are multiple GPUs on a node, we extend the single-GPU algorithm by dividing data into chunks that fit into the device memory of each GPU. Finally, for large datasets that span over a cluster of multiple nodes, we partition the data into chunks that fit into the main memory of each worker node, pipeline the three phases, and experiment with the send/receive (Send/Recv) versus broadcast (Bcast) communication patterns among nodes to pick the better alternative.

We have implemented all of these three-phase cross-match algorithms and evaluated them using a billion-object real-world astronomical catalog on a heterogeneous cluster of nodes with various models and numbers of multi-core CPUs and GPUs. The results show that, (1) our single-GPU cross-match performance achieves a speedup of up to 50 times over the optimized sequential CPU algorithm; (2) doubling the number of GPU cards on a single node from 1 to 2 and from 2 to 4 nearly doubles the cross-match performance; (3) On clusters of varied degrees of heterogeneity, Send/Recv outperforms Bcast on cross-matching large catalogs.

In summary, we make the following contributions. First, we proposed a general three-phase solution to cross-matching astronomical catalogs. This solution is applicable to data with various spatial index methods and to different hardware platforms. Second, we designed and implemented efficient HEALPix-based cross-match algorithms for multi-core CPUs, a single GPU, a multi-GPU node, and a cluster of multiple nodes. Third, we evaluated our algorithms using a real-world billion-record astronomical catalog on heterogeneous clusters consisting of various CPU and GPU nodes, and achieved a cross-match performance of under 10 minutes for such workloads for the first time in the literature.

The remainder of this paper is organized as follows. Section II discusses the background and related work. Section III proposes the general CPU-based three-phase algorithms. Section IV presents the GPU-based cross-match algorithms together with optimization strategies. Section V evaluates our algorithms and Section VI concludes the paper.

II. Background and Related Work

In this section, we first briefly introduce the problem of cross-matching astronomical catalogs, the HEALPix indexing method, and then discuss work related to ours. Table I summarizes notations used throughout the paper.

A. Cross-Match of Astronomical Catalogs

Raw digital images taken by telescopes are converted into astronomical catalogs for archival and analysis. These catalogs contain physical attributes, e.g., magnitude, position, and brightness of observed celestial objects. Due to differences in telescopes, calibration errors, and other factors, a celestial object may be recorded with slightly different positions in multiple catalogs. Therefore, cross-match algorithms usually set a threshold on the distance between two objects, or called the *search radius*, as the matching condition.

The position of a celestial object is recorded as a two-dimensional point (α,δ) in the Equatorial Coordinate System [16]. In this system, the sky is projected as a sphere with the earth in the center. α is the right ascension (projection of longitude) and δ the declination (projection of latitude) of the object. The ranges of α and δ are $[0°, 360°]$ and $[-90°, 90°]$ respectively.

For distance calculation in cross-match, equatorial coordinates are usually converted into Cartesian coordinates by formulas 1 - 3.

$$x = cos\delta sin\alpha \tag{1}$$

$$y = cos\delta cos\alpha \tag{2}$$

$$z = sin\alpha \tag{3}$$

Distance between two objects $O_1(x_1, y_1, z_1)$ and $O_2(x_2, y_2, z_2)$ is calculated as

$$D(O_1, O_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{4}$$

To achieve a high precision for small angles, the angular distance threshold θ is converted to γ in the Cartesian coordinate system by formula 5 [14].

$$\gamma = \sqrt{4sin^2(\frac{\theta}{2})} \tag{5}$$

While in some scenarios cross-match is done among more than two catalogs, two-way cross-match is most common and can be extended to more than two catalogs. Therefore, we focus on two-way cross-match, as most of the previous work, and define the problem as follows:

Two-Way Cross-match: Given two point sets R and S, a search radius θ, return all pairs of (p, q) where $p \in R$, $q \in S$, and $distance(p, q) \leq \theta$.

In other words, cross-match is to compute a spatial join with a distance threshold θ, $R \bowtie_\theta S$.

B. The HEALPix Indexing Method

HEALPix, short for the *Hierarchical Equal Area isoLatitude Pixelization*, is a scheme for fast numerical analysis on data over the sphere [12]. In this scheme, the celestial sphere is hierarchically divided into small areas of equal size, or *pixels*. The index of a celestial object is associated with the index of the pixel that contains the object. This scheme is widely used in various astronomical missions to index the objects in the

catalogs. In this paper, we choose HEALPix as our indexing method.

