SAND2014-0372C

An Evaluation of BitTorrent's Performance In HPC Environments

***Abstract*—A number of novel decentralized systems have recently been developed to address challenges of scale in large distributed systems. The suitability of such systems for meeting the challenges of scale in high performance computing (HPC) systems is unclear, however. In this paper, we begin to answer this question by examining the suitability of the popular BitTorrent protocol to handle dynamic shared library distribution in HPC systems. To that end, we describe the architecture and implementation of a system that uses BitTorrent to distribute shared libraries in HPC systems, evaluate and optimize BitTorrent protocol usage for the HPC environment, and measure the performance of the resulting system. Our results demonstrate the potential viability of BitTorrent-style protocols in HPC systems, but also highlight the challenges of these protocols. In particular, our results show that the protocol mechanisms meant to enforce fairness in a distributed computing environment can have a significant impact on system performance if not properly taken into account in system design and implementation.**

I. Introduction

In this paper, we present our research to evaluate the suitability of the BitTorrent protocol in an high performance computing (HPC) context. We describe and evaluate a system, BitLib, that attempts to solve the data distribution and access problem of dynamic shared libraries (DSLs) on HPC systems. BitLib is built using off-the-shelf components, such as the ctorrent BitTorrent client and the Linux Filesystem in Userspace (FUSE) module.

This work is part of a larger effort to examine the suitability of distributed system techniques for handling communication and data distribution challenges in HPC systems [16], [4]. By identifying existing protocols that can be adapted to HPC environments and evaluating their strengths and weaknesses, we seek to expand the tool set available to solve HPC communication and data distribution problems. BitTorrent, one of the more mature and well-known distributed systems protocols, is particularly interesting in this regard, due to its potential suitability for solving DSL distribution problems.

We use shared library dissemination as the HPC data distribution problem for our evaluation. While dynamic executables are common on single systems, until recently, most scalable HPC applications have been statically linked. Dynamic linking has several benefits, such as updating libraries without recompilation and reduced memory overhead. With increasing application size, dynamic linking has become more desirable in an HPC context. Simply pulling dynamic libraries from HPC filesystems, which are frequently optimized for write performance, can result in poor system performance.

Our work makes several contributions. First, we evaluate the use of BitTorrent in an HPC environment, identifying its strengths, weaknesses, and sources of those weaknesses. Secondly, we describe the architecture and implementation of a system, BitLib, which uses BitTorrent to address the DSL distribution problem. Finally, we evaluate the performance of this system and analyze the potential advantages and pitfalls that the BitTorrent protocol presents in this situation.

In the rest of this paper we present further details on our work. In Section II, we discuss the background of the DSL problem and the BitTorrent protocol. Section III contains a description of the architecture of our system. Section IV presents the results of an evaluation of this system on two mid-sized clusters, as well as an initial scaling study on a large-scale supercomputing system. Section V then presents an in depth analysis of the BitTorrent protocol, detailing the weaknesses and the changes we made to our system to mitigate them. Section VI discuses other work done in the area. Finally, Section VII concludes and presents directions for future work.

II. Background

In this section we discuss the background for this work. First we discuss the BitTorrent protocol. We then discuss the issues behind the distributed shared library problem we use to evaluate the suitability of BitTorrent to solving data movement problems in HPC systems.

A. BitTorrent

BitTorrent [3] is a distributed, peer-to-peer protocol designed to distribute sets of files in a decentralized manner. It's expected use case is transferring a small number of very large files. It also assumes comparatively low-bandwidth, high latency links, that peers that mostly download all the files in a torrent in a single download session, and that peers may attempt to freeload on network services.

1) Overview: The basic BitTorrent system includes three types of participants: an *uploader*, a *tracker*, and one or more *clients.* The uploader (also known as the initial seeder) begins by creating a torrent description file for a file or set of files within a directory, and dividing its list of files into pieces that can be requested independently. It then registers the torrent and itself with a tracker, and waits for a client to connect and request pieces.

Clients also use the description file to connect to the associated tracker. When a client connects to the tracker, it registers as a peer. It then performs an announce operation, which gets the client a list of peers from the tracker and requests to connect with other clients. As the other clients respond and establish connections, the client receives piece status data, indicating what data each client has. The client then requests pieces from other clients. The other clients receive these requests and serve the data. This protocol works well on the Internet for distributing movies and large open-source software projects, for example.

2) Key Mechanisms: BitTorrent includes a number of protocol mechanisms and policies designed to make it scale in a distributed environment with limited trust between clients. These include a fairness algorithm and an endgame mode, as well as other mechanisms described in the basic BitTorrent specification referenced above.

a) Fairness Algorithm: The fairness algorithm is an essential part of the BitTorrent protocol that encourages peers to contribute to the swarm, prevents adversarial peers from having an adverse effect on the swarm, and optimizes the performance of the swarm. This algorithm is based on the assumption BitTorrent makes about peers: they care more about the information they receive than the information they distribute, so peers need an incentive to ensure they interact fairly.

