Parallel Computing 39 (2013) 167–176

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

LIBI: A framework for bootstrapping extreme scale software systems

J.D. Goehner [a], D.C. Arnold [a,*], D.H. Ahn [b], G.L. Lee [b], B.R. de Supinski [b], M.P. LeGendre [b], B.P. Miller [c], M. Schulz [b]

[a] Computer Science, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001, USA
[b] Lawrence Livermore National Laboratory, P.O. Box 808 L-557, Livermore, CA 94551-0808, USA
[c] Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706-1685, USA

ARTICLE INFO

Article history:
Available online 8 October 2012

Keywords:
Infrastructure bootstrapping
Job launching
System software

ABSTRACT

As the sizes of high-end computing systems continue to grow to massive scales, efficient bootstrapping for distributed software infrastructures is becoming a greater challenge. Distributed software infrastructure bootstrapping is the procedure of instantiating all processes of the distributed system on the appropriate hardware nodes and disseminating to these processes the information that they need to complete the infrastructure's start-up phase. In this paper, we describe the lightweight infrastructure-bootstrapping infrastructure (LIBI), both a bootstrapping API specification and a reference implementation. We describe a classification system for process launching mechanism and then present a performance evaluation of different process launching schemes based on our LIBI prototype.

1. Introduction

In high-performance computing (HPC) environments, system sizes continue to grow dramatically. On the most recent Top 500 list [1], 254 (or 50.8%) of the entries have over 13,000 cores, compared to seven (or 1.4%) just 5 years ago. On the most recent list, five have over 200 K cores, ten others have over 128 K cores, and nineteen more have over 64 K cores. Lawrence Livermore National Laboratory (LLNL) has a 1.6 million core system, Sequoia [2]. Further, exascale systems are projected to have on the order of tens to hundreds of millions of cores within the current decade [3]. Capability class and other very large application instances that utilize large portions of the entire system, as well as HPC system software and tools, must scale to these massive sizes.

All distributed software systems require a bootstrapping phase in which their processes are started and some basic information is exchanged. As Fig. 1 depicts, given an allocation of physical computational nodes, we define distributed software infrastructure bootstrapping as the procedure of instantiating the infrastructure's composite processes on the specified computational nodes and exchanging the information that these processes require to complete their setup and to enter their primary operational phases.[1]

An efficient bootstrapping process can be critical, and inefficiencies in this process can become an impediment for software deployment and utility. For example, a few seconds are generally the upper bound on acceptable delay for interactive operations. At current scales, it can take several minutes to deploy an interactive software tool, when often the tool can perform its key functions much more quickly. Our experiences with our own Stack Trace Analysis Tool (STAT) [4] demonstrated

* Corresponding author.
E-mail addresses: jgoehner@cs.unm.edu (J.D. Goehner), darnold@cs.unm.edu (D.C. Arnold), ahn1@llnl.gov (D.H. Ahn), lee218@llnl.gov (G.L. Lee), bronis@llnl.gov (B.R. de Supinski), legendre1@llnl.gov (M.P. LeGendre), bart@cs.wisc.edu (B.P. Miller), schulzm@llnl.gov (M. Schulz).

[1] Technically, bootstrapping is not complete until the processes act upon exchanged information; this final activity is infrastructure-dependent.

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.09.003

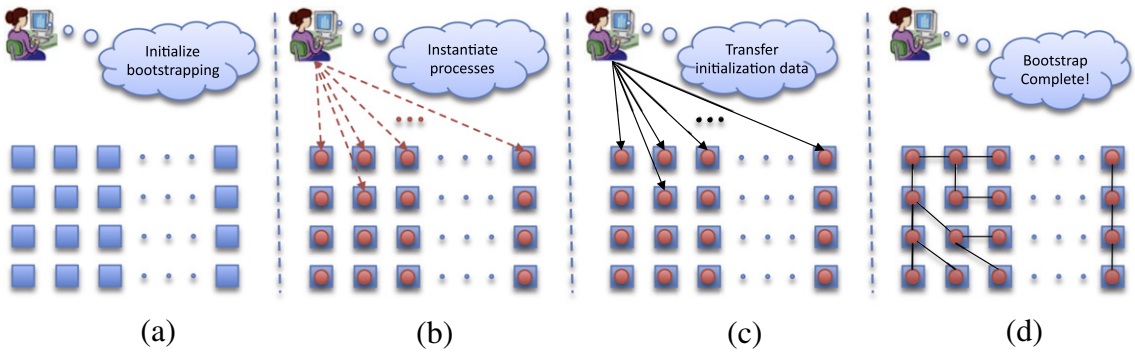

Fig. 1. Distributed application bootstrapping.

this problem: a full-scale instance of STAT on the Lawrence Livermore National Laboratory's BlueGene/L system could take minutes to start-up and subsequently, less than a single second to perform its analysis. Efficient bootstrapping can also be critical in the many-task computational model. In this model, applications are decomposed into many (thousands and sometimes millions of) tasks, and processes are launched continuously on the available computational resources throughout the application's execution. The finer-grained the tasks, the greater the impact of inefficient bootstrapping.

