Chapter 1

Filesystem Partitioning via Hierarchical Cluster Analysis

Contributed by Tim Shaffer

1.1 Introduction

As research computation achieves larger scales and takes greater advantage of hardware accelerators such as FPGAs, GPUs, and TPUs, the availability and performance of the storage system becomes a bottleneck in the overall performance of an application. Checkpointing, scratch storage, and data distribution all place storage in the critical path of the application, forcing the storage system to keep up. Parallel filesystems offer a way of delivering large file I/O bandwidth, scaling with storage hardware by striping blocks, files, and volumes across different devices, and maintaining consistent metadata and organization.

While data servers can easily increase replication and shift load away from overloaded servers, metadata servers must maintain stronger consistency guarantees. It is common to use filesystem partitioning to balance the load of requests. Choice of partitioning scheme depends on user activity and filesystem organization. While it is possible in some cases to shift excessive loads between metadata servers, the requirement for consistent semantics of operations makes it difficult to distribute a single part of the filesystem tree across multiple metadata servers. When load on a particular metadata server becomes excessive, all requests (including some data requests) cannot be served efficiently. Users then experience degraded performance or loss of service despite. In this situation, accesses to a single part of the filesystem translate to load on a single metadata server responsible for a partition. Thus the overall system's performance is degraded while leaving data storage nodes underutilized [15]. For parallel filesyetems at increasing scales, metadata bottlenecks become the limiting factor for performance and usability. These pathological metadata access patterns motivate many of the design choices of modern parallel filesystems.

Widely deployed shared filesystems such as Panasas [14], Lustre [2], Ceph [13], Gluster [9], and HDFS [11] rely on separate data and metadata servers to make a filesystem tree and its data visible from anywhere in the system. Typical access patterns observed for file data allow for optimizations to servers that can simplify operation and improve throughput and parallel access. Large reads and write of file data are especially suited to bulk access operations. Unlike the generally simple and flat structure of data stores, metadata servers must maintain consistent views of hierarchical files and directories. In order for a node to to read a file's data, path resolution is the responsibility of metadata servers. These lookups and permission checks eventually determine the location where

the actual data are stored. Efficient path resolution and metadata lookup is thus critical for the performance of a shared filesystem.

Metadata performance becomes a key bottleneck in most storage systems at extreme scale. Strategies such as caching, striping, and larger transactions allow for improvements in file data access. Unfortunately, serving the system's metadata poses distinct challenges that preclude approaches used for data access. Metadata elements are small (e.g. user-visible inode information), require stronger consistency than data, and are accessed through small update transactions rather than large linear reads and writes. It also becomes difficult to maintain a consistent view of directory structures among a large number of nodes with (potentially) an extremely large number of directory entries.

The problem of metadata handling is in the general case extremely difficult to solve. The consistency semantics of general-purpose filesystems severely limit potential optimizations. The different ways a shared filesystem is used each require somewhat different semantics. Data passed between concurrent processes, for example, requires strong consistency at each filesystem interaction. Between sequential processes, filesystem data requires strong consistency eventually. Static data such as software will not change during the execution of a given task does not require additional consistency guarantees.

Metadata activity is often extremely unbalanced, as production high performance computing (HPC) applications generate irregular bursts of metadata access. Application startup is particularly problematic for the storage system. While researchers often treat HPC applications as "simple" programs to be launched, the steps required are almost always more complicated. What appears to be a single "application" may actually be a complex collection of interpreted programs, dynamic libraries, configuration files, and calibration data. Loading the complete structure of the application into cluster memory at startup results in tens of thousands of interactions with the filesystem at each node. When thousands of nodes of the cluster attempt to load the same application at the same time, the filesystem must handle thousands of small transactions from each node before any one can make progress.

For scientific applications specifically, metadata behavior often becomes the limiting factor for performance. Scientific software is likely to use the shared filesystem in all of the previous ways. A single application may store intermediate files, synchronize between steps of an analysis, distribute application software, and collect results from multiple worker nodes. A general-purpose filesystem must adequately support the strain imposed by each. Efficient handling of common cases such as software distribution can be the key factor in attempting to scale up an analytic workflow. Poor choice of filesystem use or missed optimizations will make shared computing resources unusable for a given researcher, or in some cases for all users of a site.

