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 on large groups of distributed files. The idiom provides new semantics for using file groups in standard file operations to eliminate costly iteration. A file-based idiom promotes conciseness and portability, and eases adoption. With explicit semantics for aggregation of group 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 communication and data aggregation. We integrated group file operations into several tools: parallel versions of common utilities including `cp`, `grep`, `rsync`, `tail`, and `top`, and the Ganglia Distributed Monitoring System. Our experience verifies the group file operation idiom is intuitive, easily adopted, and enables a wide variety of tools to run efficiently at scale.

Keywords*:* group file; tools; distributed; scalable, aggregation

1. INTRODUCTION

Distributed systems for high-performance computing (HPC), corporate intranets, and cloud computing can contain tens of thousands of hosts. HPC systems on the horizon are expected to contain millions. Developers of tools and middleware for distributed systems face the daunting task of changing their software to operate at ever-increasing scale. Our work introduces a new idiom, *group file operations*, which provides a simple, intuitive interface that tools and middleware can use for scalable operations on large groups of distributed files. We provide solutions targeted at the largest current systems, and scalable techniques for continued future use.

Several classes of tools and middleware operate on groups of distributed files. Distributed system management tools control software and configuration files on many hosts, and distributed monitoring uses process and host information found in files on each monitored system. Distributed computing middleware may distribute applications or data as files to groups of hosts, and collect groups of result files for analysis. On systems using a file abstraction for processes (e.g., the `/proc` file system on Plan 9, UNIX, and Linux), file operations are used for process control and inspection. Application run-time environments control distributed process groups, performance monitors use group process or host inspection, and distributed debuggers and computational steering systems require both group control and inspection.

Little attention has been paid to supporting group operations on distributed files in a scalable manner. Each tool is forced to support the required group operations, leading to redundant effort and limiting the generality of the solutions and adoption by others. In defining the group file operation idiom, we provide a common, reusable abstraction that lends itself to scalable solutions.

The cornerstone of the group file operation idiom is a new `gopen` operation that returns a group file descriptor for use with existing file system operations (e.g., `read` and `write`). The key benefit of the group file abstraction is eliminating explicit iteration that forces serial behavior when applying the same operation to a group of files. Other benefits include program conciseness and familiarity. File operations are well-understood and universally supported in programming languages and operating systems. However, introducing group semantics for file operations presents several challenges. In particular, we address the interface and scalability issues associated with group status and data results. Our approach is to define intuitive group semantics for existing file operations and extend the file system interface when necessary.

The group file operation idiom alone cannot provide scalable operations on large groups of distributed files. The mechanisms underlying group file operations must be scalable. 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. Distributed aggregation is a powerful technique for managing large-scale data processing and analysis.

Using newly defined group file operation semantics, file system interface extensions, and a TBON-FS prototype system, we integrated group file operations into several tools: parallel versions of common utilities including `cp`, `grep`, `rsync`, `tail`, and `top`, and the

Ganglia Distributed Monitoring System [15]. Our experience has validated our assumptions on the expressive power of the group file operation idiom, and our evaluation shows the scalability benefits for tools and middleware.

2. GROUP FILE OPERATIONS

The group file operation idiom provides an intuitive interface for operating on groups of distributed files in a manner that eliminates explicit iteration. 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 semantics of group file operations, focusing on the use of aggregation to handle group status and data results. Finally, we address the impact of errors on group file operations. Section 3 describes how to provide scalable mechanisms in support of our new idiom.

2.1. Group file abstraction

The primary abstraction used in our new group file operation idiom is the *group file*, which we define as a set of open files that are operated upon as a single entity. A group file is created using our new `gopen` operation.

Group membership is specified using a directory, which is a natural file system abstraction for grouping sets of files. Placing files in the same directory often implies a logical association, and thus there is a good chance that files in the same directory will be operated on as a group. If a user wants to create groups of files not already located in the same directory, a new directory can be created containing copies of (or links to) existing files.

The new `gopen` operation has the same function signature as open, as shown in Table 1, but is passed a directory instead of a file as the first parameter. `gopen` can be considered equivalent to calling open on each file corresponding to an entry in the named directory, using the specified access flags and creation mode. Upon successful completion, `gopen` returns a group file descriptor (gfd) that serves as a handle to the group file. A *group file operation* is performed by passing a gfd to a file system operation that has a file descriptor operand (e.g., `read` and `write`). The semantics of group file operations may differ from the POSIX specification. We discuss group file operation semantics in Section 2.2.

Special consideration is given to errors that occur during `gopen`. Two modes, *default* and *best-effort*, differentiate how failures in opening individual directory entries affect `gopen`'s completion status. In *default* mode, `gopen` will succeed only when all the files in the named directory can be opened with the specified flags and creation mode. In *best-effort* mode, which is specified by including a new `O_BESTEFFORT` value in the flags operand, `gopen` will succeed when any of the files are successfully opened. In both modes, a failed completion will return -1 and set errno to `E_GROUP`.