In this work, we design and develop a set of abstractions and mechanisms that address the challenges of scalable software system bootstrapping and deficiencies in current bootstrapping approaches. We present the lightweight infrastructure-bootstrapping infrastructure (LIBI), a reference implementation of our system for launching distributed applications. LIBI is not intended to replace existing resource managers (RMs); LIBI is designed to provide a more intuitive and flexible system bootstrapping interface and mechanisms for portably leveraging RMs. LIBI also provides very efficient and scalable rsh-based process launch for situations in which RMs are unavailable or cannot be used for one reason or another. In this article, we describe LIBI's design and implementation and make the following contributions:

- We offer a system for classifying process launching and bootstrapping mechanisms (Section 3). This classification gives us a way to compare different bootstrapping facilities, particularly with respect to performance and scalability;
- we describe the architecture, design and implementation of our LIBI software prototype (Section 4); and
- we present an evaluation of our LIBI prototype, which demonstrates how LIBI can improve large scale software system bootstrapping (Section 5).

2. Distributed software system bootstrapping

As previously described, software bootstrapping generally entails starting or launching a set of processes and propagating some information to these processes. In this section, we present a background of the existing mechanisms and services available for process launching and start-up information propagation.

2.1. Bootstrap process creation

In the basic approach for distributed infrastructure bootstrapping, a master process sequentially uses a remote process creation mechanism, like `rsh` or `ssh`, to instantiate the other processes. The master process then communicates directly with each to deliver the required initialization information. Fig. 2 shows the time to launch a set of processes sequentially using `rsh` on LLNL's Atlas cluster. This approach does not scale; extrapolation suggests instantiation of 2048 processes would take over 14 s. Additionally, each process requires a small amount of resources, which places a limit on the maximum number of sequentially launched processes.

When available, applications and tools can leverage RMs like LoadLeveler [5], LSF [6], PBS [7] and SLURM [8] for process creation. Typically, an RM uses persistent daemons on each computational node to support efficient process starting. However, RMs do not generally provide facilities to disseminate application data to complete the infrastructure-bootstrapping process. Also, while the myriad of RMs provide similar process starting services, they have different, incompatible interfaces that require different job launching scripts.

For improved portability, many infrastructures implement custom mechanisms to launch their processes. For example, MRNet [9] (which we further describe in Section 4.1.1) uses a hierarchical approach in which the master process starts a small subset of the processes which in turn start other processes, and so on, until all processes of the system are instantiated. This approach is more portable than RM-based approaches and more scalable than basic, centralized approaches. However, such mechanisms still are limited by the mechanism used to start the processes (again, usually `rsh` or `ssh`) and the fan-out of the process starting. Also, other distributed infrastructures cannot leverage these mechanisms since they are embedded into the specific systems.

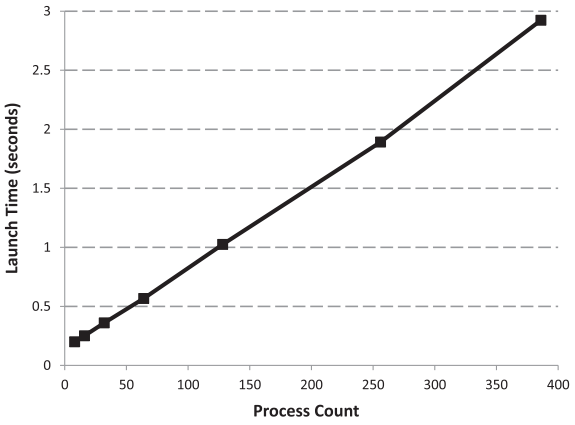

Fig. 2. Sequential process instantiation.

2.2. Bootstrapping inter-process communication

Many libraries and services implement group communication operations. In high-performance computing environments, the MPI standard [10] specifies the most popular of these services. Generally, MPI implementations provide a wide variety of group communication operations, most of which are not needed for bootstrapping. Furthermore, MPI's focus on general application data communication render it a heavyweight solution if it is to be used only for bootstrapping. Lastly, MPI implementations often rely on a distributed set of cooperating daemons that themselves need to be launched and configured. In the remainder of this section, we describe lower-level communication services that directly target the bootstrapping process.

2.2.1. PMI

The communication mechanisms used by the Process Management Interface (PMI) [11] are based on the storage and retrieval of key-value pairs from a key-value space (KVS). PMI defines three operations: Put, Get, and Fence. Put stores a set of key-value pairs in the KVS, Fence synchronizes the KVS of every process, and Get retrieves the values of a given set of keys. The KVS is also used for bootstrapping additional processes. If an additional set of processes needs to connect to the currently running processes the new processes can query the KVS associated with the running processes in order to acquire their connection information.

2.2.2. PMGR

The PMGR collective library takes a different approach to parallel library bootstrap communication [12]. PMGR provides seven group communication operations: `barrier`, `broadcast`, `gather`, `scatter`, `allgather`, `alltoall`, and `allgatherstr`. Other than `allgatherstr`, these operations perform the same functionality that is defined by the MPI operations of a similar name. `Allgatherstr` performs a similar function as `allgather` except the data in question are null terminated strings, whose length can vary from string to string.

Unfortunately, using these operations makes setting up connections between two distinct sets of processes more difficult. With the Put, Fence, and Get operations of PMI, this task is simple. The second set of processes can simply access the KVS of the first. To accomplish this task through PMGR, the first set of processes must explicitly create connection setup data, and then somehow make it available to the new processes.

