論文の内容の要旨

EASILY DEPLOYABLE, SCALABLE, AND HIGH-PERFORMANCE DISTRIBUTED FILE SYSTEM FOR DATA-INTENSIVE DISTRIBUTED COMPUTING
（データ集約型の分散コンピューティングのための簡単、スケーラブル、かつ高性能な分散ファイルシステム）

氏名　　頓　　楠

The advance of parallel and distributed computing provides a promising approach for researchers to harness distributed computing resources to solve their larger-scale scientific problems with large amount of data, where distributed file systems have become the de facto standard solutions for data sharing. Though efficient data sharing plays an important role in data-intensive distributed computing, there are still several problems of conventional distributed file systems when they are deployed on resources across different domains in the wide-area environments. First, deploying distributed file systems usually requires root permission and non-trivial installation effort, which suggests that non-privileged users are only able to use resources having a distributed file system installed by administrators in advance. If users would like to use resources across different administrative domains, they have to rely on the collaboration of administrators who are allowed to do so. Second, even though a distributed file system could be installed on resources from multiple domains, it will encounter a problem of inefficiency and non-scalability if there is a single central component in the system. Specifically, in a distributed file system with centralized metadata management server, high- latency data accesses occur when a client that is actually close to the target data but needs to acquire information of the target data from a distant metadata catalog. Moreover, when applications have a tendency of locality of data access, this problem leads to an overall low metadata performance in wide-area environments because metadata of locally processed data always needs to be updated in the metadata server. Third, conventional distributed file system usually consists of a fixed number of servers settled at installation time, which indicates that system capacity is unable to scale on demand with the number of clients.

To tackle these problems, we propose GMount, a distributed file system that can be effortlessly and instantaneously deployed by non-privileged users on clusters, clouds, and supercomputers. GMount consists of one single building block called SSHFS-MUX, and uses GXP shell as its distributed loader. SSHFS-MUX is a standalone remote file system that uses two widely available modules in most Unix/Linux servers, i.e., SSH and FUSE. SFTP server of SSH is used for data transfer and FUSE allows non-privileged users to mount a user-level file system without interfering the system kernel. SSHFS-MUX enables users to manipulate files on multiple remote servers via one single mount point on local file system where a virtual namespace is made up of merged directories from target servers. By GXP distributed/parallel shell, users are able to interact with heterogeneous resources from one to many machines through a friendly and uniform interface and invoke commands or processes efficiently on arbitrary target nodes in parallel. GXP shell is written in Python and does not require other packages either. Thus,

the installation of GMount does not depend on additional software packages but only requires compilation of small code of SSHFS-MUX, and it can be done rapidly by using GXP shell when there are many nodes. The idea of GMount is using GXP to let nodes mount each other by SSHFS-MUX to construct a global shared namespace from export directory of each node. GMount first groups nodes by their network proximity, e.g., IP addresses or network topology. Then lower sharing layers are built by SSHFS-MUX mount within each group of nodes, and finally upper sharing layers are created among these groups in a hierarchical way using similar mount algorithms. By this approach, GMount achieves locality-aware file lookup because one target file is searched within local groups first and then in other groups in order of network affinity. Therefore a client node can quickly find and access target data without sending remote messages if it is close to the node having target data. This feature of GMount not only solves the problem of high latency access when using central metadata server, but also improves the scalability of metadata performance when applications have more local data reference than global data reference. Additionally, more clients suggest more storage and bandwidth in GMount. It is because each client is actually a server node and can make use of its own local file system. Nevertheless, GMount is not designed to be a distributed storage solution for users to store their data permanently but an instantaneous and efficient sharing approach for data-intensive computing. Although GMount has a limited data transfer rate by using default SSH channel, better I/O performance over wide-area links can be achieved by using SSH bypassing technique or HPN-SSH patch. Our routine use of GMount shows that it can be deployed on hundreds of machine from scratch in a couple of minutes. Evaluation using both parallel micro-benchmark and real-world application illustrates that GMount has both highly scalable metadata and I/O performance in multiple clusters environments. In a configuration composed of 64 nodes from 4 clusters in wide-area, GMount achieves up to an average metadata performance of 50,000 ops/sec (50x speedup comparing to Gfarm) when all metadata accesses are local. For the worst case when every client searches non-existing files or creates new files in root directory, GMount has 3 to 4 times lower overall metadata performance than Gfarm. Accordingly, if application itself or smart scheduler can bring on sufficient access locality, GMount delivers better overall metadata performance than Gfarm in common case. By investigating the execution of real- world scientific applications on multiple clusters, we demonstrate that GMount has practically useful performance (15% speedup comparing to Gfarm) and are suitable for usual practice of data-intensive workflows in various distributed computing environments.