Thus the goal of this work is to identify "related" parts of the filesystem where optimizations will have the most impact. By tracing applications, we can observe their filesystem access patterns. We can then use this linear log of events to express an application's behavior as a graph. From there, we identify clusters in the graph as sets of strongly related filesystem entries. Using the knowledge of these clusters, we can apply optimizations such as local caching and pre-fetching more effectively. Before starting an application run, for example, we could make a local copy of an entire cluster of related filesystem entries in one bulk operation. This approach can reduce load on the overall system without changing the application's behavior. By choosing clusters effectively, we can minimize the amount of data copied/transferred/etc. while still maintaining the performance benefits of optimizations.

To match real-world usage, we would like to be able to process data from multiple application runs, allowing for additional samples to be added as they become available. We would also like to be able to get some recommendation even before all runs are completed and processed, because in the

extreme case there could be a continuous stream of data with images or cache entries being prepared periodically based on observed data. This design would require the use of efficient, streaming graph algorithms.

1.2 The Problem as a Graph

Applications at HPC centers are composed of (a possibly large number of) individual processes. Running a process requires, at minimum, path search and library loading. It is also common for processes to search for and read in input, configuration, or calibration data. Some applications use the shared filesystem as a means to synchronize between components. To allow the application to recover from failures, checkpoints can be written out as well. Finally, there is usually some output data flushed to the filesystem. The latter three uses rely on strong filesystem semantics and consistency guarantees, limiting the optimizations available. The former uses, however, are good candidates for further examination.

By identifying parts of the filesystem that are used together, for example all of the libraries a single program loads, it becomes possible to use more target optimizations such as pre-staging a frequently used subset of static metadata entries on nodes. Unfortunately, determining which parts of a filesystem are relevant to a particular application is difficult. There is generally a set sequence of operations for path search, library loading, etc. The exact operations can be non-deterministic or data dependent, however. Furthermore, each individual process of a complete application has distinct behavior, though some patterns are likely to be repeated across processes. There is thus no way to determine a priori which parts of the filesystem an application relies on. It is also not sufficient to try to identify one or a few "program directories" containing all needed pieces. Research applications can (and very often do) use creative filesystem organizations that are not amenable to automatic dependency tracking.

Lacking an a priori method to determine the filesystem dependencies of an arbitrary research application, it is instead necessary to observe filesystem behavior over multiple application runs. It is possible to trace the behavior of individual processes using tools like `strace` to capture syscalls. Since there is a performance penalty in tracing, it is generally better to trace some fraction of processes.

Using these process-level traces, the next step is interpreting the sequences of accesses and choosing groups of related filesystem entries. It is not sufficient to simply collect every filesystem entry that was accessed. This approach collects broad trees of filesystem entries that in practice include substantial amounts of irrelevant data. Instead, a better approach would take into account the fine-grained behavior of the processes. For example, a directory that is listed once during library search and never accessed again is a poor candidate for optimization. On the other hand, a set of libraries that are accessed consecutively by every process should be grouped together and pre-staged on nodes to reduce traffic to the shared filesystem.

To capture both the frequencies and orderings of filesystem accesses, we can build a directed graph based on the syscall traces of each process. In this representation, filesystem entries are the vertices of the graph. The events comprising the syscall traces are used to derive edge weights, with large numbers of accesses resulting in high edge weights. For a process that most recently accessed filesystem entry A and next accesses B, we increase the edge weight of $A \rightarrow B$ by one, creating the edge if it does not exist. This representation includes far fewer vertices and edges than events in the syscall trace. In addition, it is amenable to streaming updates as new traces become available or the input data and configuration change.

1.3 Some Realistic Data Sets

The `strace` utility is widely available on Linux based systems, making it possible for researchers to collect syscall traces from their applications. Aside from the previously mentioned performance overhead, there is little barrier to profiling as the process does not require changes to the application. It might be possible to refer to a graph database or to generate synthetic graphs, but simply profiling the actual application will be more effective. As an example application that researchers actively use in an HPC context, we collected syscall traces of MAKER [5], a complex bioinformatics application. Aside from the application itself and its input data, MAKER depends directly or indirectly on an additional 40 languages and libraries. Each phase of a MAKER analysis uses some subset of these dependencies and inputs. Over the course of an analysis, MAKER spawns a large number of individual processes, each of which goes through library loading, input data selection, etc. In our test run on a small dataset, MAKER performed 1.8 million I/O operations. This run included bursts of thousands of metadata I/O operations per second. The total running time was 35 minutes. Note that it is not uncommon for HPC applications to run for hours or even days. A longer-running application would produce a corresponding increase in the number of I/O operations. For the following sections, we will explore additional applications and datasets.

