Trans. Tianjin Univ. 2011, 17: 062-067
DOI 10.1007/s12209-011-1491-x

Improved Parallel Processing Function for High-Performance Large-Scale Astronomical Cross-Matching*

ZHAO Qing (赵　青)[1], SUN Jizhou (孙济州)[1], YU Ce (于　策)[1], XIAO Jian (肖　健)[1], CUI Chenzhou (崔辰州)[2], ZHANG Xiao (张　啸)[1]

(1. School of Computer Science and Technology, Tianjin University, Tianjin 300072, China;
2. National Astronomical Observatories, Chinese Academy of Sciences, Beijing 100012, China)

Abstract: Astronomical cross-matching is a basic method for aggregating the observational data of different wavelengths. By data aggregation, the properties of astronomical objects can be understood comprehensively. Aiming at decreasing the time consumed on I/O operations, several improved methods are introduced, including a processing flow based on the boundary growing model, which can reduce the database query operations; a concept of the biggest growing block and its determination which can improve the performance of task partition and resolve data-sparse problem; and a fast bitwise algorithm to compute the index numbers of the neighboring blocks, which is a significant efficiency guarantee. Experiments show that the methods can effectively speed up cross-matching on both sparse datasets and high-density datasets.

Keywords: astronomical cross-matching; boundary growing model; HEALPix; task partition; data-sparse problem

Multi-waveband data cross-matching among multiple catalogs is a basic and unavoidable step to make distributed digital astronomical archives accessible and interoperable. Because current astronomical catalogs often contain millions or billions of objects, multi-waveband data cross-matching is a typical data-intensive problem[1]. There are astronomic tools for performing cross-matching locally or online, such as Aladin, VizieR, TopCat, OpenSkyQuery and MAST. However, they were all designed to deal with limited amount of data. For example, VizieR and Topcat could only process small datasets because they adopted in-memory algorithms. The US National Virtual Observatory (NVO) proposed complicated efficiency-optimizing measures[2-4] named "zoned" on this problem, which was carried out completely in SQL commands to avoid expensive procedures or table-valued functions. However, this function is too elusive, so to some extent its popularization among other astronomical organizations is limited. Unlike NVO, AstroGrid and some other astronomical organizations attempt to resolve this efficiency problem in different ways[5-8]. Gao *et al*[9,10] proposed a cross-matching function based on HTM index and kd-tree, but they did not consider the border-data problem when doing data partition, therefore source loss is inevitable, and at the same time, it is not suitable for large data sets.

Previously, we have proposed a parallel cross-matching measure in Ref.[11]. An important concept adopted both in our previous function and the improved function in this paper is HEALPix (hierarchical equal area isolatitude pixelization of a sphere) index. It plays a significant role in speeding up database query and realizing data partition in the program.

1 Related work

1.1 HEALPix indexing function

HEALPix is a typical virtual spatial indexing function. As shown in Fig.1, it partitions the whole sky in a recursive quad-tree pixel subdivision way, then each celestial object with its position information RA and DEC will fall into one of the blocks. Therefore a many-to-one mapping can be built from the position (RA, DEC) to the block numbers, i.e., 2-dimensional space is mapped to line space. Finally, the B-tree indexing function supported by all the DBMSs can be used to index the data.

Accepted date: 2010-04-29.
*Supported by National Natural Science Foundation of China (No.10978016), Natural Science Foundation of Tianjin (No. 08JCZDJC19700) and Key Technologies Research and Development Program of Tianjin (No.09ZCKFGX00400).
ZHAO Qing, born in 1983, female, Dr.
Correspondence to YU Ce, E-mail: yuce@tju.edu.cn.

HEALPix has a nested numbering function, see Fig.2. In each partition-depth, each block is divided into 4 sub-blocks, which means only the additional 2-bit-length code need be added to the end of the father block's number. The detailed numbering function is given in Fig.2. By the hierarchical partitioning strategy and nested numbering function of HEALPix, most sky objects which are close physically also have close HEALPix index numbers. This attribute is useful in task partition and fast database query. The detailed data storage structures in database are listed in Tab.1.