The basic algorithm uses tit-for-tat choking based on tracking the upload/download ratio of peers. Peers are classified into two groups in this algorithm: seeders and leechers. Seeders have all the data they have marked for download, while leechers are peers that are actively downloading from the network. Seeders keep a certain number of unchoked peers available to upload to, and opt to keep peers with the highest transfer rate. Leechers focus on uploading to the peers that it can download from at the fastest rate. All peers look for new connections by randomly unchoking another peer at a set interval, and peers choke peers based on their upload/download ratio.

This algorithm optimizes performance when seeking to transfer an entire archive, as leechers follow a policy that allows them to become seeders, and seeders greedily seek to push new data in the network. Unfortunately, peers can get snubbed: if a leecher only has data that its connected peers already have, it looks like an adversarial peer and can get choked by all other nodes. BitTorrent performs a periodic anti-snubbing check to attempt to find new peers in this case.

b) Peer Distribution Selection: In addition, BitTorrent assumes a large number of drop-in/drop-out peers and that most of the peers want the full data set. Because of this, seeders have distribution preferences that focus on sending full copies of the data in a distributed manner to the swarm. To get this effect, seeders prefer handling requests of files they haven't sent out recently and from nodes they haven't sent to recently.

c) Endgame Mode: Since pieces are downloaded individually from different peers, a peer can end up getting stuck downloading a piece from a slow peer. If this piece is one of the last to be downloaded, it can extend the time of download by however long it takes to download that piece. The endgame mode feature addresses this by downloading the final pieces from multiple peers. This assumes, however, that BitTorrent clients know the full set of files they want to download a priori.

B. Dynamic Shared Libraries

Application programmers' demand for shared library support on large scale HPC platforms has greatly increased. These techniques were originally developed to increase resource efficiency for multi-user timeshared desktop and enterprise level platforms, however. As a result, standard implementations contain obstacles to scalability, especially on large clusters and HPC platforms.

Historically, large-scale HPC systems have used statically linked executables, where all references are resolved at compile time and all executable code is contained in the resulting binary. Such binaries can be efficiently distributed to tens of thousands of nodes in

several minutes, typically using a hierarchical distribution over the network. While distributing a single object at launch has its advantages, the increasing use of large solver libraries can result in executables exceeding 1GB in size. Since code paths are decided based on the specific problem or input, large portions of unused code must be statically built into executables which can bloat executables with unnecessary code.

Dynamically linked binaries, in contrast, defer most of the linking process until run-time. When a binary begins executing, the dynamic linker attempts to resolve all unresolved references, causing a large number of file stat operations. While these operations can be efficiently accomplished on a single system, on a large HPC platform performance declines with the number of concurrent processes. This can cause tens of thousands of simultaneous file operations for the same set of shared libraries on a shared filesystem to resolve references. These operations are latency-sensitive; handling them quickly is essential. Even small delays can cause timeouts which result in duplicate requests, further increasing load on the file server. All references must be resolved before the program can begin execution.

Once the program is executing, if any of the shared object code resolved during the initial run-time linking process is required, the dynamic loader is employed to service the request. Typically, this is done using the OS filesystem interface. Requested objects which were memory mapped during the linking phase are demand paged. Again, this process is efficient for single systems. On HPC systems, bulk synchronous executions make requests roughly at the same time. Because of this, it is possible that tens of thousands of nodes will make simultaneous requests to a single shared object. All of the processes will request a page (or pages) of the shared object at roughly the same time from the single shared library on the shared filesystem. The bandwidth and meta-data coordination of this data distribution poses the second key challenge.

Possible solutions to these key challenges are unfortunately quite disparate. Parallel filesystems are typically optimized for delivering large amounts of bandwidth. While this seems encouraging on the surface, parallel filesystem performance for shared files (N to 1) is typically much worse than can be achieved in an N to N organization. File operations like directory searches, file stats, and opens often expose bottlenecks in parallel filesystem meta-data services.

As a result, modern HPC systems, for example the Cray XT/XE/XK line of supercomputers, generally use a hierarchy of file caches for system shared libraries. The recent **A**lliance for **C**omputing at **E**xtreme **S**cale (ACES)[1] capability platform, Cielo, uses this strategy for user-provided shared libraries. This approach is more efficient than any other available file system on Cielo [9]. However, even when using this more efficient hierarchy of caches, the application runtime can still increase substantially when compared with a statically linked binary [2].

III. Architecture

To evaluate the ability of BitTorrent to meet the challenges dynamic shared libraries present, we designed BitLib. This system includes four main features to support dynamically linked and demand loaded libraries in HPC systems:

1) Data distribution to nodes that can handle moving large shared libraries to tens or hundreds of thousands of nodes without bottle-necking a single or small number of file system nodes;
2) Incidental loading of libraries, such as those loaded by a `dlopen` call but were not linked against the executable;
3) Meta-data management that can handle repeated directory queries for searching the file system for appropriate shared libraries (e.g. `LD_LIBRARY_PATH` searches); and
4) Straightforward integration with the compute-node operating systems so that the resulting system can be deployed to and maintained for production systems with minimal extra effort.