Rather than representing each metadata operations directly, the graph representation of the execution trace aggregates these events into edge weights. Each vertex in the graph corresponds to a filesystem entry accessed during the run. For this small run, the resulting graph included roughly 25,000 vertices. An analysis on a larger data set or using an application with more dependencies would result in a larger number of vertices. The graph for this particular run included roughly 130,000 edges. Again, the number of edges and ratio of vertices to edges is strongly dependent on both the application and the dataset. Since the programs spawned during a MAKER analysis exhibit both non-deterministic and data dependent behavior, there can also be limited variation in the properties of the graphs of otherwise identical runs. Streaming events from multiple analysis runs into the same graph will also result in changes to the graph properties, though it is hard to predict behavior in the general case over all applications. As general trends, merging multiple distinct applications into the same graph should result in a largely disconnected cluster for each application's activity. Combining analyses of different data sets in the same application should result in an increased graph size with some core vertices common to all runs. Finally, merging events from analysis of the same data in the same application should not significantly affect the properties of the graph.

We also recorded traces for two smaller applications for comparison. The program `true` is a standard Unix utility that simply exits successfully without performing any work. As a dynamically linked GNU program, executing `true` still results in library search, user lookup, reading locales, etc. The `bash` trace records a shell starting up and immediately exiting. In addition to the activities of `true`, `bash` loads a number of configuration files, inspects the user's home directory, and spawns several other processes. The sizes of the three traces and the resulting graphs are tabulated below.

	Events	Nodes	Edges
`true`	47	46	45
`bash`	5,499	840	1,569
MAKER	1,813,544	24,897	129,153

Here `true` has the smallest number of events and appears to follow a linear sequence of filesystem events. Since it consists of multiple process operating in parallel, the trace for `bash` is less clearly defined. MAKER, as the largest application traced, includes significantly more events which in

turn produce a graph with two orders of magnitude more nodes and edges than the next largest application.

1.4 CLUSTERING-A Key Graph Kernel

To detect clusters of related filesystem entries, we propose applying hierarchical cluster analysis to the execution trace graph of an application. Clusters in the graph serve as reasonable groupings of filesystem entries for partitioning or pre-staging at worker nodes. In addition, the series of hierarchically-related clusters that results from this analysis allows some flexibility in the granularity of clusters. When applied to real-world applications, a completely automated approach is difficult in the general case. Providing a choice in cluster granularity allows the user to apply domain knowledge to make the best decision while still providing some automated assistance.

To illustrate hierarchical cluster analysis, we use the Girvan–Newman algorithm [6], a well-studied method. Girvan–Newman repeatedly removes edges from the graph, with the remaining connected components as the clusters. The edge to be removed each step is chosen based on **edge betweenness**, a centrality measure of the number of shortest paths in the graph passing along a given edge. The underlying assumption is that by iteratively removing non-central edges, you can gradually cut apart the clusters. The algorithm proceeds until all edges have been removed. Thus the user can choose precise cluster granularities from individual vertices up to the entire graph. The algorithm itself (taken from [6]) is given in Figure 1.

Algorithm 1 The Girvan–Newman Algorithm

Require: Graph $G = (V, E)$.
  Calculate betweenness $g(e)$ for each edge $e \in E$.
  while $|E| > 0$ **do**
    Remove the edge e with the highest betweenness $g(e)$.
    Recalculate betweennesses for all edges affected by the removal.
    Identify clusters as connected components in G.

The betweenness can be calculated using [8] in $O(mn)$ time, where m is the number of edges and n is the number of vertices in G. Since betweenness is calculated at the removal of each edge, the algorithm runs in $O(m^2n)$ time. Determining connected components can be performed in $O(m+n)$ time. Since only changes to betweenness need to be calculated after the first time, this repeated computation can be confined to a single connected component rather than the whole graph. One way to perform this optimization would be to store the set of shortest paths passing through each edge. On removal, it is only necessary to recompute the shortest paths between the pairs of vertices in that list. As the clusters become smaller and smaller, in practice this optimization significantly reduces the computation.