To help users further optimize the orchestration of their data-intensive workflows, and since profiling is an effective approach to investigate realistic behaviors of such complex applications, we developed a fine-grained profiler called ParaTrac. Different from traditional kernel-level file system tracer, ParaTrac uses entirely user-level tracing techniques, i.e., FUSE for file system call trace, /proc file system, kernel task accounting, and ptrace for processes trace. This allows non-privileged users to conduct the profiling on unmodified executables for both data and processes simultaneously. Comparing to other profilers using particular workflow management systems interfaces that only give coarse-grained jobs and files information, ParaTrac uses general-purpose tracing techniques and provides more fine-grained processes and data information. There are two components in ParaTrac: a FUSE-based tracer implemented in C and a profile generator implemented in Python. To profile an application, users only need to change the working directory of application to the file system directory mounted and monitored by tracer, then invoke profile generator to process trace logs to produce application profiles. The tracer records parameters and latencies of file system calls, and monitors the activities of corresponding processes that execute these file system calls. The profiler generator integrates runtime log, system call log, and process logs produced by tracer into one database and uses statistical or causal analysis to retrieve workflow information from database. ParaTrac uses statistical analysis to examine trace logs and produces low-level I/O profiles, which enables users to quickly understand the I/O characteristics of from entire application to specific processes or portion of data. For example, whether an application is metadata-intensive, I/O intensive, or CPU-intensive can be identified by the ratio of execution time spent on corresponding operations. Besides, ParaTrac uses temporal and causal analysis to explore fine-grained interactions between data and processes in workflow execution and provides users with intuitive and quantitative workflow profiles for detailed analysis. To achieve this, ParaTrac first constructs a directed acyclic graph according to the dependencies between data and processes, and annotates vertices and edges with corresponding runtime information. Then various graph-theoretic algorithms can be applied to this graph for different types of analysis. For example, if edge weight is annotated by elapsed time between vertices and the critical paths are found, the makespan of workflow can be accordingly estimated. Though user-level trace has a 16% overhead comparing 7% overhead by kernel-level trace, it provides higher usability for general users to study their applications. Experiments on thoroughly profiling real-world workflow by using different underlying data sharing approaches demonstrate that high-latency metadata operations can be easily detected from I/O profiles. The actual data transfer among jobs illustrated in workflow DAG can give hints to data dispatcher to use a data prefetching or throttling strategy, or to job scheduler to assign tasks to close data.

To draw a comparison between different distributed/parallel file systems, we designed and implemented a parallel benchmark called ParaMark. While other existing parallel benchmarks usually report only overall system performance as the benchmark results, ParaMark allows users to investigate the performance results in more fine-grained and suggestive way. Besides overall system performance, ParaMark can aggregate performance at different granularity level, from threads to processes, nodes and clusters, which can be used to reveal bottlenecks that exist in specific system components but are usually hidden in whole-system-oriented tests. In addition, ParaMark is able to plot the variation of throughput during benchmarking time in its performance report, thus user can accurately analysis the dynamic behaviors of target systems or specific components. It is especially useful to evaluate those systems where performance varies from time to time. By these features, ParaMark provides more comprehensive comparisons and analysis between different parallel file systems with better grounds than simply head-to-head comparison with end results. ParaMark achieves this by thoroughly logging every system call invoked in benchmarking processes for later global analysis. ParaMark is written in Python and, like GMount and ParaTrac, uses GXP shell as the back-end for parallel execution and collective operations. We have used ParaMark for routine evaluation of distributed file systems deployed in multiple clusters and supercomputer, e.g., NFS, Lustre, Gfarm, and GMount.

This research targets data-intensive applications and its contribution is threefold: First, GMount is novel because it demonstrates an approach to build a distributed file system from simple and small components with considerable low development cost. It is practically useful for realistic data-intensive computing practice because it significantly lowers the barrier to conduct efficient and scalable data sharing among arbitrary resources for general users, and is able to utilize locality file access to achieve better performance in wide-area environments. Second, ParaMark and ParaTrac, as useful auxiliary tools, help researchers solve problems in data-intensive computing study from different angles, i.e., file systems and applications. In particular, they benefit recent effort in industry on standardizing system benchmark by seeking common factors in different distributed file systems and applications. Finally, GMount, ParaMark, and ParaTrac are free and open source software that especially values the practical usability for general users, which makes active and tangible contributions to the research community.