Computer Sciences Department

Group File Operations for Scalable Tools and Middleware

Michael J. Brim
Barton P. Miller

Technical Report #1638

June 2008

UNIVERSITY OF
WISCONSIN
MADISON

Group File Operations for Scalable Tools and Middleware

Michael J. Brim and Barton P. Miller

Computer Sciences Department
University of Wisconsin, Madison, Wisconsin, U.S.A.
{mjbrim,bart}@cs.wisc.edu

Abstract. Group file operations are a new, intuitive idiom for tools and middleware - including parallel debuggers and runtimes, performance measurement and steering, and distributed resource management - that require scalable operations over large groups of distributed files. The group file operation idiom provides new semantics for using file groups as operands in standard file operations, thus eliminating costly iteration. A file-based idiom promotes conciseness and portability, and eases adoption. With explicit semantics for aggregation of group data results, the idiom addresses a key scalability barrier. We have designed TBON-FS, a new distributed file system that provides scalable group file operations by leveraging tree-based overlay networks (TBONs) for scalable distribution of group file operation requests and aggregation of group status and data results. Using a prototype TBON-FS system, we have integrated group file operations into several tools. Our experience verifies the group file operation idiom is intuitive and easily adopted, and improves performance at scale.

Key words: distributed parallel filesystem group scalability

1 Introduction

The size of distributed systems continues to expand at a rapid pace to meet the computational demands of the world. High-performance computing (HPC) systems currently exist having processor counts in the hundreds of thousands [22], and several imminent systems will have counts in the millions. Similarly, large companies have tens of thousands of workstations and servers located across local- and wide-area networks. The emergence of cloud computing to harness the power of Internet hosts has also resulted in distributed host groups on the order of thousands and millions, and groups consisting of billions are not unrealistic. Developers of tools and middleware for these large-scale distributed systems face the daunting task of enhancing or redesigning their software to operate at scale. These tasks are often hindered by system designs that narrowly focus on running applications while ignoring the requirements for tools and middleware.

Several classes of tools and middleware share a requirement for performing operations on groups of distributed files. Distributed system management tools update or view software or configuration files across large groups, and middleware such as distributed monitoring uses process and host information found in files on each monitored system. Distributed computing middleware may require distributing applications or data as files to groups of hosts, and collecting groups of result files. Many HPC environments use a file abstraction for control and inspection of processes (e.g., the `/proc` file system on Plan 9, various UNIX implementations, and Linux), where control and

inspection involves operating on files associated with a single process. Application runtime environments [3][6] control distributed process groups, resource [17][19][32] and performance monitors [18][25] perform group process and host inspection, and distributed debuggers [9] and computational steering environments [20] require both group control and inspection. Unfortunately, little regard has been paid at the overall system level to support the group operation requirements of tools and middleware. As a result, each tool is forced to support the required group operations, leading to replication of effort and limiting the generality of these techniques and adoption by others.

Our work addresses the need for a common, scalable infrastructure for operating on large groups of distributed files using a new idiom, group file operations, that provides a simple, intuitive interface. The cornerstone of the idiom is a new `gopen` operation that creates a group file handle for use with existing file operations (e.g., `read` and `write`). The key benefit of the group file operation idiom is eliminating explicit iteration when applying the same operation to a group of files. A file-based idiom also promotes conciseness and portability for group operations. File operations are well-understood and intuitive, and support in programming languages and operating systems is ubiquitous. Also, file operations are data format agnostic and work on files containing binary data, text, or both.

Despite the benefits, introducing group semantics for existing file operations while maintaining intuitive behavior presents several unique challenges. In particular, we address the interface and scalability issues associated with group status and data operands. Our approach is to define intuitive group semantics for existing file operation interfaces and only make extensions when necessary.

Unfortunately, the group file operation idiom alone does not provide scalability when operating on large groups of distributed files. The mechanisms underlying group file operations must be scalable. To this end, we have designed TBON-FS, a new distributed file system that provides scalable group file operations by leveraging tree-based overlay networks (TBONs) for scalable distribution of group file operation requests and aggregation of group status and data results.

TBON-FS provides scalable group file access to a large set of independent file servers from a single client. The single-client multiple-server model is distinctly different from existing parallel and distributed file systems. Parallel file systems [7][29][30] enable large sets of cooperating clients to access large shared datasets that span few to many servers. Distributed file systems [13][28][31] provide many independent clients access to shared files exported from a small set of servers. Current distributed and parallel file systems focus on scaling the number of simultaneous clients, while TBON-FS instead focuses on scaling the number of independent servers that can be accessed concurrently from a single client. Group file operations mix properties of both distributed and parallel file systems. Similar to distributed file systems, a client accesses files located on independent servers. Akin to parallel file systems, a single operation may require access to multiple servers. Due to the differences in client-server model, no existing distributed or parallel file system makes sense as a starting point for TBON-FS.

Using our newly defined group semantics for existing file operations, file system interface extensions, and a TBON-FS prototype system, we demonstrate our techniques

by integrating group file operations into several tools: parallel versions of common UNIX utilities including `grep`, `tail`, and `top`, a simple parallel debugger similar to *mpigdb* [6], and the Ganglia Distributed Monitoring System [17]. These demonstrations have validated our assumptions regarding the usefulness and expressive power of the group file operation idiom for tools and middleware. Further, our evaluations show the potential for increasing the scalability of group file operations.

The rest of the paper is organized as follows. Section 2 introduces the abstractions and semantics of the group file operation idiom, and our solutions for handling group operands. Section 3 presents the design and architecture of scalable TBON-FS. In Section 4, we describe the integration of group file operations and TBON-FS into several tools, and evaluate the performance and scalability benefits. Section 5 discusses related work, and the paper concludes with a summary of current and future work in Section 6.

2 Group File Operations

The group file operation idiom provides an intuitive interface for operating on groups of distributed files. A file-based idiom has benefits that include programmer familiarity, conciseness, and portability. This section describes the group abstractions and operational semantics for our new idiom. First, we examine the creation of file groups using directories and the new `gopen` operation that enables group file operations. Second, we discuss the general semantics of group file operations, focusing on the use of aggregation to address the issue of group operands. Finally, we comment on a few open issues with regard to group file operation abstractions and semantics. Section 3 describes how to provide scalable mechanisms in support of our new idiom.

2.1 Group File Abstractions

Directories are a natural and existing file system abstraction for grouping, and existing directory operations provide straightforward definition and management of file groups. To create a group, one simply creates a directory to contain the group. To add or remove members, files are added to or removed from the directory. Groups can consist of newly created files, existing files, or both. Symbolic links can be used to add existing files without moving or copying. When a group is no longer needed, the directory and constituent files may be removed.

