Easy and Instantaneous Processing for Data-Intensive Workflows

Nan Dun* Kenjiro Taura† Akinori Yonezawa‡
*‡Department of Computer Science, †Department of Information and Communication Engineering
The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo, 113-8656 Japan
*dunnan@yl.is.s.u-tokyo.ac.jp †tau@logos.t.u-tokyo.ac.jp ‡yonezawa@is.s.u-tokyo.ac.jp

***Abstract*—This paper presents a light-weight and scalable framework that enables non-privileged users to effortlessly and instantaneously describe, deploy, and execute data-intensive workflows on arbitrary computing resources from clusters, clouds, and supercomputers. This framework consists of three major components: GXP parallel/distributed shell as resource explorer and framework back-end, GMount distributed file system as underlying data sharing approach, and GXP Make as the workflow engine. With this framework, domain researchers can intuitively write workflow description in GNU make rules and harness resources from different domains with low learning and setup cost. By investigating the execution of real-world scientific applications using this framework on multi-cluster and supercomputer platforms, we demonstrate that our processing framework has practically useful performance and are suitable for common practice of data-intensive workflows in various distributed computing environments.**

I. Introduction

Data-intensive scientific applications generally require large number of computing resources to process large amount of data. However, for most researchers, there still exists a significant gap between having many heterogeneous resources and easily executing applications by fully making use of these resources. We identify following major facts that lead to this problem in data-intensive computing practice.

- *Deployment effort*: Common data-intensive computing practice needs a number of underlying middleware, such as resource management system, distributed file systems, and workflow management systems, to be properly installed, configured, and maintained on all servers before target applications can run on the top them. This nontrivial task is usually undertaken by experienced system administrators for general users. Additional burden of re-installation or re-configuration will be introduced when changes are made to computing resources, such as servers are appended or removed.
- *Resources across domains*: Once specific servers are set up by administrators, users are only able to run their applications on resources where required middleware is available. However, in real world, more and more users tend to have access to computing resources that belongs to different administrative domains, such as companies, universities, or institutions. The heterogeneity of resources lies not only in hardware, software, configuration, but also in usage interface, policy, quota, etc. This further limits the usability and scalability of harnessing all available resources as a unified platform. For example, there is not a unified interface to allow users to manipulate resources via different channels such as RSH (Remote Shell), SSH (Secure Shell), or batch queue. It is also practically prohibitive to ask administrators to install a distributed file system across different domains only for individual user's temporary data sharing purpose.
- *Data sharing efficiency*: The performance of underlying data sharing approach is critical to overall workflow execution because large amount of data need to be passed between jobs distributed in separated servers. Though data staging is doable for wide-area sharing, it is tedious, non-extensible, non-scalable, and error-prone in practice. Using distributed file systems is more convenient but also suggests more challenges in large-scale, and especially wide-area, environments. A suitable distributed file system should: 1) be easy to deploy across domains; 2) have scalable metadata and I/O performance; 3) be aware of potential data locality property in workflow execution and take advantage of it; 4) have optimal data transfer rate in wide-area networks.
- *Workflow synthesis effort*: Usually different workflow engines have different workflow description languages and require users to learn and translate processing procedures of complex workflows into corresponding workflow description file for submission. Therefore, a workflow description language should be sufficiently expressive and easy to learn for users.

Our work targets domain researchers who wish to use arbitrary accessible resources from multiple domains to run their own data-intensive applications. First, domain researchers implies ordinary users who do not have to have sufficient system knowledge to install middleware for themselves. The only thing for domain researchers to learn is how to describe their workflows in specific workflow languages. Second, we assume that users have access to a set of computing resources, or servers, via standard interfaces such as RSH, SSH, or batch queue. No middleware is required to be installed on servers in advance. Third, the data-intensive workflows stands for a category of applications consisting of many data processing

978-1-4244-9705-8/10/$26.00

tasks, where tasks are synchronized and communicate through files instead of messages [1].

Accordingly, we propose a light-weight and scalable processing framework that enables non-privileged users to effortlessly and instantaneously describe, deploy, and execute data-intensive workflows on arbitrary computing resources from clusters, clouds, and supercomputers. This framework consists of three major components: GXP parallel/distributed shell [2], [3] as resource explorer and framework back-end, GMount distributed file system [4] that can be quickly built using one single building-block SSHFS-MUX [5], and GXP Make workflow engine [6] based on GNU Make [7]. All of them follow a consistent design philosophy that users start with minimum initial cost and then are able to seamlessly extend to various environments in an ad-hoc manner. Our framework is practically useful especially for those domain-specific users who know little about workflow execution context but want to straightforwardly run their applications on available resources with minimum learning and setup cost. Since the resource explorer, distributed file system, and workflow engine are all integrated within GXP shell, our all-in-one solution further lower the barrier for more researchers to leverage the power of distributed resources for high performance data-intensive computing.

II. Related Work

Workflow systems virtually depends on an underlying data sharing systems for passing data in workflow execution. Though data staging system can be integrated into the workflow system, like Makeflow [8], a more generic way in practice is still using distributed file system. Conventional parallel/distributed file systems, such as NFS [9], [10], PVFS [11], [12], Lustre [13], and GPFS [14], are mostly used in single cluster or supercomputer environment. Evaluation of these file systems in wide-area environments [15], [16] shows that they require non-trivial administrative complexity to deploy and have limited performance in multi-cluster environments. Other distributed file systems for multi-cluster includes LegionFS [17], WAN-GPFS [18], HDFS [19], Gfarm [20], [21], and Ceph [22], [23]. To deploy such distributed file systems in wide-area environments, some of them require root privilege to make kernel modification or custom low-level storage device, and most of them need nontrivial effort to install a heavy stack of software and conduct the performance tuning. Therefore, they can not be easily installed by a single non-privileged user. In our routine practice, we use Gfarm [20], [21] (installed beforehand by system administrators), SSHFS [24], or GMount [4] as file sharing alternatives for different environments and configurations.