In the remainder of this section, we describe the overall architecture of BitLib and our approach to addressing the challenges above.

A. Overall Architecture

In the general approach, shown in Figure 1, user actions take place when specifying a job to run, when the job is launched, and when requests are made at runtime for the requested files. When constructing a job to submit, users declare a set of files, for example a directory of shared object files, to make available to the application. Our system creates a small *description file* that contains information needed about the files to make available. Because clients selectively download libraries, this description file can be set-up to be all-inclusive of

[1]ACES is a collaboration between Sandia National Laboratories and Los Alamos Laboratory to design and field capability class platforms for the DOE NNSA Tri-lab community

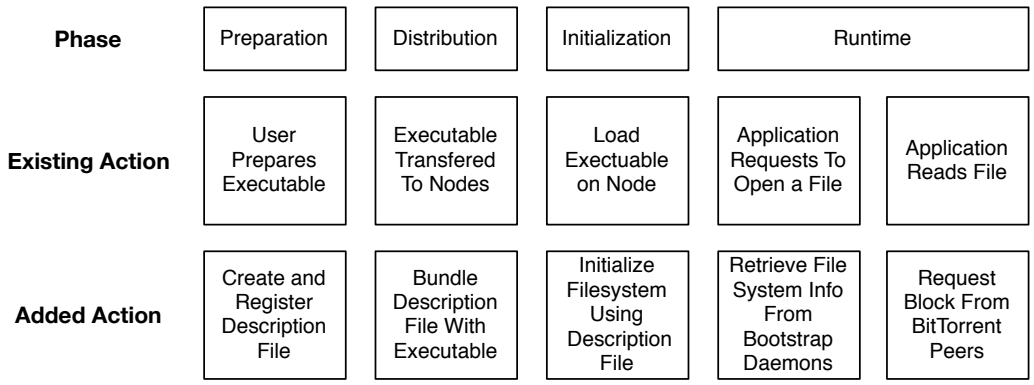


Fig. 1. Steps in System Operation

the available libraries allowing reuse of the description file, even between applications.

When a job is launched, the description file is distributed to each compute node as part of the launch process. This file specifies what local directory on the compute node the files should be mounted on, as well as all of the information necessary to handle file lookups at runtime. This information is used to begin the transfer of data to compute nodes when data is requested, and to serve, in a peer-to-peer fashion, requests from other compute nodes. In addition, *bootstrap daemons* on system service or I/O nodes also process the description file so that they can handle requests from compute nodes to bootstrap data movement onto the compute nodes for peer-to-peer distribution.

As the job runs, metadata and data requests (e.g. resulting from a call to `dlopen` library function) are handled by communicating with other compute nodes and, when necessary, with bootstrap daemons on service nodes. We describe the details of our approach to handling such requests in the remainder of this section.

B. Data Movement

BitLib supports three different strategies for downloading shared libraries: bulk download, per-file download, and prefetch mode. Bulk download mode always downloads the the entire torrent. This is the use-case for which BitTorrent is optimized, but can download files from the torrent that are not needed, which wastes memory and network bandwidth. Per-file download requests individual files from the torrent swarm as the application requests them, allowing on-demand downloads from a large number of libraries. Finally, BitLib also supports a hybrid mode that prefetches a pre-defined subset of torrent files, generally those dynamically linked with the executable, while supporting per-file download of additional files such as those opened using `dlopen`. This hybrid mode is the default BitLib mode.

We use the Enhanced ctorrent [8] BitTorrent client as the back-end for our data movement. This client has been tested over many years, is relatively stable, and has a command line interface which we utilize to control it programatically. We use Opentracker [5] as the BitTorrent tracker in this project, because it is one of the few BitTorrent trackers that runs from the command line. Both of these components are open source and were chosen for their small set of dependencies.

We use Enhanced ctorrent directly for our initial seeders. The initial seeders are given the full set of data, so they can service requests from the clients. The BitLib client in our system use Enhanced ctorrent programmatically, controlling the instance with pipes. Opentracker is run on a service node and serves as the coordinator that all the clients and initial seeders connect to.

C. Meta-data Management

Metadata management in BitLib is currently straightforward. When BitLib starts up, it parses the torrent description file for file names and sizes. The necessary meta-data this leaves out is permissions. Because permissions are strictly read only, and the user has to have read access to them to run the system, we use a default set of permissions for all of the files.