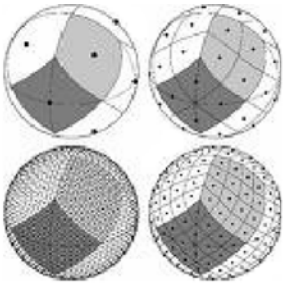

Fig.1 Sky partitioning function in HEALPix

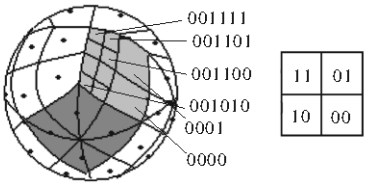

Fig.2 Nested numbering function in HEALPix

Tab.1 Data structures in database

Field	Type	Description	Index or not
ID	BIGINT	Object's ID	B-tree index
RA	DOUBLE	Right ascension	Not
DEC	DOUBLE	Declination	Not
HEAL-Pix_ID	INT(8)	HEALPix index number	B-tree index

1.2 Our previous function

Our previous function[11] is shown in Fig.3. First, the whole sky is split into many small index cells using HEALPix indexing function; then the whole cross-matching computation is partitioned into a number of sub-tasks, each of which is responsible for a small region (containing a certain number of index cells). These small regions called computing blocks are the basic units for task partitioning and scheduling from the master process to the child processes. However, because of telescopes' position errors, if one object falls in a border region of one computing block in database A, it is possible that its corresponding source in database B is categorized into another neighboring block. Therefore, for a block of data in A, not only the data inside the same number block of B but also its neighbors should be considered. This problem named border data problem is common and inevitable in astronomical cross-matching. As shown in Fig.3, in this paper, the data inside a computing block is called block data, and the data on the border outside the block is called border data. Towards the border data problem, our previous strategy is for a given block waiting for cross-matching in database A, the program first computes HEALPix index numbers of the border data, and then loads the border data into memory from database B as well as the central block.

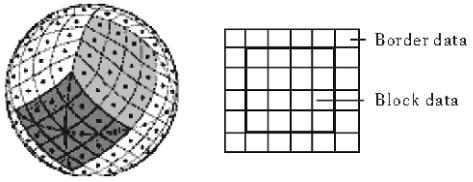

Fig.3 Our previous X-Match function[11]

The database query operations are time-consuming, especially for the database query of border data. The low-efficiency of the database query of border data results directly from the nature and characteristics of B-Tree index. As the index numbers of borders are discontinuous, the low-efficiency is inevitable. Therefore, the first aim of this paper is to reduce time on the database query of border data. And then, we also find that the block data's database query operations can also be further compressed by a series of methods, such as redesigning the task partitioning strategy, optimizing the process flow, and filtering the empty areas.

2 Database query and processing flow

2.1 Acceleration of database query of border data

The time-consuming database query of border blocks can be removed by enlarging the distance computation scope in database B, i.e., for one computing block in database A, we do not select the same position computing block or the border blocks around it in database B to do the distance computation. Instead, we read the central block and all of its 4 neighboring blocks into memory. This change is shown in Fig.4. Using this approach, the distance computation workload will be increased by nearly 5 times, but it can be reduced through increasing the partition granularity of computing blocks. Another problem is that the whole database reading time will increase again as the block number increases. However, an experiment shows that the partition depth can be changed

from 8 to 9 (which means to change the number of computing blocks from 12×4^8 to 12×4^9) without an obvious increase of database reading time. As shown in Fig.5, only when the partition depth of computing blocks is greater than 9, the database reading time of the whole computing blocks increases obviously.

