Fast Scalable File Distribution Over Infiniband*

Dennis Dalessandro
Ohio Supercomputer Center
Suite 310
1 S. Limestone St
Springfield, OH 45502
dennis@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1124 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

One of the first steps in starting a program on a cluster is to get the executable, which generally resides on some network file server. This creates not only contention on the network, but causes unnecessary strain on the network file system as well, which is busy serving other requests at the same time. This approach is certainly not scalable as clusters grow larger. We present a new approach that uses a high speed interconnect, novel network features, and a scalable design. We provide a fast, efficient, and scalable solution to the distribution of executable files on production parallel machines.

Keywords

RDMA, Scalable File Distribution, Infiniband, File system, Cluster

1. Introduction

Often times in the cluster environment of today a program is executed on a cluster by a number of nodes receiving the executable code from a single source. Sun's Network File System (NFS) [14] is a prime candidate for such a task as it is popular, stable, and widely available. As executables grow larger, and the number of nodes increases, the single NFS server struggles to handle the additional load, as it was not designed to handle these types of situations. Not only must NFS send more data to more nodes, it also has to be available to service file requests from other users on the cluster.

Two immediate observations show why using NFS is the wrong approach for executable distribution. First, the parallel processes are cooperating and all need exactly the same set of files at exactly the same time. Having each node request the file from NFS independently causes unnecessary load on the server and much duplicated effort among the clients. Second, high performance clusters generally have some sort of fast interconnect for message passing traffic, such as Myrinet [21], Quadrics [16], or Infiniband [1]. However, installations generally use a fast or gigabit Ethernet network for file system traffic, perhaps because using NFS generally requires using an IP stack that may not be supported on the high performance network or on the file system server.

Our approach to these problems starts by recognizing the collective nature of parallel executable startup. Collective operations appear in many contexts. In message passing, the MPI library [5] offers primitives for functions such as broadcast and reduce that optimize the data transfers when compared to the equivalent collection of point-to-point primitives. In peer-to-peer file

*Support for this project was provided to the Ohio Supercomputer Center through the Department of Energy ASC program.

sharing, collectives are used to save link bandwidth at the source host and fully utilize all network paths among downloading hosts. The parallel startup problem is similar to both of these, but the nature of the interconnection network and the types and sizes of files lead us to a particular solution.

Modern networks offer features that can be used to enhance message passing performance. We investigate the advantages and drawbacks of using Remote Direct Memory Access (RDMA) [2] to perform the file distribution. It allows us to move data between nodes without the involvement of the operating system or host processor, exposing opportunities for overlap during the startup phase.

In Section 2, we have a brief introduction to Infiniband [1] and RDMA [2], followed by a discussion in section 3 of the environment that Fast Startup is intended to take part. In section 4 we focus on the basic layout and the decisions which influenced the engineering of the Fast Startup software. Next in section 5 we turn to experimental results to show that Fast Startup is indeed a better choice. Section 6 looks at related work and we conclude in section 7.

2 Infiniband and RDMA

Infiniband is a technology to interconnect processor and I/O nodes to form a system area network [1]. It is a switched, point-to-point network with high speed data transfer and low latency. RDMA [2] is a technology that allows the transfer of data directly from a process running on a node into the memory of a process running on a remote node. The OS and CPU are not involved in this transfer. This is what makes RDMA so appealing. Infiniband provides both RDMA read and write data transfers.

As one of our target clusters uses 10 gigabit Infiniband, we choose that as the initial implementation platform. Other alternatives include Quadrics [16] and RDMA over Ethernet (iWarp) [2], but those technologies are not yet as inexpensive or prevalent. However, as other interconnects mature, we want to take advantage of them. Thus, we were careful in designing the code to allow for this, as described in Section 4.

3. Resource Management Environment

The collective file distribution problem is constrained and orchestrated by the resource management environment. Production machines use software such as PBS [4], Platform LSF [17], or Slurm [18] to control the assignment of individual hardware elements in a parallel machine to particular tasks. Users submit job scripts that specify requirements, such as number of processors and amount of memory, and a list of shell commands to run to perform the tasks. MPI jobs, in particular, use a job launcher such as mpirun [7] or mpiexec [6] to spawn the executable on each processor and initialize communications among the nodes. There are three phases to starting a parallel executable: signaling a daemon on each node to begin a task, launching the executables themselves, and initializing communications in the context of the new parallel job. The final phase, initializing communications among nodes, has been the subject of much previous work and is discussed briefly in Section 6.