3. Classification of bulk process launching frameworks

Generally, there are two models for process launching: individual and bulk. The difference between the individual and the bulk launch model is the number of processes that a single request can launch; individual launchers can only launch a single process (for example, `rsh`/`ssh`-based mechanisms), while bulk launchers can launch multiple processes at once. In this section, we offer a classification of bulk launching frameworks.

Generally, bulk launch frameworks are implemented using a system of daemons, background processes that are not under direct user control. When used in a bulk launch framework, daemons communicate with each other to propagate and to execute the requested commands. We classify such bulk launch frameworks according to two dimensions: framework *persistence* and framework *connection topology*.

3.1. Framework persistence

The set of daemon processes that comprise the framework and the set of connections that inter-connect these processes define the organization of a bulk launch framework. The persistence of a bulk launch framework can be mapped to the per-

sistence of these two components between subsequent jobs. Therefore, persistence is a spectrum from completely impersistent (no daemons or connections exist between jobs) to completely persistent (all daemons and connections exist between jobs).

The MPD bootstrapping service [13] is an example of a completely persistent bulk launcher. To service a job launch request, MPD only needs to distribute the process creation command to the relevant daemons using existing connections. Bulk launch frameworks that use persistent daemons and connections are the most responsive since they do not create new daemons or connections at request time. However, they use more resources than necessary: not every process and every connection is needed for every request.

Services like SLURM [8] and ALPS [14] fall in the middle of the persistence spectrum: only daemon processes persist between jobs. These services need two steps to launch a job: (1) create connections between the relevant daemons and (2) distribute the process creation command. Persistent daemons still provide fast bulk launch, but do not maintain unnecessary connections. However, their deployment requires system administrator privilege so the daemons persist. Thus, users cannot easily choose which bulk launcher to use or to configure the bulk launcher. As we describe in latter sections of this article, customizing a bulk launcher (particularly with respect to its inter-connection topology) can significantly impact performance.

Bulk launch frameworks that have no persistent daemons (or connections) between subsequent uses, like ScELA [15], generally allow for the highest degree of user customizability and use no unnecessary resources. However, such services are the least responsive and require three steps to launch a job: (1) create relevant daemons, (2) create connections between the daemons and (3) distribute the process creation command.

3.2. Framework connection topology

A second way in which bulk launch frameworks differ is the connection topology used to launch the processes. In this article, we focus on tree topologies, which are best-suited for process launching.[2] The shape of the tree determines the scalability and efficiency of process launch operations. In general, the scalability and efficiency of a tree is determined by how productive each node in the tree is during a launch or communication operation. Higher node productivity leads to better performance. We discuss four types of connection trees: sequential, chain, k-ary trees, and greedy trees.

The least scalable connection trees occur at the extreme ends of the tree shape spectrum, the broadest trees and the tallest trees. The sequential tree, Fig. 4a, is the broadest possible tree. The root is the parent of and communicates with all other processes. Therefore, launch or communication operations using this tree require $n-1$ steps, where n is the number of nodes in the tree. The chain tree, Fig. 4b, in which each process but the leaf has exactly one child, also requires $n-1$ steps for launch or communication operations. While these topologies do not scale well, they are often used because they are easy to set up and work sufficiently well at small scales.

Many bulk launch frameworks, including ScELA, SLURM and ALPS, use k-ary trees, Fig. 4c, in which each process has at most k children and the height of the tree is minimized. K-ary trees offer good scalability executing operations in at most $\lceil \log_k n \rceil$ steps. Scalability can be adjusted by adjusting k, but improper choice of k can lead to poor performance, and in current practice, this choice generally is made in an ad hoc manner.

In other related work, we studied optimal trees for bulk process launch and designed a greedy algorithm that maximizes the overall productivity of the process launch tree [16]. Our greedy algorithm accounts for environment-specific features and is based on two key parameters, the time it takes to launch a process remotely and the necessary delay between subsequent remote launches by a single process. We proved that our greedy algorithm outputs a tree topology that would launch a set of processes in a minimal amount of time. We call this topology the greedy topology, Fig. 4d.

4. LIBI overview

We have designed and developed the lightweight infrastructure-boostrapping infrastructure (LIBI) to address the various challenges previously described including (1) a myriad of available resource management services with incompatible interfaces and varying levels of scalability, (2) the general unavailability of communication services for software system bootstrapping and (3) scenarios in which resource management services are unavailable or undesirable. LIBI is both a set of abstractions and a set of services for the efficient and scalable bootstrapping of extreme scale software systems. In this section, after describing the primary usage scenarios that motivated LIBI's design, we present the LIBI API and its implementation.

We have two primary goals for LIBI: First, LIBI's abstractions and API should serve as a model for the process launch and communication services needed for general software bootstrapping. Second, LIBI should provide a framework which offers portability and the best available performance to the application that uses it. When used on a platform that provides bulk-launch and/or group-communication services, LIBI should use these when adequate. Otherwise, LIBI should provide a suitable alternative.

[2] Indeed, some services, use non-tree topologies. For example, MPD uses a ring topology, but for the purposes of an individual job launch, the communication topology is effectively a chain, a tree in which every process has at most one child.