HEALPix has already been implemented in C, C++, F90, IDL, Python and Java [17]. Two key functions are developed in these open-source software routines: *getPix* and *queryDisc*. getPix calculates the HEALPix index of a point (α,δ). queryDisc returns a list of search intervals that cover a circle with the origin (α,δ) and radius θ. In this paper, we extracted the original implementation [17] of these two functions in C for our CPU-based algorithms and implemented our own CUDA C version for our GPU-based algorithms.

C. Related Work

There have been a number of studies on cross-matching large astronomical datasets.

Gray et al. [14] [15] proposed a zones algorithm to split the sky into stripes and performed cross-match in related zones instead of the entire sky. They implemented the zones algorithm in Microsoft server with some SQL extensions. Subsequently, Nieto-Santisteban et al. [9] [10] parallelized the zones-based cross-match on multiple SQL servers and performed cross-match of SDSS DR3 (142 million objects) and 2MASS (28,445,694 objects) in 20 minutes with 8 servers. Kumar et al. [11] further optimized the zones algorithm and cross-matched two catalogs with sizes of 3 million objects and 30,551 objects respectively in seven seconds on a hybrid MYSQL cluster. Most recently, Wang et al. [3] parallelized the zones-based cross-match in a single GPU, and Budavári and Lee [4] [5] implemented it in a single node with multiple GPUs. These zone-based GPU algorithms processed in-memory cross-match of million-scale catalogs in several seconds. Additionally, Fan et al. [6] proposed a modified version of the zones algorithm that is suitable for cross-matching partially overlapping catalogs. With taking the sky coverage information of catalogs into consideration, irrelevant proportion of objects were excluded up at a very first step.

Other than the zones algorithm, the hierarchical triangular mesh (HTM) proposed by Kunszt et al. [13] and HEALPix proposed by Gorski et al. [12] are two main-stream spatial index methods for astronomical data. Zhao et al. [7] adopted HEALPix to partition the sky into chunks and cross-matched SDSS DR6 (100 million objects) and 2MASS (470 million objects) in 32 minutes on a single MYSQL server. Pinearu et al. [8] employed HEALPix to split the cross-match task into pieces and processed them in parallel: With two hyper-threaded quad-core CPUs on a single machine, cross-match 2MASS (470 million) and USNOB1 (1 billion) was done in 30 minutes.

In addition to distance thresholds, Budavári et al. [18] proposed to take physical properties, such as colors, redshift, and luminosity, into consideration when cross-matching multiple catalogs. Most recently in year 2015, Fan et at. [19] applied this method to cross-match celestial objects recorded in radio catalogs.

In comparison, our work focuses on developing a general three-phase solution to the cross-match of large datasets on a heterogeneous cluster, and designing and implementing efficient HEALPix-based cross-match algorithms for multi-core CPUs, GPUs, and clusters consisting of various kinds of GPU nodes.

Algorithm 1: SCPU-CM($R_{file}, S_{file}, \theta, L$)

Input: R_{file}: Reference index file
Input: S_{file}: Sample index file
Input: θ: Search radius
Output: L: List of matching pairs

```
// Phase 1
R ← R_file ;
S ← S_file ;
sort S by index key;
// Phase 2
for i ← 0 to N_R do
    SI[i] ← queryDisc(R[i], θ);
// Phase 3
for i ← 0 to N_R do
    intervalSet ← SI[i] ;
    for j ← 0 to |intervalSet| do
        [start_j, end_j] ← intervalSet[j] ;
        k ← binarySearch(S, start_j) ;
        while k < N_S and S[k].id ≤ end_j do
            if distance(R[i], S[k]) ≤ θ then
                add (R[i], S[k]) to result list L;
            k ← k + 1
return L ;
```

III. CPU-Based Cross-Match

In this section, we present our CPU-based sequential and parallel algorithms for multi-core CPUs. As the context is a cluster of multiple nodes, we assume that the data for each node fit into the main memory.

A. Algorithm Overview

Two-way cross-match is a binary spatial join with a non-equality comparison predicate on a distance threshold. As such, a straightforward approach is to perform the comparisons in nested loops on the two catalogs, which is inefficient as not every pair is a match. A more efficient approach is to build indices on one or both catalogs and perform an indexed nested-loop join. Our three-phase algorithm is essentially an optimized indexed nested-loop join with the spatial distance predicate:

- Phase 1 (Preprocessing): Compute the spatial indices on both the reference and sample catalogs. Load both index files into memory. Sort the sample index file.
- Phase 2 (Computing search intervals): For each point in the reference index file, compute the list of search intervals based on the distance threshold.
- Phase 3 (Join): For each reference point, find all sample points that fall into a search interval of the reference point. For each such sample point, compute its distance from the reference point to determine whether it is a match.