For the first phase, the MPI job launcher program contacts each node in turn to instruct it to fork and execute one task of the parallel process. Using

MPICH's [19] mpirun [7], this involves using rsh or ssh in a loop to contact the inetd on each node to start the tasks. With LAM [15], MPD [10], and mpiexec [6], an existing collection of daemons are used to start up the processes efficiently. In the case of mpiexec, these daemons are the individual resource managers themselves. Regardless, this signaling process is relatively lightweight in terms of resource consumption on the network and scales well.

The second phase generally has each process independently invoke the exec() system call directly on a shared file system, such as NFS [14], to start the executable. As discussed earlier, this activity is a major limitation to scalability. Besides the executable itself, shared libraries associated with the code also must be read into the memory of each node at this time. The job may further specify common input files to be used by all the processes. Distributing all these files collectively on the message passing network is the subject of the next section.

4. Design

The basic design of Fast Startup consists of three components. First, there is a transport specific module that currently handles all the transferring of data from a high level view, such as sending and receiving a file. There is another module which handles helper functions for the transport module, such as initializing the Infiniband network. Lastly, we have the main module that oversees all operation of the application. It handles making connections, coordinating sending and receiving, and everything else related to the parallel distribution of the file.

4.1 API

The API we have chosen to use is VAPI [8] which is the lowest level API that enables us to interface directly with the Infiniband hardware. This is a verbs API, and is the least common denominator on Infiniband systems. We have chosen not to use something like DAPL [3] because it is not very common yet. Other API's such as IPoIB [20] do not expose the necessary function calls to do RDMA directly.

4.2 Topology

One of the most important design considerations is to use a tree-based approach for the distribution of the files. This way we have a O(*log N)* algorithm. Taking a step back we can see this is already a better approach and the efficiency of the distribution will out perform the traditional NFS [14] file distribution, which happens to scale linearly as we show in section 5.

Things can only get better by increasing the performance of the network. One of the aspects we wanted to look at is the effect of arity, or the maximum number of nodes one node in the tree may send to. It was important to make this an option that can be set at runtime. For extremely large numbers of nodes, it may be better to allow nodes to send to more than just two others, so instead of a binary tree requirement we have an n-ary tree flexibility. The biggest advantage of the tree based distribution is that it is easily scalable, which is an absolute requirement for this research.

4.3 Communication Manager

In the future, porting this software to DAPL [3] to make use of a communication manager (CM) could be quite useful. This would allow for a reduction in the connection build up time between nodes that we have to endure at the present time. More on this follows. As of right now there is no accepted standard for a communication manager on Infiniband so we should not rely on any one in particular.

Since we are not using DAPL [3] we use TCP as a means of building up the initial connection between two nodes. This amounts to sending Infiniband connection related information which can be handled more efficiently when using a communication manager. We show the portion of time it takes to complete this later.

4.4 File Transfer

One of the major advantages offered by this work is the utilization of advanced networking features, namely Remote Direct Memory Access (RDMA). There are, however, limitations on the use of this mechanism, and thus we have identified three different schemes to send a file. Those are *mmap all, multi send multi map,* and *multi send one map*. All three use RDMA to move data around. The differences are in how the memory is allocated and if the file can be sent at one time or not.

Presently, Infiniband allows a maximum transfer of 2GB at one time [9]. Clearly we need a way to send larger files since this is meant to be scalable in every aspect. It may also be desired that a limit be set on the maximum size that can be sent at one time to take advantage of pipelining. One other important configuration choice is the *chunk size*. Chunk size refers to the amount of data that can be sent at one time. All of these factors significantly impact performance as we will see later.