Fig. 3 shows how LIBI is intended to be interposed in an HPC system software stack. Software infrastructures can use LIBI services for bootstrapping themselves. LIBI leverages the best resource management and communication services available to deliver an efficient bootstrapping service. LIBI is designed to access these lower-level services via LaunchMON [17] or directly when LaunchMON is unavailable or not suitable for the application at hand. Alternatively, LIBI offers self-contained mechanisms that do not rely on external job launch or communication services. This ensures maximum portability and flexibility.

4.1. Motivating use cases

We have identified three general modes of infrastructure bootstrapping based on the number of different executable files used for launching the processes and whether some processes cannot be created by our infrastructure, for example, because they are already running or must be created by some specific third party service like mpirun. In the basic case, we can create all processes from a single image. This case supports distributed systems based on the single program, multiple data (SPMD) model that do not rely on special environmental features (like those provided by MPI).

In the second case, we can create all processes, but from multiple images. A concrete example of this latter scenario is given below. Currently, we focus on these two cases. In the future, we plan to support the third case in which a subset of the processes are started by a separate service. In this situation, we need "out-of-band" mechanisms for communicating with those processes to give them the information needed to properly join into a single distributed system session.

4.1.1. A concrete bootstrapping example

In this section, we describe a real distributed bootstrapping scenario. Specifically, we describe the MRNet instantiation process. MRNet, the multicast-reduction network, is a prototypical tree-based overlay network (TBON), a network of hierarchically organized processes that leverages the scaling properties of the tree organization to provide scalable data multicast, data gather, and in-network aggregation. The root of the MRNet tree is called the *front-end*; the leaves are called *back-ends*, and the intermediate processes are the *internal processes*. The MRNet start-up process entails instantiating the MRNet internal and back-end processes and propagating information such that each process can establish a connection with its parent in the tree. After MRNet instantiation is complete, MRNet-based applications can use MRNet's communication and aggregation services.

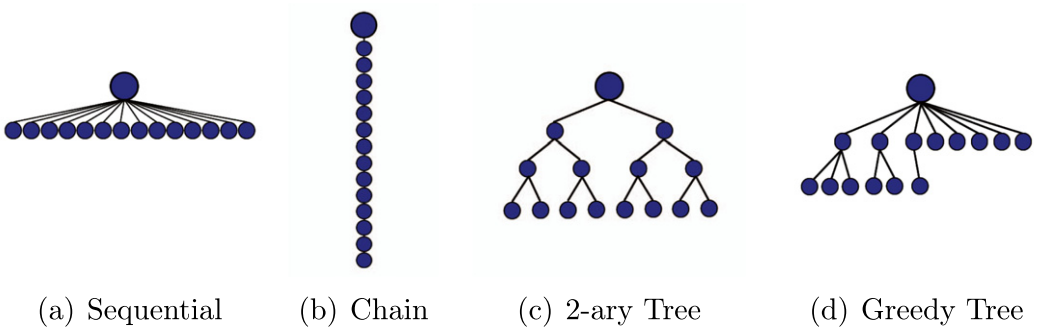

Fig. 3. Connection trees of 15 daemons.

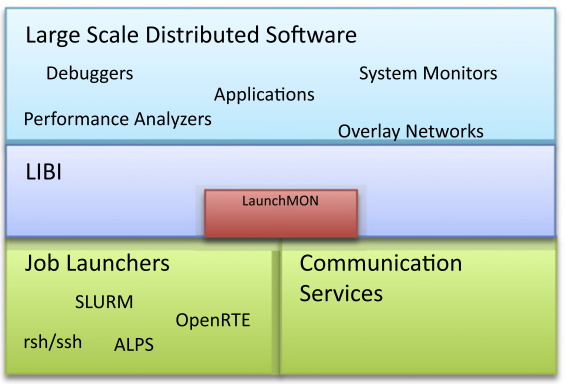

Fig. 4. The LIBI Architecture.

MRNet's primary bootstrap mode implements a *parent-creates-children* scheme in which process instantiation and information dissemination are integrated. The front-end uses a remote shell mechanism, like `rsh` or `ssh`, to create its children processes for the first level of the tree. As command line arguments during this creation process, each parent propagates the information necessary for their children to connect back to it, for example, parent IP address and port number. Each newly created child process establishes a connection back to its parent process and receives the portion of the topology configuration relevant to that child. Each child then uses this information to instantiate its immediate children. This procedure is repeated until the entire tree of communication and application processes is created. As a final step in this start-up procedure, MRNet propagates the complete topology information to all processes (for fault-tolerance purposes).

By concurrently instantiating processes in disjoint branches, the MRNet start-up procedure is much more efficient than the baseline sequential approach. However, this procedure still suffers from serialization bottlenecks due to each parent's responsibility to instantiate its children. For example, with a fan-out of 64, we frequently experience start-up times on the order of tens of seconds for our STAT tool [4,18].

4.2. LIBI API

LIBI provides services for both process instantiation and rudimentary data communication. We present the core LIBI API functions, simplified for presentation.

LIBI's primary abstraction is a *session*, which encapsulates the set of processes to be created. There are several abstractions related to the session: the *session member*, a process within the session, a *session master*, one session member designated to manage the others, and the *session front-end*, the process that launches the session. The session front-end communicates with the session master, who then in turn communicates with the other session members.