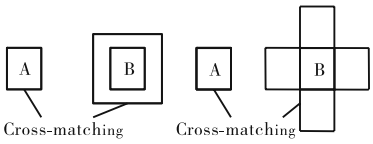

Fig.4 Enlarging distance computation scope

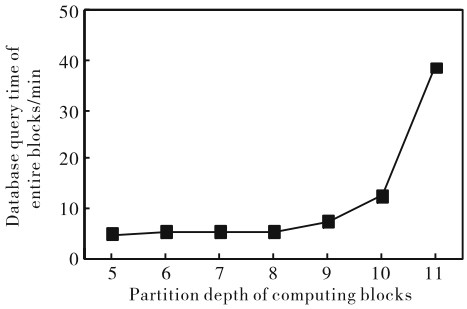

Fig.5 Database query time of block data

2.2 Reconstruction of processing flow based on boundary growing model

A repeated data query problem exists in the above function: if the processing sequence is from No.1 computing block to No. Max computing block, then each computing block in database B is required to be read out from database for 4 times, because each computing block is a neighbor of 4 blocks. In this section, a new processing flow based on boundary growing model is described to resolve this problem. The idea is that when a computing block from one database is read into memory, the procedure will try to arrange all of its distance computation tasks with its four neighboring blocks as quickly as possible, therefore it can be released after the computation and will never need to be loaded again. The data processing flow is shown in Fig.6. The whole area in Fig.6 is a growing block which contains a number of computing blocks. The central 4 computing blocks at Round 0 are regarded as a growth point, and then the data loading and computing scale will grow round by round from the growth point.

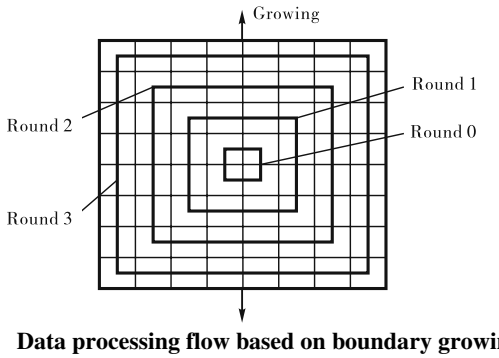

Fig.6 Data processing flow based on boundary growing model

The processing flow based on boundary growing model is described as follows.

Step 1 The inner round (Round 0) consisting of 4 blocks is read into memory from database A.

Step 2 The blocks of Round 0 and Round 1 in database B are loaded so that the blocks of Round 0 from database A can accomplish their entire distance computation tasks, and then all of them will be released as they will never be needed again.

Step 3 The blocks of Round 1 and Round 2 in database A are read into memory, then the data in Round 0 and Round 1 from database B can run their entire distance computation tasks, and they will be released when the computation is finished.

…

The data loading rule is the same as the above steps.

For the purpose of not exceeding memory limit, the growing scope should be limited. One easy method is to partition the whole sky into a number of growing blocks of equal area. But it is not proper because the density of astronomical objects in different areas is highly uneven, the static size can only be set as a small number conservatively; otherwise, an "out of memory" exception may occur again. The computing blocks which fall in the outmost round of a growing block will be queried from database twice, since it is also needed by its neighboring growing blocks. Therefore, as long as memory allows, the larger the size of growing blocks is set, the less the database query operations are required.

3 Determination of the biggest growing blocks

Sometimes, one or both of the 2 database are sparse, i.e., a great number of regions are empty. The empty areas should be first filtered out, so we choose a proper block size to identify which blocks are empty, and then try to find growing blocks as large as possible in the remnant spherical areas. A big growing block can be divided into 4 equal blocks by HEALPix indexing mechanism if the data amount exceeds the allowable value.

The growing blocks of various sizes are the basic task assignment units, and their determination process can be described as follows: First, the upper limit of growing blocks is initialized, which means that the whole

sky is divided into a certain number of "initial_biggest" blocks. Then an appropriate size used to check whether an area is empty or not will be chosen. Using its HEALPix index, the procedure can submit a "Select Count ()" query to databases to check whether the area in at least one of the catalogs is empty or not. If yes, the index number of the empty block will be recorded in the empty table. This means that, for every "initial_biggest" block, the procedure firstly establishes an "empty or not" situation map for it. As shown in Fig.7, according to the empty table, all of the biggest growing blocks can be found in a recursive quad-tree way quickly.