Using directories as file groups, we require an abstraction for operating on all files in a group. The file descriptor abstraction is used by many file system operations to name a target file. Thus, a similar abstraction where a descriptor names a group of files is intuitive and compatible with existing interfaces. As file descriptors are created using `open`, we define a new group open operation, `gopen`, that uses the same function signature as `open` and returns a group file descriptor (gfd) for operating on all files within a named directory. Table 1 shows the interface for `gopen`.

Similar to `open`, `gopen` returns a valid gfd only if the user has permission to open all files in the directory using the specified access flags. The gfd can be used in file system operations that have a file descriptor operand (e.g., `read` and `write`), although the semantics of these operations may differ from the POSIX specification. We discuss the semantic differences of group file operations in the following subsection.

Table 1 New Group File Operations: Interface & Description

`int gopen(const char* dirname, int flags)` Opens all files in directory `dirname` using specified access `flags` and returns a group file descriptor.
`int gsize(int gfd)` Returns the number of files in the group specified by `gfd`.
`int gfiles(int gfd, char** files)` Fills user-allocated array `files` with character strings naming all files in the group specified by `gfd`.
`int gindex(int gfd, const char* file)` Returns the index within group results of the named `file` for the group specified by `gfd`.
`int gstatus(int gfd, int* status_array)` Fills user-allocated `status_array` with the individual member status results of the last group file operation on the specified `gfd`. Returns positive number indicating number of individual errors.
`int gloadaggr(const char* library, const char* function)` Loads the named aggregation `function` located in the shared object file `library`. Returns a new unique identifier for the aggregation that can be used with `gbindaggr`.
`int gbindaggr(int gfd, FileOp fop, AggrType typ, int ag)` Binds the loaded aggregation `ag` to the file operation `fop` for the group `gfd`. *AggrType* is an enumeration indicating status or data aggregation. If `gfd` equals -1, the binding is a default for future groups.

Once a gfd has been obtained, there is no requirement that the directory used to define the group cannot be modified or removed. As such, a gfd represents the set of files in the directory at the time of the `gopen`. Tools acting on dynamic file group views can take advantage of this aspect by reusing a directory to create a new group after adding or removing files. In allowing the underlying directory to change, new operations are required for querying a gfd to obtain group information. Table 1 presents three new operation interfaces, `gsize`, `gfiles`, and `gindex`, for retrieving the group size and information about member files.

2.2 Group File Operation Semantics

By reusing the file descriptor abstraction for file groups, group file operations can use common, well-understood file system interfaces. However, existing interfaces are designed for operating on single files, not file groups. Thus, we define new, intuitive group semantics for these operations using the following guiding principles:

1. Maintain POSIX interfaces, making extensions or additions only when necessary.
2. Choose default group semantics for existing interfaces that are intuitive and handle the common case well.
3. Allow users to easily define and use custom data aggregation when the default semantics do not meet their needs.
4. Summarize group results whenever possible to improve performance and scalability, yet provide methods for users to view detailed group results as necessary.

Intuitive behavior is achieved when the actions performed by a group file operation appear equivalent to applying the operation individually to each member file. Current file system call interfaces have one or more parameters and a status (return) code. Input parameters are the simplest to map to group behavior. Intuition suggests that the same input values should be used for operating on each group member. For example, data

provided as input to a group `write` operation should be written to each member file. In contrast, each member operation will produce separate status and possibly data results. To provide intuitive behavior for group results, individual results should be available to users. However, there are two challenges to providing individual results. First, group results must be returned to users using an interface designed for a single result. Second, group results grow linearly in the size of the group, which can lead to performance and scalability problems when processing data and status results for large groups. Our solution to both of these problems is to aggregate group results.

Aggregation combines individual pieces to form a whole. Depending on the situation, a tool may require various levels of data resolution for group results. It may be interested in all individual results or only a single summary result. To support scoping of group results, we explicitly incorporate data aggregation into group file operation semantics. We provide default aggregations that fit existing interfaces and are suitable for common usage, and allow users with alternative needs to specify custom data aggregation of group results. For convenience, a small, general set of pre-defined aggregations is provided that represents common functions for processing group results. The initial set includes six summary aggregations (AVERAGE, EQUAL, MAX, MIN, SUM, and ZERO) and a CONCATENATE aggregation that combines individual results into an array. In Section 3 we describe the characteristics of aggregations and discuss their definition by users.

It is important to note that adding aggregation semantics to file system interfaces would not have a clear advantage over a tool post-processing group results if the aggregation was executed on a single machine. However, when aggregations that reduce data are used in distributed environments, performing the aggregation in a distributed manner often reduces the centralized processing, memory, and network overhead. Aggregations that summarize group results provide the most performance improvement when distributed, while techniques that provide complete information about individual results (e.g., data compression based on equivalence classes) can still be beneficial by eliminating the processing and network transmission of duplicate data.

For our default group file operation semantics, we require aggregations that permit group status and data results to be returned to users via existing interfaces. Current interfaces share a convention for returning status results using a single integer value that indicates success or failure. To fit these interfaces, we have chosen default aggregations that produce a single summary group status result for each existing operation; our choices are shown in Table 2. In using summary aggregations, we lose information about individual status results. To rectify this deficiency, we have defined a new `gstatus` operation, described in Table 1, to allow users to query individual results. We expect that in the common case, users will only use `gstatus` when the summary result indicates unexpected behavior. For example, an anomalous status value for a group `read` would be less than the expected sum (`nbytes * gsize(gfd)`). Observing such a value might prompt a user to query individual results to see which members did not read the requested amount.

A notable form of unexpected behavior comes as the result of one or more of the individual operations returning error codes. If the pre-defined summary aggregations were not aware of the possibility of error values, summary group results could be cor-

Table 2 Default Summary Aggregations for Group Status Results

Group File Operations	Status Aggregation
`close` `ftruncate` `fchmod` `fchown` `fstat` `fsync`	**ZERO** Return zero when all group member operations are successful.
`pread` `pwrite` `read` `write` `readv` `writev`	**SUM** Return the total number of bytes read or written across all group members.
`lseek`	**EQUAL** Return the common offset when all individual file offsets are equal. Otherwise, return the invalid offset value.

rupted. For example, in the SUM aggregation, adding negative error codes to the computed value would produce an invalid group status. Therefore, our default semantics for all group file operations require that a generic error code is returned as the summary status value when any members return an error, indicating that `gstatus` should be used to identify faulty members. Thus, all group status aggregations, both pre-defined and user-defined, must be error-aware.

