12-1-2011

Efficiently bootstrapping extreme scale software systems

Joshua Goehner

Joshua David Goehner

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dorian Arnold , Chairperson

Patrick Bridges

Lydia Tapia

Dong Ahn

Greg Lee

Efficiently Bootstrapping Extreme Scale Software Systems

by

Joshua David Goehner

B.S., Computer Science, Gonzaga University, 2008
B.A., Psychology, Gonzaga University, 2008

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2011

Acknowledgments

I would like to thank my advisor and committee chair Dorian Arnold, for working with me on the content of this thesis and helping me revise this document. I would like to thank my committee members, Patrick Bridges and Lydia Tapia from UNM and Dong Ahn and Greg Lee from LLNL[1]. I would also like to thank Scott Levy for helping me count to infinity.

[1]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.

Efficiently Bootstrapping Extreme Scale Software Systems

by

Joshua David Goehner

B.S., Computer Science, Gonzaga University, 2008
B.A., Psychology, Gonzaga University, 2008

M.S., Computer Science, University of New Mexico, 2011

Abstract

Many scientific fields and commercial industries need to solve computationally large problems. These problems can often be broken into smaller tasks, which can be executed in parallel, on large computer systems. In pursuit of solving ever larger problems in a more timely manner, the number of nodes in these large computer systems have grown to extremely large scales. All of the extreme-scale-software systems that run on these extreme-scale-computational systems go through what we call a bootstrapping phase. During this phase, the software system is deployed onto a set of computers and its initialization information is disseminated.

In this thesis, we present a framework called the Lightweight Infrastructure-Bootstrapping Infrastructure (LIBI) to support extreme-scale-software systems during their bootstrapping phase. The contributions of this thesis are as follows: a classification system for process-launching strategies, an algorithm for creating an optimal-process-launching strategy, an implementation of LIBI and a performance evaluation of LIBI. Our performance evaluation demonstrates that we decreased the time required for software system bootstrapping by up to 50%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Solving Computationally Intensive Problems

Many scientific and commercial fields have problems which can be solved with computationally intensive code. Physicists want to simulate the formation of planetary bodies in the solar system [18]. Chemists want to simulate the interaction of chemicals at the atomic level [5]. Animation studios want to render every pixel of every frame for their entire movie [12]. Financial firms want to analyze the entire financial market, so that they can place their buy and sell orders before their competitors [10].

Solving these computationally intensive problems on a single computer would take too long. However, these large tasks can often be broken into smaller pieces and spread out across computational clusters. Each node of the cluster can work on their individual task, in parallel, resulting in faster times to a solution.

In the pursuit of more timely solutions, the resources available in these clustered systems have reached extreme scales, and the system sizes and capabilities are continuing to increase [27]. In November of 2005, the BlueGene/L computer at Lawrence Livermore National Laboratory was the largest supercomputer in the world, with 65,536 processors [6]. By November of 2007 a successive iteration of the BlueGene/L computer had

212,992 processors. As of June 2011, the largest and fastest supercomputer in the world, the K computer at the RIKEN Advanced Institute for Computational Science, has 548,352 processors [2].

Specialized software is needed to support these extreme-scale systems (as well as smaller scale systems). For example, tools are needed to debug, monitor, and optimize the software system. Resource managers are needed to efficiently schedule and allocate resources for individual jobs.

All these types of software systems have what we call a *bootstrapping* phase. During this phase, the software system is deployed onto an allocated set of nodes and its initialization information is disseminated. For extreme-scale software to be used successfully, it has to address the scalability and efficiency of the bootstrapping phase. Let us assume that on average it takes four minutes to bootstrap a software system which fully utilizes one of the aforementioned supercomputers. Let us also assume that on average a typical software system requires two hours of computation time. Over the course of one year, 12 days will be devoted exclusively to the bootstrapping phase.

1.2 Extreme Scale Software Bootstrapping

Distributed software bootstrapping consists of two procedures: the *process launching* procedure and the *initialization dissemination* procedure. Given a set of computational nodes, a set of executables, and a mapping between the instances of the executables and the nodes, the *process launching* procedure will execute the given executables, on the given nodes, according to the given mapping. Given a set of running processes and a set of 2-tuples of <information, operation>, the *initialization dissemination* procedure will disseminate the given information in a manner dictated by its associated operation, among the given processes. These definitions are similar to those discussed in previous research [11, 26, 4, 13].

There are several existing bootstrapping services. The Process Management Interface

(PMI) offers a bootstrapping interface for parallel programming libraries, such as MPI [4]. PMI's process launching abstraction does not allow for the placement of processes on specific nodes. The Scalable and Extensible Launching Architecture (ScELA) is also aimed at parallel programming libraries [26]. It specifies a process launching architecture which is not readily separable from its MPI implementation. LaunchMon on the other hand is aimed at the needs of software-system tools [1]. It is mainly focused on mapping the abstractions of existing launching and communication services to a common set of abstractions.

The novel bootstrapping approach of this thesis does not focus on a specific type of distributed software. Instead we focus on two goals: providing portability to distributed software and providing the best performance available on the given system. The bootstrapping service described in this thesis is called the Lightweight Infrastructure-Bootstrapping Infrastructure (LIBI) [11].