Many bulk launchers use a distribution concept to specify how processes should be placed on computational nodes, for example, block or cyclic (aka round-robin) distributions. Accordingly, LIBI uses a *process distribution* to specify how and where to create the requested processes. A process distribution is a 5-tuple, {sid, exe, args, hd, env}, where sid is a session handle, exe is the path to an executable file, args are the arguments to pass to the executable during process creation, hd is a *host distribution*, and env is the environment to use for the created processes. A host distribution is a 2-tuple {hostname, num-procs}, which defines how many processes to create on the named host.

Since LIBI targets only a scalable lightweight bootstrapping service, LIBI only supports mechanisms for communication between the session master and the other session members. In other words, non-masters cannot communicate directly with each other via LIBI. For bootstrapping, the goal is typically to disseminate some configuration information from the session front-end, to the session members. A summary of LIBI's launch and communication abstractions can be found in Table 1.

4.3. A LIBI example

The code snippets in Fig. 5 outline the two parts of a simple software system that we refer to as the LIBI micro benchmark. They show the creation and launch of a process distribution. Once the process distribution has been launched, the front-end waits for the session to complete the group communication operations. Once complete, the session master sends a message to the front-end, signaling that it is done.

4.4. LIBI implementation

This section discusses the two implementations of LIBI. The first implementation integrates with LaunchMON. The second implementation uses an individual launch based approach to launch and connect the requested processes.

4.4.1. Integration with LaunchMON

LIBI is a follow-on project to our earlier LaunchMON work which was intended to provide a more intuitive and flexible system bootstrapping interface and mechanism for leveraging RMs in a portable manner. With LaunchMON, we aimed to provide an abstraction layer to wrap resource management services. LaunchMON's abstractions explicitly targeted HPC tools and were shaped primarily by existing RM mechanisms. These design decisions meant that LaunchMON is not always flexible and ideal for more general use cases. For example, LaunchMON supports the capability to deploy tool processes on the

Table 1
The LIBI launch and communication abstractions.

launch (process-distribution-list) instantiate the appropriate sets of processes according to the input process distributions.
[send—receive] (msg) transfers data between the session master and the session front-end. The session members wait until the data transfer is complete.
broadcast (sendbuf, nbytes) transfers `nbytes` bytes of data from `sendbuf` at the session master to all other session members.
[scatter—gather] (sendbuf, nbytes, receivebuf) transfers `nbytes` bytes of data from/to the session master to/from all session members.
barrier () blocks until all session member calls this routine.

```
front-end() {
  LIBI_fe_init()

  LIBI_fe_createSession(s1);

  proc_dist_req_t pd;
  pd.sessionHandle = s1;
  pd.proc_path = get_ExePath();
  pd.proc_argv = get_ProcArgs();
  pd.hd = get_HostDistribution();

  LIBI_fe_launch(pd);

  LIBI_fe_recvUsrData(s1, times, len);

  return 0;
}
```

```
session_member() {
  LIBI_init();

  msg_sz=sizeof(msg);

  LIBI_barrier();

  LIBI_broadcast(msg, msg_sz);
  LIBI_gather(buf, msg_sz, msg);

  LIBI_scatter(buf, msg_sz, msg);
  LIBI_gather(buf, msg_sz, msg);

  LIBI_sendUsrData(times, len);

  LIBI_finalize();
}
```

Fig. 5. LIBI micro benchmark front-end and session member code.

same nodes of already running MPI applications. The mechanisms to support such capabilities can encumber using LaunchMON in cases where the capabilities are not needed. LIBI is designed to alleviate these issues. (In Section 6, we describe how this work will culminate eventually in a re-factorization of LaunchMON's services. Our first version of LIBI was implemented completely on top of LaunchMON. In this case, the LIBI API was mainly a pass through wrapper for LaunchMON's launch and communication mechanisms.

4.4.2. Individual launch based implementation

For maximum portability and efficiency, LIBI must provide efficient, scalable, lightweight and self-contained launch and communication mechanisms when scalable versions of these are not available on a target system. Therefore, in our second version of LIBI, we added scalable `rsh-based` launching and communication support. In this approach, LIBI launches its processes in three phases. During the first phase, the front-end launches the master process and specifies a connection tree topology – LIBI can be made to use any connection tree topology. During the second phase, LIBI uses `rsh` to launch and connect, one process per node. These processes will be launched and connected in a manner defined by the specified connection tree. During the third phase, the processes created in phase two will be used to launch any collocated processes on their local node. By using only one individual launch per node, LIBI eliminates all of the additional network connections that would have been created for each co-located process. This approach has been used in previous bootstrapping services, with good results [15,19].

During the individual launch, LIBI will pass the current node's name, and the port number used by LIBI, to the new process, as command line arguments. Once the new process is running, it will connect back to the process which launched it. In this way, LIBI's launch topology will also be used as its communication topology.

5. LIBI evaluation

In previous work [20], we demonstrated the performance improvements of an early prototype of LIBI which completely leveraged available resource management services via the LaunchMON framework. In this evaluation, we focus on LIBI's extensible framework that allows one to control the topology LIBI uses to launch jobs and communicate initialization information. A secondary goal is to demonstrate the performance boost that can be attained by replacing an existing bootstrapping mechanism with LIBI. First, we evaluate LIBI by doing some basic process launching and information dissemination. Then we demonstrate the improvements in bootstrapping performance of MRNet using LIBI.