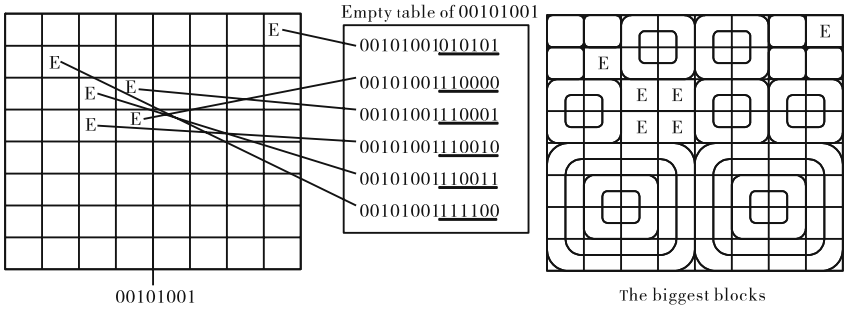

Fig.7 Processes of determining the biggest growing blocks

4 Fast computation of index numbers of neighboring blocks

It is important to compute the index numbers of neighboring blocks of a computing block. The computation efficiency should be very high, since the computation is called frequently throughout the whole procedure. In this section, by utilizing the nested numbering function of HEALPix, all the index numbers of neighboring blocks can be computed by only one or two steps of XOR bitwise operation.

The block "11001111" in Fig.8 is taken as an example. In fact, only the index number computation of block 2, 4, 6 and 8 are needed since block 1, 3, 5 and 7 can be deduced by them. As an example, first, we compute the index number of block 2 — the upper block of "11001111" according to the following steps.

The last 2 bits of "11001111" are "11", which means that in the last division, block 2 is positioned at the upper-left corner, so to find the index number of its upper neighbor, only changing the last bits "11" is not enough, we need to find the first 2 bits which are "00" or "10" from lower bits to higher bits, since the number of the upper neighbor can be obtained by replacing "00" with "01" or replacing "10" with "11". Therefore, in this example, the inner "00" should be changed to "01", and the next upper bits "11" should not be changed. By now, the first 4 bits of the index number of the upper neighbor has been determined as "1101", and the big dotted region in Fig.8 is what it represents.

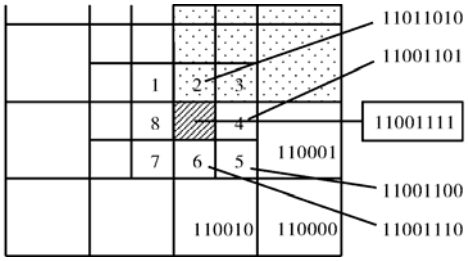

Fig.8 Deduction of the index numbers of neighboring blocks

The last 4 bits of the initial block is "1111", which means that it has been divided into the upper-left position in the last two division rounds. Correspondingly, the index number of its upper neighboring block should be divided to the lower-left corner of the dotted area, i.e., its last 4 bits should be "1010". Up to now, the index number of the upper neighboring block has been completely resolved as "11011010".

In fact, the index number of the upper neighboring block can be achieved by simple XOR bit-operation. Similarly, the lower neighbor, the left neighbor, and the right neighbor can also be identified using XOR bit-operation. All of these XOR formulas are given as follows.

Upper neighbor: Lookup the first 2 bits which is "00" or "10" from lower to higher bits, so the two bits at this position and its entire lower bits of the second operand should be "0101…01". Then do the operation: 11001111 ^ 00010101= 11011010.

Lower neighbor: 11001111 ^ 00000001= 11001110

Left neighbor: 11001111 ^ 00101010 = 11100101

Right neighbor: 11001111 ^ 00000010 = 11001101

Then the index numbers of all the other neighboring

block 1, 3, 5 and 7 can be deduced by two steps of the above operations.