While `gopen` is executing, the named directory's contents cannot be changed. After `gopen` returns, this restriction is removed. As such, a gfd represents the set of files that resided in the directory at the time of the `gopen` and were successfully opened. Defining new groups by adding or removing directory entries often is convenient. For example, after operating on a particular group file, a tool may identify a subset of the group upon which it wants to operate as a new group. Rather than requiring a new directory to be created containing the subset's files, the user can simply remove files from the directory.

By allowing the contents of a directory used to define a group file to change, new operations are required to obtain information about the group file. Fig. 1 presents three new file system operations, `gsize`, `gfiles`, and `gindex`, for retrieving the group size and information about member files.

2.2. Group file operation semantics

The group file descriptor (gfd) abstraction allows group file operations to use existing, well-understood file system operations. As these existing operations are designed for operating on individual files, our challenge is to define semantics for these operations when used with a gfd. We follow four guiding principles in defining the semantics for group file operations:

1. Maintain POSIX file system operation interfaces, making extensions or additions only when necessary.
2. Choose default group semantics for existing operations that are intuitive and handle the common case well.
3. Allow users to easily specify custom behavior when the default group 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.

Conceptually, group file operations can be viewed as equivalent to applying the file operation individually to each group member. The complexity in defining group file operation semantics concerns the treatment of operation parameters and status results. We refer to parameters whose values are used only as inputs to an operation as *input parameters*, and those whose values are modified as *output parameters*. Input parameters are the simplest to map to group behavior, as intuition suggests that the same values should be used when operating on each group member. For example, a `write` operation has only input parameters: the gfd, a byte count, and a data buffer. A `write` on a gfd will copy the requested bytes from the provided data buffer to each member at its current offset. In contrast, the values of output parameters and status results produced when operating on each group member can differ. The individual values may be of interest to users and must be provided upon request.

There are two obstacles to providing individual results. First, existing file operations have system call in-

TABLE 1. NEW FILE SYSTEM OPERATIONS FOR MANAGEMENT OF GROUP FILES: INTERFACE AND DESCRIPTION

`int gopen(const char* dirname, int flags, int mode)` Opens all files in directory `dirname` using specified access `flags` and creation `mode`. 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)` Copies directory entry names corresponding to files in the group specified by `gfd` into the user-allocated array of character buffers `files`. The `files` array should be allocated to contain `gsize(gfd)` character buffers, each able to hold a maximum length directory entry name.
`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, const char* params_fmt, ...)` Binds aggregation `ag` to the file operation `fop` for the group `gfd`. If `gfd` equals -1, the binding is a default for future groups. *AggrType* is an enumeration indicating status or data aggregation. *params_fmt* is a varargs format string that describes a variable number of parameters.

terfaces designed to return individual status results and output parameters. Second, the collections of individual status and output parameter values have sizes that grow linearly in the size of the group, which can lead to performance and scalability problems for large groups. Our solution is to aggregate each collection into a group result. We refer to the computed aggregates as *group status results* and *group data results*.

By choosing appropriate aggregations, group status and group data results can fit existing interfaces. Group status results should take the form of a single value that can be returned by the group file operation. When users need more detailed information, we provide the new `gstatus` operation that retrieves all individual results, described in Section 2.3. For group data results, we observe that existing interfaces use pointers for parameters that may be modified. A straightforward method to deliver group data results is to require users to pass a pointer to a buffer that can hold an array of individual results.

For each type of group file operation, we must identify default aggregations for group status results and group data results (if applicable) that can be returned to users via existing interfaces and are suitable for common usage. In addition, users need the ability to specify custom aggregations when the default aggregations are insufficient for their needs. Section 2.3 discusses our choices for default aggregations, and introduces new operations that permit the use of custom aggregation.

Fig. 1 lists the steps required for each group file operation. It is important to note that the implementation of these steps can take advantage of parallel techniques, as we show in Section 3. First, the group file descriptor is used to retrieve group file information (line 1). The file operation is called for each of the group members, and individual status values are stored (5-9). The status aggregation function is used to compute the group status result (11-12). If necessary, group data results are computed using the data aggregation function (14-17). Finally, the group status result is returned, and group data results are returned in the output parameters.

2.3. Group status and data aggregation

For convenience, we provide a small set of pre-defined aggregations that includes common functions for processing group status and data 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.

Summary aggregations should be used for group status results to produce a single value that can be returned by the group file operation. In Table 2, we have chosen default status aggregations for each type of group file operation that are reasonable for common use. When a summary of the group file operation's behavior is insufficient, we have defined the new `gstatus` operation, described in Fig. 1, to allow users to query individual results from the last group file operation issued for a gfd. We expect that in the common case, `gstatus` will be used only when the summary value indicates unexpected behavior. For example, an anomalous status for a group read using the *sum* aggregation would be less than the expected value `num-bytes-to-read` × `gsize(gfd)`. The user could then query individual results to see which members did not read the requested amount.