1.3 Contribution of This Thesis

This thesis makes several contributions to software-system bootstrapping:

- We have created a classification system for process launching strategies. This classification system can be used to guide the implementation of a process launching service.
- We have created an optimal algorithm for process launching and have proven the conditions under which optimality is maintained.
- We have created a prototype implementation of LIBI's bootstrapping abstractions.
- We have evaluated the performance of the LIBI prototype, using MRNet as a benchmark.

1.4 Thesis Outline

The remaining chapters of the thesis are organized as follows. Chapter 2 discusses background information and related work. This includes a discussion of several resource managers, a look into process launching strategies, as well as connection hierarchies. Chapter 3 introduces LIBI. This chapter examines MRNet's bootstrapping use cases, outlines MRNet's port to the LIBI abstractions, and discusses the different LIBI implementations. Chapter 4, discusses the optimal process launching hierarchy and provides a proof of optimality. Chapter 5 gives a detailed analysis of performance of the LIBI prototype as well as MRNet's LIBI port. Chapter 6 concludes this document and discusses future work.

Chapter 2

Background Information

2.1 Overview

In this chapter we provide information that is necessary to understand the discussion of software-system bootstrapping as well as some related work. We discuss the different models of process launching. We categorize the different mechanisms used to implement the bulk launch model. Finally, we discuss the process launching and communication abstractions provided by other bootstrapping services.

In this thesis, we discuss software-system bootstrapping. The functionality that is included in software-system bootstrapping is similar to the functionality that is included in process management. The difference between software-system bootstrapping and process management centers around the duration of the provided service. Bootstrapping occurs only at the beginning of a process' life, while process management deals with the entire life of the process. Most definitions of process management have requirements that are more than those of bootstrapping. These can include I/O redirection, signal redirection, process health monitoring, and job termination [17, 8, 15, 14, 4, 7, 26]. These added requirements do not conflict necessarily with the requirements of bootstrapping. As such, we will refer to process managers as bootstrapping services.

2.2 Process Launching Models

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 are capable of being launched with a single request; individual launches are capable of launching only a single process while bulk launches have the capability of launching more.

Several services offer individual launch. One of the most basic is exemplified by rsh, the remote shell. rsh uses a client/server model. The rsh client sends a command to the rsh server, which then executes the command on its local node. ssh, the secure shell, does the same thing as rsh, except it encrypts all of the data sent between the client and server. These individual launch services are widely available and hold a consistent interface on all platforms.

Bulk launch services on the other hand are not as widely available, and their interfaces vary from platform to platform. High Performance Computing (HPC) system vendors and resource managers generally offer their own specialized bulk-launch services. BlueGene systems and Open RTE each provide an implementation of the bulk-launch service, mpirun. CRAY systems provide their own bulk-launch service, aprun. Likewise, SLURM provides yet another bulk-launch service, srun.

2.2.1 Bulk Launch Frameworks

Bulk launch frameworks are built out of daemons. For the purposes of this thesis, a daemon is a process that is not directly controlled by the user. When used in a bulk launch framework, daemons communicate with each other to propagate and execute the requested commands. There are two areas in which bulk launch frameworks differ, the *framework persistence* and the *connection hierarchy* of the daemons.

Framework Persistence

The persistence of a bulk launch framework is based upon the number of framework components that persist between jobs. The more persistent a framework is, the less has to be setup for each launch, and the faster a launch can be. The less persistent a framework is, the more portable it can be and the less resources it needs.

The greatest amount of persistence occurs when both the daemons and the connections between the daemons persist. This type of persistence can be found in the MPD bootstrapping service [8]. Here daemons persists on each node, connected to each other in a ring topology. To launch a job, the user sends the executable path and command line arguments to one of the daemons. The contacted daemon then sends the job information around the ring. The daemons that are involved with the job then spawn a manager process on their local node. These manager processes form a smaller ring topology that is used for job control. Once the job is complete, the manager processes disconnect and terminate, but the daemons remain connected in a ring.

By having persistent connections, a bulk-launch service can quickly relay the launch command to all of the relevant nodes. There is no delay from having to form new connections at the start of each job, but this also means that it uses more system resources than necessary. Unless all nodes in the system have a persistent connection to a central node (which has scalability issues) the launch message has to be relayed through a set of intermediary nodes. If the intermediary nodes are dedicated to the resource manager, then there are less nodes available for computation. If the intermediary nodes are not dedicated to the resource manager, they could be in use by other jobs. This results in one job using resources that are allocated to another job.

A more economical use of resources can be achieved when only the daemons persist. This type of persistence can be found in resource managers like SLURM [14] and ALPS [15]. In SLURM, a daemon persists on each node. When a job needs to be launched,

connections are formed between the daemons on the relevant nodes. Once the job is complete, the connections are disconnected, but the daemons remain.

Persistent daemons still allow for fast bulk launch. They remove the daemon launch time from the total bulk launch time. However, they are not portable. Persistent daemons have to be integrated into the resource management system. This make it harder to move this type of bulk-launch service to a different platform.