To handle group data results, we observe that current interfaces use pointer operands to indicate a destination buffer. It is straightforward to specify group semantics that require buffers to be allocated large enough to hold an array of individual results. We can then use a default CONCATENATE aggregation that combines individual results into an array, with individual results accessible using a member file's index as returned by the new `gindex` operation. Note that concatenation is by no means the most scalable of aggregations, but it is intuitive and functional for arbitrary data (e.g., binary data structures and text). Tools that know the format of output data a priori are unlikely to use concatenation. Rather, we expect them to use custom aggregation to improve performance and scalability.

We have defined two new operations in support of specifying custom aggregations, `gloadaggr` and `gbindaggr`, as shown in Table 1. The former is used to load new aggregations for use with group file operations, while the latter binds status and data aggregations to a specific group file operation. Aggregations can be bound to group file operations for a particular group or as a default for future groups.

To illustrate the use of default and custom aggregation of group results, we describe an example using `read` on a group of `/proc/loadavg` files from many distributed Linux hosts. These files contain text indicating the one-, five-, and fifteen-minute system loads. We assume the tool is interested in the average loads. Figure 3a shows pseudo-code for the example using default aggregation, while Figure 3b shows how custom aggregation may be used. With default aggregation, the tool performs the group `read` and then iterates over the result array to scan and compute the averages. With custom aggregation, the tool uses a load average calculation when performing the group `read`, and the computed averages are directly extracted from the result buffer. Using custom aggregation, the iteration over individual results is eliminated, as is the requirement for a memory allocation whose size increases linearly in the group size.

```
int gfd = gopen("loadavg_grp", O_RDONLY);
int gsz = gsize(gfd);

// Collect individual system load averages
int buf_size = gsz * LOADAVG_FILE_SZ;
void* buf = malloc(buf_size);
int rc = read(gfd, buf, LOADAVG_FILE_SZ);
if( rc != buf_size ) handle_error_condition(gfd, rc);
close(gfd);

// Compute overall averages
double avg_one = 0.0, avg_five = 0.0, avg_fifteen = 0.0;
for( i = 0; i < gsz; i++ ) {
    double one, five, fifteen = 0.0;
    char* data = ((char*)buf) + (i * LOADAVG_FILE_SZ);
    sscanf(data, "%f %f %f", one, five, fifteen);
    avg_one += one;
    avg_five += five;
    avg_fifteen += fifteen;
}
avg_one /= gsz;
avg_five /= gsz;
avg_fifteen /= gsz;
```

(a) Default Data Aggregation

```
int gfd = gopen("loadavg_grp", O_RDONLY);

// Load and Bind Custom Aggregation
int ag = gloadaggr("tool_aggr.so", "calc_load_avgs");
int rc = gbindaggr(gfd, OP_READ, DATA_AGGR, ag);

// Collect and aggregate load averages
unsigned buf_size = 3 * sizeof(double);
void* buf = malloc(buf_size);
rc = read(gfd, buf, LOADAVG_FILE_SZ);
close(gfd);

double* avgs = (double*) buf;
double avg_one = avgs[0];
double avg_five = avgs[1];
double avg_fifteen = avgs[2];
```

(b) Custom Data Aggregation

Figure 1 Group `read` Example: Default vs. Custom Data Aggregation

2.3 Open Issues

Although intuitive, using existing operations for defining file groups still requires iteration when adding or removing files. For tools that create groups of files occasionally, this may not be a problem that limits performance. However, tools that create groups often or manage highly dynamic views of the same group are likely to be limited. We are investigating more scalable methods of group definition and management, such as using regular expressions to add or remove files sharing similar paths or names.

In Table 2, we have not exhaustively addressed group semantics for all POSIX file operations that operate on file descriptors. Two particularly interesting sets of operations we are studying with respect to group semantics are communication (e.g., `socket` and `poll`) and asynchronous I/O operations. Similarly, the semantics of passing a group file descriptor to `mmap` are intriguing.

3 TBON-FS: Scalable Group File Operations

The previous section introduced the key abstractions and semantics of our new idiom, group file operations. We now describe TBON-FS, a new distributed file system designed for scalable group file operations on thousands of distributed files. First, we present the TBON-FS architecture, focusing on how it addresses the key scalability barriers for group file operations. Second, we discuss design issues for TBON-FS at the client and server side. Third, we address concerns for introducing aggregation to file system operations. Finally, the prototype TBON-FS system used for the evaluation presented in Section 4 is summarized.

3.1 TBON-FS Architecture

The group file operation idiom is essential for eliminating explicit iteration when performing the same operation on a group of files, yet the underlying mechanisms also must be scalable. The two key scalability barriers for group file operations are distribution of group operation requests and collection of group status and data results. The central component of the TBON-FS architecture for addressing these barriers is a tree-based overlay network (TBON). TBONs provide outstanding scalability for one-to-many multicast communication and many-to-one data aggregation [1][2][15][26][27].

As shown in Figure 2, TBON-FS employs a TBON for communication and aggregation of data sent between a single client and thousands of independent servers. The processes at the TBON vertices assist to distribute and parallelize the communication and computation for group file operations. Using aggregation to process group results, TBON-FS can significantly reduce the computation required at the client. Necessarily, proper load distribution at scale requires a tree topology with sufficient fan-out and depth, and client load reduction is fully dependent upon the aggregations used.

Using a TBON also helps to improve the scalability of information management. By enforcing a policy where neither the client nor servers maintain global state, TBON-FS avoids any storage overhead that might grow linearly in the number of servers or group members. Instead, servers maintain only local knowledge about the mounted files and file groups, and the client simply contains summary information such as the number of mounted servers and the size of each file group. Additionally, each client and server requires only a local handle for communication with the TBON, and are agnostic to its topology.

3.2 Client Design Issues

On the client side, the two main design issues are how to present the mounted files from thousands of servers within the local file system name space and whether TBON-FS functionality should be implemented as a real file system or at the user level.

From the client perspective, TBON-FS provides the abstraction of a global mount, combining many remote server file system hierarchies under a single, local mount point. To avoid naming conflicts resulting from the same directories or files being mounted across many servers, the default layout for TBON-FS places each server's hierarchy into a directory named after the server. Unlike traditional distributed file systems where mounts are performed by an administrative user, TBON-FS is intended for management by regular users. User-initiated mounts are widely supported for adding

