ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp.17-21
doi: 10.18178/wcse.2017.06.003

A Distributed Cache Framework on Massively Parallel Computing Architecture

Jie Yu [1+], Guangming Liu [1, 2], Wenrui Dong [1] and Xiaoyong Li [1]

[1] College of Computer Science, National University of Defense Technology
[2] National Supercomputer Centre in Tianjin

Abstract. Current high performance computers (HPC) mostly adopt massively parallel computing (MPP) architecture, which uses parallel file system to accommodate all the data, instead of having local storage located on the compute node side. MPP architecture prevents large scale applications dumping terabytes of temporal or intermediate data into local storage and causes severe I/O bottleneck. In this paper, we proposed a distributed cache framework to merge scattered memory spaces of up to thousands of compute nodes into a unified cache pool, so that data-intensive applications can store their intermediate data into those compute nodes of light I/O burden. We evaluate the framework with IOR and a realistic benchmark BTIO on a HPC system TH-1A, which indicates that our approach can bring significant performance boost to data-intensive applications.

Keywords: HPC, Distributed Cache, Data-intensive Application, Cooperate Cache, MPP.

1. Introduction

The rapid development of high performance computing (HPC) technology have enabled scientists and engineers to utilize the tremendous computing power of HPC to solve more and more complex scientific problem and engineering simulation. For example, data-intensive application need ingest enormous amount of raw data to gain insightful knowledge, and large scale simulation would generate terabytes of data during every checkpoint [1].

Current HPC systems mainly adopt massively parallel computing (MPP) architecture to simplify the design of compute nodes and storage system. In MPP architecture, there is no local storage media exists in compute node (CN) side, like hard disks (HDD). Instead, they always attach a parallel file system (PFS), such as Lustre [2], GPFS [3] or PVFS [4], to the compute sub system through an identical or specified network. All the CNs in the HPC system can access the PFS through the high speed network, which can easily cause I/O bottleneck when PFS handling a massive influx of I/O requests from thousands of CNs. The reason of I/O bottleneck mainly lies in two aspects. Firstly, any single I/O flowing out from CN needs to be forwarded to remote storage servers, and the network latency introduces significant cost. Secondly, any single storage server needs handle tremendous amount of concurrent I/O requests at the same time. The performance of HDD based storage servers can be seriously affected under server contention. PFS becomes incompetent to deal with daily growing I/O demand. The performance gap between compute and storage is widening.

As a step towards narrowing the performance gap, we proposed a distributed cache framework to accommodate massive temporal and intermediate data that flushed out from data-intensive application. By making use of memory of CNs under light I/O burden, the framework reduces the costly interaction with

+ Corresponding author. Tel.: +86 022-65375500; fax: +86 022-65375501.
E-mail address: jcommon@163.com.

remote PFS. The experiments conducted on TH-1A in National Supercomputer Centre in Tianjin (NSCC-TJ) shows that the proposed framework can improve the efficiency of data-intensive applications greatly.

2. Design and Implementation

2.1. Architecture

TH-1A is the first system of Tianhe HPC series, which adopts MPP architecture. It consists of three subsystems, which are compute subsystem, interconnect subsystem and storage subsystem. There are 7168 CNs in compute subsystem, each of which has 2 Xeon CPU and 12 cores. CNs are connected with storage subsystem via high speed interconnect network. TH-1A adopts Lustre to merge hundreds of storage servers (OST) into a unified storage subsystem.

The architecture of the distributed cache framework is shown in Figure 1. To be noted that, we design the proposed framework to be portable in the first place, so it can be used in any MPP systems. The HPC architecture shown in the figure is also a typical MPP system.