In Phase 1, indices are usually computed and stored in files in advance. We index both files, after which we only need to

use the index files as opposed to the original data files in the subsequent phases. We sort the sample index file by the index key so that in the join phase, we can perform a binary search on the sample index file to find the first key that falls into a search interval.

B. CPU-based Sequential Cross-Match

The pseudo-code of CPU-based sequential cross-match is listed in Algorithm 1. This algorithm takes R_{file}, S_{file} and θ as input, and returns a list L of matching point pairs. For $\forall(p,q) \in L$, we have $p \in R$, $q \in S$ and $distance(p,q) \leq \theta$. Both R_{file} and S_{file} are index files.

The algorithm runs in three phases.

Phase 1: Preprocessing (line 1-3). This phase loads both reference and sample index files into main memory. For $i \leq N_R(N_S)$, $R[i]$ contains the coordinate ($R[i].coordinate$) and the index key($R[i].id$). Line 3 sorts S by the index key of S.

Phase 2: Compute search intervals (line 4-5). This phase computes the list of search intervals for all points in the reference point set R. $\forall i \leq N_R$, $SI[i]$ is a list of search intervals that takes the form $SI[i] = \{[start_1, end_1], [start_2, end_2]...[start_n, end_n]\}$. Each search interval $[start_j, end_j] \in SI[i]$ specifies the range of spatial indices of which the points fall into a circle with point $R[i]$ as the origin and θ as the radius.

Phase 3: Join (line 6-15). This is the filter-and-refine stage. For each point $R[i]$, we first find all points in S whose spatial indices fall in some of $R[i]$'s search intervals. $\forall R[i] \in R$, line 7 stores $R[i]$'s search interval list in $intervalSet$. $\forall[start_j, end_j] \in intervalSet$, line 10 performs a binary search on S and returns k. k is the position of the first element in S whose spatial index is greater than or equal to $start_j$. Then in the refinement stage (line 11-14), for each point in S starting from position k, we calculate its distance from origin $R[i]$ to test whether it is within the distance threshold. Finally, we return all satisfying points in result list L.

C. CPU-based Parallel Cross-Match

Parallelizing the CPU-based sequential cross-match for multi-core CPUs is simple. As shown in Algorithm 1, the loops in both phase 2 and 3 can be parallelized. Therefore, we add an OpenMP clause *#pragma omp parallel* before the for-loops (line 4 and line 6 in Algorithm 1) so that the piece of code inside the for-loop will be executed by multiple CPU threads in parallel.

IV. GPU-Based Cross-Match

A. Design Considerations

As both phase 2 and phase 3 in Algorithm 1 have high degrees of data parallelism, our main idea is to parallelize these two phases on the GPU by assigning GPU threads to process data points. Moreover, we have the following considerations in the GPU algorithm design.

- Global memory capacity: In the cluster environment, either R or S can be entirely stored in the main memory of each node. However, large catalogs cannot fit into the GPU global memory, which is only a few gigabytes per GPU card. Therefore, we need to partition the files for GPU processing.
- Overhead of data transfer: As both phases will be executed on the GPU, R and S will be transferred from main memory to the GPU device memory. At the end of the algorithm, the result list L will be transferred from the device memory back to the main memory. These transfers of large arrays will take a considerable amount of time, and it would be best to overlap it with GPU computation.
- Memory access latency: When performing the binary search on S (line 10 in Algorithm 1), all threads in a block will access S frequently. As the GPU global memory has a high access latency, we consider putting part of S into the on-chip shared memory for fast access.

Algorithm 2: SGPU-CM($R_{file}, S_{file}, \theta, L$)

```
Input: R_file: Reference index file
Input: S_file: Sample index file
Input: θ: Search radius
Output: L: List of matching pairs
// Phase 1
1  R ← R_file ;
2  S ← S_file ;
3  for m ← 0 to N_R / |R_chunk| do
4      R_d ← m-th R chunk ;
5      run thrust::sort_by_key on R_d by index key;
       // Phase 2
6      invoke kernel computeSI(R_d, θ, SI, start_min, end_max) ;
7      for n ← 0 to N_S / |S_chunk| do
8          S_d ← n-th S chunk ;
9          run thrust::sort_by_key on S_d by index key;
           // Phase 3
10         invoke kernel join(R_d, S_d, θ, SI, start_min, end_max, L) ;
```

B. Single GPU Cross-Match

The pseudo-code of single GPU cross-match is listed in Algorithm 2. As shown at line 3, we partition R into chunks of size $|R_{chunk}|$ to fit each chunk R_d into the global memory. Furthermore, we allocate the memory as page-locked and use the GPU multi-stream technique so that data transfer and kernel execution can overlap in time.