Other processing models and workflow systems include Condor [25], MapReduce [26], Hadoop [27], Swift [28], [29], Falkon [30], Pegasus [31], Kepler [32], Dryad [33], etc. They target different scenarios with different expressiveness and features, but most overlooked the usability of being an easy system for end-users. Thus our GXP Make focuses on providing an expressive, but easy to learn/use workflow system. A detailed comparison between GXP Make and other workflow systems has been described in [6].

The Cooperative Computing Tools (cctools) [34] are a collection of software that shares similar objectives as our framework. It also includes a user-level distributed file system Chirp [35] that can be quickly deployed and a workflow engine Makeflow [8] also using *Make* like language. However, our framework differs from cctools in several aspects. First, cctools requires to be installed beforehand in all servers but our framework can be initialed from one single node and automatically extended to many nodes during the resource exploration phase. This property is especially useful for non-dedicated computing resources that has to be cleaned and recycled after usage, such as cloud servers. Second, Chirp distributed file system employs a central metadata server (i.e. *directory server*) architecture like Gfarm and may have a scalability problem when the number of servers increases. Third, Chirp does not address the locality-aware problem existing in wide-area environments.

Different from our another publication on the design and implementation details of GXP Make [6], this paper conducts a comprehensive study on challenges of executing data-intensive workflows in wide-area environments, from the point of view of both underlying shared file system and workflow system.

III. Processing Framework

A. GXP Parallel/Distributed Shell

GXP parallel/distributed shell [2], [3] is a shell-like remote command/process invoker for distributed multi-cluster environments. Using GXP, users can execute commands on many nodes simultaneously and manipulate (i.e. select, deselect, append, or remove) nodes dynamically. A remarkable advantage of GXP shell is that it does not need to be installed on target nodes in advance but can be rapidly deployed from one single initial node. Since GXP is written in Python, the installation process does not require other software and compilation. We refer interested users to [2], [6] for details of GXP design and implementation.

GXP is used as the resource explorer and back-end in our processing framework. We illustrate the resource exploration here and leave the description of back-end usage in Section III-B and Section III-C. Interactively using GXP shell to explore resource includes following steps:

1) *Installation*: Download and extract GXP to *one initial node* which user interact with. Then the user is ready to issue `gxpc` commands without any setup in the rest of target nodes.
2) *Configuration*: Specify login channel (e.g. SSH, RSH, or TORQUE batch queue [36]) and connection rules (i.e. which hosts are reachable from which hosts) to use by `use` command.
3) *Resource Acquisition*: Issue `explore` command with specifying target nodes, which make GXP discover, login to, start GXP daemons on all target hosts, and finally report successfully explored resources.

Then users are able to manipulate explored resources using other GXP commands. In practice, we also start VGXP [37], [38], a visual extension of GXP, to monitor the real-time status of resources, such as CPU usage, memory usage, I/O traffic, network traffic, etc.

B. GMount Distributed File System

GMount is a distributed file system that can also be easily and instantaneously deployed by non-privileged users on arbitrary resources [4]. It is designed to adapt the target scenario we are discussing in this paper. Comparing to other wide-area distributed file systems referred in Section II, GMount has considerable low setup cost, locality-aware file operations, and scalable storage and performance. GMount is composed of one single small software called SSHFS-MUX [5]. Like SSHFS [24], SSHFS-MUX uses FUSE [39] technique to allow unmodified binaries to access the file system, but SSHFS-MUX also allows users to simultaneously mount multiple servers to create a merged namespace at the local mount point. Figure 1 shows two common mount operations used in GMount.

Merge mount

```
X$ sshfsm Y:/export Z:/export ... /mount
```

Now X can access Y's export and Z's export from a unified namespace at mount.

Cascade mount

```
B$ sshfsm C:/export /mount
A$ sshfsm B:/mount /mount
```

Now both A and B can access C's export from mount.

Fig. 1. Two basic SSHFS-MUX mount operations

The basic idea of GMount is to invoke `sshfsm` commands simultaneously on all nodes and let them properly mount each other to construct a shared file system. Therefore, we use GXP shell as the distributed loader and implement a GXP script/command to automatically achieve above complex job. The detailed mount algorithms of GMount has been illustrated in [4], where locality-aware file operations and scalability of GMount is also fully described.

Recent implementation achieved a significant improvement of GMount I/O performance. It includes using raw socket as the data transfer channel to bypass default OpenSSH [40] that suffers from low throughput due to limited buffer size and SSH encryption overhead [41]. Though this problem can be solved by patching OpenSSH client and server using HPN-SSH [41], [42], it is still introduces additional installation effort for GMount deployment. Another improvement enables nodes in GMount to transfer data through independent direct link instead of GMount overlay connections.

C. GXP Make Workflow Engine

GXP Make [6] is a workflow system built on top of, and integrated in GXP shell. Besides many advantages inherited from GXP shell, such as easy installation, portability, and unified interface across platforms, GXP Make provides fully compatibility with unmodified GNU Make [7] and fast dispatching performance. Different from other workflow systems, GXP Make adopts Unix *make* language as its workflow description language, which is straightforward for data-oriented application and sufficient to describe various processing paradigms, including popular MapReduce [26]. We refer interested users to [6] for further details of GXP Make workflow system.

D. Workflow Execution from A to Z

In this section, we use actual commands to illustrate the simplicity of how users start from scratch to final workflow execution by using our framework.