The highest degree of portability can be achieved when the daemon image remains the same between jobs. This type of persistence can be found in ScELA [26]. When a job needs to be launched, ScELA launches one daemon per node. ScELA uses an executable as the daemon image, but a library based image is also possible. During the launch event, ScELA's daemons connect to each other and are then used for job control. Once the job is complete the connections disconnect and the daemons terminate.

Since there are very little preconditions required for this type of bulk-launch service it is very portable. The only requirement is that an individual launch service be available. The down side is that it does not have the potential to be as fast as a persistent daemon bulk-launch service. A daemon has to first be started on a node, before that node can further relay the launch command.

Connection Hierarchies

A second way in which bulk-launch services differ is the connection hierarchy used to launch the processes. Bulk-launch-connection hierarchies perform similar functionality as multicast-connection hierarchies. During typical multicast communication, information is relayed from node to node until all nodes have the information. For bulk launch, that information is the executable and the corresponding arguments. One important aspect of multicast communication, that affects performance, is the connection hierarchy between the nodes. Similarly, the connection hierarchy affects the scalability of the bulk-launch

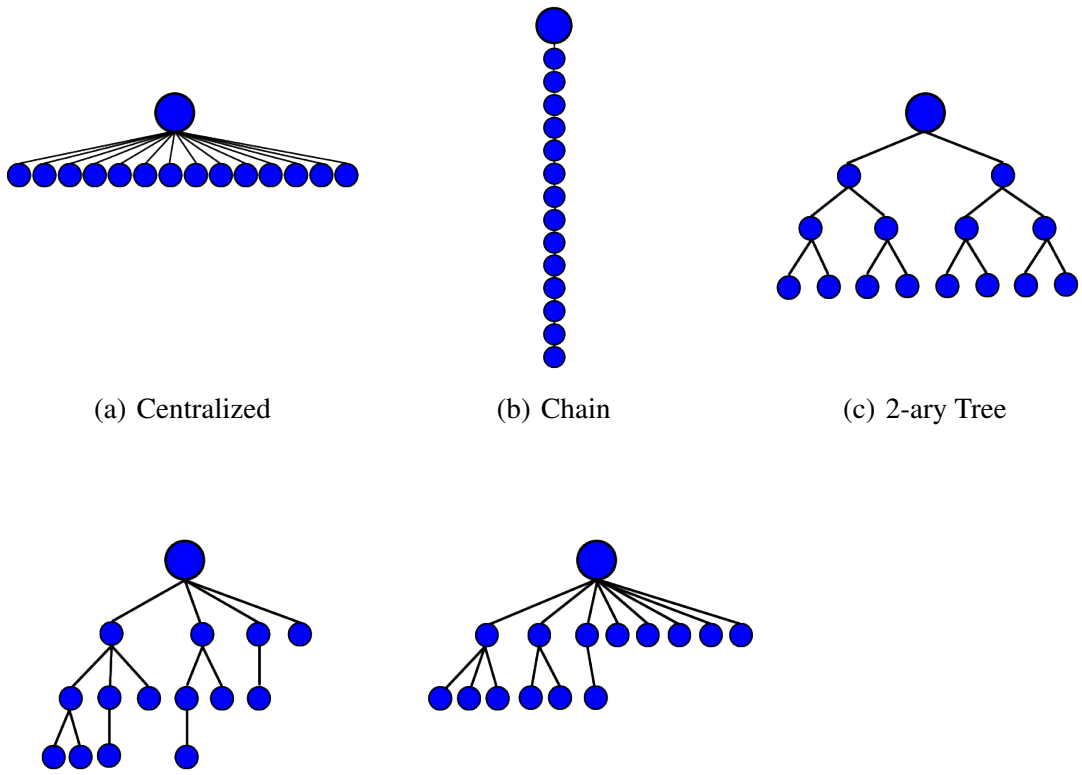
service.

We will be focusing on tree-based hierarchies in this thesis. There are many types of non-tree-based hierarchies, but they are not best suited to the type of communication needed for process launching. A hyper-cube, for example, provides excellent redundancy, but the path taken by the launch command would not utilize every connection. Instead, the non-redundant subset of connections used during process launching would resemble a tree. We will discuss 5 types of tree-based hierarchies: centralized, chain, k-ary trees, recursive-divide-and-conquer trees, and optimal-multicast trees.

The scalability of these hierarchies can be discussed in terms of productivity of the individual nodes. Productivity can be described as the amount of time a process does not spend idling. The more productive a node is, the faster the launch command is disseminated, and the better the scalability. This concept is discussed in more detail in Chapter 4.

The least scalable connection hierarchies occur at the extreme ends of the tree spectrum. This means the broadest trees and the tallest trees. The centralized connection hierarchy is the broadest possible tree based hierarchy (see Figure 2.1(a)). All of the processes are connected to the root. This means the last process has to wait until the root process sends the information to $N - 1$ other processes before it can receive the information. The chain hierarchy is worse. It is the tallest of the tree based hierarchies (see Figure 2.1(b)). All of the processes, except the root and the last process, have a parent and one child. This is similar to a ring topology without the connection between the last process and the root. For a message to reach the last process, it has wait for the message to be relayed through $N - 1$ processes.