For group file operations that produce group data results, we believe the logical choice for default aggregation is *concatenate*, which combines individual results into an array. Individual results are then accessed in the group data result using a member file's index as returned by the new `gindex` operation. Although array concatenation is by no means the most scalable of aggregations,

```
  int fileop(gfd, in_arg, out_arg)
  {
1    group_file* gf = get_group(gfd);
2    int stat[gf->size];
3
4    // Call fileop for each member
5    for( i=0; i < gf->size; i++ ) {
6       int fd = gf->members[i];
7       o_arg = out_arg + i;
8       stat[i] = fileop(fd, in_arg, o_arg);
9    }
10   // Compute group status result
11   status_aggr_fn = gf->saggr(fileop);
12   int grp_status = status_aggr_fn(stat);
13
14   data_aggr_fn = gf->daggr(fileop);
15   if( data_aggr_fn != NULL )
16      // Compute group data results
17      data_aggr_fn(stat, in_arg, out_arg);
18
19   return grp_status;
  }
```

Figure 1. Group file operation algorithm
General algorithm for applying a file operation to a group file and computing group status and data results.

it is intuitive and functional for arbitrary data (e.g., binary data structures and text). Users that know the format of output data a priori are encouraged to use custom aggregation for improved performance and scalability.

We have defined two new operations in support of specifying custom aggregations, `gloadaggr` and `gbindaggr`, with interfaces as described in Fig. 1. The former is used to load new aggregations for use with group file operations, while the latter binds status or data aggregations to a specific group file operation. Aggregations can be bound to group file operations for a particular group file or as a default for future groups. 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. File systems directly supporting group file operations specify their own aggregation interfaces as appropriate.

To illustrate the use of default and custom aggregation of group results, we give an example using read on a group of `/proc/loadavg` files from many 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. Fig. 2a shows pseudocode for the example using default aggregation, while Fig. 2b shows how custom aggregation may be used.

2.4. Error semantics

The semantics for group file operations are complicated when one or more of the individual member operations return an error status. Our current specification requires a group file operation to return an error status (`E_GROUP`) if any individual errors occur. This status indicates to the user that `gstatus` should be used to identify faulty members. Both pre-defined and user-defined aggregation functions must be error-aware in order to avoid returning corrupted group status or data results. For example, if the *sum* aggregation was not error-aware, adding one or more negative error values to the computed sum would produce an invalid value.

TABLE 2. GROUP STATUS AGGREGATION DEFAULTS

Summary Aggregation & `Group File Operations`
zero: Return zero if all member operations succeed. `close, fchmod, fchown, fstat, fsync, ftruncate, lio_listio, aio_fsync, aio_read, aio_write, aio_suspend`
sum: Return total bytes read/written across all members. `pread, pwrite, read, readv, write, writev`
equal: Return common offset when all individual file offsets are equal. Otherwise, return invalid offset value. `lseek`

3. TBON-FS: A FILE SYSTEM FOR SCALABLE GROUP FILE OPERATIONS

The group file operation idiom permits the use of scalable implementation techniques by removing explicit iteration at the file system interface when operating on file groups. We now describe TBON-FS, a new distributed file system designed for scalable group file operations on thousands of distributed files. First, we describe the global mount abstraction of TBON-FS. The TBON-FS architecture is presented next, with focus given to the design choices that address the scalability barriers for group file operations. Finally, the prototype TBON-FS system used for the evaluation in Section 4 is discussed.

3.1. TBON-FS global mount

TBON-FS combines a collection of remote file system directories under a single mount point on the client. The global mount abstraction provided by TBON-FS is inspired by previous work on single system image (SSI) administration of distributed systems [3,9,12,16,19,20]. The goal of a global mount is to provide a unified file namespace that hides the distributed nature of files, which simplifies tool development. The two most common approaches to provide the global mount abstraction are to use a distributed operating system, or build upon existing distributed (or parallel) file systems.

Since we endeavour to provide a scalable approach to group file operations that can be used on a wide variety of distributed systems, and distributed operating systems are rarely used for large-scale production systems, we believe that there is little benefit in adding group file operations to a specific distributed operating system.

Relying on existing distributed file systems has two problems when considering group operations on files distributed across thousands of servers. First, distributed file systems are designed to provide good performance for multiple clients making requests to a single server, while group file operations invert the relation to a single

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

// Collect load average file data
char* buf = malloc( gsz * LDAVG_FILE_SZ );
read( gfd, buf, LDAVG_FILE_SZ );
close( gfd );

// Scan data and compute overall averages
double avg_one, avg_five, avg_fifteen;
avg_one = avg_five = avg_fifteen = 0.0;
for( i = 0; i < gsz; i++ ) {
   double one, five, fiftn;
   char* data = buf + ( i * LDAVG_FILE_SZ );
   scandata( data, &one, &five, &fiftn );
   avg_one += one;
   avg_five += five;
   avg_fifteen += fiftn;
}
avg_one /= gsz;
avg_five /= gsz;
avg_fifteen /= gsz;
```

(a) Using Default Aggregation

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

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

// Collect and aggregate load averages
double* avgs = malloc( 3 * sizeof(double) );
read( gfd, avgs, LDAVG_FILE_SZ );
close( gfd );

double avg_one = avgs[0];
double avg_five = avgs[1];
double avg_fifteen = avgs[2];
```

(b) Using Custom Aggregation

Figure 2. Default vs. custom data aggregation
(a) With default aggregation, a group `read` generates an array of results that is iterated over to scan and compute averages.
(b) With custom aggregation, `gbindaggr` is used to specify a load average calculation function for use with the `read`, and the averages are directly extracted from the result buffer.

client interacting with many servers. Without significant changes to the client-server model, client support for group file operations in existing distributed file systems still requires iteration over servers. Second, the majority of distributed file systems directly manage the underlying storage. To include files in the global mount, the files must reside in the distributed file system. This precludes access to a large set of useful files in memory-backed file systems (e.g., `/proc`), and forces all disk files on each server that may be of interest to the client tool to reside in the distributed file system, which may introduce unnecessary overhead for server-local file operations.