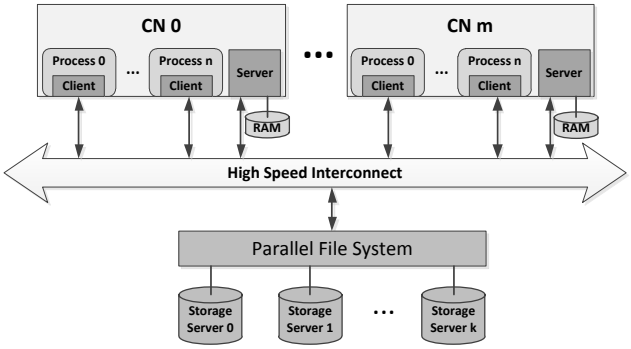

Fig. 1: Architecture of distributed cache framework on a MPP system.

There are two roles in the framework, server and client. Server is a daemon process resides in each CN, which is in charge of all the metadata and data requests. Each application process initiates a client in its address space when it first open a file. This is realized by intercepting all the I/O related POSIX functions. The client is responsible of handling all the I/O requests that it intercepts from application process, and forwarding them to a specified server.

We take the read operation to demonstrate the I/O path of our cache framework. The client first makes sure the requested file is not in the directories that hold system library files. If so, client would call the original I/O function to handle this I/O so that the request can be directly issued to underlying file system. In this way it can lighten the burden of caching these small, read-once files. Then, client ask a specified metadata server which is determined by a metadata management algorithm (depicted in Section 2.2), for the data server where the requested data located. Finally, the client retrieve the data from the specified data server. The procedure of write operation is similar except that data will be buffered in data server immediately. When data can be retrieved from local server, the data is directly read from local memory. Otherwise, data is transferred from remote server with Remote Direct Memory Access (RDMA).

2.2. Metadata Management

When designing distributed cache system, data coherence is a primary issue to concern. As multiple clients cache their own data independently, it is probable that their cached data overlaps. If any of the client modifies the overlapped data, data incoherency occurs, which means there are multiple different copies of a same data piece exist in the system.

To address this problem, traditional approach let the storage servers maintain the locks of all the data pieces. When lock conflicts occurs, storage servers would notify related clients to invalidate the stale data or write back the latest data. The drawback of this approach is that, the cost of maintaining coherence can easily exceeds the benefits of caching. In a distributed cache of a HPC system, the cost can be much higher. Firstly, in HPC environment, different processes of a same application work together to solve a large and complex problem, the possibility of data overlapping is higher. Secondly, in order to accommodate massive amount of data, the size of cache need to be larger than traditional ones. Larger size of cache brings higher possibility of data conflicts.

We propose an alternative approach by separating cache metadata and data. Cache metadata is scattered in all the clients through a hash-based algorithm, and cache data is preferentially stored locally and can be stored wherever suitable when local memory is limited. The way to decide where to store the metadata of requested data is shown in following equation.

$$ID_{metadata\_server} = Hash(file\_path)\%Number_{metadata\_server}$$

The hash function takes the absolute path of the requested file as input parameter. The hash values are uniformly distributed in its hash range, which ensures the load balance of servers. Clients can find the right metadata server of every I/O requests they handle with above equation.

Hash based metadata management has been widely used in parallel file systems like Ceph and Gluster. However, it has long been criticized for its poor performance of directory traversing. The proposed cache framework perfectly averts this drawback, because there is no user initiated metadata traversing in cache framework so that users would never experience the time consuming response.

3. Evaluation

3.1. Experimental Setup

We develop a prototype system of distributed cache framework. The code is written in C and has about 8K lines. The server is a daemon process starts along with the booting up of CN. The client is packed into a library and will be linked to application software when compiling.

Each CN has 24 GB RAM, and we take up about 8 GB to act as the cache space. The size of the space can be changed flexibly and dynamically according to different memory pressure of different applications. The memory space is organized with *tmpfs*.

3.2. Cache Block Size

In order to decide the block size of our cache framework, we conduct an experiment to inspect the performance of transferring different sized data blocks over network. The result is shown in Figure 2.