At line 5, we sort R_d by its spatial index value. This step is to reduce the time of subsequent binary search on S, because neighboring reference points with similar search intervals are stored consecutively. At line 6, we call kernel compute_SI: for each thread block, let SI_{block} be the union of $SI[i]$, where $R[i]$ will be processed in this thread block. We perform a parallel reduce function on SI_{block} to find $start_{min}$ and end_{max} (line 5-6 in Algorithm 3). Thus, interval $[start_{min}, end_{max}]$ covers all search intervals involved in this thread block. Because R_d is sorted, this result interval is minimized. Furthermore, in the subsequent invocation of kernel join (Algorithm 4), we only load S chunks whose indices are in the range of $[start_{min}, end_{max}]$ into the shared memory (line 6-9 in Algorithm 4).

At line 7-9 of Algorithm 2, we partition S into chunks S_d,

Algorithm 3: kernel computeSI($R, \theta, SI, start_{min}, end_{max}$)

Input: R: Reference point set
Input: θ: Search radius
Output: SI: List of search intervals
Output: $start_{min}$: Array of minimum start index in each block
Output: end_{max}: Array of maximum end index in each block

```
1 tid ← threadIndex ;
2 bid ← blockIndex ;
3 SI[tid] ← queryDisc(R[tid], θ) ;
4 syncthreads() ;
5 start_min[bid] ← parallel_reduce (SI_block, MIN) ;
6 end_max[bid] ← parallel_reduce (SI_block, MAX) ;
```

load it to the GPU global memory, and sort it. As each S chunk will be accessed frequently in the join, we consider loading it into the shared memory. However, the maximum amount of shared memory per multiprocessor is only tens of kilobytes, e.g., 48 KB in GPUs with compute capability of 2.X, 3.0 and 3.5. Therefore, we split S_d into smaller chunks of size $shared\_size$ that fit into shared memory and only load those chunks that match the search intervals of R_d into the shared memory.

C. Multi-GPU Cross-Match

In practice, each server node can contain multiple GPU cards, typically of the same model. For a single server node equipped with GPU_N GPU cards, we split the reference point set R into GPU_N chunks with an equal size. Each GPU takes a chunk to cross-match with the entire sample point set S. If S is too large to fit into device memory, S is also split into small chunks to be loaded by all GPUs. We use the OMP clause *omp_set_num_threads()* to set the number of CPU threads equal to the number of GPUs in the node. Then we use *#pragma omp parallel* and *cudaSetDevice()* to invoke all GPU cards in the node concurrently.

D. Multi-Node Cross-Match

Similar to the single-node algorithms, a general approach to perform cross-match on a multi-node cluster is to partition both reference and sample point sets into chunks and have each node responsible for several reference chunks. All sample chunks will be loaded and retrieved by all nodes. When a pair of reference and sample chunks are loaded in main memory on a single node, a multi-GPU cross-match can be performed then.

There are two issues to consider in the multi-node cross-match. The first issue is the parallel pipeline between nodes and the inter-node communication pattern, and the second the sample chunk size.

Inter-node Communication As in our experiments, GPUs perform cross-match much faster than CPUs, we designate GPU nodes as worker nodes and use a CPU-only node as the master node. The master loads the entire sample point set into its main memory. The workers get sample point chunks from the master.