To avoid the inherent limitations for group file operations in an existing distributed file system, we designed TBON-FS to provide scalable group file operations on arbitrary server files from the beginning.

3.2. TBON-FS scalable architecture

In designing TBON-FS, we adopted techniques that have been shown to be scalable for large distributed systems. In particular, we target methods that permit parallel execution and do not require knowledge of global state to be maintained at each distributed host. The key piece of the TBON-FS architecture that allows for parallel execution is the integration of a tree-based overlay network, which provides both scalable group multicast communication and scalable aggregation of group data.

The group file operation idiom allows for parallel execution when member files are distributed across many file servers. With both `gopen` and group file operations, each file server can operate independently. Thus, the client can multicast the operation request, and servers can fulfill the request in parallel. As servers finish their local operations, the resulting status and data values can be returned to the client. Using distributed aggregation, the group status and data results can be computed in a parallel fashion. By distributing communication and processing load over a tree overlay, client load and completion time for group file operations can be greatly reduced.

TBONs easily scale to any size distributed system by simply increasing fan-out or depth. For example, a TBON-based debugging tool was the first to run at full scale on the BlueGene/L system, which has over 200,000 processors [13]. Using reasonable fan-outs (e.g., 32 or 64) and depths (e.g., 4 or less), a tree topology has the desirable property that the number of non-leaf nodes is a small percentage of the total number of nodes. A balanced tree with fan-out 32 and depth 4 can be used to access one million servers, and has only 3% non-leaf nodes. TBONs also provide desirable properties for recovery and reconfiguration that help tolerate faults, which occur with higher frequency at larger scales [1].

As shown in Fig. 3, the TBON-FS architecture interposes a TBON between a client at the root and many servers at the leaves. The client uses the TBON to multicast group file operation requests to servers. The servers transform requests into standard operations on local member files, and send the status and data results back to the client via the TBON, which executes the aggregation functions (i.e., status/data aggregations bound to the current group file operation) to produce group results.

A tree-based approach to data aggregation requires functions that support hierarchical execution. An aggregation function at a non-leaf tree vertex will be passed a set of input data, where each input datum is the output produced by the function at one child. An example aggregation function that supports hierarchical execution is presented in Fig. 4, which shows psuedocode for the custom load average calculation described in Section 2.3 and used in Fig. 2b. A `file_data` structure is defined to encapsulate raw or aggregated file data and a list of group file indices. For raw data (e.g., data obtained di-

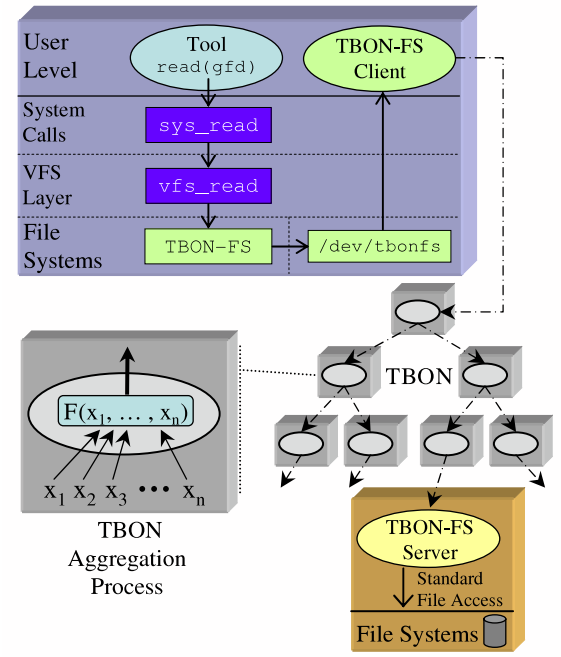

Figure 3. TBON-FS architecture
A tool group `read` is a system call to the Virtual File System, which maps the request to a TBON-FS file system operation. The TBON-FS client process polls a character device for new requests, and multicasts them to servers via the TBON. Servers transform requests into local file operations, and send results back through the TBON, which performs aggregation.

rectly from read), the list will contain a single integer denoting the origin file index. With aggregate data, the list will contain each of the indices that contribute to the aggregated result. The aggregation function accepts an array of `file_data` structures as an input, and stores aggregated results in a `file_data` output parameter.

To avoid shared global state that may require costly synchronization or consensus, TBON-FS adopts a policy where neither the client nor servers maintain complete information for the state of each group file. Servers keep track of the local member files, last operation status results, and bound aggregations for each group file. The client keeps only summary information such as the size of each file group. This policy introduces some extra collection overhead when global information is required, such as when gstatus, gfiles, or gindex is used. Fortunately, TBONs are efficient for data collection.