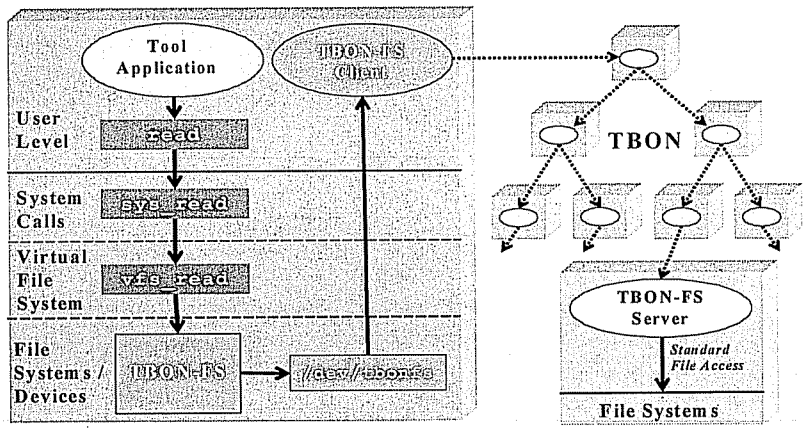

Figure 2 TBON-FS Architecture

An application `read` is a system call to the Virtual File System (VFS) layer, which maps to the TBON-FS file system. Requests are passed to a TBON-FS client process through a character device (`/dev/tbonfs`). The client forwards requests to servers via a TBON. The servers provide a simple file operation proxy service by transforming requests into local file system operations. Operation results are sent back to the client using the TBON, which aggregates results.

the contents of portable storage devices to the current file name space, so a similar approach can be used for TBON-FS. Once mounted, a TBON-FS client can use group file operations on the remote files as previously described in Section 2.

To evaluate the choice between implementing TBON-FS client side functionality as a file system or at the user level, we note the primary advantages and disadvantages of each approach, then conclude with our preference.

In a pure user level approach, functionality can be implemented within a library that is linked with clients. The library must contain functions for mounting and navigating the TBON-FS name space, as well as functions supporting single file and group file operations. The primary advantage of a user level approach is ease of implementation, as operating system development environments are often restricted to a specific programming language and a limited set of runtime support functions for necessary operations such as memory allocation. At the user level, developers can use familiar software languages, runtime support libraries, development environments, and tools to enable rapid prototyping, testing, and debugging. Although straightforward to implement, this approach does not provide the complete look or feel of a real distributed file system, since only clients written to use the library can mount and operate on remote files. Thus, the use of existing software utilities such as `grep` to search files or a command shell to navigate the file system and run local or remote programs on the remote files is prohibited.

Implementation of TBON-FS as a real file system avoids the key problems of the user level library approach by enabling the use of existing software with remote files and allowing multiple clients to share the same mounted file system. Furthermore, using

a framework such as FUSE [33], a majority of file system functionality can be implemented at the user level to ease in development and testing. However, the primary disadvantage is that the file system call interfaces need to be extended to include the new group file operations. From a practical perspective, such interface extensions are straightforward. Unfortunately, gathering support in the real world to extend operating system interfaces is a much harder problem.

Developers of operating system extensions must show that the changes are beneficial to a wide community and will not adversely affect the performance and functionality of existing software. In this paper, we show the benefit to the community through real and useful application of the group file operation idiom. For the latter concern, we have designed an extension to the Linux Virtual File System (VFS) layer that transparently provides support for group file operations with both existing, group-unaware file systems and new, group-aware file systems such as TBON-FS. The extension requires no changes to existing file systems, and has almost negligible impact on single file operations, requiring only a simple check to determine if the file identifier represents a single file or a group. As the Linux VFS layer is similar in design to the equivalent file system abstraction layers of other current UNIX operating systems, the extension should be widely portable.

Our design for extending the Linux VFS to support group file operations is shown in Figure 3. The current VFS design supports only the traditional single file case, as shown in Figure 3a. Application-level file operations correspond to Linux system calls, which in turn call functions within the VFS layer. The VFS layer determines which file system contains the target file, looks up the address of the function within the file system that implements the current file operation, and calls the function.

Our extension, shown in Figure 3b, supports the group file case by first checking if a file identifier represents a group or single file, then mapping to the appropriate underlying group file or single file implementation. If the underlying file system does not provide functions for operations on group files, as is the case for existing, group-unaware file systems, a new generic function within the VFS layer will be called. These generic group file operations simply iterate over each group member, calling the underlying file system's single file operation. From a performance perspective, the generic group file operation functions can provide benefit over user-level group operations by eliminating the per file system call overhead of switching from user to system level.

We believe that extending the operating system to support group file operations and implementing TBON-FS as a real file system is the cleanest and most advantageous option. Further, we assert that direct operating system support for group file operations can benefit even single-host applications. Many utilities such as `grep`, `top`, and recursive application of `chmod` or `chown` target groups of files in one or more directories, and can be modified to use group file operations. Also, distributed and parallel file systems may be able to improve performance when operating on file groups by using explicit group file operation implementations to avoid serial interactions with servers.

3.3 Server Design Issues

TBON-FS servers provide a proxy service for accessing the file systems already present on each server, similar to NFS [28]. Client requests for single file and group file oper-

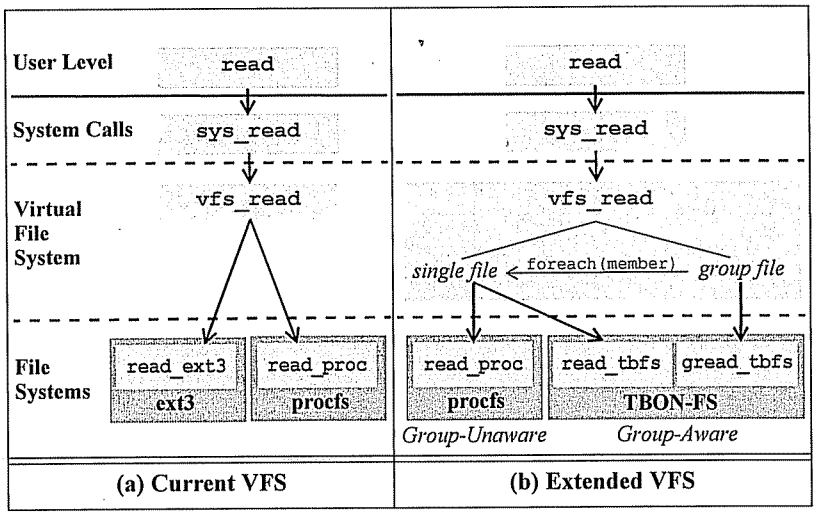

Figure 3 Linux Virtual File System (VFS) Design

(a) User level file operations correspond to Linux system calls, which in turn call functions within the Virtual File System (VFS) layer. The VFS layer maps operations to the appropriate file system. **(b)** The current VFS design supports only the *single file* case. Our extension introduces the *group file* case. The design will allow use of group file operations with both *Group-Unaware* (e.g., existing) and *Group-Aware* file systems.

ations are transformed to one or more local file operations, and status and data responses are passed to the TBON for aggregation. The primary design issues for the TBON-FS server are support for concurrent clients and whether to provide service at the user or operating system level.