The main reason they are used is because they perform well enough at small scales. As seen in chapter 5, relaying a message through a single process or waiting for your parent to send the message to a single prior process, does not take much time. These types of hierarchies only run into performance problems as the scale of the hierarchy increases.



(a) Centralized (b) Chain (c) 2-ary Tree

(d) Recursive Divide and Conquer Tree (e) Optimal Multicast Tree

Figure 2.1: Connection hierarchies of 15 daemons.

The k-ary tree offers a more scalable solution. A k-ary tree is a tree where each process has at most k children and the height of the tree is minimized. For the last process to receive the information it only has to wait for the information to be relayed through at most $\lceil \log_k N \rceil$ processes.

The k-ary tree is a popular connection hierarchy. ScELA, SLURM, and ALPS are just a few of the many bootstrapping services that use this hierarchy. One of the aspects of the k-ary tree that makes it so popular is that its scalability can be adjusted by the parameter k.

The scalability of the k-ary tree is not perfect, processes are not as productive as they could be. For example, in a 2-ary tree of 15 nodes (see Figure 2.1(c)) after the root has sent the information to both of its child processes it is no longer productive. It waits until the information has been relayed to the last process. The time that the root spends waiting is time that could have been spent sending information to additional processes.

The recursive-divide-and-conquer tree is designed to increase the productivity of the parent processes. This type of tree has not been used for process launching, but it has been used for communication. This is the connection hierarchy used by COBO, a group-communication service created at Lawrence Livermore National Laboratory [9]. It was designed to provided sufficient group-communication services to allow for software-system bootstrapping. COBO is used by LaunchMON.

The recursive-divide-and-conquer tree is built using a divide and conquer approach (see Figure 2.1(d)). Starting with a list of processes, the root process selects one of the processes to be its child and divides the remaining processes between itself and its new child. The root and the new child then continue this process of selecting a new child and dividing the remaining list until all of the processes are accounted for.

The recursive-divide-and-conquer tree increases the productivity of the parent nodes, but it does not maximize it. After the list of remaining processes is divided a parent has to send the executable, the arguments, and half of the list to the new child. Since the time it takes the parent to send this information and move on to its next child is not equal to the time it takes the new child to receive and parse the information, there is a delay between when the parent and the new child start their next recursive step. This delay means that the parent process will finish launching its half of the list before the new child launches its half. In the case of process launching, parsing the information can mean executing the command, and then letting the resulting process launch the remaining list of processes. This can result in an order of magnitude difference between the delay of the parent and the delay of the child, when starting the next recursive step.

The optimal-multicast tree accounts for this delay (see Figure 2.1(e)) [20]. Just like the recursive-divide-and-conquer tree, the optimal-multicast tree has never before been used for process launching. It has only been used for communication.

The optimal-multicast tree described by Park, Choi, Nupairoj, and Ni uses two parameters: t_{hold} and t_{end}. In their communication model, t_{hold} defines the time required for a process to send information to one child and then move on to the next. t_{end} defines the time required for the parent to send information to a child, for that information to traverse the network, and for the child to receive that information. The time required to perform a multicast using a given tree is modeled using the parameters of t_{hold} and t_{end}. Using this model of communication time, a dynamic-programming algorithm is used to create an optimal-multicast tree of a given size; they create larger optimal-multicast trees by combining smaller optimal-multicast trees.

2.3 Existing Bootstrapping Abstractions

In this section, we will discuss the highlights and deficiencies of existing bootstrapping abstractions. There are two sets of abstraction that are relevant to bootstrapping: process launching abstractions and group communication abstractions. The services that we will discuss are PMI, PMGR, LaunchMON, and ScELA.

2.3.1 Process Launching Abstractions

PMI

The Message Passing Interface (MPI) is the standard programming model used in HPC systems. The standard practice for launching MPI applications is to execute either `mpirun` or `mpiexec` with the appropriate arguments. These two programs are not part of the MPI standard [19]. MPI is mainly concerned with inter-process communication, not

process launching. As such it does not specify a standard command line interface which could preclude the full implementation of MPI on some systems. This helps explain the variety of the bulk-launch services discussed in Section 2.2.

Instead of a command line interface, MPI defines two process launching abstractions: MPI_COMM_SPAWN( ) and MPI_COMM_SPAWN_MULTIPLE( ). These abstractions provide the functionality to launch numerous copies of one executable or multiple executables, respectively. Many MPI implementations choose to pass this functionality onto PMI [4].

Since PMI is intended for use in process management, it specifies more requirements than what is needed for bootstrapping. These features includes I/O redirection, propagating signals, and stopping processes. PMI was designed to be used by parallel libraries, such as MPI. This use case has guided its process launching abstraction.

PMI's process launching abstraction offers a single distribution type [22]. This distribution is specified by a 3-tuple of <executable, arguments, count>. This distribution type works fine for parallel libraries, but it is unsuitable for software-system tools. A debugger needs the ability to launch its processes on a specified set, or subset, of nodes. This functionality is not possible through the PMI process launching abstraction.

LaunchMON

LaunchMON is a bootstrapping framework for software-system tools [1]. One of the differences between software-system tools and software systems is that software-system tools need to "attach" to another set of processes. When a tool attaches to a running process, it is given control of the running process' execution. The tool can start and stop execution, manipulate the values of variables, and trace the stack.