In our multi-node algorithm, the three phases of cross-match are pipelined in iterations:

Algorithm 4: kernel join($R, S, \theta, SI, start_{min}, end_{max}, L$)

Input: R: Reference point set
Input: S: Sample point set
Input: θ: Search radius
Input: SI: List of search intervals
Input: $start_{min}$: Array of minimum start index in each block
Input: end_{max}: Array of maximum end index in each block
Output: L: List of matching pairs

```
1  bid ← blockIndex ;
2  begin ← binary_search(S, start_min[bid]) ;
3  end ← binary_search(S, end_max[bid]) ;
4  intervalSet ← SI[tid] ;
5  while begin ≤ end do
6      shared_Sid[block] ← (S[begin].id,...,S[begin + shared_size].id) ;
7      k ← begin;
8      begin ← begin + shared_size ;
9      syncthreads() ;
10     for j ← 0 to |intervalSet| do
11         [start_j, end_j] ← intervalSet[j] ;
12         pos ← binarySearch(shared_Sid[block], start_j) ;
13         while pos < shared_size and shared_Sid[pos] ≤ end_j do
14             if distance(R[tid], S[pos + k]) ≤ θ then
15                 add (R[tid], S[pos + k]) to result list L;
16             pos ← pos + 1
```

Iteration 1. The time line of the first iteration is shown as follows. At first, the master loads the first sample chunk and sorts it while each worker loads one reference chunk, and sorts it and computes search intervals on the GPU. Then, the master broadcasts the first sample chunk to all workers via *MPI_Bcast*. After the broadcast finishes, the master continues to load the second sample chunk, while the workers execute the join on the first pair of chunks. Then the master broadcasts the second sample chunk, and this pipeline of loading, sending, and executing join continues until all sample chunks have been loaded. At the end of this iteration, all sample chunks are loaded into the master's main memory, and each worker has finished cross-matching one reference chunk and the entire sample point set.

Iteration 2 and on. In the remaining iterations, all workers continue to load reference chunks and get sample chunks for cross-matching.

We consider two options for communications between the master and the workers in these iterations.

- Option 1 (MPI_Bcast). In this option, sample chunks are sent to all workers through broadcast (MPI_Bcast) at the same time. When all sample chunks have been retrieved by all workers, the next iteration starts. Synchronization is needed between all nodes at broadcast time.
- Option 2 (MPI_Send/Recv). Different from Option 1, the sample chunks are sent to workers on demand via MPI_Send/Recv. After a worker finishes cross-matching the current reference and sample chunk, it requests the master for the next sample chunk and receives it via MPI_Send/Recv. No synchronization is needed between nodes in the cluster.

Which communication options to choose? MPI_Bcast fully utilizes the network broadcast mechanisms and therefore

TABLE II: Multi-GPU Cluster Configuration

Node Name	Configuration	Processor Name	Clock (Hz)	Number 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
4×M2090	2 * CPU	Intel E5-2650	2.00G	2 * 8	512KB(L1) 2MB(L2) 20MB(L3)	64GB
	4 * GPU	Tesla M2090	650M	4 * 512	48KB	4 * 6GB
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	Master Node	Worker Node
Cluster-M	CPU-ONLY	Six 4×M2090 nodes
Cluster-K	CPU-ONLY	Three 2×K20 nodes, two 2×K20x nodes, one 2×K40 node

TABLE III: Data Sets

Data Set	Description	File Size	# of Objects	Avg # of Matching Pairs in Self Cross-Match
d_1	$\alpha \in [0^\circ, 360^\circ], \delta \in [-0.8343^\circ, 0.0209^\circ]$	$20MB$	$1,000,000$	4.65
d_5	$\alpha \in [0^\circ, 360^\circ], \delta \in [-0.8343^\circ, 1.2817^\circ]$	$100MB$	$5,000,000$	5.45
d_{25}	$\alpha \in [0^\circ, 360^\circ], \delta \in [-0.8343^\circ, 1.3154^\circ]$	$500MB$	$25,000,000$	15.80
dr_{12}	$\alpha \in [0^\circ, 360^\circ], \delta \in [-17.7573^\circ, 84.9799^\circ]$	$24.6GB$	$1,231,051,050$	30.21

is more efficient than MPI_Send/Recv for large data chunks. However, it requires all workers to synchronize at broadcast time. So, if all workers finish cross-matching at roughly the same time, broadcast will work better. Otherwise, the overhead of synchronization between all workers will be large. In our experiments, we find that cross-matching different reference chunks with the same sample chunk take quite different amounts of time, even if all nodes are homogeneous. Therefore, We choose Option 2 over broadcast.

Sample chunk size Let $|S_{chunk}|$ denote the sample chunk size. In each iteration, the master node sends the entire sample set in $\frac{|S|}{|S_{chunk}|}$ rounds to a worker node. Also, these chunks will be transfered from main memory to the device memory on each worker node. For data transfer in both situations, transferring large chunks is more efficient than transferring smaller ones separately due to better bandwidth utilization and fewer rounds of transfer. In addition, each worker node performs $\frac{|S|}{|S_{chunk}|}$ rounds of binary search to cross-match one reference chunk and the entire sample set. Therefore, the size of each sample chunk should be set as big as possible. Device memory size and number of elements in a chunk supported by MPI APIs are two constraints on the chunk size. Therefore, we set the chunk size to be just under the limit of both factors.