When shared among many clients, a server process needs elevated privilege to service requests from clients representing multiple users. The server also is responsible for verifying client credentials to prevent unauthorized access to files. As a result, supporting concurrent clients introduces complexity to develop a secure server and properly isolate failures among clients. With TBON-FS, the expectation is that a specific user mounts the file system and sharing among multiple users is not necessary. By running server processes as a specific user, TBON-FS can rely on existing operating system and file system support for authentication and file access control.

Often, the choice to provide server functionality at the operating system versus the user level is done to improve performance, as client requests can be handled without the overhead of context switches to a user level server process. The same reasoning is applicable to TBON-FS. However, a user level server is easier to develop and test, at the cost of potentially reduced performance. Furthermore, a user level server is easier to deploy in existing distributed environments, as no operating system changes are required. For these reasons, we prefer a user level server.

3.4 Aggregation Considerations

The final design issue for TBON-FS concerns the definition and use of aggregations. The inclusion of aggregation semantics in group file operations is tightly correlated with the notion of using a TBON. However, the use of aggregation can not be specific to TBON-FS if we wish to justify extending operating systems with interfaces for loading and binding of aggregations. Currently, the new `gloadaggr` and `gbindaggr` system calls do not specify the interface to aggregation functions. Instead, they simply allow for finding a function pointer to the compiled code of an aggregation within a shared library. Future file systems supporting group file operations can thus specify their own aggregation interfaces as appropriate. For TBON-FS, aggregation definitions need to be compatible with tree-based computation, and therefore must support composition (i.e., the output of an aggregation may be passed as an input to another instance of that aggregation function).

3.5 Prototype System

In implementing the TBON-FS prototype system, our goal has been rapid development and testing. The prototype provides a framework for evaluating group file operations with respect to applicability and ease-of-use within tools and middleware. In this vein, we chose to implement the prototype as a user level shared library that is linked with client applications and a user level proxy file server. MRNet [26] is our TBON infrastructure for communication and aggregation between the client library and servers.

The client library implements the operations defined in Table 1, as well as group versions of the standard `read`, `write`, and `lseek` file system calls. In addition, the library contains functions for mounting and unmounting TBON-FS. The server implements the proxy service previously described, and contains functions for handling requests generated for the group file operations supported by the client.

The MRNet infrastructure is instantiated when a user mounts TBON-FS. The name of a file listing the remote server directories is passed as the *device* operand to `mount`, and the TBON topology to use is specified in a file whose name is passed as part of the file system options. During infrastructure instantiation, MRNet communication processes are started at internal TBON vertices, servers are launched at the leaves, and the overlay communication is established. For each new file group defined using `gopen`, a set of MRNet streams are created including a group control stream, a group file operation request stream, and streams for aggregating group status and data results. The request stream is used for multicasting group file operation requests to servers. One result stream is created for each status and data aggregation that is set as the default for any group file operation. The control stream supports the new group operations from Table 1. TBON-FS aggregations are written to conform to the MRNet filter specification [34].

4 Evaluation

Our goal for evaluating group file operations and our prototype TBON-FS system is to demonstrate the power of the idiom and the benefits of aggregation at scale. In Section 4.1, we demonstrate the ease in creating new tools for large scale distributed systems by developing parallel versions of three simple, yet powerful Linux command-line tools: `pgrep`, `ptail`, and `ptop`. Section 4.2 shows how we improve upon the usability of

an existing parallel debugger with a simple tool using group file operations and custom aggregation. Finally, Section 4.3 relays our experience in quickly integrating group file operations into the Ganglia Distributed Monitoring System [17].

All experimental results were collected on Thunder, a 1024-node Linux cluster at Lawrence Livermore National Laboratory. The cluster uses a Quadrics QsNet^{II} Elan4 interconnect, and each node has four 1.4GHz Intel Itanium2 processors and 8GB of memory. Due to job resource limits, we could use up to 493 nodes at a time. To overcome this limit, we ran all experiments with four TBON-FS servers on each node. We used five topologies: 1x24x24 (576 servers), 1x28x28 (784 servers), 1x32x32 (1024 servers), 1x8x10x16 (1280 servers), and 1x6x16x16 (1536 servers).

4.1 Parallel UNIX Tools

4.1.1 Parallel `grep`

One of the most useful utilities provided by UNIX and Linux machines is `grep`. System administrators use `grep` for various tasks including searching configuration files, scanning system and application logs for interesting or alarming events, and gathering system or process information. Leveraging `grep` on distributed files provides the ability to spot configuration differences, correlate distributed events, and monitor system resource use. Thus, we have developed `pgrep`. Currently, `pgrep` supports simple textual search strings, as opposed to regular expressions. For scalable results, a line-based equivalence aggregation is used that keeps track of the constituent host files. Line numbers can be prepended, in which case equivalence matches the line number and text.

Table 3 compares `pgrep` on distributed files to standard `grep` on files served by NFS for the same number of files searched. In each experiment, we measured the completion latency in seconds, output line count, and output size. We searched files backed by both disk and memory, and used searches that returned few or many unique matches. *Disk File - Many Matches* searched for the abundant string 'udp' in the `/etc/services` file. *Disk File - Few Matches* searched for 'Kernel' to retrieve the single kernel boot command line from the `/var/log/dmesg` file containing system startup messages. *Memory File - Many Matches* searched `/proc/meminfo` for 'MemFree'. Since the amount of free memory is highly variable at runtime, this search returns many unique lines. *Memory File - Few Matches* searched for 'MemTotal' in `/proc/meminfo`. This search returns a single line that is typically the same across all hosts in a homogenous environment, although our experiments show that there are at least two different totals for Thunder.

In all cases, `grep` exhibits linear scaling in completion time, number of output lines, and output size. Due to `pgrep`'s equivalence aggregation, the number of output lines is simply the number of unique lines across all hosts. The output size for `pgrep` is dominated by a linear factor representing the list of constituent host files prepended to each matched line, since we do not yet compress host file lists into ranges. In all cases, `pgrep`'s output is smaller in size and easier to interpret than `grep`, and provides up to an order of magnitude reduction in cases with significant similarity across files. For `pgrep`, completion time includes the time to initialize the file group using `gopen` and `gbindaggr`, `read` the files, and print the aggregated results. We report total time and the individual component times. The time for the group `read` provides insight into

Table 3 `pgrep` vs. `grep`

		Disk File - Many Matches					*Disk File - Few Matches*				
	Files	**576**	**784**	**1024**	**1280**	**1536**	**576**	**784**	**1024**	**1280**	**1536**
grep	**Time**	.70	.96	1.32	1.62	1.97	.35	.48	.65	.83	1.03
	Lines	131,904	179,536	234,496	293,120	351,744	576	784	1,024	1,280	1,536
	KBytes	7,948	10,826	14,153	17,754	21,354	90	122	160	200	240
pgrep	**Time**	.59	.76	.94	1.30	1.55	.08	.09	.11	.13	.17
	(read)	.44	.56	.69	1.00	1.20	.02	.03	.04	.04	.06
	(init)	.04	.05	.06	.07	.08	.05	.05	.06	.07	.07
	(print)	.10	.13	.18	.22	.27	.00	.00	.01	.01	.02
	Lines	229	229	229	229	229	1	2	2	1	1
	KBytes	763	1040	1368	1768	2168	3	5	6	8	10