1) Workflow Synthesis: First, user needs to describe the processing procedure of target application in a *makefile* by using make language. This is the only thing that user should spend some time to learn if s/he is not familiar with make. Since make has been widely used, it is not hard for users to find related manuals and examples to learn. Furthermore, users can use local installed make program to easily debug and refine their makefiles with useful features such as *dry run* '`-n`' option and *parallel control* '`-j`' option [7].

2) Workflow Execution: After the makefile is created, the rest of work is as easy as issuing several GXP commands at the interactive node. Figure 2 illustrates the exact steps to run a workflow on many servers. Suppose GXP has been downloaded and installed on the initial node and we use three clusters named as *A*, *B*, and *C*.

Though there are many other options that can be used to fine-grained control resource exploration settings, distributed file system structure, and workflow execution configurations, users are supposed to follow this straightforward way as shown in Figure 2 to execute workflows without being aware of underly heterogeneity of environments. Note that the GMount, and GXP Make are independent components and can be used with other distributed file systems or workflow systems for comparison and other purposes.

IV. Experiments and Evaluation

A. Experimental Settings

We evaluated our framework on InTrigger multi-cluster platform [43] and HA8000 supercomputer [44]. InTrigger is a cluster collaboration that consists of 16 clusters distributed around Japan. It has approximately 400 nodes and 1,600 CPU cores in total, and each cluster has different specification of machines. The HA8000 system, located at the University of Tokyo, has 952 Hitachi HA8000 servers. Each node is equipped with a four way Quad Core AMD Opteron 8356 (2.3GHz) and 32GB memory.

We used both micro-benchmark and real-world scientific applications in evaluation. We also developed a parallel benchmark for distributed file systems called ParaMark [45] as the micro-benchmark. For real workflow, an event recognition workflow that extracts and classifies bio-molecular events mentioned in English text from PubMed database [46] was

1. Configure

```
$ gxpc use ssh A B
$ gxpc use torque A C
```

Now GXP knows A nodes connect to B nodes via SSH, and A nodes connect to C nodes via RSH.

2. Explore resource

```
$ gxpc explore A[[000-015]] B[[000-031]]
$ gxpc explore C[[000-015]]
```

Now total 64 nodes from A, B, and C are explored if all nodes are available. It usually takes up to a few minutes for hundreds of nodes from multiple clusters in wide-area.

3. Deploy GMount

```
$ gxpc e install_sshfsmux.sh
$ gxpc mw gmnt /export /mount
```

'e' command is used to install SSHFS-MUX in parallel on all nodes, which usually takes couples of seconds. 'mw' command is used to execute GMount script and invoke sshfsm commands on all nodes, which takes less than 10 seconds for over 300 nodes in wide-area [4]. Now every node can access a shared namespace via mount directory.

4. Run workflow

```
$ gxpc make -j N -f makefile
```

Workflow is started to run on all nodes with maximum N concurrent jobs using the procedure specified in makefile.

5. Unmount distributed file system

```
$ gxpc mw gmnt -u
```

After the workflow is finished, user can easily destruct the file system. All sshfsm processes are terminated.

6. Exit GXP shell

```
$ gxpc quit
```

Stop all GXP daemons, perform cleanup, and disconnect from all nodes.

Fig. 2. Basic steps to run a workflow on distributed resources

used. The brief description and further references of this application have been given in [6] and its main workflow structure is shown in Figure 3.

When running the workflow across multiple clusters, we used two data sharing configurations. One straighforward way is using GMount file system. Another is using the mount scheme as shown in Figure 4. In this scheme, we first arbitrarily choose a *master site*. Then we let all nodes in master site (marked by M) *directly* mount the NFS server raid (since it has better throughput than local disk) by SSHFS-MUX. Finally, we use *cascade mount* introduced in Section III-B to let nodes from other clusters (marked by N) mount the M nodes in a round-robin way. Here M nodes play a role as "buffer" that mitigate the I/O traffic from many N nodes, and they also share the load of sftp-server daemon started for each SSHFS-MUX client. We use this simple mount

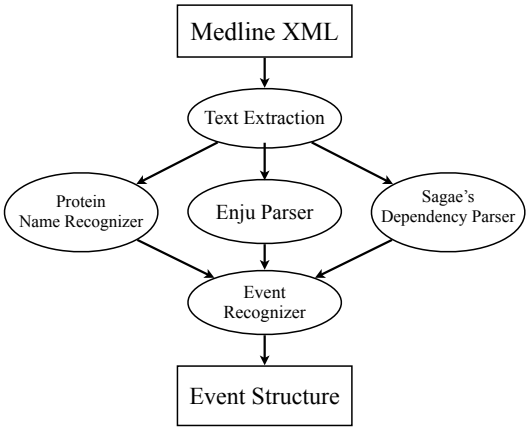

Fig. 3. Event Recognition Workflow

scheme for several reasons. First, it allows us to investigate the robustness and performance of M node under the load of serving many incoming data requests. Second, this scheme is useful in practice when GMount can not be deployed in some environments. For example, HA8000 system does not have FUSE installed but has a powerful Lustre file system to server their nodes. Then we can let nodes from other clusters directly mount HA8000 nodes to use the Lustre file system.

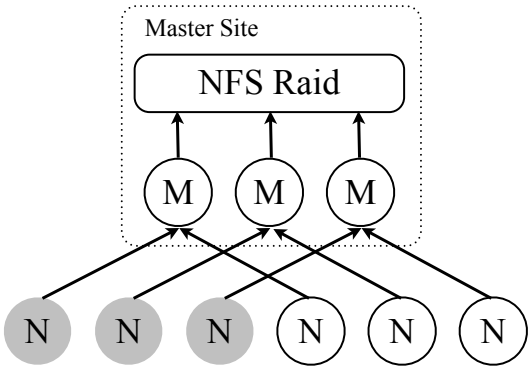

Fig. 4. All-to-one mount scheme

B. SSHFS-MUX Performance

Since our workflows are data-intensive, underlying data transfer rate in LAN (Local-Area Network) and WAN (Wide-Area Network) are critical to the overall performance of workflow execution. A FUSE-based file system always has overhead coming from context-switch between kernel and user space [47], [48]. We also discussed limited SSH data transfer rate and possible workarounds in Section III-B. In following experiments, unless otherwise indicated, SSHFS-MUX (denoted by SSHFSM-Direct) is used with these configurations: using socket to bypass SSH, allowing big writes and kernel cache in FUSE module.