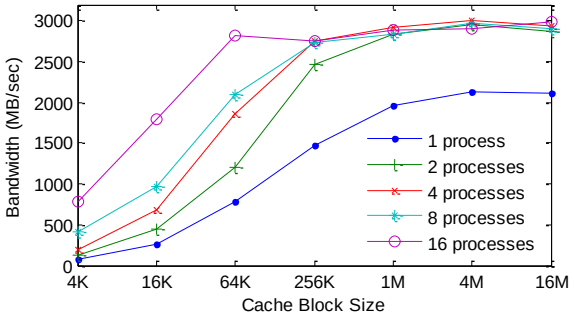

Fig. 2: Performance of transferring different sized data blocks over network.

Through the experiment we can conclude that, when transferring data blocks of smaller size, the network latency plays a more important role than the latency of access data in RAM. When the size grows larger than 1 MB, the network latency becomes insignificant. From this perspective, any cache block size larger than 1 MB is suitable.

On the other hand, smaller cache block size leads to finer cache granularity, which brings much flexibility to data placement. And as the stripe size of underlying PFS is usually set to 1 MB, it is better to align cache block to data stripe in PFS. Based on the above two considerations, we set the cache block size to 1 MB.

3.3. Metadata Performance

Hash-based metadata management policy distributes metadata to all the clients, which has better performance and reliability since it avoids the single metadata server becoming a hot spot. To evaluate this we develop a single metadata server version of our framework by directing all the metadata queries to just one server. The experiment is conducted in 64 CNs, on each of which we runs a process continuously access its own file with different I/O size. All the files are not cached in the cache framework before the experiment. The comparison is shown in Figure 3.

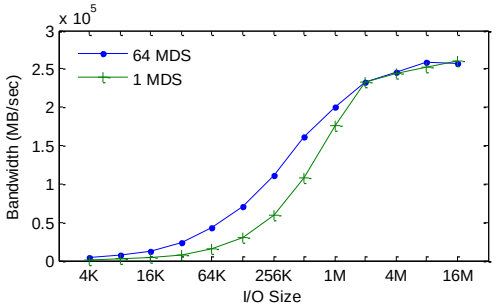

Fig. 3: Performance of hash-based metadata management and single metadata server.

When the I/O size is smaller than 2 MB, the performance of hash-based mechanism is higher than the single metadata server. When the performance gap is greatest at I/O size of 256 KB, the hash-based mechanism is almost 3X higher. As I/O size continues to grow, the amount of metadata queries decreases, and the performance gap is eliminated.

3.4. IOR Performance

IOR is a widely used benchmark to test the I/O performance in HPC system. In this test we set the –C –Z option, under which each IOR process will first write out a data bulk and then read in one produced by a random process. The experiment is conducted in 1024 CNs and on each CN runs an IOR process.

As shown in Figure 4, with the growing number of CNs, the performance of native approach without cache framework increases constantly until arriving the peak performance of Lustre, which is about 10 GB/s. But the performance of enhanced approach with cache framework increases without a upper bound. With 64 CNs, the performance of enhanced approach is already 20X higher than native approach.

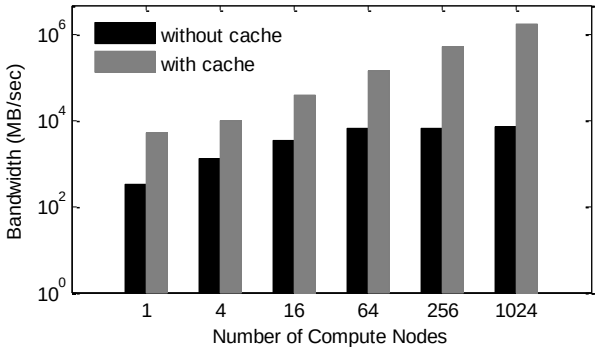

Fig. 4: Performance of IOR benchmark.