To attach a software-system tool to a software-system, LaunchMON needs to know the location of each process in the software system. To do this LaunchMON attaches itself

to the bulk-launch service (mpirun, srun, aprun, ...) that is responsible for launching the software system and looks for the locations of each of the launched processes. Once LaunchMON has the location of each process it can launch its own processes at the same locations and perform a normal attach for each process.

LaunchMON's "attach" abstraction assumes the existence of a bulk-launch service. The process distribution that is used by attach is a 1-tuple of <pid>, where pid is the process id of the bulk-launch process. Since this bulk-launch service is guaranteed to exist, LaunchMON uses the same bulk-launch service to launch its own processes.

This dependence on existing bulk-launch services has influenced its own process launching abstraction. The process distribution used by LaunchMON's process launching abstraction is specified as a list of 4-tuples of <exec, args, count, dt> where exec is the path to the executable, args are the arguments for the executable, count is the number of processes needed to be launched, and dt is a distribution type. A few of the distribution types include *block*, where each node is given a consecutive block or processes, *cyclic*, where each node is cycled through, adding a process to each in turn, and *hostslist*, where the node of each process is specified. These distribution types are supported by many existing bulk-launch services, including: srun, aprun, and most implementations of mpirun.

For this thesis, we will be basing our process launching abstraction off of LaunchMON's process launching abstraction. This allows us to easily work with existing bulk-launch services.

2.3.2 Group Communication Operations

Many libraries provide a set of group communication operations. As noted in section 2.3.1, MPI is the most popular one. MPI provides numerous group communication operations. The wide variety of operations ensures that almost any type of communication is defined in MPI.

For software-system bootstrapping, most of these operations are not needed. One could use MPI for bootstrap communication, but it would be overkill. When MPI itself is bootstrapped, it only relies on a small subset of its own group communication operations.

PMI

As mentioned in section 2.3.1, PMI is designed for bootstrapping parallel libraries, like MPI. The communication abstractions defined by PMI involve the storage and retrieval of key-value pairs from the key-value space (KVS). To accomplish this task, 3 operations are defined: Put, Get, and Fence. Put stores a set of key-value pairs in the KVS, Fence is a group communication operation that 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.

These three operations were first introduced as the communication interface for MPD in 2001 [8]. This means there are a number of resource managers that already support them [4]. However, it does not mean they are not efficient. For the KVS to work properly, all of the key-value pairs need to be distributed to all nodes or they need to be accessible from all nodes. The trade off is either wasting a lot of network resources and memory by broadcasting and then storing unneeded key-value pairs on every node or processing an individual value request for each Get operation [25].

PMGR

The PMGR Collective takes a different approach to parallel library bootstrap communication [21]. PMGR provides seven group communication operations. These include barrier, broadcast, gather, scatter, allgather, alltoall, and allgatherstr. These operations perform the

same functionality that is defined by MPI operations of a similar name. The only exception is allgatherstr. Allgatherstr performs a similar function as allgather except the data in question are null terminated strings, whose length can vary from string to string.

One of the downsides of using these operations is that it makes setting up connections between two distinct sets of processes harder. 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 has to explicitly create connection setup data, and then somehow make it available to the new processes.

LaunchMON

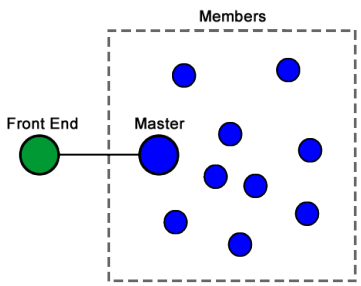


Figure 2.2: Process types used by LaunchMON's communication operations.

LaunchMON's bootstrap communication abstractions address the issues of efficiency and bootstrapping additional sets of processes. LaunchMON's abstractions are built around three process types: front-end, master, and member (see Figure 2.2). The front-end process is the one which launches all of the other processes. The front-end process can communicate with the master process but cannot directly communicate with any of the other member processes. When bootstrapping an additional set of processes the front-end can act as an intermediary, storing the connection setup data, and passing it on to the additional processes.

To accommodate the three process types, 6 group communication operations are given: send, recv, broadcast, scatter, gather, and barrier. Send transfers data from the front-end to the master process while all of the other members wait. Recv transfers data in the opposite direction. The remaining operations are all performed between the member processes. Broadcast sends data from the master to all members, scatter sends distinct data from the master to each member, gather sends data from each member to the master, and barrier blocks until all members have entered the group communication operation.

ScELA

With the exception of PMGR's allgatherstr, the Scalable and Extensible Launching Architecture (ScELA) provides the group communication operations of both PMI and PMGR [26]. In addition to these group operations, ScELA provides point-to-point of communication.

Offering PMGR's communication operations in addition to that of PMI's, lessens the advantages of PMI. The advantage that is taken away is the ability to easily bootstrap additional sets of processes. If an application now has the option to perform broadcast, gather, and scatter operations that means that all of the relevant information is not guaranteed to be in the KVS. Any additional processes now have to be dealt with in the same manner as when using PMGR.