V. Evaluation

We have implemented and optimized our algorithms for multi-core CPUs, a single GPU, multiple GPUs on a node, and a cluster of multiple nodes. In the following, we evaluate our algorithms on a heterogeneous cluster consisting of both CPUs and GPUs.

A. Experimental Setup

The cluster on which we conducted our experiments is a research facility maintained at our university. It consists of 22 nodes, with 16 of which containing GPUs. All nodes are connected with InfiniBand (IB) at 40 Gbit/s. Due to the acquisition time differences and other considerations, the nodes vary in their processor models.

The specifications of the CPUs and GPUs in the cluster are listed in Table II. In total, there are five kinds of nodes. The first type is a CPU-only node with 4 10-core Intel Xeon E5-2665 v3 processors (CPU-ONLY). The second is a CPU-GPU node with 2 8-core Intel Xeon E5-2665 processors and 4 NVIDIA Tesla M2090 cards (M2090). These Tesla cards are of the Fermi architecture, and their CUDA compute capability is 2.0. The third to fifth types are each of 4 10-core Intel Xeon E5-2665 v2 CPU processors, and 2 NVIDIA Tesla K20/K20X/K40 cards respectively. All of the K-series nodes are of the Kepler architecture, and their CUDA compute capability is 3.5.

Considering the available nodes in the cluster and our purpose, we experimented with two subsets of seven nodes, Cluster-M, and Cluster-K. Both clusters have a CPU-only node as the master node, and six worker nodes with GPU co-processors. Cluster-M is homogeneous in that all of its six worker nodes are of the M2090 node type. In contrast, Cluster-K is heterogeneous, consisting three K20 nodes, two K20x nodes, and one K40 node.

The entire cluster runs the CentOS 6.5 64-bit Operating System. Intel MPI 5.0.3. was used in our multi-node cluster implementation. Our CPU-based implementation used the *tbb_parallel_sort* provided by Intel(R) Threading Building Blocks library [20]. Our CPU programs and CUDA programs were compiled with GNU g++ 4.4.7 and CUDA 6.5, respectively. All CPU programs were compiled with -O3 optimization option. We measured the time performance by *gettimeofday()* on the CPU or *cudaEventRecord()* on the GPU. We repeated each experiment 10 times, and report the average

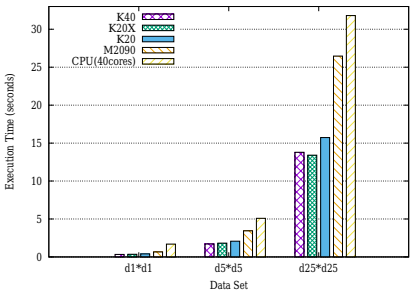

Fig. 1: CPU/GPU parallel execution time.

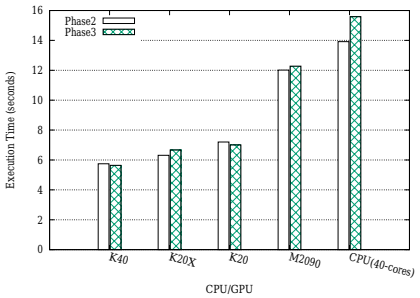

Fig. 2: Performance of individual phases.

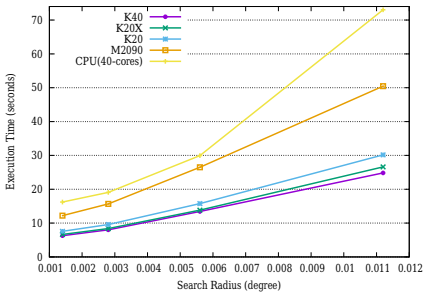

Fig. 3: Impact of search radius.

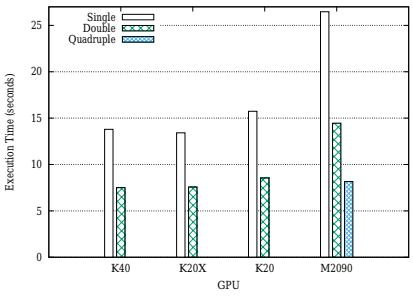

Fig. 4: Number of GPUs on a node.