D. Bootstrapping

Our system bootstraps before the target process runs by running via a script that sets up BitLib. The steps to bootstrap BitLib are:

1) Launch Opentracker on a service node
2) Create a torrent file that points to this tracker

3) Push the library folder to a service node and run ctorrent to seed from that node
4) On each node, compare the linked libraries to the torrent to generate a bulk fetch list and start BitLib

The way we currently have implemented this bootstrapping process is through run scripts specific to running pynamic on Cielo's environment. However, there are a number of ways this can be improved; creating a global list of supported libraries with a single instance Opentracker would allow an admin to preform the first three steps once, for all users. This would also result in the sharing of libraries between jobs, if they both used the same library. To allow for users to add additional files, this process can be easily modified to crete two BitLib directories on a node, each serving different files.

E. OS Integration

To integrate BitLib into the host operating system, we use the FUSE (Filesystem in USErspace) filesystem framework. FUSE provides an interface by which developers can override filesystem functions to provide different types of functionality. By implementing a FUSE module, BitLib can provide peer-to-peer file data transport while still remaining transparent to the application programmers.

To illustrate our FUSE integration, we describe the BitLib data path at the file granularity. Initially the application makes a request for a file, such as in a `dlopen` call. This request is routed through the kernel to the FUSE module. The FUSE module, upon receiving a request to open a file, requests the file from the BitTorrent library. The BitTorrent library changes the priority of these files from undesired to normal or high. The BitTorrent library then starts actively looking for the file from the BitTorrent swarm, saving the file into a RAM disk. As pieces of the file are downloaded, the BitTorrent library will advertise and upload these pieces to other nodes. After the file is downloaded, FUSE will provides an interface for the application to access the file on the RAM disk.

IV. Evaluation

In this section, we present an evaluation of BitLib with different configurations and on different systems. We also present results from an initial scaling study of BitLib on the Cielo system.

A. Experimental Setup

We primarily evaluated BitLib on two different systems, Sandia's Teller and Muzia testbed systems. Teller is a 104 node cluster with the AMD Trinity Fusion processors, 16 GB of RAM per node, and QDR Infiniband and Gigabit Ethernet networks. Muzia is a 20 node Cray XE6 testbed connected via Cray's Gemini interconnect. Both machines run versions of the Linux operating system, the Tri-labs OS in the case of Teller, and the Cray Linux Environment in the case of Muzia.

In addition, we also had limited access to up to 1024 nodes of the ASC Cielo supercomputer for an initial scaling study. Cielo is a 8,944 node Cray XE6 system, again connected via Cray's Gemini interconnect. Like Muzia, it runs the Cray Linux Environment. It uses a 160 GB/sec Lustre filesystem for back end file storage.

We evaluated BitLib performance on these systems using the Pynamic distributed shared library benchmark [11]. Pynamic generates a number of shared libraries, and a python MPI program which dynamically links and imports each of the shared libraries, visits each of the generated functions, and then does a small computation phase where it does some simple MPI command to generate a small fractal image. We used the default Pynamic set-up with 495 shared libraries, 215 of which are utility libraries, each with 1850 functions, with each function name being 100 characters long. This amounts to about 1.4GB of shared libraries. The utility libraries are C libraries without python hooks, and they are linked to the executable but they are not imported within the benchmark. It is important to note that a significant fraction of Pynamic's runtime is spent to importing and visiting the modules. For instance, on Muzia, it takes roughly 190 seconds after the libraries have been loaded into the RAM disk for Pynamic to complete.

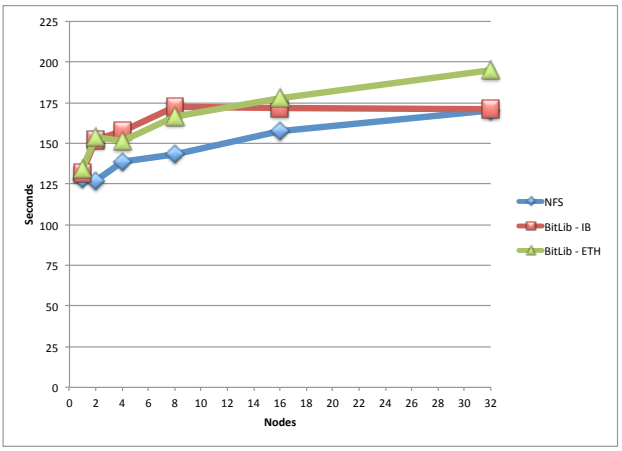
Unless otherwise stated, each datapoint is the average of three runs, and BitLib was run using the hybrid prefetching mode of file transfer. For Pynamic, this results in all of libraries being pre-fetched, rather than being transferred on demand.