Mmap all, as the name implies, involves memory mapping the entire file. On the Root Node the file is opened through the regular OS method and *mmap()* is used to map the file to the send buffer. There are no *memcpy()* calls made. On the receiving node an empty file is created and mapped to its receive buffer. The RDMA that occurs moves the file directly from the local disk of the root node to the local disk of the remote node. Now the node that just received the file can open the new local file and map it in.

Multi-send multi-map, unlike the above, maps only portions of the file and copies the data to a buffer to send. This type of send involves multiple memory maps to be made. This is essential to support large files. It may be the case that the file being transferred is too large to fit into a contiguous area of memory, thus requiring multiple memory maps.

Similarly, *multi-send one-map* sends the file in chunks. The difference is that *multi-send one-map* will map the entire file on both ends like with *mmap all*. This achieves the greatest performance, but the available memory may not always be available to hold the entire file at one time.

In both of the multi send methods the sender will poll its completion queue [9] to verify the send request was properly formatted and wait for an acknowledgment from the remote node that it has received and processed the data. The remote node knows when data has been received by polling its

completion queue. It is an RDMA write with immediate data [9] that makes this possible. We show the performance implications of the three send types in the next section. We support all three send types in the software to ensure that Fast Startup can handle any size file.

5. Results

The results that follow were gathered on OSC's P4 Cluster which includes 112 compute nodes connected with 10 gigabit Infiniband and 1 gigabit Ethernet. Each with two 2.4 GHz Intel P4 Xeon processors and 4GB of RAM. The file sizes used may seem quite large compared with the common executable. This, however, is intentional. It is meant to show the scalability of Fast Startup, and is motivated by the need to send shared libraries and input files. It should be noted that all data was gathered on a 2-ary (binary) tree, with the exception of the last data set which looks at multiple arities.

5.1 Total Time to Distribute

One of the most important measures we need to look at is the overall time to distribute the file to a number of nodes. Having measured the mean time it takes to send a 100MB file, 361 milliseconds, and the mean time it takes to receive a 100MB file, 193 milliseconds (see table 1), we can know how long it will take to distribute amongst a varying number of nodes. Using the send history of runs of various size nodes, from 2 to 100 in increments of 3, we can plug in the mean send and receive times to calculate the total expected distribution time. We do this in software to account for the parallelism of a tree based distribution. We now have the most general case possible of sending to a certain number of nodes. The results are shown in Fig 1.

The reason we do not measure this directly is because it is difficult to get a very accurate result without the notion of a global clock, which is not possible to achieve [22]. The root node is the only node that knows when the distribution started, and the only way for it to know when the distribution is completed is to have every node report when finished. This adds significant overhead and skews the result. The same situation arises in section 5.5.

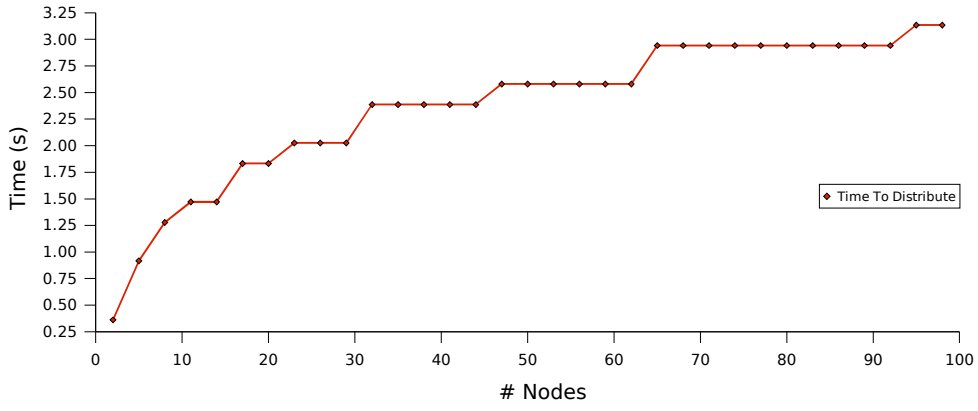

Fig 1. Time to distribute 100MB.

We see some flat areas on the graph, and that illustrates perfectly the advantage gained from a tree based algorithm. The flat areas are due to different numbers of nodes receiving the file in the same amount of sends. Ultimately the cause of this is the ability to have many sends and receives happening in parallel with each other.