For intra-cluster benchmarking, we used one of InTrigger clusters consisting of 18 nodes. Each node has Xeon E5530

2.4GHz and 24GB memory, and connects with other nodes via 10Gbps network. Within the same cluster, we compare SSHFSM-Direct with NFSv3 server for both single-client performance and parallel access performance. For SSHFSM-Direct, we use SSHFS-MUX from client nodes to directly mount the NFS server node.

Figure 5 shows the comparison of metadata operation performance. SSHFSM-Direct achieves an average 30% of NFS performance. Some popular metadata operation, such as `stat` operation, still have a 60% of NFS performance because of cache effect. This is basically due to the FUSE context switch overhead in SSHFS-MUX.

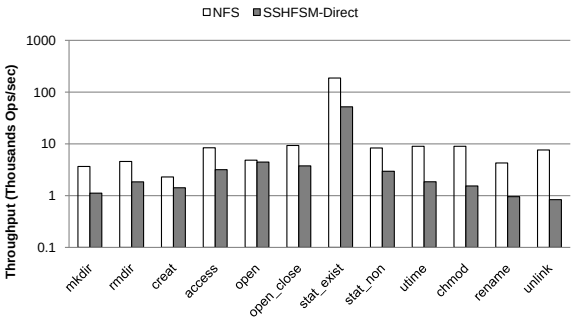

Fig. 5. Metadata performance in LAN

The single-client write and read performance by I/O request size are shown in Figure 6 and Figure 7, respectively. SSHFSM-Direct has close performance as NFS when I/O request size is large than 64KB. It is because small requests are dispatched immediately in SFTP protocol and does not batched for buck transfer as in NFS. If read takes place right after writes on the same file, local kernel cache will leads to a significant high read performance as in NFS.

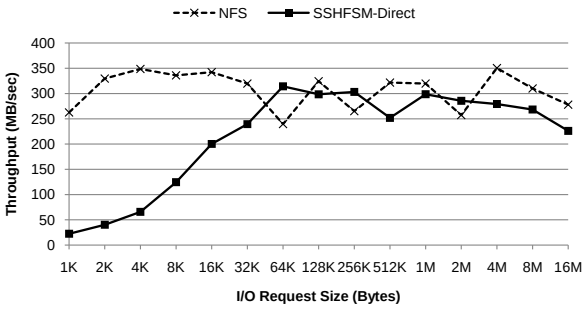

Fig. 6. Write performance by request size in LAN, file size set to 4GB.

According to I/O performance comparison by file size, as shown in Figure 8 and Figure 9, we found that SSHFSM-Direct generally outperforms NFS when I/O request size is large. Note that here we eliminated the cache effect by letting client start to read after all write tests are done, so cache for previously accessed files is already swapped out.

To investigate the I/O scalability of our approach, we increase the number of concurrent clients from 1 to 8, where

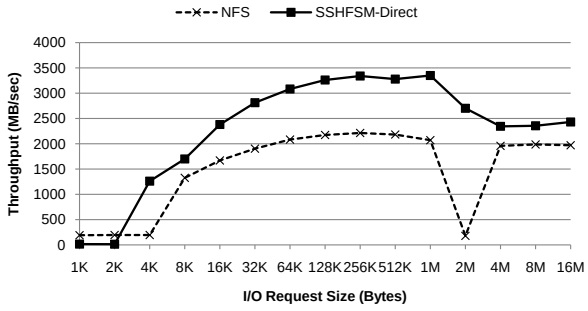

Fig. 7. Read performance by request size in LAN, file size set to 4GB.

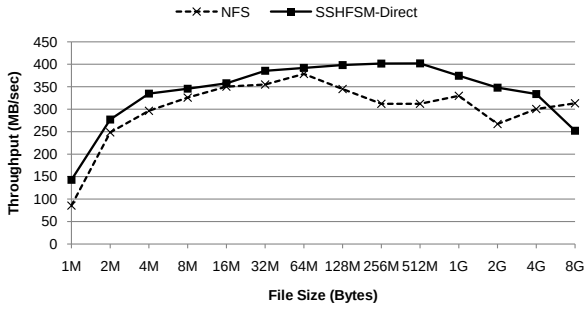

Fig. 8. Write performance by file size in LAN, I/O request size set to 1MB.

each client write/read a separate file. Figure 10 and Figure 11 shows the results. Write is saturated when the number of clients reaches 4 because the NFS server node becomes the bottleneck. While read performance scales to 8 clients because of local cache on each client.

We used two hosts located in two distant clusters to measure the I/O performance of SSHFSM-Direct in wide-area environments. By Iperf [49], the TCP bandwidth between these two hosts is 150Mbits/sec and average RTT time is 23.6 msec. From the comparison to ordinary SSHFS [24], shown in Figure 12, SSHFSM-Direct achieves a significant improvement (close to optimal TCP bandwidth) than SSHFS since kernel can do auto TCP-tuning and chose optimal TCP buffer for high-latency links when using raw sockets for data transfer.