2.4 Summary

In this chapter we discussed the difference between individual launch and bulk launch. We then categorized the different methods of bulk launch based on the persistence of the framework and the connection hierarchy of the daemons. The more persistent a framework is, the faster it can be. However, increased persistence also means less portability and a higher resource requirements. For connection hierarchies, their scalability can be

determined by the productivity of the individual process.

We discussed the process launching and group communication abstractions used by existing bootstrapping services. We discussed the motivating use cases of the launch abstractions. We discussed the tradeoffs of the group communication operations.

Chapter 3

A Framework For Bootstrapping Extreme Scale Software Systems

3.1 Overview

Our approach to software-system bootstrapping is to create a set of abstractions that are sufficient for bootstrapping as well as to create a framework for implementing these abstractions. The framework we created is called the Lightweight Infrastructure-Bootstrapping Infrastructure (LIBI). We have discussed the concepts involved with the abstractions and framework in a previous paper [11]. This chapter will review these concepts and expand upon them.

LIBI has two goals, one for its abstractions and one for its framework. The first goal is for LIBI's abstractions to serve as a model for the type of services a resource manager should provide. The second goal is to provide a framework which gives 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.

LIBI is designed to sit between large-scale-software systems and the underlying bulk-launch and group-communication services. Since LaunchMon already provides access to a

number of bulk-launch services and group-communication services, LIBI will sit on top of LaunchMon when appropriate. This relationship may change in the future. If a particular platform is not supported by LaunchMon, LIBI will be able to use its own implementation of the services provided by the resource manager or even rely on its own individual launch based bulk-launch and communication service. Figure 3.1 depicts this relationship.

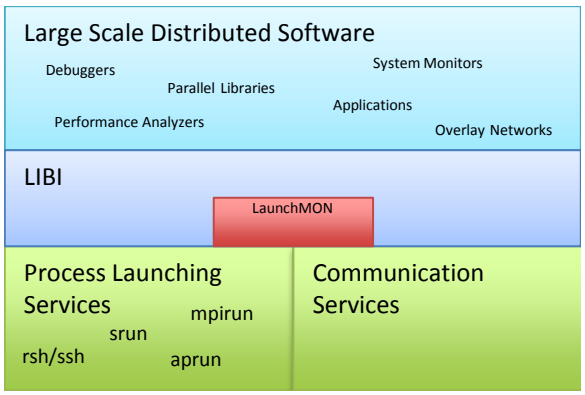


Figure 3.1: The relationship between LIBI and the software it interacts with.

3.1.1 Use Cases

The LIBI abstractions are designed to accommodate two use cases. The first use case is to bootstrap a software system, on a specified set of nodes, where the processes are launched from a single executable. The second use case is to bootstrap a software system, on a specified set of nodes, where the application is comprised of multiple executables.

These use cases are similar to those of PMI. PMI's use cases are based off of the needs of parallel libraries, mainly MPI. MPI's process-launching requirements center around the functions: MPI_COMM_SPAWN( ) and MPI_COMM_SPAWN_MULTIPLE( ). The difference between LIBI's use cases and the needs of these two functions is that these two functions do not require a specified set of nodes. This additional requirement does not preclude LIBI from bootstrapping MPI, it just requires an additional step. MPI would first have to request an appropriately sized allocation of nodes before using LIBI.

To facilitate LIBI's use cases, the process launching and communication abstractions will be based on the concepts of a front-end and a session. These concepts are similar to those used by LaunchMon. A session is a group of processes that are launched at the same time and can communicate with each other. Each process within the session is a member. Each member is given a unique identifier called a rank. One of the members is the master of the session. The front-end can only communicate with the master, but not directly with any of the other members.

3.2 LIBI Abstractions

3.2.1 Process Launch Abstraction

As noted in the discussion of existing process launching abstractions (Section 2.3.1), a distribution type is the method in which processes are placed on nodes. PMI offers a single distribution type which allows the user to specify the number of processes to create from a given executable, but not how those processes should be distributed among the nodes. LaunchMon offers multiple distribution types, which can be mapped to the distribution types of many bulk-launch services.

LIBI's process launching abstraction will focus on the host list distribution type. There are a couple of reasons for this. First, it will support both of our use cases. Second, the other distribution types (block, cyclic, unspecified, ...) can be built on top of the host list distribution type, but the host list distribution type cannot be built on top of any other. Third, the host list distribution type allows for the direct implementaion of a bulk-launch service that is built on top of individual launches. Extra information would have to be added to the *block* distribution type, such as a list of possible nodes, before any individual launch could take place. Finally, most bulk-launch services already support the host list distribution type.

LIBI's process launching abstraction uses a list of process distributions. A process distribution is a 5-tuple of <sid, exe, args, hl, 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, hl is a host list, and env is the environment to use for the created processes. A host list is a list of 2-tuples of <hostname, num-procs>, which defines how many processes to create on the named node.

launch(process-distribution-list) instantiates the appropriate sets of processes according to the input process distributions.

Table 3.1: The launch abstraction used by LIBI.

3.2.2 Group Communication Abstraction

The group-communication abstraction employed by LIBI will be similar to LaunchMon's. It will make use of the front-end, master, and member process classes. The group-communication operations include: send, receive, broadcast, scatter, gather, barrier. This allows LIBI to easily piggyback on top of LaunchMon, so that it can make use of the bulk-launch and group-communication services that LaunchMon supports.