We ran all experiments on Lawrence Livermore National Laboratory's Atlas system, a cluster of 1,152 AMD Opteron nodes. Each node has 8 2.4 GHz CPUs, and the nodes are interconnected via a double data rate InfiniBand network. To increase test scales, we placed up to eight processes on a single node in our testing configurations.

5.1. Micro benchmark experiments

For our micro-benchmark experiments, we use our rsh-based LIBI implementation to launch a test application numerous times and measure process launch time and the time to perform some basic collective operations amongst the processes.

Specifically, we measure (a) the time to start the processes, (b) the time required for all processes to arrive at a barrier, (c) the time to broadcast 128 bytes of data followed by a gather of 128 bytes from each process and (d) the time to scatter 128 bytes of data to each process and gather 128 bytes from each process.

For these experiments, we designed a small, standalone, LIBI-based software system (shown in Fig. 5), comprised of two executables. The first executable uses LIBI to launch the second executable, which is 238 KB in size. These executables were compiled to be statically-linked executables.

All experimental runs were executed on the same allocation of nodes. Primarily, this strategy simplifies the batch scheduling requirements of our managed cluster environment. This also allows every test to execute in roughly the same time frame and on the same resources, minimizing environmental variations, for example due to varied levels of network congestion. Furthermore, this strategy ensures that test runs do not occur concurrently, eliminating possible inter-test interferences and contentions, for example, contending for the same executable files on the same file system.

For these experiments, we deploy a single process per node, the maximum process count is limited by the system's maximum job size limit: 386 nodes. We test chain, sequential, greedy, 2-ary (or binary), 16-ary and 32-ary trees. (Recall these topologies were defined in Section 3.2.) Each scenario was executed ten times and averaged after removing outlying points. Fig. 6a–d shows the results for increasing numbers of processes using different connection trees. These results show that the connection topology of bulk launching frameworks impacts bootstrapping performance. LIBI's framework allows a user to readily customize the topology LIBI uses therefore allowing for optimal choices to be made.

5.2. Macro benchmark experiments

To demonstrate the performance of LIBI in a real context, we replace MRNet's traditional bootstrap mechanism, which uses a process launch topology based on the topology MRNet should use for its runtime operation, with LIBI. We modified

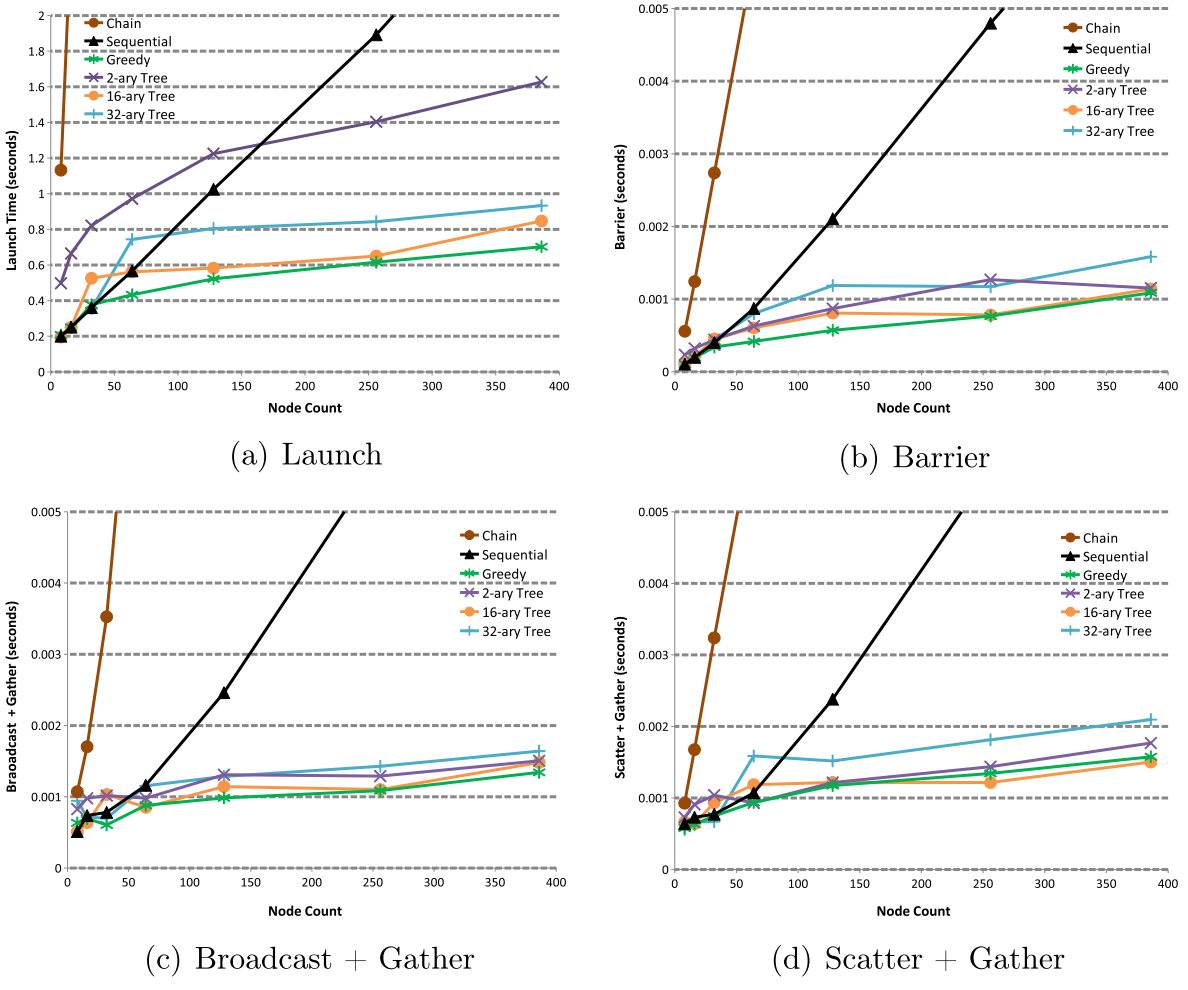

Fig. 6. Micro benchmark results.