C. GXP Make scheduling performance

On the same InTrigger platform, a particular stress test for GXP Make by dispatching large-amount of small jobs was conducted and reported in [6]. It is reported that GXP make reaches a useful dispatch rate of 62 tasks per second, comparing to a 56 task per second rate of Swift-Falkon combination in TeraGrid and a few tasks per second in other Grid middleware [28].

D. Workflow on Single Cluster

We start our investigation by using real applications first in one single cluster. Another InTrigger cluster consisting of one NFS server and 10 computing nodes was used. Each

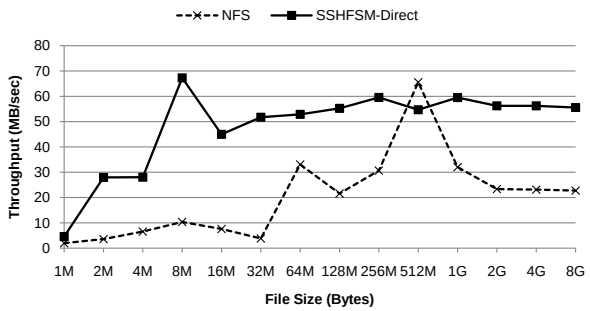

Fig. 9. Read performance by file size in LAN, I/O request size set to 1MB.

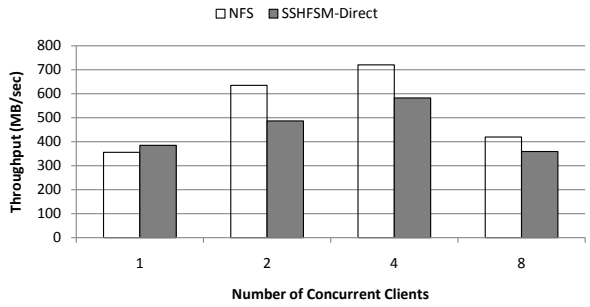

Fig. 10. Parallel write performance by concurrent clients in LAN, file size is 1GB and I/O request size is 1MB.

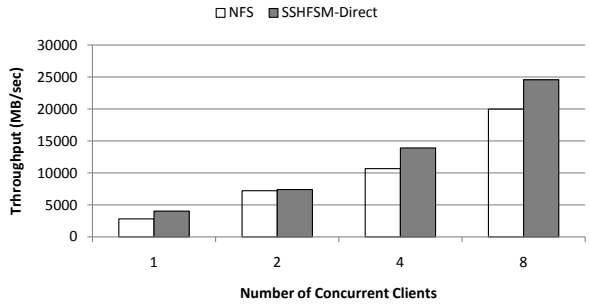

Fig. 11. Parallel read performance by concurrent clients in LAN, file size is 1GB and I/O request size is 1MB.

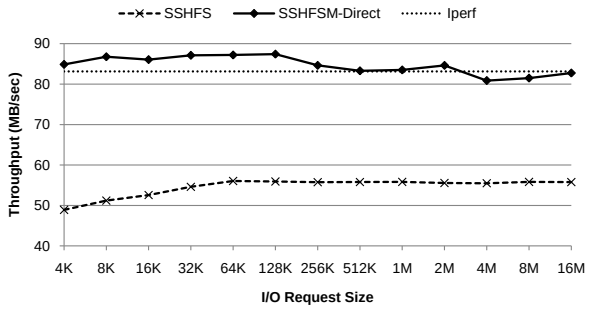

Fig. 12. Write performance by file size in WAN, file size is 4GB.

TABLE I
RUNTIME SUMMARY OF WORKFLOW

DFS	Total		Per-Job Elapsed (sec)	
	Jobs	*Elapsed (sec)*	*Avg*	*StdDev*
NFS	159	2684.08	798.786	393.99
SSHFSM	159	2660.19	796.09	395.95

node has Xeon E5410 2.33GHz CPU, 16GB memory, and 1Gbps link. In this experiment, we used one PubMed [46] data that splits to 159 XML input files as denoted in Figure 3. Except the dispatcher (GXP make master node), a number of 8 concurrent jobs (8 cores per CPU) was specified for all workers. Therefore, the allowed maximum number of concurrent jobs is set to 72 (i.e. '`-j 72`') in our experiment.

The comparison of workflow execution using SSHFSM-Direct all-to-one mount and NFS is given in Table I. When SSHFSM-Direct uses similar file sharing architecture (i.e. multiple clients mount one single server) as NFS, SSHFSM-Direct has close performance as NFS.

We also studied the correspondence between task processing time and input data size. Here one task refers to the entire processing cycle from initial input to final output, as illustrated in Figure 3. The task processing time is defined as the creation time of first XML input data subtracting from the creation time of final tag file. Figure 13 illustrates the correspondence of these two values. Basically small input data requires less processing time. Average long processing time happens when file size is between 30KB to 50KB. However, when file size becomes larger than 60KB, the processing time does not increase accordingly but remains around 1000 seconds. This results indicates that the event searching time depends not only on the amount of text but also on the internal structure of text.

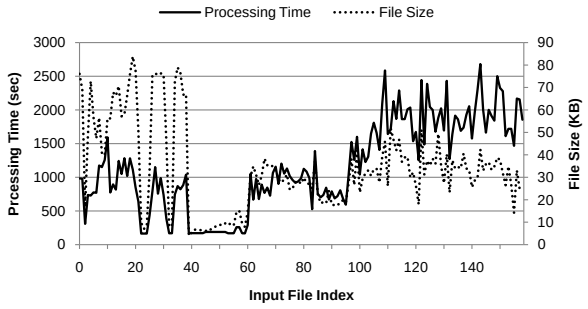

Fig. 13. Processing time vs. file size

Using task processing time as a metric, we compared the difference of per-task processing time by using SSHFSM-Direct and NFS. Figure 14 shows the results, where positive value indicates a task takes more time by using SSHFSM-Direct than NFS, and vice versa. From the distribution in Figure 14, it not hard to find that SSHFSM-Direct saves more time than NFS for individual tasks. As a result, SSHFSM-Direct gained 1183 seconds in total if we aggregate all

difference values of task processing time.