It is also good measure to compare this with how long it takes NFS to distribute to a certain number of nodes. Data in Fig 2 was generated on OSC's large mass storage system. This is comprised of multiple dual P4 2.4GHz with SCSI RAID disks and a 1Gbps Ethernet interface. These NFS servers fan out via high performance switches to connect to the compute nodes on the cluster. Due to the fact that NFS is so slow and inefficient we have only tested NFS up to 30 nodes. See Fig 2.

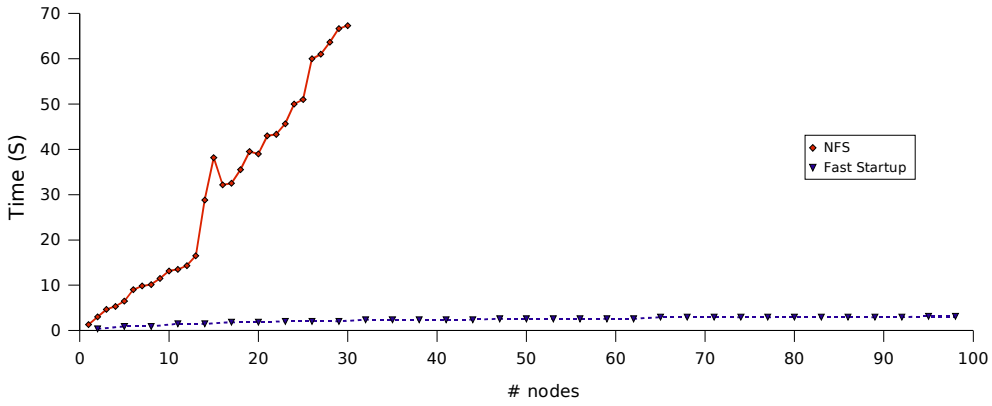

Fig 2. Time to distribute 100MB, NFS compared to Fast Startup.

We can see from Fig. 2 that it takes roughly 1.5- 2 seconds per node. While with Fast Startup and Infiniband it takes roughly 3 seconds to distribute to 100 nodes. Potential reasons for the poor performance of NFS include, lack of server software stack scalability and heavy load on the network. We see 30 nodes take roughly 70 seconds to receive 100MB each. This is an aggregate bandwidth of 43MB/s, well under the potential 125MB/s available in the gigabit Ethernet network. Therefore, we suspect the biggest drain on performance comes from the overheads associated with the NFS protocol and costly send/receive network semantics rather than problems with load on the network.

5.2 Basic Send Performance

We next look at how long Fast Startup takes to send a certain amount of data. Using files ranging from 4KB all the way up to 1GB we show the rate at which Fast Startup can send the file to the remote node, including all overhead incurred by our software. Then the remote node saves the file locally and acknowledges the sender. The data in Fig. 3 represents the mean time to send the various file sizes. The mean was taken over a large number of trials. The standard deviations of these means were always less than 1%. We see that the mean is pretty accurate, and that the trend of the graph is very linear as we

would expect. The slope is linear across the entire range.

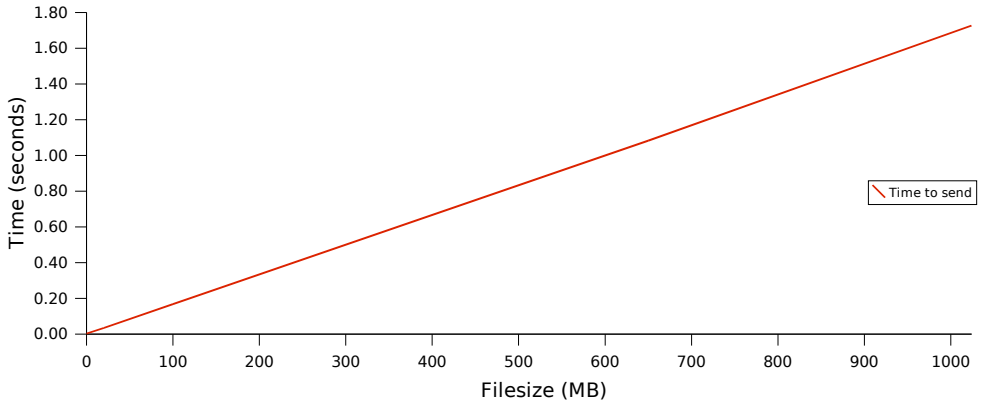

Fig 3. Time to send various size files.