B. Comparison Against Default Filesystems

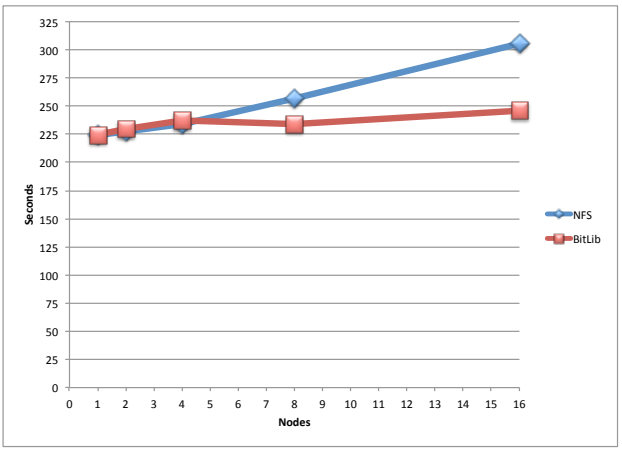
We began by examining the performance of BitLib at different node counts on the Teller and Muzia clusters. Figure 2 presents the results of these two clusters compared against their default NFS filesystem performance. On Teller, BitLib performs approximately the same as NFS while running over Ethernet and Infiniband. On Muzia, BitLib's approach outperforms NFS over Gemini.

C. Impact of File Transfer Modes

We then examined the impact of file transfer mode on BitLib performance, specifically whole archive, per-



(a) Teller



(b) Muzia

Fig. 2. BitLib Scaling Performance on Different Clusters

file, and prefetching downloads of shared libraries. As described in Section III-B, BitLib can communicate in three different modes, with a hybrid prefetching mode as the default. Figure 3 shows the results of changing the file download strategy when running over Infiniband on a single node of the Teller testbed. Bulk download generally performs best in these single-node tests, with the prefetching strategy performing competitively. In contrast, the per-file downloading strategy works very poorly. This is because sequences of independent file requests violates BitTorrent's assumed use case, causing a severe performance hit. Section V provides further discussion of the BitTorrent features that caused this downloading scheme to perform poorly.

D. Impact of Network Architecture

We then examine the performance of BitLib on multiple network fabrics, as shown in Figure 4. The Infiniband and Ethernet numbers are from on the Teller cluster

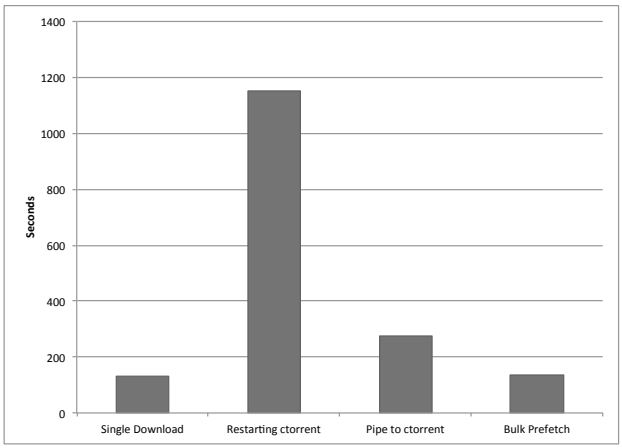


Fig. 3. Impact of file download strategy on BitLib Performance

and the Gemini numbers are from the Muzia cluster. As these are different systems, we attempt to isolate the effect of the BitTorrent traffic by examining only the benchmark slowdown compared to the single-node case on that system.

In this comparison, we can see that BitLib slows down linearly on Ethernet, quickly becoming limited by network bandwidth. In contrast, it performs well on Infiniband and Gemini interconnects, both of which have greater bandwidth and a more scalable topology compared to the 1Gb Ethernet network.

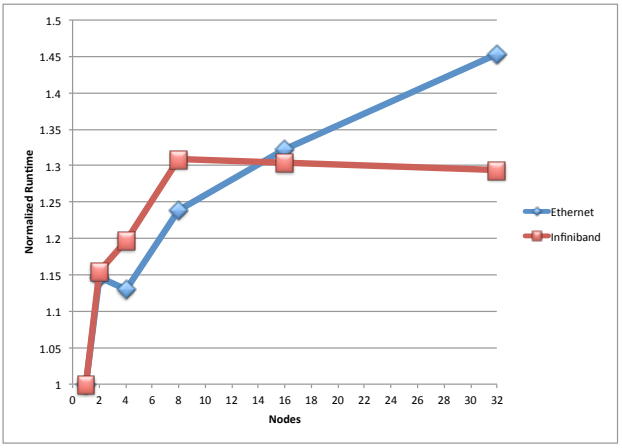
E. Scaling Study

Finally, we were also able to perform an initial evaluation of the scalability of BitLib on up to 1024 nodes of the ASC Cielo supercomputing systems. Because of time and access limitations, we were only able to run a single test at each system size, however, so these results are only an initial study.