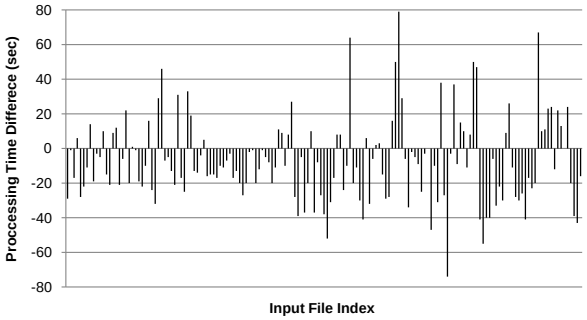

Fig. 14. Per-task gained time by using SSHFSM-Direct

The parallelism during the workflow is shown in Figure 15. Note that the curve of NFS is very close to the one of SSHFSM-Direct, thus it is not plotted for clarity. The peak parallelism reached soon after the workflow was started, and lasted for about half of runtime of entire workflow.

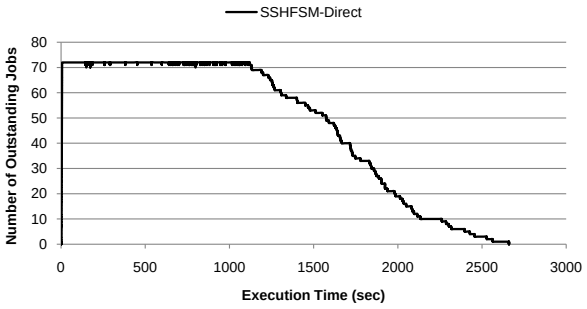

Fig. 15. Parallelism during workflow execution using single cluster

In Figure 16, the load of job dispatcher (GXP make master node) also boosted at the very beginning because it was busy dispatching many jobs to initially idle nodes. Those minor peaks appearing at about 1600 seconds stand for the dispatching of jobs for aggregation in workflow. The outstanding load of NFS during 1000-1500 seconds comes from a system disturbance. Figure 17 shows the received, sent, and finished jobs during the workflow. The curves of NFS are also very close to the ones of SSHFSM-Direct in Figure 17 so it is ignored for clarity.

E. Workflow on Multiple Clusters

1) Using All-to-One Mount Scheme: For evaluation in multi-cluster environments using the all-to-one mount scheme shown in Figure 4, we used 147 available nodes from 11 sites of InTrigger and 5 PubMed datasets that produce 821 XML files. Here we assigned 4 parallelism for each node and thus results an overall maximum parallelism of 584.

The workflow successfully finished in 4422.72 seconds. The average and standard deviation of job elapsed time are 872.98 seconds and 481.19 seconds, which are both larger than the values in single-cluster case.

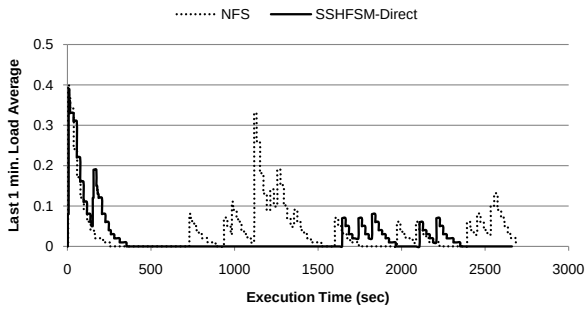

Fig. 16. Load of dispatcher during workflow execution using single cluster

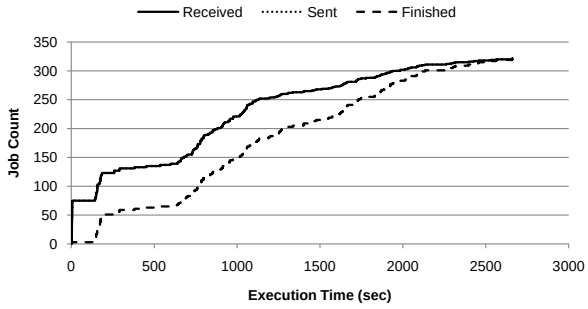

Fig. 17. Job counts during workflow execution using single cluster

From Figure 18 we can find a potential problem of running workflow in multi-cluster environments. In wide-area, the peak parallelism only lasted for about 850 seconds or about 20% of total runtime, then followed by a very long tail (lasting for about 2400 seconds) starting from about 2000 seconds. We investigated these long lasting jobs by running them indvidually on local file system on the same node and achieved a much lower runtime. This is due to some of sites are distant from the master site, therefore the wide-area links between them are high in latency and low in bandwidth. Accordingly, an optimal data sharing approach should preserve data processing locally as much as possible.

Fig. 18. Parallelism during workflow execution using multiple clusters

Comparing to the load of dispatcher and jobs count in single-cluster case (in Figure 16 and Figure 17), both the load of dispatcher and scheduling throughput are higher (in Figure 19 and Figure 20). Furthermore, both the values scales

TABLE II
RUNTIME SUMMARY OF WORKFLOW

DFS	Total		Per-Job Elapsed (sec)	
	Jobs	*Elapsed (sec)*	*Avg*	*StdDev*
GMount	159	2759.98	994.22	431.85
Gfarm	159	3257.27	926.30	663.99

with the parallelism. For example, when the parallelism is increased by 8 times from 72 to 588, the corresponding load and scheduling throughput are also grows by 8-10 times.