		Memory File - Many Matches					*Memory File - Few Matches*				
	Files	**576**	**784**	**1024**	**1280**	**1536**	**576**	**784**	**1024**	**1280**	**1536**
grep	**Time**	.34	.46	.62	.79	1.08	.34	.48	.62	.80	.99
	Lines	576	784	1,024	1,280	1,536	576	784	1,024	1,280	1,536
	KBytes	17	27	30	39	47	17	24	30	39	47
pgrep	**Time**	.07	.10	.14	.18	.18	.07	.10	.11	.16	.16
	(read)	.02	.04	.06	.08	.07	.02	.04	.03	.04	.05
	(init)	.04	.05	.06	.07	.08	.04	.05	.06	.07	.08
	(print)	.01	.01	.01	.01	.01	.00	.00	.01	.01	.01
	Lines	352	507	663	723	856	2	1	2	2	2
	KBytes	16	23	30	34	41	3	4	6	8	8

the scalability of our equivalence aggregation. We observe sub-linear completion time for all experiments except the two largest scales for *Disk File - Many Matches*, where the aggregation of host file lists dominates. In total completion time, we see a measurable increase when the depth of our TBON increases from two to three between the 1024 and 1280 scales.

4.1.2 Parallel `tail`

The `tail` utility with the "-f" option is often used to follow system log activity in real-time. Packages have been developed that enhance the functionality of `tail` to include monitoring multiple files [21] and multiple hosts [16], and to highlight interesting lines of output [12][21]. Combining all three of these enhancements, we have developed `ptail` for following large groups of distributed files. To improve correlation of events across hosts and reduce output, we extended the line equivalence aggregation used for `pgrep` with an option to strip host-specific information (e.g., hostname and process ids) from lines using the standard `syslog` message format. Correlation of events across distributed hosts can be beneficial for applications such as identifying misconfigured network services (e.g., multiple clients of the service notice a problem and generate an error) and security (e.g., distributed intrusion or denial-of-service attacks). To evaluate `ptail`, we wrote a simple synthetic log file generator that allows us to control the rate of log entries and the percentage of entries that are equivalent across hosts.

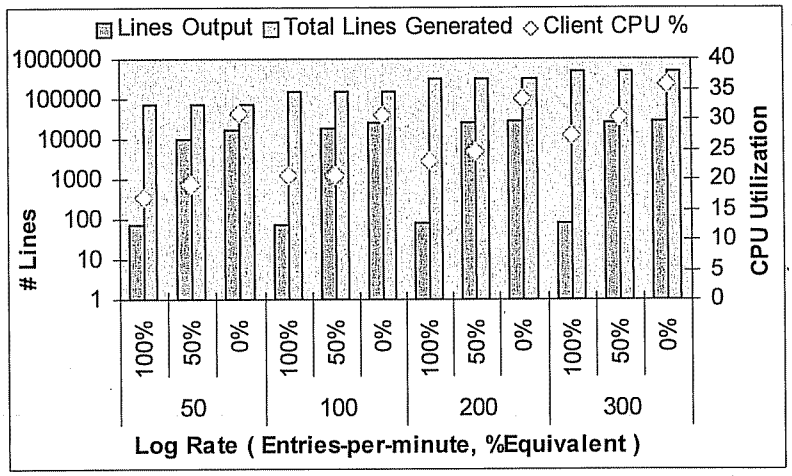

Figure 4 `ptail` Scalability

Figure 4 shows the client CPU utilization of `ptail` and compares the number of output lines to the aggregate number of lines generated during a 60-second run for a file group of size 1536. We tested high log rates from 50 up to 300 per-minute per-host, and equivalence percentages of 0%, 50%, and 100%. The results show that `ptail` is able to significantly reduce the number of output lines, up to three orders of magnitude for 100% equivalence. Since equivalence for log entries implies correlated events, this reduction should improve greatly the ability of users or tools to analyze the logs. In the case of no equivalence across hosts, the number of output lines is still reduced from the number generated due to the fact that several identical messages are generated per second on a single host. Across all experiments, the client CPU utilization remains fairly close for 100% and 50% equivalence, with a noted increase for the 0% worst-case scenario due to the extra processing needed to output a large number of log entries.

4.1.3 Parallel `top`

For many administration tasks, it is useful to know the processes that are using the most resources. The `top` utility is a simple yet powerful method for displaying real-time resource utilization by processes on a single host. Unfortunately, we know of no existing tool that provides the same functionality for a large set of distributed hosts and processes. Hence, we created `ptop`. As with standard `top` for Linux, `ptop` gathers information from files in the `proc` file system. Aggregation is used to summarize across hosts and support the sorting and filtering capabilities of `top`. To provide greater insight into the use of distributed resources, we also provide two new grouping facilities that enable display of summary information for groups of processes with the same command name, both for a specific user and across all users. When grouping processes, the user has the option of viewing total, average, or maximum resource utilization from any group. With `ptop`, users can answer many interesting questions: what parallel applica-

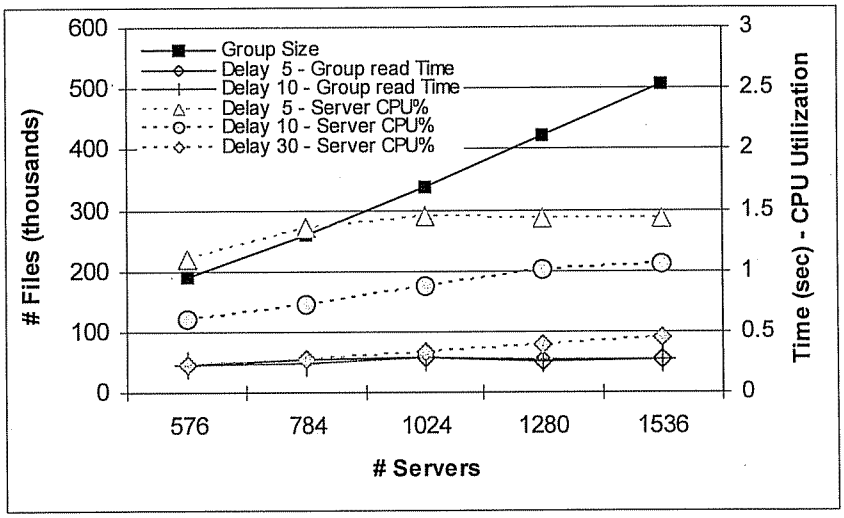

Figure 5 `ptop` Scalability: Server CPU Utilization and Group `read` Time

tion is using the most physical pages of memory, how many `bash` shells are running on all system nodes, which one of my processes is using the most CPU, and how many users are playing solitaire or surfing the web?