3.3. Prototype system

The TBON-FS prototype system provides a framework for evaluating group file operations with respect to adoption and ease-of-use within tools and middleware. Our implementation effort focused on rapid development and testing, rather than performance optimization.

The resulting system consists of a shared library that is linked with client tools and a user-level file server. The client library implements the operations defined in

```
struct file_data {
  unsigned data_len;  // length of data
  void* data;         // data (any format)
  unsigned files_len; // number of files
  int* files;         // group file indices
}

int calc_load_avgs( unsigned num_inputs,
                    struct file_data inputs[],
                    struct file_data* output )
{
   // calculate average of inputs
   double avg1 = 0, avg5 = 0, avg15 = 0;
   unsigned total_num_files;
   for( unsigned u=0; u < num_inputs; u++ ) {
      int nfiles = inputs[u].files_len;
      void* idata = inputs[u].data;
      if( nfiles == 1 ) {
         // scan raw file data
         double one, five, fiftn;
        scandata(idata, &one, &five, &fiftn);
         avg1 += one;
         avg5 += five;
         avg15 += fiftn;
      } else {
         // data contains aggregated loads
         double* avgs = (double*) idata;
         avg1 += avgs[0] * nfiles;
         avg5 += avgs[1] * nfiles;
         avg15 += avgs[2] * nfiles;
      }
      total_num_files += nfiles;
   }
   avg1 /= total_num_files;
   avg5 /= total_num_files;
   avg15 /= total_num_files;

   // prepare output data
   allocate_output_data( output );
    store_averages(output, avg1, avg5, avg15);
   fill_output_file_list( inputs, output );
}
```

Figure 4. TBON aggregation function example
Pseudocode for a hierarchical version of the custom system load average calculation used in Fig. 2b.

Fig. 1, group-aware versions of the standard file system calls in Table 2, and functions for mounting TBON-FS. The server implements a simple proxy file service that uses standard file operations on local files, and manages the local state for each group file. MRNet [22] serves as our TBON infrastructure, and TBON-FS aggregations are written conforming to the MRNet filter API.

The MRNet infrastructure is instantiated when a user mounts TBON-FS. The name of a file containing pairs of the form (*server host*, *directory*) is passed as the *device* operand to mount, and a TBON topology file is given in the file system options. During overlay instantiation, communication processes are started at internal TBON vertices, and servers are launched at the leaves. For each new group file, a set of MRNet data streams are created: a group control stream, a group file operation request stream, and streams for aggregating group status and data results. The control stream supports the new group operations from Table 1. 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.

4. EVALUATION

Our goal for evaluating group file operations and the prototype TBON-FS system is to demonstrate the idiom's power and the benefits of aggregation at scale. Section 4.1 describes our use of group file operations to create parallel versions of five Linux command-line tools: cp, grep, rsync, tail, and top. Section 4.2 relates our experience in quickly adding group file operations to the Ganglia Distributed Monitoring System [15].

Experiments were run on two Linux clusters, Thunder and Atlas, located at Lawrence Livermore National Laboratory. Thunder has 1024 hosts connected via Quadrics QsNet^{II} Elan4, and each host has four 1.4GHz Intel Itanium2 processors and 8GB of memory. Atlas contains 1152 hosts, each with eight 2.4GHz AMD Opteron processors and 16GB of memory, and uses a 4X-DDR Infiniband interconnect. Thunder experiments used four TBON-FS servers per host, and Atlas experiments used eight servers per host. In all experiments, the TBON-FS client and each MRNet communication process were run on hosts separate from those with TBON-FS servers. On Thunder, we used five tree topologies: 1×24×24 (576 servers), 1×28×28 (784), 1×32×32 (1024), 1×8×10×16 (1280), and 1×6×16×16 (1536). On Atlas, four topologies were used: 1×4×8×32 (1024), 1×8×8×32 (2048), 1×8×12×32 (3072), and 1×8×16×32 (4096).

4.1. Parallel UNIX tools

4.1.1. Parallel `cp` and `rsync`

File distribution is a common task when managing large distributed systems. In environments such as clusters where hosts are similarly configured, file distribution can be used to update configuration files local to each host. For distributed computing, application, data, and script files may be staged to hosts before execution.

To improve the scalability of file distribution, we developed parallel versions of `cp` and `rsync`. `pcp` uses group `write` operations to multicast a source file to all servers. `psync` uses the `rsync` block checksum comparison algorithm [26] to identify differences between the servers' copies of a file and the client source file, and only multicasts the data that has been changed or added. psync uses group `read` operations with an aggregation that checksums file data blocks. The aggregation identifies sub-groups of servers whose file data is identical. The block checksums for each sub-group are compared to the source file, and updated data is multicast using a group `write`. In homogenous environments with one unique group, the computational load for `psync` on the client is similar to `rsync` with a single server.