LaunchMon's set of group-communication operations were also chosen based on functionality and efficiency reasons. As noted in Section 2.3.2, the front-end, master, member process classes facilitate the bootstrapping of additional processes. They are also more efficient than PMI's Put, Fence, and Get operations.

3.2.3 Example Usage

To demonstrate how the LIBI abstractions could be used to bootstrap a software system, we have conducted a case study of MRNet. MRNet is a multicast/reduction network that

[send|receive](sid, 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.

Table 3.2: The communication abstractions used by LIBI.

provides a scalable data aggregation service [23]. It is comprised of three different process types: the front-end, the communication daemons, and the back-end daemons. MRNet's bootstrapping needs include launching its communication daemons onto a specified set of nodes, launching the back-end daemons onto a specified set of nodes, and then connecting these two types of daemons to form a *Tree Based Overlay Network* (TBON). The front-end is the root, the communication daemons are the internal nodes and the back-end daemons are the leaf nodes.

MRNet's normal bootstrap mechanism involves multiple individual launches. The front-end individually launches its child communication daemons. The child communication daemons connect back to their parent, the root. Once a connection is established, the parent sends the MRNet TBON topology. The children then launch their own children. This process is repeated until all processes have been launched.

MRNet's back-end attach mode offers a second bootstrapping scenario. In the attach mode, the back-ends are started by a separate entity. Here, MRNet is only concerned with launching the communication daemons, and connecting them into a TBON. The back-ends will attach themselves to the communication daemons at a later time.

For this case study we have replaced MRNet's normal bootstrapping mechanism with

LIBI. MRNet's front-end process creates two LIBI sessions; session one is for the communication daemons and session two is for the back-ends. MRNet's front-end translates the MRNet topology into separate process distributions for each LIBI session. After adding MRNet specific environment variables to each session's environment, the front-end calls LIBI's *launch* abstraction to launch both sessions.

Each communication daemon and back-end daemon needs to know the hostname and port number of its parent. To start the communication process, the front-end *sends* the MRNet topology to each session master, who then *broadcasts* it to their session members. The parent/child relationships are contained within the MRNet topology.

Since there could be multiple processes per node, the front-end *sends* a host list containing the host name of each process in the session, in LIBI rank order, to each session master. The master *broadcasts* the host list to its session members. Each member scans the host list, counting the number of occurrences of its own hostname before reaching the position equal to its LIBI rank. The member processes then scan the MRNet topology, to find the MRNet rank with the same hostname, that has the same number of occurrences of its own host name preceding it. Using this information, each process can find the host name and MRNet rank of its parent.

The final piece of information needed is the port number where its parent is listening for connections. The communication daemons bind to a port number and then use LIBI to *gather* all of the <port number, MRNet rank> tuples to the the session master. The session master *sends* these tuples to the front-end. The front-end *receives* the tuples and adds its own < port number, MRNet rank> tuple. These tuples are *sent* to the master of each session, who in turn *broadcasts* the tuples to their members. Each member, knowing the MRNet rank of its parent, scans the list of tuples to finds its parent's port number.

After every process knows its parent's hostname and port number, LIBI is no longer needed. From here, MRNet finishes its own bootstrapping process. All communication

daemons and back-end daemons connect to their parent. These connections form MRNet's TBON, which completes MRNet's bootstrapping.

3.3 Prototype Implementations

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.

3.3.1 Integration with LaunchMon

The purpose of integrating LIBI with LaunchMon is to offload some development work. As discussed in Seciton 2.3.1, LaunchMon has both an attach and a launch abstraction. LaunchMon's attach implementation already has support for several bulk-launch services. The implementation of LaunchMon's launch abstraction should not be far behind. These bulk-launch services *should* be able to provide the best (and sometimes only) process launching option on a given platform.

For the most part LIBI's launch and communication abstractions pass the information directly to LaunchMon's abstractions. The differences between the two abstractions are mainly implementation specific.

3.3.2 Individual Launch Based Implementation

When a bulk-launch service is not available on the given platform, or if it is not performing as well as desired, LIBI needs to provide an adequate alternative. To do this, LIBI will be relying on a persistent image daemon. As noted in Section 2.2.1, persistent image daemons are the least persistent. They are alive only for the duration of the job. Since they

do not need to be integrated with the resource manager, they are very portable. The only thing they rely on is the existance of an individual launch service, such as rsh.

LIBI's daemon images will be library based, instead of executable based. This turns the target executable into a LIBI daemon, at least while bootstrapping. The library based daemon should help maintain LIBI's *lightweight* status.

LIBI will use three process launching phases. During the first phase, the front-end will launch the master process and define a connection hierarchy. During the second phase, LIBI will use multiple individual launches to launch and connect, one process per node. These processes will be launched and connected in a manner defined by the connection hierarchy that was created in the first phase. During the third phase, these processes 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 collocated process. This approach has been used in previous bootstrapping services, with good results [26, 13].