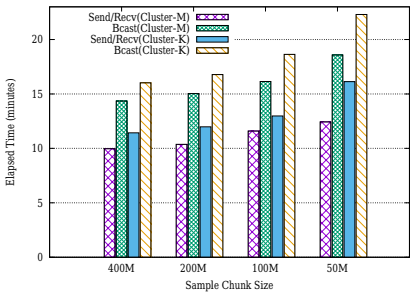

Fig. 5: Performance on two clusters.

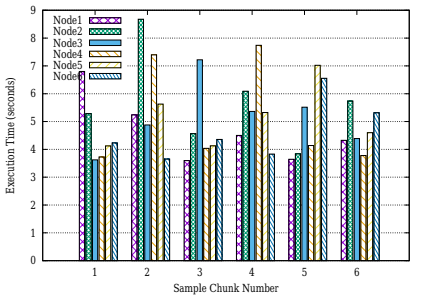

Fig. 6: Execution time in a single iteration on Cluster-M.

time. The standard deviation was small.

We downloaded a real-world dataset, data release 12 (dr12) of SDSS [21] for our evaluation. This dataset is one of the largest astronomical catalogs currently available in public. The entire table, *PhotoObjAll*, contains 1,231,051,050 objects. To study the impact of data size and the performance on a single GPU and a single node, we selected fixed numbers of objects within certain celestial areas from PhotoObjAll to form several smaller datasets. Table III summarizes the four test datasets with different sizes used in our evaluation.

All of our cross-match experiments were self-matches on the same catalog. Such self-matches are more computationally intensive than matching different catalogs in that every object must have matches in the other catalog. The number of matching results for each object depends on both the datasets and the distance threshold. Unless otherwise specified, the distance threshold, or search radius θ, is set to 0.0056°, a value suggested by the astronomer. The correctness of the matching results is also verified by the astronomer.

B. Single Node Performance

We first evaluate the performance of our cross-match algorithms on a single node. We used the three smaller test datasets d1, d5 and d25, each of which fit into the device memory of a single GPU. As phase 1 of our algorithm, loading reference and sample index files, is executed on the CPU on all nodes, we separate its time from the execution time of phase 2 and 3 on the CPU or the GPU. The size of GPU thread block was set to 512, which achieved the best performance in our experiments.

TABLE IV: CPU Sequential Execution Time (seconds)

DataSet	Phase 1	Phases 2 and 3
d1 * d1	1.6	17
d5 * d5	7.8	85
d25 * d25	40	618

The phase 1 time and the execution time (phase 2 and 3) of our sequential CPU program are listed in Table IV. As shown in the table, phase 1 is only a small fraction, less than 10% of the sequential CPU execution time of phases 2 and 3.

Fig. 1 shows the parallel execution time of phase 2 and 3 on a multi-core CPU or GPU on the three datasets. Compared with the sequential CPU execution time, the parallel CPU program on 40 cores achieved a speedup of 9.49 - 19.44. In comparison, the speedups of M2090 were 21.8 - 25.2, and those of the K-series GPUs (K40/K20X/K20) 35.41 - 52.0.

Fig. 2 illustrates the time breakdown of phase 2 and 3 on the dataset d25. Across the multi-core CPU and the GPUs, phase 2 and 3 took a similar amount of time, except on the CPU phase 3 considerably longer than phase 2.

We varied the value of the search radius around the default value, and plot the execution time of phase 2 and 3 in Fig. 3. With the increase of the search radius, the execution time increases more than linearly due to the increased number of objects tested.

Finally, Fig. 4 shows the performance comparison of single-GPU and multi-GPU execution time on one node. The multiple GPUs on a single node share the main memory but not the

device memory. Both reference and sample catalogs were dataset d25. Two GPUs on a single node reduced the execution time of a single GPU of the same type almost by half. Four M2090 GPUs reduced the execution time of two M2090s by another 40% roughly. The performance of cross-match on a node with two M2090 cards was roughly the same as that of one K40 card, whereas the performance of four M2090 cards on a single node was almost equal to that of two K40 cards.

C. Multi-Node Cross-Match

After investigating the cross-match performance on a single node, we next study the performance of self-matching the entire catalog *PhotoObjAll* of SDSS dr12 on clusters of multiple nodes.