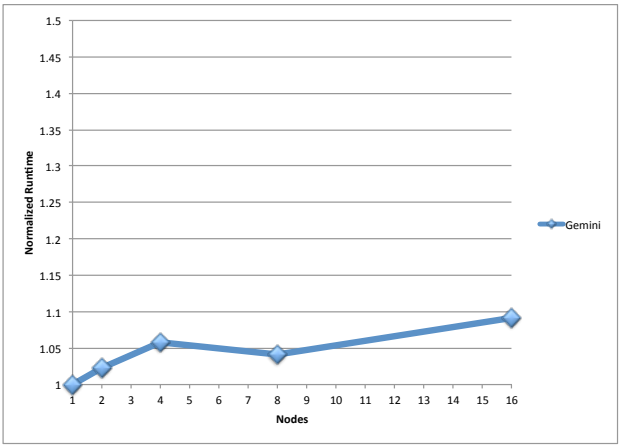
Figure 5 presents the results of tests of Cielo Pynamic performance running both BitLib and the standard Lustre file system. In this comparison, we can see that, for a small number of nodes, BitLib performs about as well as Lustre, paying a little bit of overhead for the extra communication. In the 256 node case, BitLib performance is poor, likely due to an outlier measurement. At 512 nodes and beyond, Lustre performance slows more rapidly, while BitLib continues to scale well, resulting in BitLib outperforming Lustre in the 512 and 1024 experiments.

V. Using BitTorrent in HPC

As described in Section II, BitTorrent's expected use-case, single, bulk transfers of large set of files over



(a) Normalized Slowdown on Teller



(b) Normalized Slowdown on Muzia

Fig. 4. Normalized Slowdown

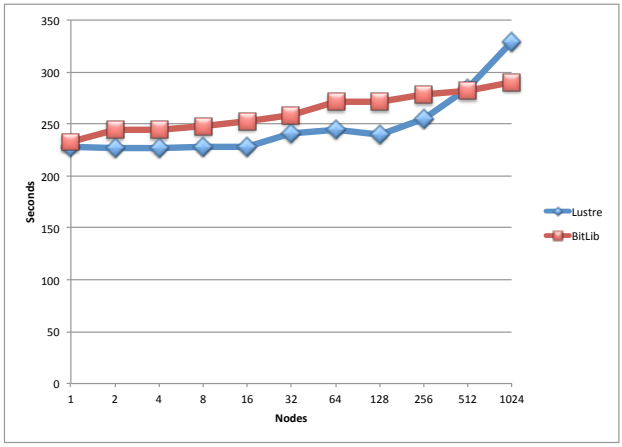


Fig. 5. Initial Performance Results on Cielo

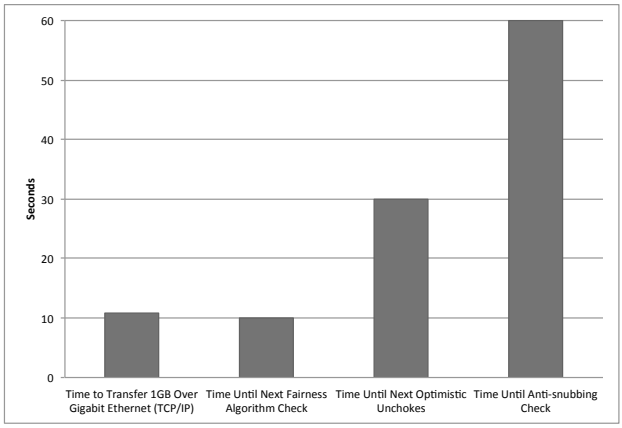


Fig. 6. Granularity of BitTorrent's Fairness Algorithm

slow networks, is different from that of a file system supporting demand-loading of shared libraries. BitTorrent includes multiple mechanisms, namely a fairness algorithm, peer distribution selection scheme, and an endgame mode, that all depend on these assumptions. As was shown in Section IV-C, these mechanisms can have significant negative impact on BitLib performance. In this section, we analyze the impact of these mechanisms, and describe how we mitigated them in BitLib.

A. *Protocol Mechanism Challenges*

As shown in Figure 6, the rate of optimistic unchoking and anti-snubbing checks in BitTorrent are designed for low bandwidth, high latency networks. In HPC systems, however, greater bandwidths can transfer 1 gigabyte files in roughly 10 seconds over gigabit Ethernet, while optimistic unchokes are only done once every 30 seconds and anti-snubbing checks are done after 60 seconds. As a result, better connections will frequently not be found until after the download has completed and a snubbed peer will not react until almost a minute after their download stops.

In addition, peer distribution selection mechanisms can adversely impact system performance in cases where clients all ask for the same data from a small number of seeders. In particular, the distribution scheme will only give the data to the first requester, serializing the distribution of critical data. This case can occur in per-file mode in BitLib, where all clients are asking for the same piece of a small shared library at startup.

Finally, endgame mode also impacts BitLib performance, as it is invoked every time BitLib nears completion of a file download. In addition to downloading data multiple times unnecessarily, it also reduces the upload/download ratio of the client. Again, this is particularly problematic in BitLib per-file mode.

B. Combined Mechanism Impact

The combined impact of these mechanisms was a catastrophic performance failure in per-file mode in BitLib. In particular, while the system worked well for one client and a single initial seeder, it performed poorly with the addition of more clients in per-file mode. This is because, in per-file mode, all clients begin by requesting pieces from the same shared library, as opposed to different pieces of a large archive. When this happens, the initial seeder receives two requests for the same file, services the first one, and disregards the second one.