A small 4KB executable took almost no time (0.3 milliseconds), while even the 1GB file takes less than 2 seconds. The mean time to send a 100MB file is 169 milliseconds, which we will look at as a general case through out.

5.3 Chunk Size

Another interesting performance aspect that we wanted to look at is the effect of chunk size. We can see what impact chunk size and the type of send has on the overall performance by looking at Fig. 4. The graph in Fig. 4 depicts the mean time to send a 100MB file. As with the above, this time includes the time to send the file to the remote node, then for the remote node to save the file locally and acknowledge the sender.

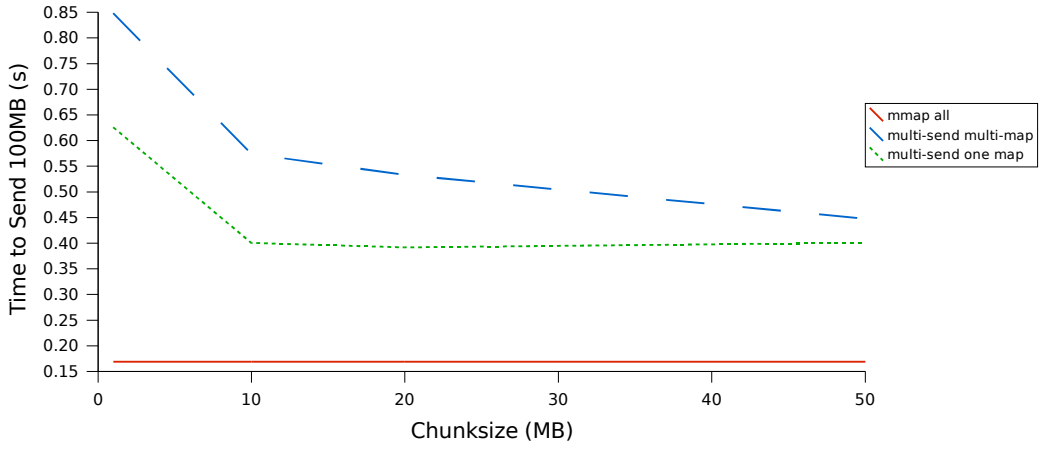

Fig 4. Time to send in chunks.

From the above graph we see that a *mmap_all* is far superior to the other two. This is due to the overhead involved in each RDMA operation. The reason that

the *mmap_all* line is flat is that chunk size has no effect since the file is sent all at once with this type of send. The reason for even having the other two types of sends are for files larger than can be sent in one send through the interconnect.

The *multi-send multi-map* type performs slightly worse than *multi-send one-map*. As we can see the two lines start to get closer and closer together. This is as expected because as we increase the size that we are mapping it is getting closer to one large mapping. Clearly we want to map as much as possible at one time, and minimize the number of mappings that we need to do in order to gain the best performance.

The lesson to be learned here is that if a file can not be sent in one send due to restrictions, either on the interconnect or the restrictions on mapping memory, it is best to make the chunk size as large as possible and use the *multi-send one-map* type of send. Unfortunately mapping memory turns out to be the biggest problem due to the fact that it needs to map contiguous area of memory, so for large files, lots of memory is needed.

5.4 Time Breakdown

For the *mmap_all* case we break down the entire process of sending and receiving a file to see just where the time is spent and how much time is spent in each stage. Over many nodes the average to send a 100MB file is 361.3 milliseconds. This time is different than that shown in Fig. 3 as it is the total send time which includes all overheads. The components that make up this time are illustrated below in Table 1.

Send	***Event***	***Time Spent (ms)***
	Build Connection	178.0
	Wait QP Ready	3.0
	Send Initial RDMA	3.4
	RDMA Executable	171.9
	Send Overhead	5.0
	Total	***361.3***
Receive		
	Receive Overhead 1	4.9
	Receive File	183.1
	Receive Overhead 2	5.1
	Total	***193.1***

Table 1. Time breakdown.