To evaluate `ptop`, we measured average latency to collect and aggregate process information and average CPU utilization of the TBON-FS servers. We ran `ptop` for 60 seconds with and without command grouping, using delay intervals of 5, 10, and 30 seconds (the standard `top` default interval is 5 seconds) and specifying that the top 35 or 100 processes should be displayed. We observed no measurable difference between the results for displaying 35 or 100 processes, so we present those for 100. Similarly, results with and without command grouping were indistinguishable, so we show only the grouping case. As shown in Figure 5, `ptop` is able to aggregate resource utilization for file groups consisting of hundreds of thousands of distributed processes (several hundred processes per host) using the same default delay interval as `top`. Further, the time to perform the collection and aggregation scales logarithmically compared to the file group size. We also note that the TBON-FS server CPU utilization was under 0.5% using the 30 second interval for all experiments, which suggests that `ptop` could be run continuously for low-impact, real-time monitoring.

4.2 Parallel GDB Debugger

When MPI applications are run at larger scales, new problems often develop. A parallel debugger may be critical in diagnosing and correcting bugs or logic errors. *mpigdb* [6] is a facility provided with MPICH for running a parallel debug session consisting of multiple independent GDB debuggers, each controlling one MPI process rank (logical process id). Users issue normal GDB commands that are sent to each GDB for execution, and the resulting output is annotated by rank, collected, and grouped by line-equiv-

alence for display. mpigdb uses the MPD (MPICH multipurpose daemon) [6] ring infrastructure for control and user command distribution, and a binary tree overlay among the MPD processes to collect GDB/application output.

The line-based equivalence comparison used by mpigdb often results in unintelligible results for large groups, due to interleaved, multiple-line responses (e.g. stack traces and code listings) and variances caused by host-specific information (e.g. program and variable addresses). Additionally, although a tree structure is used for collection of output, there is no extensible mechanism for aggregation as is available in TBON-FS. As a result, the entire set of output must be collected to the root process before the equivalence comparison is performed. Unfortunately for mpigdb, this equivalence computation has O(NG) behavior, where N is the number of processes and G is the number of equivalence groups. The presence of host-specific information often leads to G being equal to N, and thus exponential time for equivalence grouping.

To overcome these deficiencies, we investigated the benefits of using a more powerful aggregation amenable to use with TBONs. If mpigdb could be extended to provide aggregation of output at each tree process, or alternatively adopt an infrastructure such as TBON-FS, use of this aggregation should result in both performance and usability benefits. We developed a new aggregation providing response-based equivalence, where multiple-line responses are treated as a single comparison entity. The aggregation also filters out host-specific addresses from the output.

To test our aggregation on real GDB output, we implemented a new facility within TBON-FS that launches a user-specified program on all servers and captures the standard input, output, and error streams as a group file descriptor. Writing to the gfd provides input to the individual programs, and the resulting output can be read from the gfd. Using a Linux cluster located within our department, we conducted a qualitative study of the interactivity and intelligibility of command output for our new tool for running parallel GDBs versus mpigdb while debugging a 100 process MPI application. Both provided interactive behavior at this small scale. However, for multiple line responses such as stack traces, the output from mpigdb became unintelligible past 32 processes, and was difficult to understand at much smaller numbers due to the interleaving of stack frames. Since our tool removed program addresses and treated entire stack traces as a comparison entity, we were able to easily identify equivalent group behavior.

4.3 Ganglia Distributed Monitoring System

Ganglia supports host resource monitoring for local-area clusters and wide-area grids. It uses a TBON architecture consisting of *gmetad* cluster/grid aggregator processes, where each leaf gmetad records summary and host information for a cluster, and gmetads at higher-level tree nodes record grid summaries. Within a cluster, a *gmond* monitor running on each host collects local resource utilization and regularly multicasts updates to fellow gmonds in the cluster. The gmetad for a specific cluster queries a representative of the set of gmonds to collect the latest resource use for all cluster hosts. On Linux hosts, the gmonds read `/proc` files to gather resource utilization.

With no previous knowledge of the Ganglia software internals, we integrated group file operations into version 3.0.4 in a span of a few weeks. We unified the Ganglia architecture to be completely tree-based by removing the requirement for multicast

among cluster hosts. As the size of a cluster grows, the use of multicast causes each host in the multicast group to incur a linear increase in network load and memory storage of group state [17]. The updated version, Ganglia-tbonfs, replaces the gmonds with TBON-FS servers and uses custom aggregations that store metric data to local round-robin databases at internal TBON processes, mimicing the behavior of gmetads. To support the web-based interface and its recursive grid/cluster/host views, we developed aggregations that enabled retrieval of summary or host state as XML documents and graphs created from the round robin databases.

5 Related Work

As discussed in Section 1, TBON-FS uses ideas from both distributed and parallel file systems, but has a radically different client-server model that focuses on scaling the number of independent servers that can be accessed concurrently from a single client, rather than increasing the number of concurrent clients or files that can be served.

Gropp and Lusk [11] used parallel versions of command-line utilities for scalable management of massively parallel processors where each node runs a full UNIX environment. The C3 tools [4] are a recent implementation of parallel commands with the same goal. Our parallel UNIX tools share the same motivation for scalable distributed system administration. Unfortunately, the previous tools do not address one of the key scalability issues for group operations, the aggregation of group results to aid in analysis or presentation. Instead, both annotate output with the originating host and expect users to post-process results when aggregation is desired. Thus, these tools could benefit from the integration of a TBON infrastructure to eliminate redundancy in output and summarize group behavior, as we have done with our parallel tools built on top of TBON-FS.

Due to its simplicity and power, many efforts have extended `grep` for parallel operation on distributed data. Dean and Ghemawat [8] exploited the highly data parallel nature of search in a version of `grep` based on MapReduce to scan a terabyte of distributed data using 1700 machines in parallel. It is unclear if any attempt was made to aggregate matching lines of output as part of the reduce phase. The Trellis-SDP data parallel programming system [10] gives an example of how to program a parallel `grep` of distributed files and collect the results to a master process, but does not address solutions for scalable presentation of results when a large number of matches are found. Biogrep [14] matches sets of genetic patterns against large genetic sequence databases, and uses threads to parallelize the work on multiprocessor systems. If the sequence databases were distributed, a version of Biogrep based upon group file operations and TBON-FS may help to further reduce the latency of genetic pattern matching.