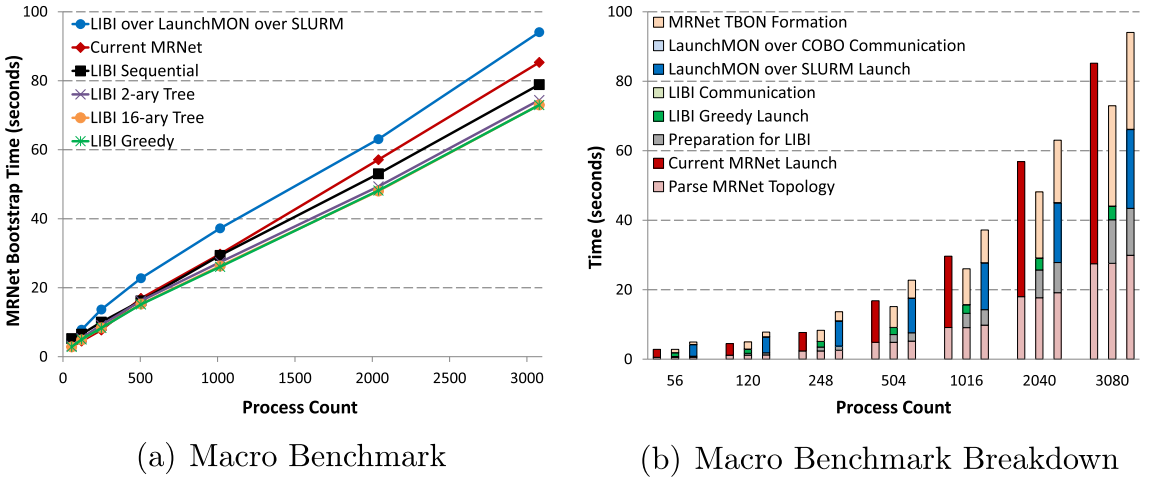

Fig. 7. Macro benchmark bootstrapping results.

MRNet to use LIBI for process creation and for disseminating the topology information needed by child processes to connect to their parent process. Previously, MRNet's start-up process integrated process launch and information dissemination: when a parent created its children, it passed on the command line the necessary port information the children needed to establish a connection with the parent. LIBI completely separates the process launch and information dissemination interactions. In the new LIBI-based MRNet, the session master gathers the relevant information and scatters it to the other session members.

Our macro benchmark experiments used MRNet's current bootstrapping method as well as our rsh-based and srun-based LIBI implementations. We launch an MRNet tree, with a fanout of 16, and measure (1) the total bootstrapping time, (2) the time required to parse the mrnet topology, (3) the time required for LIBI preparation, (4) the time required to launch all processes, (5) the time required to disseminate configuration information and (6) the time required to form MRNet's TBON.

Figs. 7a and b compare these times for increasing numbers of processes using different bootstrapping methods. MRNet over LIBI performs the best, for all connection trees, followed by the current version of MRNet, and LIBI over LaunchMON over SLURM. That being said, all of the bootstrapping conditions appear to scale linearly. The cause of the linear scaling is apparent when viewing the breakdown in Fig. 7b. The largest portion of MRNet's bootstrapping performance is the time to parse the topology file and the TBON formation, which entails the launched process creating TCP socket connections to their peers in the MRNet runtime tree. These operations do not use LIBI. The "LIBI Greedy Launch" component scales less than linearly and the "LIBI Communication" component never took more than an eighth of a second.

Perhaps most surprisingly, "LaunchMON over SLURM" performs much worse than we expected. Especially considering that SLURM uses persistent daemons, we expected that the SLURM launch time would be smaller than the LIBI's best rsh-based launch time. There are several factors that could account for this result. First, the LaunchMON executables are more than double the size of the LIBI executables. This would require additional time to transfer the file from the NFS server. Second, the administrators of the Atlas system have added some plugins to SLURM which perform tasks like detecting if a node has run out of memory. These additional plugins incur extra overheads. Third, LaunchMON is using SLURM for bulk launching with a completely independent communication topology. After the daemons have been launched, the communication topology has to independently be setup. The launch and connection tasks are separate under LaunchMON while they are integrated for the rsh-based LIBI implementation.

6. Conclusion and future work

The goal of this paper was to improve the current state of software-system bootstrapping. Our approach was to create LIBI to facilitate bootstrapping portability for software systems as well as provide the best performance available on a given platform. In this paper, we presented the description and an evaluation of a LIBI prototype. We created a system for classifying bulk-launch strategies based on framework persistence and inter-connection topology. This classification can be used to compare existing bulk-launch strategies as well as inform the design and development of new ones. We also demonstrated the improved bootstrapping performance and portability of MRNet through the use of LIBI.