We studied the performance on two clusters: Cluster-M consists of six homogeneous nodes of four M2090 GPUs each, and Cluster-K six heterogeneous nodes of two K-series GPUs each. Both cluster share the same CPU-only node as the master node. The chunk size of the reference file was set to $20M$ (20 million objects) in all experiments for the best performance. Note that $400M$ and $200M$ were the maximum sizes of the sample chunk that fit into a node with $24GB$ and $10GB$ device memory respectively.

Fig. 5 shows the overall time of our cross-match algorithm in two clusters with different chunk sizes of the sample file and communication patterns between nodes. On both clusters, the processing time increased with the decrease of the sample chunk size, and the Send/Recv communication outperformed broadcast (Bcast) significantly. Interestingly, Cluster-M outperformed Cluster-K even though they had a similar single-node performance. These phenomena suggest that inter-node communications and synchronization had a major impact in the performance.

To study the timing of inter-node communication, we measured the execution time of each sample chunk in a single iteration on each node of Cluster-M. The sample chunk size was $200M$ and the entire sample set was sent in six rounds. Fig. 6 shows that, on a single node, the time of cross-matching one reference chunk differs by the sample chunk. Conversely, with the same sample chunk, the cross-matching time varies with the reference chunk. These time differences were significant even on a homogeneous cluster such as Cluster-M. As a result, the asynchronous Send/Recv communication outperformed Bcast, which required synchronization.

VI. Conclusion

In this paper, we proposed a three-phase common algorithm for cross-matching large astronomical catalogs. We further designed and implemented such three-phase algorithms for multi-core CPUs, a single GPU, multiple GPUs on a single node, and a cluster of heterogeneous nodes with various types of GPUs. We evaluated our implementations on self-matching a real-world billion-object astronomical catalog on clusters of heterogeneous nodes. Our results show that (1) a single GPU can speed up the sequential cross-match algorithm by 50 times; (2) doubling the number of GPUs on a node nearly doubles the cross-match performance; (3) cross-matching on clusters favors large data chunks and send/receive inter-node communications; and (4) our algorithm was able to self-match the billion-object catalog within 10 minutes on a seven-node cluster.

VII. Acknowledgement

This work was supported by grants 616012, 617509, and 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft SQL Server China R&D.

References

[1] C. P. Ahn *et al.*, "The tenth data release of the sloan digital sky survey: First spectroscopic data from the sdss-iii apache point observatory galactic evolution experiment," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 17, 2014.

[2] D. G. Monet *et al.*, "The usno-b catalog," *The Astronomical Journal*, vol. 125, no. 2, p. 984, 2003.

[3] S. Wang, Y. Zhao, Q. Luo, C. Wu, and Y. Xv, "Accelerating in-memory cross match of astronomical catalogs," in *eScience (eScience), 2013 IEEE 9th International Conference on*. IEEE, 2013, pp. 326–333.

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

[5] M. Lee and T. Budavári, "Cross-identification of astronomical catalogs on multiple gpus," in *Astronomical Data Analysis Software and Systems XXII*, vol. 475, 2013, p. 235.

[6] 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*, vol. 125, no. 924, pp. 218–223, 2013.

[7] 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*. Springer, 2009, pp. 604–614.

[8] 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*, vol. 442, 2011, p. 85.

[9] 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*, vol. 351, 2006, p. 493.

[10] 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.

[11] V. S. Kumar, T. Kurc, J. Saltz, G. Abdulla, S. R. Kohn, and C. Matarazzo, "Architectural implications for spatial object association algorithms," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[12] 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*, vol. 622, no. 2, p. 759, 2005.

[13] P. Z. Kunszt, A. S. Szalay, and A. R. Thakar, "The hierarchical triangular mesh," in *Mining the sky*. Springer, 2001, pp. 631–637.

[14] 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.

[15] J. Gray, M. A. Nieto-Santisteban, and A. S. Szalay, "The zones algorithm for finding points-near-a-point or cross-matching spatial datasets," 2006.

[16] Equatorial coordinate system. [Online]. Available: https://en.wikipedia.org/wiki/Equatorial-coordinate-system

[17] Healpix home page. [Online]. Available: http://healpix.sourceforge.net

[18] T. Budavári and A. S. Szalay, "Probabilistic cross-identification of astronomical sources," *The Astrophysical Journal*, vol. 679, no. 1, p. 301, 2008.

[19] 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*, vol. 451, pp. 1299–1305, 2015.

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

[21] Sdss dr12. [Online]. Available: http://www.sdss.org/dr12/scope/