5 Experiments and performance evaluation

5.1 Environment condition

The parameters of server Dell PowerEdge T300 on which experiment programs ran are as follows: CPU, Quad-core Xeon X3323; Memory, 2.0 G 667MHz DDR2; OS, Linux Fedora 10 (kernel: 2.6.23.1-42.fc8 SMP); Swap, 2G; Software, C programming language + MPICH2, MySQL.

5.2 Experimental results and analysis

In the first experiment, the entire data of two large catalogs are used. Since they are typical sparse datasets, they were also used in our previous paper[11]. In the second experiment, we build a non-sparse dataset selecting some regions which are dense both in the two original catalogs.

Experiment 1 On the large sparse dataset containing many empty areas

Data: SDSS data 6 catalog (containing about 100 million objects, refer to Sloan Digital Sky Survey DR6, http://cas.sdss.org/dr6/en) and 2MASS catalog (containing about 471 million objects, refer to Two Micron All Sky Survey at IPAC, http://www.ipac.caltech.edu/2ma ss/).

Number of processes: 1 master process and 3 child processes.

Results of our previous function and the improved function under exactly the same environment are shown in Tab.2 and Tab.3. It can be seen that in the improved function, by filtering out many empty blocks, the time consumed on database query of SDSS is reduced from 317 s to 191 s. Furthermore, the time-consuming part of database query of border data of 2MASS in our previous function has been removed by the strategy given in Section 2.1, so there is only one column about 2MASS database query in Tab.3. And the total running time has decreased to 46%.

Tab.2 Experimental results of our previous function

Partition granularity of computing blocks	Time consumed on operations /s					
	Database query of SDSS	Database query of 2MASS (block)	Database query of 2MASS (border)	Distance computation	Others	Sum
12×4^7	307	59	335	580	40	1 321
12×4^8	317	40	639	151	44	1 191
12×4^9	427	54	1 177	51	72	1 781

Tab.3 Experimental results of the improved function

Partition granularity of computing blocks	Time consumed on operations /s				
	Database query of SDSS	Database query of 2MASS	Distance computation	Others	Sum
12×4^7	120	78	2 489	48	2 735
12×4^8	127	79	690	58	954
12×4^9	191	102	199	57	549
12×4^{10}	374	239	58	67	738

Experiment 2 On the small non-sparse dataset built by ourselves

Data: part of SDSS data catalog (47 949 212 objects) and part of 2MASS catalog (35 476 377 objects).

Number of processes: 1 master process and 3 child processes.

Results of our previous function and the improved function under exactly the same environment are shown in Tab.4 and Tab.5.

In this experiment, the datasets are very dense, and the time consumed on database query of SDSS cannot be reduced because there is no empty block which can be filtered out in advance. Instead, because every computing block in SDSS on the border of the biggest growing blocks needs to be loaded twice, the time consumed here even rises to 36 s from 33 s in our previous function (both for the partition granularity of 12×4^8). However, because the datasets are dense, there is a higher proportion of database query time consumed on 2MASS datasets. In our previous function of Experiment 1, the proportion is 57%, while in our previous function of Experiment 2, the proportion is 74%. This is because in our previous function, when SDSS data are sparse, if one block of SDSS has been found empty, the program will not try to load the same number block in 2MASS or its neighboring blocks, so not all of the blocks in 2MASS need to be queried in Experiment 1. Therefore, the time-decreasing effect of the method described in Section 2.1 is more obvious when the datasets are non-sparse, since it

has avoided the time-consuming operations — database query of the value-discrete border data. This experiment has proved that the improved function can work well on the dense datasets as well as on the sparse datasets.