3.5. BTIO Performance

BTIO is a part of NASA parallel benchmark suite that solves 3D Navier-Stokes equations. BTIO partitions several multi-dimensional arrays across a square number of processes. Each process handle some subsets of the global arrays. As arrays are stored in row-major order in PFS, so each process will generates lots of small I/O requests to retrieve its subsets. If MPI collective mode is set, the small requests will be rearranged in a shuffle phase and merged into bigger request. We test both the independent I/O mode and collective I/O mode to investigate our cache framework under various I/O scenario.

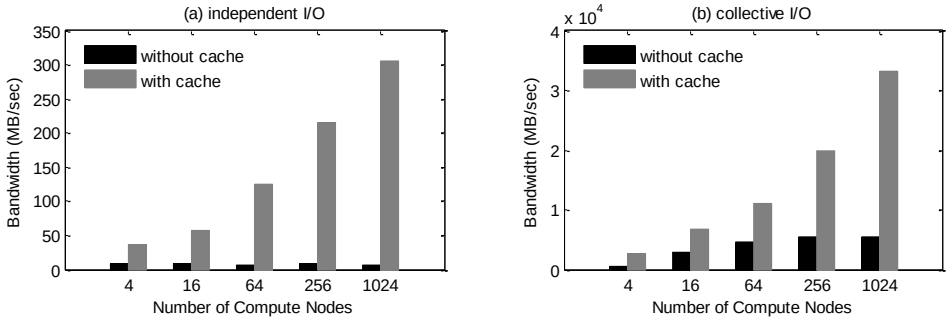

Fig. 5: Performance of BTIO (a) under independent mode and (b) under collective mode.

Figure 5(a) shows the experimental result of independent I/O. The performance of native approach without cache framework is extremely poor since Lustre is incompetent to handle small I/O. The enhanced

approach is much better as it caches a whole data block into cache framework on the first time it access a small piece of data. Most I/O requests are satisfied in local RAM or remote RAM. Although Lustre has prefetching in its readahead mechanism, the effect is poor since the I/O pattern is random. In Figure 5(b), the performance of native approach is a lot better than independent mode, because random, small I/O are aggregated into bigger requests. It soon reaches its peak performance, which is about 6 GB/s. The performance of enhanced approach is better than native approach at any number of CNs, and the performance scales out well with larger number of CNs.

4. Related Work

Lustre [2] and GPFS [3] file system provide coherent client-side caching with distributed locks. I/O requests get grants before accessing data, and the client will invalidate stale data or write back latest data if locks conflict. Due to the expensive maintaining of coherency, the size of their dirty cache is kept relatively small (32 MB in Lustre). Cooperative cache[5] was proposed to coordinate multiple distributed cache in a network to constitute a global cache. As a kind of cooperative cache, Collective Caching [6] is proposed to establish a global cache in the memory of MPI processes. Dong et al. [7] presented a distributed cache in the POSIX level, but most the cached data is stored in remote clients.

5. Acknowledgements

We thank Jinhua Feng, Jian Zhang, Fuxing Sun and Yuqi Li at NSCC-TJ for their generous help of setting up experimental environments. This work is supported by the National Natural Science Foundation of China under grand number 61502511.

6. References

[1] Q. Koziol et al., High performance parallel I/O. CRC Press, 2014.

[2] P. J. Braam et al., “The lustre storage architecture,” 2004

[3] F. B. Schmuck and R. L. Haskin, “Gpfs: A shared-disk file system for large computing clusters.” in FAST, vol. 2, 2002, p. 19.

[4] Ross R B, Thakur R. PVFS: A parallel file system for Linux clusters //Proceedings of the 4th annual Linux Showcase and Conference. 2000.

[5] M. D. Dahlin et al., “Cooperative caching: Using remote client memory to improve file system performance,” in Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation. 1994.

[6] W.-k. Liao, et al., “Using mpi file caching to improve parallel write performance for large-scale scientific applications,” in Proceedings of the 2007 ACM/IEEE conference on Supercomputing. ACM, 2007.

[7] W.-r Dong, et al. Sfdc: File access pattern aware cache framework for high-performance computer. In High Performance Computing and Communications, 2015 IEEE 17th International Conference on (2015).