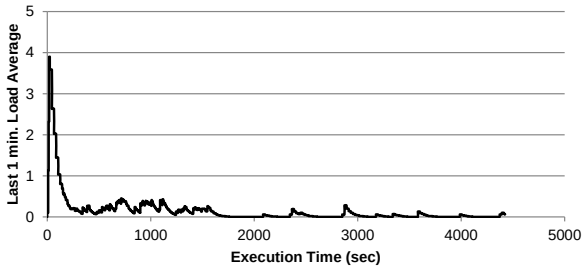

Fig. 19. Load of dispatcher during workflow execution using multiple clusters

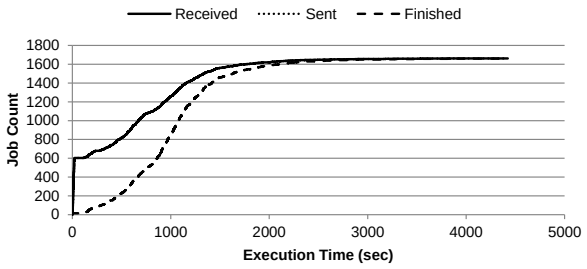

Fig. 20. Jobs count during workflow execution using multiple clusters

2) Using GMount: For evaluation using GMount, we used 64 nodes from 4 sites, assigned 4 jobs for each node and results an overall maximum parallelism of 252. The dataset used in this experiment is the same as the one in single cluster experiment (Section IV-D). We also used Gfarm [20], [21] as an alternative distributed file system for comparison.

Table II shows the comparison of execution summaries. Using GMount achieved 2759 seconds execution time that is close to the runtime in single-cluster using NFS/SSHFS-MUX (Section IV-D) and has 15% speedup over using Gfarm.

The peak parallelism 159 when using Gfarm was reached at 28 second, and 156 parallelism when using GMount was reached at 198 second. Workflow on Gfarm having faster initial performance is because many files are created at the beginning. In Gfarm, metadata server can quickly reply these requests since it holds all metadata information. In GMount, a file should be search in all nodes to check its existence before the file creation is allowed, which results lower performance than Gfarm when many new files are created.

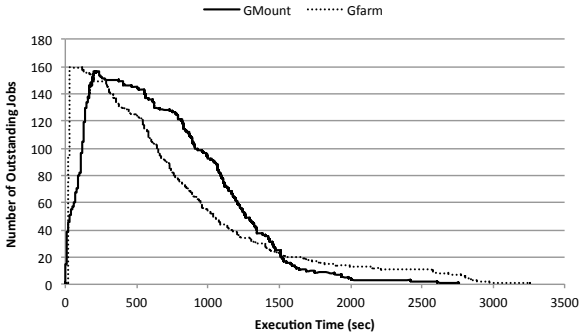

Fig. 21. Jobs count during workflow execution using multiple clusters

F. Workflow on Supercomputer

Our practice on large systems also includes supercomputer. As reported in [6], we deployed GXP make on one InTrigger cluster and HA8000 system [44]. However, in that experiment, we used a hybrid data sharing approach because HA8000 system does not have FUSE for GMount. We put GXP Make dispatcher in the InTrigger cluster (i.e. out of HA8000 system), and then used SSHFS to share output data between two clusters via several HA8000 gateway nodes. The mount schema is shown in Figure 22. Input data is also initially placed to both clusters' local shared file systems (e.g. NFS and Lustre). By this approach, all nodes can fetch input data from their local shared file systems and only write output data to Lustre file system in HA8000 system.

The overall execution elapsed approximately ten hours. HA8000 was allocated only for first six hours. After the allocated six hours, the second cluster joined the computation and remaining tasks were dispatched to it, reaching the parallelism around 400.

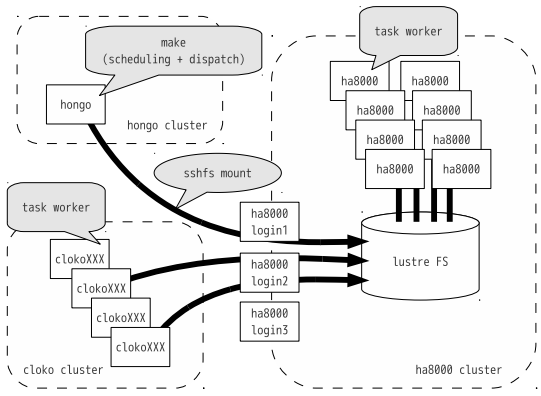

Fig. 22. File sharing scheme used in workflow execution across InTrigger and HA8000 clusters

V. CONCLUSION AND FUTURE WORK

We present an easy and instantaneous processing framework that we routinely use for running various data-intensive workflows on distributed platforms. It includes three

components with low learning cost and deployment cost: GXP shell, GMount distributed file system, and GXP Make workflow engine. This framework tackles the practical problems when using large-scale computing resources in wide-area environments for data-intensive workflows. our experiments demonstrate that the proposed framework has practically useful performance and good usability for general users.

In the future, we will continue developing and improving the performance of GMount and GXP Make. We also plan to improve the usability of this framework, such as by fully implementing GMount in Python to avoid compilation of SSHFS-MUX.

GXP is available at http://gxp.sourceforge.net/. SSHFS-MUX and GMount is available at http://sshfsmux.googlecode.com/.

ACKNOWLEDGMENT

We would like to thank our colleagues for their efforts and cooperation in the practice of developing and running workflow. They are Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Takeshi Shibata, Sungjun Choi, Jun'ichi Tsujii. The authors also would like to thank the referees for their efforts and comments to help us make this paper better. This research is supported in part by the MEXT Grant-in-Aid for Scientific Research on Priority Areas project "New IT Infrastructure for the Information-explosion Era" and Grant-in-Aid for Specially Promoted Research.