This process continues until the first client's fairness algorithms chokes the second client because of its poor download ratio. At this point, the second client is choked by both the initial seeder and the first client. The first client then completes in a normal time, and the second node is starved until it receives an optimistic unchoke. This problem prevails even as additional clients are added; one node always gets ahead and is preferred by the seeder.

C. Mitigation

We examined multiple ways of dealing with this problem. Trying to coalesce open requests, our initial approach, failed because when the dynamic loader was opening the files, a read of the file header always followed. Hence, we needed to download the file before moving on to the next open. Adding extra files to increase the download size (e.g. in bulk download mode) improved performance, but wasted time downloading potentially unnecessary files. In the end, hybrid prefetching mode was an effective pragmatic compromise. By prefetching necessary data at program startup, all clients received good download ratios while still downloading useful data. As a result, they were not unnecessarily choked later when they began requesting individual files on demand in response to `dlopen` calls.

VI. Related Work

Research related to this effort comes primarily from two areas; HPC, the target of this research, and distributed systems. Sandia National Laboratories conducted some earlier experiments specifically directed towards supporting DSLs on HPC platforms that utilized light-weight kernels and custom run-time systems. While this experiment was generally successful, it required extensive changes to the standard loader and run-time and was not portable to other commodity software stacks. This effort, and a more general survey of other potential approaches, was documented in a Sandia technical report [10].

The Blue Gene/P system uses an IO forwarding approach to address scalable IO (Blue Gene L supported only statically compiled binaries) [19], . though the authors of this paper focus on IO performance and bottlenecks in general and do not specifically address DSLs. Cray Inc. has also implemented DSL support using ta filesystem proxy, the Data Virtualization Service (DVS) [18]. Coverage of Cray's approach and an optimization that extended the scalability of this approach (a DVS optimization) can be found in [9]. Other researchers have investigated optimizing IO on HPC platforms using proxy methods and data compression ([20], [13], and [1]). These techniques can improve the scalability of using DSLs but have not focused on this issue.

Magic Ermine [12], a tool developed to aid in binary portability, was also investigated in [10] as a possible hybrid method of combining both the executable and the required DSLs into a single package which could then be efficiently distributed on the target HPC platform. While interesting, this approach would generate very large combined executables along with other limitations.

In the distributed systems area there are a number of related research efforts. Chord [17], CAN [14], Pastry [15], and Tapestry [21] are peer-to-peer Distributed Hash Table systems introduced in 2001. All share the same fundamental idea but have significantly different approaches. The original protocol for BitTorrent was also designed in 2001 but specifically targeted peer-to-peer file sharing over the Internet.

XGet [7] is a research effort for file transmission that used the 9P protocol. When evaluating their software, they compared it with BitTorrent. While BitTorrent performed faster than their software, there were notable limitations, namely, a cumbersome user interface and the use of a metadata file. We address the first limitation by creating a BitTorrent client that, instead of using an explicit user interface, takes input in the form of filesystem calls. We address the second limitation by distributing the metadata with the executable, a solution specific to our problem.

General HPC file system approaches, Lustre, GPFS, PVFS, or Panasas, use parallel techniques to achieve good performance in aggregate, but are not optimized for handling shared libraries. These packages are tuned for high volume, bursty I/O, such as writing checkpoint or reading restart files. The serial I/O to a single file on these parallel systems is not much better than a single

disk file system. In fact, most shared libraries are stored on cluster-local disk or on network attached storage (NAS) and accessed via NFS. As mentioned previously, I/O forwarding techniques from NFS servers have been developed with some, but insufficient success.

More recent research efforts have also made considerable progress on these access issues. For example, Zhao, et al. have done work supporting DSLs with their FMcache and DLcache [22] projects. Their approach is to create a cache file using a small scale run of the target program and push the cache to the nodes at launch time. While their approach has scalability and high performance, it makes the assumption that the program can be run at small scale and will require the same libraries at small scale as it will at large scale. The other caveat of their approach is that changing the DSLs used by a program requires additional small scale runs to update the cache.

Spindle [6] is a project by Frings et al. to use overlay networks to create a tree based data distribution network. Their data distribution model is similar to ours; BitTorrent distributes each piece in a tree like manner, although it is random and changes per piece. This allows our system to adapt to changes in the network. Our approaches also differ as they use a custom overlay network and we use an off-the-shelf BitTorrent implementation.

VII. Conclusions

In this paper, we have discussed the performance and issues we encountered when trying to use an off-the-shelf BitTorrent client in an HPC context. We examined BitTorrent's suitability by trying to apply it to the problem DSL distribution. While this approach has many benefits in scalably and reliably distributing data, there are issues with how it is tuned for its preferred use case.