Tab.4 Experimental results of our previous function

Partition granularity of computing blocks	Time consumed on operations /s					
	Database query of SDSS	Database query of 2MASS (block)	Database query of 2MASS (border)	Distance computation	Others	Sum
12×4^7	33	17	124	96	16	286
12×4^8	33	19	191	24	16	283
12×4^9	43	28	403	11	22	507

Tab.5 Experimental results of the improved function

Partition granularity of computing blocks	Time consumed on operations /s				
	Database query of SDSS	Database query of 2MASS	Distance computation	Others	Sum
12×4^7	32	19	421	27	499
12×4^8	36	20	130	27	213
12×4^9	46	27	39	31	143
12×4^{10}	107	60	11	32	210

6 Conclusions

The improved method proposed in this paper has been proved useful in improving the performance of parallel cross-matching computation. The boundary growing model is a new idea which can improve the parallel processing flow of cross-matching greatly. According to the processing flow, most of the data in database need to be read out only once, so the time consumed on database query can be decreased. In addition, the task partition strategy—determination of the biggest growing blocks is an effective complement to the boundary growing model by realizing the dynamic restriction and subdivision of growing blocks. Furthermore, it has resolved the data-sparse problem in cross-matching. Through experiments, the improved method has been proved effective both in sparse datasets and high-density datasets. Therefore, it should be applicable to a wide range of astronomical data access services.

References

[1] Nieto-Santisteban M A, Thakar A R, Szalay A S. Cross-Matching Very Large Datasets[R/OL]. http://citeseerx. ist. psu. edu/viewdoc/download?doi = 10. 1. 1. 129. 3240&rep = rep1&type = pdf. 2006.

[2] Gray J, Szalay A, Budavri T *et al*. Cross-Matching Multiple Spatial Observations and Dealing with Missing Data[R/OL]. Microsoft Technical Report, MSR-TR-2006-175. Redmond WA. http://research. microsoft. com/pubs/64519/tr-2006-175. pdf. 2006.

[3] Gray J, Szalay A, Fekete, G. Using Table Valued Functions in SQL Server 2005 to Implement a Spatial Data Library[R/OL]. Microsoft Technical Report, MSR-TR-2005-122. Redmond WA. 2005.

[4] Gray J, Nieto-Santisteban M A, Szalay A S. The Zones Algorithm for Finding Points-near-a-Point or Cross-Matching Spatial Datasets[R/OL]. Microsoft Technical Report, MSR-TR-2006-52. Redmond WA. http://arxiv. org/ftp/cs/papers/0701/0701171. pdf. 2006.

[5] Clive Page: Indexing the Sky[R/OL]. Technical Report, AstroGrid.http://www.star.le.ac.ud/~cgp/ag/skyindex.html. 2002.

[6] Clive Page: Comments on the XMATCH function in ADQL[R/OL]. Technical Report. AstroGrid. 2004.

[7] Report on Cross-Matching Catalogues[R/OL]. Technical Report. AstroGrid, http://wiki. astrogrid. org/pub/Astrogrid/DataFederationandDataMining/cross. htm, 2003.

[8] Spatial Joins and Spatial Indexing Revisted [R/OL]. Technical Report of AstroGrid. AstroGrid, http://wiki. astrogrid. org/bin/view/Astrogrid/SpatialIndexing , 2003.

[9] Gao Dan, Zhang Yanxia, Zhao Yongheng. The realization of cross-identification based on huge multi-wavelength catalog data[J]. *Astronomical Research and Technology*, 2005, 2(3): 186-193 (in Chinese).

[10] Gao Dan. Very Large Astronomical Data Sets Fusion System's Development and Data Mining Algorithms' Research[D]. National Astronomical Observatories, Chinese Academy of Sciences, 2008 (in Chinese).

[11] Zhao Qing, Sun Jizhou, Yu Ce *et al*. A paralleled large-scale astronomical cross-matching function[C]. In: *The 9th International Conference on Algorithms and Architectures for Parallel Processing. ICA3PP 2009*. Taipei, China, 2009. Vol. 5574 LNCS. 604-614.