1.5 Prior and Related Work

Several approaches to achieving sufficient metadata performance in shared filesystems have been explored. One approach is to separate metadata from the parallel filesystem completely. This approach maps metadata storage tables to file objects in the parallel filesystem [16, 10]. The total metadata transaction rate of the system is improved, but each client must still make many small transactions while using the service. Another approach is to introduce new operations that access

metadata in bulk or with weaker consistency guarantees. Examples of this include the proposed `getlongdir` and `statlite` system calls [12], which are, unfortunately, not widely implemented. This reduction in the transaction rate between clients and servers complements other approaches.

1.6 A Sequential Algorithm

The pseudocode given in Section 1.4 is quite simple, but uses several other algorithms as building blocks. In particular, the Girvan–Newman Algorithm repeatedly computes the edge betweenness, which itself depends on computing the all-pairs shortest paths of the graph. Thus a framework that supports a number of common graph algorithms would be best for implementing the algorithm. The graphs generated from the traces consist only of vertices and directed edges, where edges have integer weights. Thus any graph language or framework should be able to represent the generated graphs. Assuming a sparse representation, a good implementation of the Girvan–Newman Algorithm should match the time complexity given previously.

1.7 A Reference Sequential Implementation

For processing the event logs and the implementation of the Girvan–Newman Algorithm, we chose Python and NetworkX [7]. Python has good support for text processing and regular expressions, which are necessary for parsing the text event logs. NetworkX is a Python library for working with graphs. Conveniently, NetworkX includes an implementation of the Girvan–Newman Algorithm. Thus the majority of the work is in parsing the event logs and constructing the graph in a way that NetworkX can use. Each event shows the invoking PID, syscall, its arguments, and the result.

```
29204 open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
```

For example, this is a single syscall that successfully opened `/etc/passwd`. These lines were parsed to find file-related syscalls and determine the path(s) involved. To construct the execution graph, we simply note the filesystem entry accessed previously and increment the weight of edge to the entry referenced in the current event, adding this edge if id does not exist. This part was written as part of the parsing phase and independently of NetworkX, simply storing the edges in Python data structures. This snippet shows the section of the main loop for building the graph.

```
for path in EVENTS:
    path = path.strip()
    if not path in self.paths:
        self.paths[path] = self.ctr
        self.ids[self.ctr] = path
        self.ctr += 1
    k = (prev, self.paths[path])
    v = self.graph.get(k, 0)
    self.graph[k] = v + 1
    prev = k[1]
```

After processing the events of a trace and summing up all edge weights, it is straightforward to convert the graph to a NetworkX representation.

```
def to_networkx(self):
    G = networkx.Graph()
```

```
    for (k, v) in self.ids.items():
        G.add_node(k, path=v)
    for ((l, r), v) in self.graph.items():
        G.add_edge(l, r, weight=v)
    return G
```

Given a NetworkX graph, finding clusters via the Girvan–Newman Algorithm is as simple as invoking a function.

```
G = event_graph.to_networkx()
components = networkx.algorithms.community.centrality.girvan_newman(n)
for comp in itertools.islice(components, depth):
    print(tuple(sorted(c) for c in comp))
```

1.8 Sequential Scaling Results

We used the the Girvan–Newman Algorithm based approach to identify clusters in the three traces described previously. The tests were run on a machine at the Center for Research Computing (CRC) at the University of Notre Dame. The machine had an Intel Xeon E5620 CPU with 8 cores and 32 GB of RAM. The machine was running Python 2.7 with NetworkX 2.2 installed locally. The running times for computing the first round of edge betweenness and the full the Girvan–Newman Algorithm clustering analysis on the three application traces are tabulated below.

	Edge Betweenness	Girvan–Newman
`true`	0.41 s	0.43 s
`bash`	10. s	290 s
MAKER	?? ($>$ 45 min.)	??

For MAKER, the running time proved to be excessive and computation of the the Girvan–Newman Algorithm analysis timed out. We also used the times for the first round of edge betweenness to estimate the time that would be required, but found that even a single step took a large amount of time and aborted it.

The observed running times appear to be consistent with the time complexity given, but it is impossible to determine this with any certainty based on two data points. Based on the measured running times and the graph sizes, we would indeed expect an algorithm with $O(|E|^2|V|)$ complexity to time out. We will save a more definitive complexity analysis for the improved implementation.