REFERENCES

[1] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proc. The 1st Workshop on Many-Task Computing on Grids and Supercomputers*, Austin, Texas, USA, Nov. 2008.

[2] K. Taura, "GXP: An interactive shell for the grid environment," in *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Charlotte, NC, USA, Apr. 2004, pp. 59–67.

[3] GXP distributed/parallel shell. [Online]. Available: http://gxp.sourceforge.net/

[4] N. Dun, K. Taura, and A. Yonezawa, "GMount: An ad hoc and locality-aware distributed file system by using ssh and fuse," in *Proc. The 9th IEEE International Symposium on Cluster Computing and the Grid*, Shanghai, China, May 2009.

[5] SSHFS-MUX filesystem. [Online]. Available: http://sshfsmux.googlecode.com/

[6] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, S. Choi, and J. Tsujii, "Design and implementation of GXP Make – a workflow system based on make," in *Proc. IEEE e-Science 2010 Conference*, Brisbane, Australia, Dec. 2010.

[7] GNU make. [Online]. Available: http://www.gnu.org/software/make/

[8] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain, "Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions," *Journal of Cluster Computing*, vol. 13, no. 3, pp. 243–256, 2010.

[9] NFS version 4. [Online]. Available: http://www.nfsv4.org/

[10] Linux NFS. [Online]. Available: http://nfs.sourceforge.net/

[11] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proc. the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[12] Parallel Virtual File System. [Online]. Available: http://www.pvfs.org/

[13] Lustre file system. [Online]. Available: http://www.lustre.org/

[14] IBM general parallel file system. [Online]. Available: http://www-03.ibm.com/systems/software/gpfs/

[15] J. Cope, M. Oberg, H. M. Tufo, and M. Woitaszek, "Shared parallel file systems in heterogeneous linux multicluster environments," in *Proc. The 6th LCI International Conference on Linux Clusters*, Apr. 2005.

[16] S. C. Simms, G. G. Pike, and D. Balog, "Wide area filesystem performance using Lustre on the TeraGrid," in *Proc. The TeraGrid 2007 Conference*, Jun. 2007.

[17] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "Legionfs: A secure and scalable file system supporting cross-domain high-performance applications," in *Proc. ACM/IEEE Supercomputing 2001 Conference*, Nov. 2001.

[18] P. Andrews, P. Kovatch, and C. Jordan, "Massive high-performance global file system for grid computing," in *Proc. ACM/IEEE Supercomputing 2005 Conference*, Nov. 2005.

[19] Hadoop distributed file system. [Online]. Available: http://hadoop.apache.org/common/hdfs/

[20] O. Tatebe, S. Sekiguchi, Y. Morita, N. Soda, and S. Matsuoka, "Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing," in *Proc. the 2004 Computing in High Energy and Nuclear Physics*, Interlaken, Switzerland, Sep. 2004.

[21] Gfarm file system. [Online]. Available: http://datafarm.apgrid.org/

[22] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 1–7.

[23] Ceph Distributed File System. [Online]. Available: http://ceph.newdream.net/

[24] SSH filesystem. [Online]. Available: http://fuse.sourceforge.net/sshfs.html

[25] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in Condor," in *Workflows for e-Science*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. Springer-Verlag, 2006, ch. 22, pp. 357–375.

[26] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. The 6th Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004.

[27] Hadoop project. [Online]. Available: http://hadoop.apache.org/

[28] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. V. Laszewski, I. Raicu, T. Stef-praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proc. 2007 IEEE Congress on Services*, 2007, pp. 199–266.

[29] Swift workflow system. [Online]. Available: http://www.ci.uchicago.edu/swift/

[30] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proc. IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2007.

[31] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[32] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. the 2nd ACM SIGOPS European Conference on Computer System*, 2007.

[34] Cooperative Computing Tools. [Online]. Available: http://www.cse.nd.edu/~ccl/software/

[35] D. Thain, C. Moretti, and J. Hemmes, "Chirp: A practical global filesystem for cluster and grid computing," *Journal of Grid Computing*, vol. 7, no. 1, pp. 51–77, 2009.

[36] G. Staples, "Torque resource manager," in *Proc. the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[37] Y. Kamoshida and K. Taura, "Scalable data gathering for real-time monitoring systems on distributed computing," in *Proc. The 8th IEEE International Symposium on Cluster Computing and the Grid*, Lyon, France, May 2008.

[38] VGXP: Visual grid explorer. [Online]. Available: http://www.logos.ic.i.u-tokyo.ac.jp/~kamo/vgxp/

[39] FUSE: Filesystem in userspace. [Online]. Available: http://fuse.sourceforge.net/

[40] OpenSSH. [Online]. Available: http://www.openssh.org/
[41] C. Rapier and B. Bennett, "High speed bulk data transfer using the SSH protocol," in *Proc. The 15th ACM Mardi Gras Conference*, Baton Rouge, LA, USA, Jan. 2008, pp. 1–7.
[42] HPN-SSH. [Online]. Available: http://www.psc.edu/\\networking/projects/hpn-ssh/
[43] InTrigger multi-cluster platform. [Online]. Available: http://www.intrigger.jp/
[44] HA8000 cluster system. [Online]. Available: http://www.cc.u-tokyo.ac.jp/service/ha8000/
[45] ParaMark: Benchmark for parallel/distributed systems. [Online]. Available: http://paramark.googlecode.com/
[46] PubMed database. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed
[47] FUSE performance discussion. [Online]. Available: http://old.nabble.com/fuse-performance-td18271595.html
[48] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proc. the 2010 ACM Symposium on Applied Computing*, Sierre, Switzerland, Mar. 2010, pp. 206–213.
[49] Iperf network testing tool. [Online]. Available: http://sourceforge.net/projects/iperf/