For future work, we plan to look at and evaluate modifications to a BitTorrent client to increase its applicability to an HPC environment. First, we plan to look adapt a tested, open-source, and high performance BitTorrent client into a library to get quicker and more detailed communication and control. Second, we plan to examine how modest changes and tuning of BitTorrent for HPC networks would impacts its performance on individual file downloads. More generally, we plan to examine the potential benefit of features like the fairness algorithm in an HPC context. Finally, we plan to examine if there features of the HPC networking environment that can be utilized to improve the performance of BitTorrent in that context, such as non-TCP transfers and broadcast communication.

Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000.

This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program.

References

[1] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, *Scalable I/O forwarding framework for high-performance computing systems*, in International Conference on Cluster Computing, IEEE, Sept. 2009.

[2] B. Barrett, R. Barrett, J. Brandt, R. Brightwell, M. Curry, N. Fabian, K. Ferreira, A. Gentile, S. Hemmert, S. Kelly, R. Klundt, J. H. Laros III, V. Leung, M. Levenhagen, G. Lofstead, K. Moreland, R. Oldfield, K. Pedretti, A. Rodrigues, D. Thompson, T. Tucker, L. Ward, J. V. Dyke, C. Vaughan, and K. Wheeler, *Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale*, Technical Report SAND2012-1750, Sandia National Laboratories, March 2012.

[3] B. Cohen, *The bittorrent protocol specification*, 2008.

[4] M. G. Dosanjh, P. G. Bridges, S. M. Kelly, and J. H. Laros III, *A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems*, in Parallel Processing Workshops (ICPPW), 2012 41st International Conference on, IEEE, 2012, pp. 55–61.

[5] D. Engling, *opentracker–an open and free bittorrent tracker*, Web, 2010.

[6] W. Frings, D. H. Ahn, M. P. LeGendre, T. Gamblin, B. R. de Supinski, and F. Wolf, *Massively parallel loading.*, in ICS, 2013, pp. 389–398.

[7] H. N. Greenberg, L. Ionkov, and R. Minnich, *XGet: A Highly Scalable and Efficient File Transfer Tool for Clusters*, in LCI International Conference on High-Performance Clustered Computing, January 2009.

[8] D. Holmes, *Enhanced ctorrent*, htt p://www. rahul. net/dholmes/ctorrent.

[9] S. M. Kelly, R. Klundt, and J. H. Laros III, *Shared Libraries on a Capability Class Computer*, in Cray User Group Annual Technical Conference, May 2011.

[10] J. H. Laros III, S. M. Kelly, M. J. Levenhagen, and K. T. Pedretti, *Investigating Methods of Supporting Dynamically Linked Executables on High Performance Computing Platforms*, Technical Report SAND2009-5515, Sandia National Laboratories, 2009.

[11] G. L. Lee, D. H. Ahn, B. R. de Supinski, J. Gyllenhaal, and P. Miller, *Pynamic: the python dynamic benchmark*, in Proceedings of the IEEE 10th International Symposium on Workload Characterization, Sept. 2007, pp. 101–106.

[12] *Magic Ermine*. http://www.magicermine.com/erk/.

[13] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, *Optimization Techniques at the I/O Forwarding Layer*, in International Conference on Cluster Computing, IEEE, Sept. 2010.

[14] S. RATNASAMY, P. FRANCIS, M. HANDLEY, R. KARP, AND S. SHENKER, *A Scalable Content-Addressable Network*, in Special Interest Group on Data Communication (SIGCOMM), August 2001.

[15] A. ROWSTRON AND P. DRUSCHEL, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, in IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November 2001.

[16] P. SOLTERO, P. BRIDGES, D. ARNOLD, AND M. LANG, *A gossip-based approach to exascale system services*, in Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers, ACM, 2013, p. 3.

[17] I. STOICA, R. MORRIS, D. KARGER, M. F. KAASHOEK, AND H. BALAKRISHNAN, *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, in Special Interest Group on Data Communication (SIGCOMM), August 2001.

[18] S. SUGIYAMA AND D. WALLACE, *Cray DVS: Data Virtualization Service*, in Cray User Group Annual Technical Conference, May 2008.

[19] V. VISHWANATH, M. HERELD, K. ISKRA, D. KIMPE, V. MOROZOV, M. PAPKA, R. ROSS, AND K. YOSHII, *Accelerating I/O Forwarding in IBM Blue Gene/P Systems*, in Internatioinal Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM, Nov. 2010.

[20] B. WELTON, D. KIMPE, J. COPE, C. PATRICK, K. ISKRA, AND R. ROSS, *Improving I/O Forwarding Throughput with Data Compression*, in International Conference on Cluster Computing, IEEE, Sept. 2011.

[21] B. Y. ZHAO, K. J. D., AND A. D. JOSEPH, *Tapestry: a fault-tolerant wide-area application infrastructure*, SIGCOMM Comput. Commun. Rev., 32 (2002).

[22] Z. ZHAO, M. DAVIS, K. ANTYPAS, Y. YAO, R. LEE, AND T. BUTLER, *Shared library performance on Hopper*, in Cray User Group Annual Technical Conference, May 2012.