1.9 An Enhanced Algorithm

The ultimate objective for this system is to identify clusters in the filesystem as an application runs, and to easily incorporate additional execution samples to improve the results. Thus a computationally expensive, offline approach is unacceptable. Moving away from the Girvan–Newmann Algorithm, we instead consider using a streaming algorithm. Rather than first constructing the graph and then building communities, a streaming community detection would allow us to maintain approximate communities as the graph is constructed. In addition, use of a centrality-based method like Girvan–Newman resulted in significant computational cost. For the smaller traces, this was an acceptable approach. At the scale of a realistic application, however, the $O(|E|^2|V|)$ time complexity becomes unacceptable. For this application, a decrease in accuracy is acceptable to obtain a computationally tractable approximation.

Thus, we are interested in an approximate method that supports larger scales and is amenable to a streaming implementation. We chose the Louvain method [3] as our stating point. This method has been used to quickly detect communities at large scale, and is very performant in practice. The Louvain method attempts to maximize *modularity*, a measure of the degree to which a graph is structured in modules or clusters. Computing the optimal modularity of a graph, however, is known to be an NP-complete problem [4]. Thus for large graphs, it is necessary to use an approximation instead. Rather than repeatedly computing edge betweenness for the entire graph and removing an edge as in Girvan–Newman, the Louvain method combines subgraphs. For each pass, the algorithm chooses some subgraphs to merge. Since it would be computationally expensive to identify optimal choices here, the algorithm arranges the subgraphs in a sequence and only considers merging them with their neighbors. The Louvain method also takes advantage of the fact that it is easy to compute the *change* in modularity due to moving an isolated vertex. To finish the pass, the algorithm constructs a new graph in which the newly produced clusters are contracted into single vertices. By applying the same steps for a small number of passes, the Louvain algorithm in practice operates in linear time on commonly encountered sparse graphs. An evaluation [3] of the method found acceptable performance and accuracy even on graphs consisting of hundreds of millions of nodes and billions of edges. In addition, the authors found that on all graphs examined, results converged after five or fewer passes.

1.10 A Reference Enhanced Implementation

For an enhanced implementation of community detection that supports more realistic graph sizes, we chose to use STINGER [1], a high-performance graph data structure usable as either a library or through a standalone server. Since existing pieces to parse logs and generate event streams were written in Python, we chose to run STINGER as a server and feed data to it. This is also closer to the original goals of the application, i.e. to have a server that provides information on usage patterns as data become available. STINGER allows asynchronous ingestion of events, so that for example a trace could run in the background and feed into the STINGER server without blocking the application being traced. As changes to the graph are incorporated, the STINGER server maintains an approximate set of communities. At any point in time, it is possible to query the server to get a current listing of clusters.

The algorithm included with STINGER is based on Louvain's method, implemented to take advantage of parallelism when traversing the graph. While the STINGER data structure and the RPC interface used to communicate with the server are both substantial, the included implementation of community detection is concise enough to include here.

```
void community_detection(stinger_t * S, int64_t NV,
        int64_t * partitions, int64_t maxIter){
    //we need the sum of the total weights in the graph to calculate modularity
    double_t m = 0;
    STINGER_FORALL_EDGES_OF_ALL_TYPES_BEGIN(S){
                            m += STINGER_EDGE_WEIGHT;
                        }STINGER_FORALL_EDGES_OF_ALL_TYPES_END();

    //begin by setting each vertex to its own partiton
    for (int64_t i = 0; i < NV; i++){
        partitions[i] = i;
    }
```

```
    int64_t num_partitions = NV;
    partitions = louvain_method(S,partitions, NV, m, maxIter);

}

int64_t *louvain_method(stinger_t * S, int64_t * partitions, int64_t size,
        int64_t m, int64_t maxIter){
    int64_t num_moves;
    int64_t num_iter = 0;
    do {
        num_moves = 0;
        num_iter += 1;
        for (int64_t i = 0; i < size; i++) {
            double_t maxMod = -DBL_MAX;
            int64_t label = partitions[i];
            STINGER_FORALL_OUT_EDGES_OF_VTX_BEGIN(S, i)
                                    {
                                        double_t modul = modularity(S, i,
                                            STINGER_EDGE_DEST, m);
                                        if (modul > maxMod) {
                                            label = partitions[STINGER_EDGE_DEST];
                                            maxMod = modul;
                                        }
                                    }
            STINGER_FORALL_OUT_EDGES_OF_VTX_END();
            if (partitions[i] != label) {
                //we move the vertex
                partitions[i] = label;
                num_moves += 1;
            }
        }
    }while (num_moves > 0 && num_iter < maxIter);
    return partitions;
}

double_t modularity(stinger_t * S, int64_t vertex_a, int64_t vertex_b, double_t m){
    int64_t ka_out = 0;
    int64_t kb_in = 0;
    int64_t edge_between = 0;
    double_t modularity = 0;
    STINGER_FORALL_OUT_EDGES_OF_VTX_BEGIN(S, vertex_a){
                                ka_out += STINGER_EDGE_WEIGHT;
                                if (STINGER_EDGE_DEST == vertex_b){
                                    edge_between += STINGER_EDGE_WEIGHT;
                                }
                            }STINGER_FORALL_OUT_EDGES_OF_VTX_END();
    STINGER_FORALL_IN_EDGES_OF_VTX_BEGIN(S, vertex_b){
                                kb_in += STINGER_EDGE_WEIGHT;
```

```
                                    if (STINGER_EDGE_SOURCE == vertex_a){
                                        edge_between += STINGER_EDGE_WEIGHT;
                                    }
                                }STINGER_FORALL_IN_EDGES_OF_VTX_END();

    modularity = (edge_between/m) - ((double_t)(ka_out*kb_in)/pow(m,2));
    return modularity;

}
```