We evaluated the performance of `pcp` and `psync` in terms of distribution time for three file sizes on the Atlas cluster. The files were chosen by finding recently modified configuration files on Atlas for which we could retrieve the prior version. For comparison, we measured the time for every server to use the `cp` command to get the file from NFS or a Lustre parallel file system. The `cp` tasks were launched in parallel using the `srun` command of SLURM, the resource management system used on Atlas. To account for `srun` overhead, we measured the time to run `hostname` in parallel at each experimental scale, and subtracted that time from the NFS and Lustre times. Two versions of `pcp` were tested, using synchronous and asynchronous group `write`. Synchronous `pcp` was always faster, so we report those results, although the asynchronous version still outperformed the parallel `cp` from NFS and Lustre. `cp` and `pcp` experiments use `/dev/null` as the destination file to avoid measurement bias from local disk writes on the servers.

Fig. 5 reports copy times for the three files. `pcp` is always fastest, showing logarithmic performance as the number of servers (destinations) increases. Lustre comes in second, but has poor linear scaling. NFS does not exhibit linear scaling, but is an order of magnitude slower than `pcp`. For the two smaller files, `psync` outperforms NFS at all but the largest experimental scale, even though it must read and checksum all destination files. On the largest file, the overhead of processing over 500 1KB blocks becomes too costly. However, as shown in Table 3, `psync` sends much less data by only transmitting new data and information on the matched blocks. Due to its network savings, `psync` is attractive for updating files on systems where the network is highly-utilized, yet computation is comparatively cheap.

4.1.2. Parallel `grep`

`grep` can be used to search configuration files, scan system or application logs for interesting events, and gather host or process information. Leveraging `grep` on distributed files helps identify configuration differences, correlate distributed events, and monitor resource use. Thus, we have developed `pgrep`. For simplicity, it currently supports textual searches rather than regular expressions. `pgrep` aggregates matching lines into groups. Each group is a distinct line found in one or more files, and the aggregation prepends the constituent files. Files are represented as strided ranges of group indices, where each range has a start index, stride, and count. When a `pgrep` user requests line numbers in the output, the equivalence groups match the line number and text.

Table 6 compares `pgrep` on distributed files to standard `grep` on files served by NFS for the same number of files searched. We measured the completion latency in seconds and output size in kilobytes. We searched files backed by both disk and memory, and used searches that returned few or many unique matches. *services-udp* searched for the abundant string "udp" in the `/etc/services` file. *meminfo-Free* searched /proc/meminfo for "MemFree". As the amount of free memory is variable at runtime, this search returns many lines. *meminfo-Total* searched for "MemTotal" in

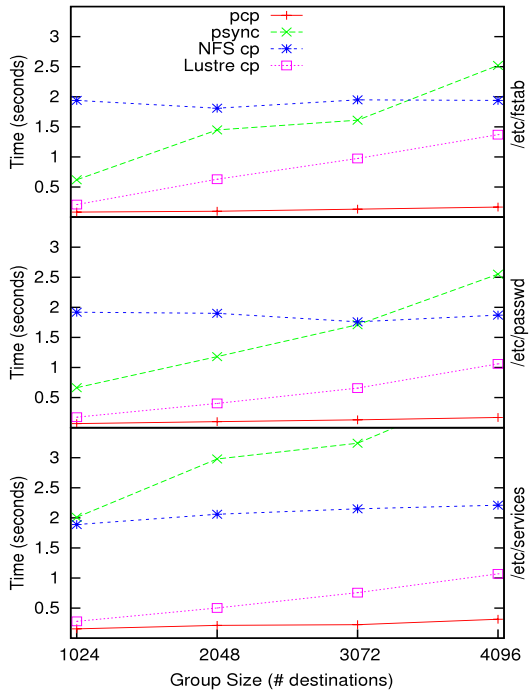

Figure 5. Parallel copy scalability
File distribution time for `pcp`, `psync`, and parallel `cp`.

`/proc/meminfo`. This search returns a single line representing the homogenous group of Atlas hosts.

`grep` exhibits linear scaling in completion time and output size. Due to `pgrep`'s equivalence aggregation, the number of output lines is simply the number of unique lines across all hosts. `pgrep`'s output is smaller in size and easier to interpret than `grep`, and is an order of magnitude smaller in cases with significant similarity. For `pgrep`, completion time includes the time to initialize the file group using `gopen` and `gbindaggr`, read the files using the equivalence aggregation, and print the aggregated results. We report total and individual component times. The group `read` time provides insight into the scalability of our equivalence aggregation. We observe sub-linear completion time for all `pgrep` experiments, and note that for smaller files the completion time is dominated by the time to initialize the file group.

4.1.3. Parallel `tail`

The `tail` utility with the "-f" option is used to follow system log activity in real-time. Existing software enhances the functionality of `tail` to include monitoring multiple files [17] and multiple hosts [14], and to highlight interesting lines of output [11,17]. Combining all three enhancements, we developed `ptail` for following large distributed file groups. 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 `syslog` message format. Correlation of distributed events can benefit applications such as identifying misconfigured network services (e.g., many clients of the service notice a problem and generate an identical error) and security (e.g., distributed intrusion or denial-of-service attacks).