Group file operations and TBON-FS are related to the Google MapReduce system [8], as both the map and reduce operations are forms of distributed aggregation of file data. For MapReduce, it is assumed that input comes from a single large data set that is partitioned into many small fixed-size chunks located across hundreds or thousands of Google File System servers. Simply treating chunks as files permits a MapReduce operation to be cast as a group file operation where both map and reduce are implemented as a single aggregation. The MapReduce system is tightly bound to the Google File System, as group file operations are currently tightly bound to TBON-FS. Group file operations are similar in spirit to the Sawzall [24] and PigLatin [23] programming languages

in the desire to expose data parallelism in a simple interface while hiding the details of the underlying parallel processing system. Unlike these languages, group file operations are based on file system interfaces that are already familiar to developers.

6 Conclusion and Future Work

Group file operations are a new, intuitive idiom for operating on large file groups. The idiom eliminates explicit iteration over group members, thus permitting scalable mechanisms for accessing large distributed file groups. We have designed TBON-FS, a new distributed file system, to support scalable group file operations by employing a TBON for distributed communication and aggregation. Using a prototype TBON-FS system, we integrated group file operations into several new tools and one existing middleware system. Our experimental and qualitative observations demonstrate the applicability, ease of use, and benefit for scalable operations on large groups of distributed files.

The prototype TBON-FS system, in its limited form, was useful for demonstrating the potential of scalable group file operations. Yet, there is still much work to be done. As discussed in Section 2.3, we would like to investigate the use of group files with more file operations. This will likely lead to an even wider range of applications that can benefit from group file operations. We wish to explore our ideas for group file operations and TBON-FS within the context of a real client file system for Linux. Using a real file system will allow us to validate our extended Linux VFS design, measure the impact on existing file systems, and observe performance for single file and group file operations. Soon, we will begin evaluating the addition of group file operations to TotalView, the most popular debugger for parallel applications. We aim to improve the scalability of TotalView for use on the largest of current HPC systems and applications.

7 References

[1] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Applications", *International Parallel and Distributed Processing Symposium*, Long Beach, California, March 2007.

[2] D. Arnold, G. Pack and B. Miller, "Tree-based Overlay Networks for Scalable Applications", *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Rhodes, Greece, April 2006.

[3] R. Brightwell and L. Fisk, "Scalable Parallel Application Launch on Cplant", *ACM/IEEE SC 2001*, Denver, Colorado, November 2001.

[4] M. Brim, R. Flanery, A. Geist, B. Luethke, and S. Scott, "Cluster command & control (c3) tool suite", *Parallel and Distributed Computing Practices* **4**, 4, 2001, pp. 381-399.

[5] D. Brownbridge, L. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!", *Software-Practice and Experience* **12**, 1982, pp. 1147-1162.

[6] R. Butler, W. Gropp, and E. Lusk, "A Scalable Process Management Environment for Parallel Programs", Recent Advances in Parallel Virtual Machine and Message Passing Interface - *LNCS* **1908**, Springer, September 2000, pp. 168-175.

[7] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters", *Fourth Annual Linux Showcase and Conference*, pp. 317-327, October 2000.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *6th Symposium on Operating Systems Design and Implmentation*, December 2004.

[9] "Debugger for Multi-core, Multi-Threaded, Multi-Processor Applications", TotalView Technologies, LLC, http://www.totalviewtech.com/productsTV.htm, 2007.

[10] M. Ding and P. Lu, "Trellis-SDP: A Simple Data-Parallel Programming Interface", *2004 International Conference on Parallel Procesing Workshops*, pp. 498-505, August 2004.

[11] W. Gropp and E. Lusk, "Scalable Unix Tools on Parallel Processors", *Scalable High-Performance Computing Conference*, pp. 56-62, Knoxville, Tennessee, May 1994.

[12] S. Hansen and E. Atkins, "Automated System Monitoring and Notification With Swatch", *7th USENIX Conference on System Administration*, pp.145-152, November 1993.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems* **6**, 1, February 1988.

[14] K. Jensen, G. Stephanopoulos, and I. Rigoutsos, "Biogrep: a multi-threaded pattern matcher for large pattern sets", http://web.mit.edu/bamel/biogrep.pdf.

[15] G. Lee, D. Ahn, D. Arnold, B. de Supinski, B. Miller, and M. Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L", *ParCo 2007 Mini-Symposium on Scalability and Usability of HPC Programming Tools*, Juelich, Germany, September 2007.

[16] "logtail: Watch Multiple Log Files on Multiple Machines", https://www.fourmilab.ch/webtools/logtail/.

[17] M. Massie, B. Chun, and D. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience", *Parallel Computing* **30**, Elsevier B.V., 2004.

[18] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", *IEEE Computer* **28**, 11, November 1995, pp. 37-46.

[19] R. Mooney, K. Schmidt, R. Studham, and J. Nieplocha, "NWPerf: A System Wide Performance Monitoring Tool for Large Linux Clusters", *2004 IEEE International Conference on Cluster Computing*, pp. 379-389, San Diego, CA, September 2004.

[20] A. Morajko, O. Morajko, T. Margalef, and E. Luque, "MATE: Dynamic Performance Tuning Environment", *10th International Euro-Par Conference*, pp. 98-107, August 2004.

[21] "MultiTail", http://www.vanheusden.com/multitail/.

[22] "TOP500 Supercomputing Sites", http://top500.org/lists/2007/11, November 2007.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing", *SIGMOD '08*, Vancouver, British Columbia, 2008.

[24] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall", *Scientific Programming*, **13**, 4, October 2005, pp. 277-298.

[25] "Open|SpeedShop", http://www.openspeedshop.org/.

[26] P. Roth, D. Arnold, and B. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", *SC 2003*, Phoenix, Arizona, November 2003.

[27] P. Roth and B. Miller, "On-line Automated Performance Diagnosis on Thousands of Processes", *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, New York, March 2006.

[28] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *USENIX Technical Conference*, pp. 119-130, Portland, Oregon, June 1985.

[29] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", *File and Storage Technologies*, pp 231-244, January 2002.

[30] P. Schwan, "Lustre: Building a File System for 1000-node Clusters", *2003 Linux Symposium*, July 2003.

[31] S. Soltis, T. Ruwart, and M. O'Keefe, "The Global File System", *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 319-342, September 1996.

[32] M. Sottile and R. Minnich, "Supermon: a high-speed cluster monitoring system", *IEEE Conference on Cluster Computing*, pp. 39, Chicago, Illinois, September 2002.

[33] M. Szeredi, "Filesystem in Userspace (FUSE)", http://fuse.sourceforge.net.

[34] "A User's Guide to MRNet v1.1", http://www.paradyn.org/mrnet/release-1.1/UG.html.