As mentioned before, the time to build the connection is by far the most costly overhead. The time spent on the RDMA transfer, which includes setting up the transfer on the NIC and polling for the results, is the only useful work. The time to build the initial connection includes significant overhead due to the TCP stack, and rivals the time to actually send 100MB of data over Infiniband.

There is a slight delay while the sender waits for the receiver's QP to be ready. This involves the receiver sending a message to the sender, as we see it

takes 3 milliseconds for this to happen. Similarly it takes another 3.4 milliseconds for the sender to send the initial RDMA which includes a list of nodes that still need the file, this includes waiting for the receiver to acknowledge it has received the initial RDMA. Another 5 milliseconds is spent maintaining data structures and tearing down the connections.

Looking at the Receive section of Table 1, we see there is very little time wasted. Clearly receiving the file is more efficient because the TCP socket to transfer the initial Infiniband connection data is already created, and waiting on the sender. This is quite desirable because all nodes must spend time doing the receive, but not all must participate in the sending stage. The overheads involved in receiving the file are small (4.9 and 5.1 milliseconds), and are basically independent of file size. So most of the time is spent doing actual work to receive the file, actually it is passively waiting for data to be written into its buffer, rather than actively receiving the file.

We hope to greatly speed up the connection management using a better interface such as DAPL [3].

5.5 Arity

The remaining aspect we need to look at is arity. Arity as used here means the number of nodes a sender is able to distribute the file to. Fig. 5 shows the effect of arity on distributing a 100MB file to 100 nodes using the *mmap_all* type of send.

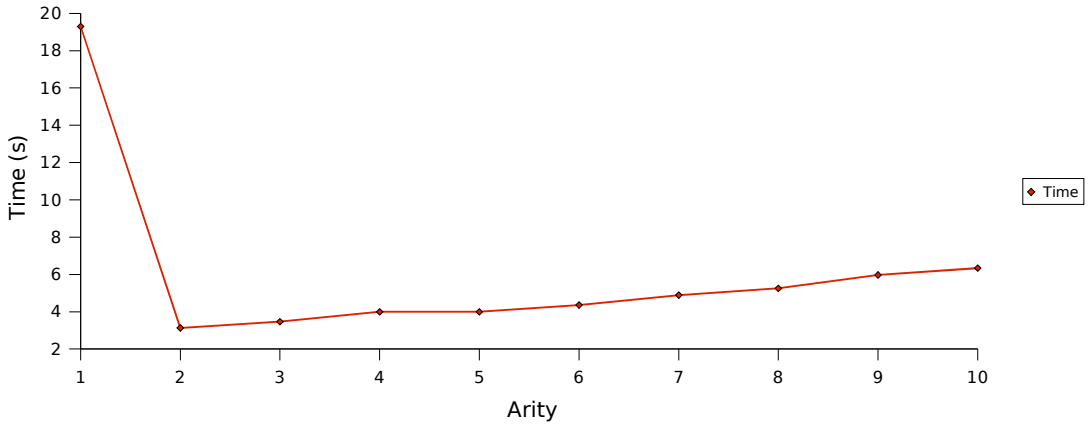

Fig 5. Time to send 100MB to 100 nodes, for various arities.

From the graph we can see that a binary tree is actually the best performing type of distribution. While this may seem surprising at first, after looking at the total send and receive times, we see that the total send time per node is in fact quite a bit more than the total time it takes to receive per node. What we can tell from this is that we want to minimize the number of sends that occur sequentially and maximize the number of sends occurring in parallel.

6. Related Work

According to [11] there are two phases to start a parallel application,

process initiation and connection set up. We propose there is in fact three phases, before a process can be started the executable has to reside locally. Other papers have been presented which focus on the two phases described in [11]. This is where Fast Startup comes in, and is what makes it different. MPD [10], is meant to be a fast way of starting parallel jobs once the file resides locally, and a flexible run time environment. In [11], the concession is made that to eliminate the impact of network communication to file access, all files were duplicated on local disks, further [11] mentions the file system performance could be a big bottleneck for a larger cluster. Efficiently starting the process once the file is there, is a vital issue to be addressed and therefore our Fast Startup work is by no means meant to be a replacement for earlier works, but in fact complimentary.