We used a synthetic log generator that controls the rate of log entries and the percentage of equivalent entries across hosts to evaluate `ptail`. `ptail` significantly reduces the number of output lines by combining equivalent log entries. In general, for a group of size **G**,

TABLE 3. PSYNC FILE STATISTICS
For each file, total size, size of new data, and size of block match metadata is given in bytes. `psync` sends only the new data and matches to servers, reported as a percentage of total size.

		psync		
File	**Size (B)**	**New (B)**	**Matches (B)**	**Sent**
`/etc/fstab`	7063	1579	216	25.4%
`/etc/passwd`	38742	1025	1332	6.1%
`/etc/services`	559563	3531	19548	4.1%

		Group Size (# files)		
services-udp		**1024**	**2048**	**4096**
grep	**Time (secs)**	12.690	27.440	52.160
	Size (MB)	376.897	754.949	1519.513
pgrep	**Time (secs)**	0.714	0.762	0.882
	read	0.669	0.698	0.780
	init	0.045	0.064	0.102
	Size (MB)	0.347	0.347	0.347
	% of grep	0.09%	0.05%	0.02%
meminfo-Total		**1024**	**2048**	**4096**
grep	**Time (secs)**	0.580	1.250	2.570
	Size (MB)	0.049	0.097	0.197
pgrep	**Time (secs)**	0.053	0.074	0.120
	read	0.008	0.008	0.007
	init	0.045	0.066	0.113
	Size (MB)	0.000	0.000	0.000
	% of grep	0.09%	0.04%	0.02%
meminfo-Free		**1024**	**2048**	**4096**
grep	**Time (secs)**	0.580	1.290	2.600
	Size (MB)	0.049	0.097	0.197
pgrep	**Time (secs)**	0.054	0.082	0.139
	read	0.010	0.014	0.021
	init	0.044	0.068	0.118
	Size (MB)	0.016	0.036	0.068
	% of grep	33.39%	36.51%	34.73%

Figure 6. `pgrep` vs. `grep` scalability
Time and output size is given for three searches. pgrep searched TBON-FS files, and grep searched NFS files. For pgrep, time is split into two main activities: **read** (`read` and aggregate) and **init** (`gopen` and `gbindaggr`).

a percentage **P** of equivalent events, and **L** log entries generated per host, the total number of output lines is reduced to **L((1-P)G + P)** from the total lines generated **LG**. The aggregated output eases log analysis, and reduces the storage required to keep log history.

4.1.4. Parallel `top`

`top` is a simple yet powerful utility for displaying resource utilization by processes on a single host. We know of no existing tool that provides the same functionality for many distributed hosts, so we created `ptop`. As with standard `top` for Linux, `ptop` gathers information from files in `/proc`. Aggregation is used to calculate summaries and support the sorting and filtering capabilities of `top`. To give greater insight into distributed resource use, we added two new grouping facilities that summarize processes having the same command name, both for a specific user and across all users. When grouping, one can view total, average, or maximum utilization. Using `ptop`, one can answer many interesting questions: what application is using the most memory pages, what is the average CPU utilization for my parallel application processes, and who is playing solitaire?

To evaluate `ptop`, we measured average latency to collect and aggregate process information and average CPU utilization at TBON-FS servers. On Thunder, we ran `ptop` for 60 seconds with and without command grouping, using delay intervals of 5, 10, and 30 seconds (`top`'s default is 5 seconds) and reporting the top 100 processes. Performance with and without grouping was indistinguishable, so we show the grouping case. Fig. 7 shows that `ptop` can 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`. The time scales logarithmically compared to the file group size. We note that TBON-FS server CPU use was under 0.5% for the 30 second interval in all experiments. Thus, `ptop` can be used for low-impact, continuous monitoring.

4.2. 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 resource utilization information from local files and regularly multicasts updates to fellow gmonds in the cluster. The gmetad for a specific cluster queries one of the gmonds to collect the latest data for all cluster hosts.

With no previous knowledge of the Ganglia source code, we added group file operations to version 3.0.4 in a span of a few weeks. We unified the architecture to be completely tree-based by removing the use of IP multicast among cluster hosts. As cluster size grows, the use of multicast causes each host in the multicast group to incur a linear increase in CPU and network load and in memory storage of group state [15]. To combat the negative effects of using multicast, the default update interval is 90 seconds, which precludes real-time monitoring.

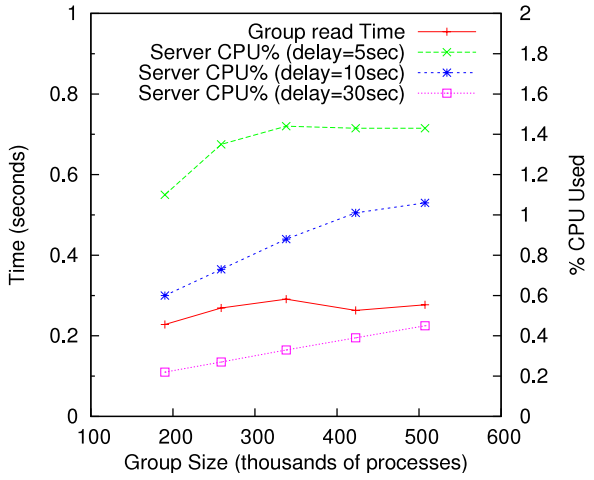

Figure 7. `ptop` scalability
Server CPU use and group `read` time vs. number of processes.