1.11 Enhanced Scaling Results

Compared to the Girvan–Newman approach, we observed marked performance improvement with STINGER's parallel community detection based on the Louvain method. This modularity-based algorithm was also more usable as it allows streaming updates to the graph, with current communities listed on demand. In all cases, there was a constant amount of time required to start the server, prepare memory allocations, register components, etc. This was not included in the computation times listed. Since STINGER uses a streaming computation model for this algorithm, processing events and updating the graph is inextricably linked with the algorithmic computation. In the table below, "Event Ingestion" refers to the time required to stream the complete stream of events for the application to the STINGER server. This is an asynchronous process, so the ingestion completes while the server performs computation in the background. "Community Detection" includes both additions/updates to edges and the community detection proper.

	Event Ingestion	Community Detection
`true`	0.01 s	0.01 s
`bash`	0.01 s	0.01 s
MAKER	0.8 s	197 s

1.12 Conclusion

In this work, we sketched a graph-based description of filesystem access patterns for scientific applications. With suitable choices for algorithms and approximations (and efficient implementations), we computed hierarchical communities among filesystem entries based on event streams captured during application runtime. STINGER's modularity-based community detection implementation gave a computationally tractable approximation. In addition, it was fast enough to process an event stream in step with the live application. As future work, we would like to use the computed community information as part of the application pipeline, e.g. to pre-fetch cache contents on worker nodes. This information would be useful in enforcing more organized IO patterns in large distributed scientific applications.

1.13 Response to Reviews

In response to reviews, we fixed a number of errors that were pointed out. We also elaborated on computing partial updates mentioned in the pseudocode. One of the reviews found the introduction to be poorly organized, so we rearranged and reworded pieces there. The reviewer was also not clear on the objectives, so we expanded the explanation there.

Bibliography

[1] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[2] R. Behrends, L. K. Dillon, S. D. Fleming, and R. E. K. Stirewalt. White paper: Lustre file system high-performance storage architecture and scalable cluster file system. Technical report, Sun Microsystems, Menlo Park, California, December 2007.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[4] U Brandes, D Delling, M Gaertler, R Goerke, Martin Hoefer, Z Nikoloski, and D Wagner. Maximizing modularity is hard. Technical report, University of Konstanz, Germany, 2006.

[5] M. S. Campbell, C. Holt, B. Moore, and M. Yandell. Genome Annotation and Curation Using MAKER and MAKER-P. *Curr Protoc Bioinformatics*, 48:1–39, Dec 2014.

[6] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[7] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[8] M. E. J. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Phys. Rev. E*, 64:016131, Jun 2001.

[9] Inc. Red Hat. Gluster. http://www.gluster.org/, 2017. Accessed 2018-09-28.

[10] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, Nov 2014.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[12] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. Extending the posix i/o interface: A parallel file system perspective. *Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-302*, 2008.

[13] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[14] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[15] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 8:1–8:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[16] Q. Zheng, K. Ren, and G. Gibson. Batchfs: Scaling the file system control plane with client-funded metadata servers. In *2014 9th Parallel Data Storage Workshop*, pages 1–6, Nov 2014.