Two novel systems that do take into account the need to get the file locally are *yod* [12] and *STORM* [13]. They both address the two phases that [10] and [11] focus on, in addition they focus on what we propose as the third phase.

While *Yod* [12] does handle all three phases it is not completely scalable, as it uses an O(*N*) file distribution technique. *STORM* [13], on the other hand, is scalable, and handles all three phases. The drawback with *STORM* [13] is that it requires the use of a hardware multicast mechanism specific to Quadrics [16] while we use only the more general RDMA operations.

7. Conclusions & Future Work

We have presented an efficient scalable method to distribute files among a number of cluster nodes in the context of a parallel process. We have shown there are three potential ways to manipulate buffers during file distribution, and we have looked at the effects of arity and chunk size on the total time to distribute the file. We have also shown that existing methods perform and scale poorly compared to our approach.

Future plans for Fast Startup involve more extensive scaling studies on large cluster environments and using multiple network types. Potential improvements specific to Fast Startup include a study on using multiple trees to distribute the file among the nodes. Involving the specifics of the underlying network topology in the construction of the distribution trees may be necessary for extreme scale heterogeneous parallel machines.

References

1. About Infiniband Trade Association: An InfinibandTechnology Overview. http://www.infinibandta.org/ibta/.

2. RDMA Consortium website http://www.rdmaconsortium.org.

3. DAT Collaborative website http://www.datcollaborative.org/.

4. Open PBS http://www.openpbs.org/.

5. MPI The message passing interface http://www- unix.mcs.anl.gov/mpi/.

6. P. Wyckoff. Mpiexec. http://www.osc.edu/~pw/mpiexec/.

7. Mpirun http://www- unix.mcs.anl.gov/mpi/www/www1/mpirun.html.

8. Mellanox IB-Verbs API (VAPI) *Mellanox Software Programmers Interface for Infiniband Verbs*, 2001.

9. Infiniband Architecture Specification vol. 1 rel. 1.1, 6 Nov 2002.

10. R. Butler, W. Gropp, and E. Lusk. Components and Interfaces of a Process Management System for Parallel Programs. *Parallel Computing* , 27(11):1417 1429, 2001.

11. W. Yu, J. Wu, D.K. Panda. Scalable Startup of Parallel Programs over Infiniband. Technical Report OSU-CISRC-5/04- TR33, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH 43210, 2004.

12. R. Brightwell, L.A. Fisk. Scalable Parallel Application Launch on Cplant. Proceedings of SC2001, Denver, Colorado, November 10 16, 2001. Available from http://www.sc2001.org/papers/pap.pap263.pdf.

13. E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, S. Coll. STORM: Lightning-Fast Resource Management. In *Proceedingsof the IEEE/ACM SC2002 Conference, 2002.*

14. B. Callaghan, B. Pawlowski, P. Staubach. NFS Version 3 Protocol Specification. June 1995 available via http://www.cse.ohio- state.edu/cgi-bin/rfc/rfc1813.html.

15. LAM/MPI Parallel Computing http://www.lam- mpi.org/.

16. Quadrics corporate website http://doc.quadrics.com/quadrics/QuadricsHome.nsf/DisplayPages/Homepage.

17. PLATFORM LSF Intelligent, policy- driven batch application workload processing — for desktops, servers and mainframes. http://www.platform.com/products/LSF/.

18. SLURM: A Highly Scalable Resour ce Manager http://www.llnl.gov/linux/slurm/.

19. MPICH A Portable Implementation of MPI http://www-unix.mcs.anl.gov/mpi/mpich/ .

20. H.K. Jerry Chu, V. Kashyap. Transmission of IP over InfiniBand. August 2004 available via http://www.ietf.org/internet- drafts/draft- ietf- ipoib- ip-over- infiniband- 07.txt.

21. Myrinet Home Page http://www.myri.com/.

22. M. Singhal, N Shivaratri, *Advanced Concepts in Operating Systems,* Mc Graw Hill 1994. p.98.