Our updated version, Ganglia-tbonfs, replaces the gmonds with TBON-FS servers. We use custom aggregations to store metric data in round-robin databases at root and 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 transform summary or host state from the databases into XML documents and graphs.

To compare Ganglia-tbonfs to Ganglia, we measured average gmetad and gmond CPU utilization over a period of 30 minutes on Thunder. Metrics were collected according to the default intervals set by Ganglia. With Ganglia, gmetad and gmond CPU use increased linearly as we increased the number of monitored hosts. In contrast, Ganglia-tbonfs suffered no ill effects from increasing the monitored group size. At the largest scale, Ganglia-tbonfs used 50% less CPU for the gmetad and just 0.37% CPU at the gmonds (versus 0.95% for Ganglia).

5. Related Work

Steere's dynamic file sets [25] provide an abstraction similar to that of the group file, and share the same motivation for eliminating serialization imposed by the file system interface when operating on groups of files. To reduce group operation latency, Steere used a threaded distributed file system client to prefetch whole files in parallel, and returned file descriptors using a set iterator construct that favored fully fetched data. The dynamic set abstraction still requires iteration over the files

for each group operation, and scalability for large groups is hindered by fetching all file data to the client.

Group file operations are related to the MapReduce system [6], as both the map and reduce operations are distributed aggregation of file data. In MapReduce, a large data set is partitioned into many small fixed-size chunks stored at hundreds or thousands of servers. Considering 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. Google's MapReduce system is tightly bound to the Google File System, as group file operations are currently bound to TBON-FS. Group file operations are similar to the Sawzall [21] and Pig Latin [18] programming languages that expose data parallelism in simple interfaces that hide the underlying parallel system. Unlike these languages, group file operations use familiar file system calls.

CFS [5] and PAST [24] use overlay networks for scalable file storage and retrieval. However, they only support read-only files and do not provide in-network aggregation of data from groups of distributed files. San Fermín [4] provides large-scale, fault-tolerant distributed data aggregation using a binomial swap forest on a peer-to-peer overlay network. For fault tolerance, each distributed data source exchanges aggregated data with peers to compute the final aggregate. Unfortunately, the overhead of this duplicate computation and communication is too high for many distributed systems and tools, most notably performance monitoring of HPC systems.

Gropp and Lusk [10] and Brim et al. [2] created parallel versions of common utilities for scalable management of distributed systems. Neither work uses aggregation of group results to aid in analysis or presentation. Instead, both annotate output with the origin host and require post-processing of results. These tools could benefit from the integration of a TBON infrastructure to eliminate redundant output using group summaries, as we have done in our parallel tools.

6. CONCLUSION

Group file operations are a new, intuitive idiom for operating on large file groups. The idiom eliminates iteration over group members, which allows TBON-FS to provide scalable group file operations by using a TBON for distributed communication and aggregation. We integrated group file operations into several tools and one existing middleware system. Experimental and qualitative observations with a prototype TBON-FS show the applicability, ease of use, and scalability of group file operations.

7. REFERENCES

[1] D. C. Arnold, "Reliable, Scalable Tree-Based Overlay Networks", Ph.D. Dissertation, Computer Sciences Department, University of Wisconsin-Madison, December 2008.

[2] 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.

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

[4] J. Cappos and J. Hartman, "San Fermín: Aggregating Large Data Sets using a Binomial Swap Forest", *5th USENIX Symposium on Networked Systems Design and Implementation*, April 2008.

[5] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS", *SIGOPS Oper. Sys. Rev.* **35**, 5, December 2001, pp. 202-215.

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

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

[8] 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.

[9] A. Goscinski, M. Hobbs, and J. Silcock, "GENESIS: an efficient, transparent and easy to use cluster operating system", *Parallel Computing* **28,** 4, April 2002, pp. 557-606

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

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

[12] E. Hendriks, "BProc: The Beowulf distributed process space", *2002 International Conference on Supercomputing*, pp. 129-136, New York, New York, June 2002.

[13] G. Lee, D. Ahn, D. Arnold, B. de Supinski, M. Legendre, B. Miller, M. Schulz, and B. Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores", *SC 2008*, November 2008.

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

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

[16] C. Morin et al., "Kerrighed: A Single System Image Cluster Operating System for High Performance Computing", *9th International Euro-Par Conference*, Klagenfurt, Austria, August 2003.

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

[18] 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.

[19] H. Ong et al., "Kernel-level single system image for petascale computing", *SIGOPS Oper. Sys. Rev.* **40**, 2, April 2006, pp. 50-54.

[20] "OpenSSI (Single System Image) Clusters for Linux", http://openssi.org/, August 2006.

[21] 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.

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

[23] 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.

[24] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", *SIGOPS Oper. Sys. Rev.* **35**, 5, December 2001, pp. 188-201.

[25] D. C. Steere, "Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency", *SIGOPS Oper. Sys. Rev.* **31**, 5, December 1997, pp. 252-263.

[26] A. Tridgell and P. Mackerras, "The rsync algorithm", Australian National University Technical Report TR-CS-96-05, June 1996.