There are a number of key research and development issues that we plan to address in the future. We conclude this paper by highlighting some of these key issues. A major goal of this project was to use our LaunchMON experiences to shape how distributed system bootstrapping should be interfaced and implemented. As its name suggests, LaunchMON provides facilities for both application/tool launching and application monitoring. One of the lessons learned is that while many tools need both these services, many tools and applications do not. We are in the process of revisiting LaunchMON's design to refactor

and separate these two functionalities. Our plan is to reverse the relationship between LaunchMON and LIBI, such that LaunchMON's launching mechanisms becoming a part of LIBI proper and LaunchMON relies on LIBI for process launching and communication. One current challenge we face in this process is the fact that certain architectures, like BlueGene-based systems, make it difficult to separate launching from monitoring capabilities for the tools that need both.

Currently, LIBI requires that applications input a list of previously allocated nodes in the form of the host distributions described in Section 4.2. We will design more flexible mechanisms that do not require a previous allocation. In this mode, LIBI will efficiently combine the acquisition of the necessary nodes and the instantiation of the specified processes.

We will also explore improving LIBI's launch efficiency with scalable file distribution mechanisms. Filesystem contention is a common problem in high-end computing systems. We have previously developed the Scalable Binary Relocation Service (SBRS) [18] to alleviate this problem. When a common file (like an executable image) is needed by many processes on different nodes, SBRS allows a single process to access the filesystem. The process can then transmit the file to the other processes in a scalable fashion.

Acknowledgments

This work was supported in part by Lawrence Livermore National Security, LLC subcontract B590510. A part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52–07NA27344 (LLLNL-JRNL-575496). It is supported in part by LLNL contracts B579934 and B580360; Department of Energy Grants 93ER25176, 07ER25800, and 08ER25842; AFOSR Grant FA9550-07-1-0210; and NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Top 500 supercomputer sites, visited June 2012. <http://www.top500.org>.
[2] ASC sequoia, visited May 2011. <https://asc.llnl.gov/computing_resources/sequoia>.
[3] P. Kogge, ExaScale computing study: technology challenges in achieving exascale systems, Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008.
[4] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G. Lee, B.P. Miller, M. Schulz, Stack trace analysis for large scale applications, in: 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07), Long Beach, CA, 2008.
[5] IBM Tivoli workload scheduler loadleveler, visited May 2011. <http://www-03.ibm.com/systems/software/loadleveler>.
[6] Platform LSF, visited May 2011. <http://www.platform.com/workload-management/high-performance-computing>.
[7] PBS works, visited May 2011. <http://www.pbsworks.com/ProductPBSWorks.aspx>.
[8] M.A. Jette, M. Grondona, SLURM: simple linux utility for resource management, in: ClusterWorld Conference and Expo, San Jose, CA, 2003.
[9] P.C. Roth, D.C. Arnold, B.P. Miller, MRNet: a software-based multicast/reduction network for scalable tools, in: 2003 ACM/IEEE conference on Supercomputing (SC '03), IEEE Computer Society, Phoenix, AZ, 2003, p. 21.
[10] MPI, message passing interface forum, visited June 2011. <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>.
[11] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, R. Thakur, PMI: a scalable parallel process-management interface for extreme-scale systems, Recent Advances in the Message Passing Interface (2010) 31–41.
[12] PMGR_COLLECTIVE, visited June 2011. <http://sourceforge.net/projects/pmgrcollective>.
[13] R. Butler, W. Gropp, E. Lusk, Components and interfaces of a process management system for parallel programs, Parallel Computing 27 (2001) 1417–1429.
[14] M. Karo, R. Lagerstrom, M. Kohnke, C. Albing, The application level placement scheduler, in: Cray User Group, pp. 1–7.
[15] J.K. Sridhar, M.J. Koop, J.L. Perkins, D.K. Panda, ScELA: scalable and extensible launching architecture for clusters, in: 15th International Conference on High performance Computing, HiPC'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 323–335.
[16] J.D. Goehner, Efficiently bootstrapping extreme scale software systems, Master's thesis, University of New Mexico, Albuquerque, New Mexico, USA, 2011.
[17] D.H. Ahn, D.C. Arnold, B.R. de Supinski, G.L. Lee, B.P. Miller, M. Schulz, Overcoming scalability challenges for tool daemon launching, in: 37th International Conference on Parallel Processing, ICPP '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 578–585.
[18] G.L. Lee, D.H. Ahn, D.C. Arnold, B.R. de Supinski, M. Legendre, B.P. Miller, M. Schulz, B. Liblit, Lessons learned at 208K: towards debugging millions of cores, in: 2008 ACM/IEEE conference on Supercomputing (SC2008), Austin, TX, 2008.
[19] A. Gupta, G. Zheng, L.V. Kalé, A multi-level scalable startup for parallel applications, in: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers – ROSS '11, ACM Press, New York, USA, 2011, pp. 41–48.
[20] J. Goehner, D. Arnold, D. Ahn, G. Lee, B. de Supinski, M. LeGendre, M. Schulz, B. Miller, A framework for bootstrapping extreme scale software systems, in: Workshop on High-performance Infrastructure for Scalable Tools, Tucson, Arizona.