During the 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 hierarchy will also be used as its communication hierarchy. For the purposes of our research, LIBI will be able to use any arbitrary hierarchy. This will allow us to test the relative performance of different hierarchies. It will also allow us to easily create our own Greedy hierarchy (see Chapter 4) and compare it to other hierarchies.

3.4 Summary

In this chapter we described the Lightweight Infrastructure-Bootstrapping Infrastructure. LIBI is designed to sit between large-scale-software systems and the underlying bulk-launch and communication services. The two motivating use cases cover bootstrapping

software systems, on a specified set of nodes, where the processes are created from a single executable and where the processes are created from multiple executables.

LIBI's launch and communication abstractions are similar to those of LaunchMon. LIBI's launch abstraction will focus on the host list distribution type. The group-communication operations offered by LIBI will include: send, receive, broadcast, scatter, gather, barrier. Using MRNet as a case study we showed how LIBI's abstractions could be used to bootstrap a software system.

LIBI currently has two prototype implementations. The first implementation sits on top of LaunchMon. The second implementation uses an individual launch based approach. This implementation relies on library based persistent image daemons. These daemons will be able to form any arbitrary hierarchy.

Chapter 4

The Optimal Process Launching Hierarchy

4.1 Overview

In this Chapter, we examine the process-launching phase of software-system bootstrapping. We begin by modeling the time required to launch a given set of processes. We then propose a new method for launching processes which minimizes the modeled launch time.

We can break the process-launching phase of bootstrapping into three separate subphases: preparation, connection, and co-location. These three phases can be seen in the LIBI's individual launch implementation. During the preparation phase, the front-end process starts the master process. During the the connection phase the first process on each node is started. Parent processes sequentially launch their child processes. The child processes then go on to launch their children. During the co-location phase, all of the other processes on each node are started.

Listing 4.1: Pseudo code for the connection phase

```
1 main( ){
2         retrieve   parameters  from parent
3         for( each child  ){
4                 create  a  new process
5                 if ( child  process  )
6                         execute  individual   launch  command
7         }
8 }
```

We will mainly be focusing on the connection phase. The algorithm for the connection phase is shown in Listing 4.1. This algorithm transforms a set of processes into a hierarchy of processes. The process which preforms the individual launch is the parent, while the process that is launched, is the child. This parent/child relationship among the processes forms a hierarchy. Horizontal connections between processes do not make sense in this context, so they are not allowed.

There are three types of hierarchies that we will build a model for: chain, sequential, and arbitrary hierarchies. The first two hierarchies are degenerate cases. These two hierarchies demonstrate the two extremes of the tree spectrum. The sequential hierarchy is the broadest possible while the chain hierarchy is the tallest. All other hierarchies will fall somewhere in between these two.

The rest of this Chapter will create a model for the time required to launch a set of processes, based on the three phases of process launching. Given a process launching hierarchy the model will estimate the amount of time it will take to launch all of the processes. This model will be used as a tool to compare separate hierarchies. It will also be used to guide the creation of the Greedy hierarchy. We will then prove that the Greedy hierarchy is optimal, meaning it minimizes the total launch time.

4.2 Modeling Hierarchy Launch Time

To create a model of launch time, we will model the time required for each phase of the launch. For any given set of set of processes we can assume that the time required by the preparation phase is a constant. We can also assume that the time required by the co-location phase is a constant. Let us label these constant times as PREP and COLOC, respectively. We can make these assumptions because any variability in PREP and COLOC is insignificant when compared to the time required by the connection phase.

Modeling the connection phase requires a little more consideration. The time required by the connection phase depends on the connection hierarchy. The general method for creating the model of the connection phase entails tracing through the algorithm in Listing 4.1. By tracing through the algorithm we can see which sections are repeated, and how often they are repeated. We then create a model of launch time, by stating that a section of the algorithm is repeated x times and takes y amount of time.

In order to create a model of the total launch time, we only need to model the code that leads to the launch of the last process. However, it is that it is hard to determine, in an apriori fashion, which process is the last. As such, we model the time it takes to launch each process and then use the maximum as the modeled launch time. This is expressed in Equation 4.1, where $Launch(h)$ is the time required to launch all of the nodes in hierarchy h and $Launch(x, h)$ is the time required to launch node x in hierarchy h. The number of nodes is represented by n.

$$Lanch(h) = PREP + max_{x=1}^{n} Launch(x, h) + COLOC \tag{4.1}$$

4.2.1 Modeling the Chain Hierarchy

The first hierarchy we will model, is the Chain hierarchy. As introduced in Section 2.2.1, in the Chain hierarchy, every node except the last, has one child. This can be thought of as a ring without the connection between the last node and the first. It can also be thought of as a 1-ary tree.

To model the Chain Hierarchy, we need to label a relevant section of the algorithm in Listing 4.1. The section of the algorithm we are concerned with spans two nodes. It starts when the parent node creates a new process and executes the individual launch command (rsh, ssh, ...) for the child node. The child node then enters main and retrieves its parameters from its parent. Let us label this trace through the algorithm as *remote_launch*. Let us label the time required by *remote_launch* as REMOTE.

For the Chain hierarchy *remote_launch* is repeated $n-1$ times, where n is the number of nodes in the hierarchy. The value $n-1$ also equates to the number of ancestors of the last node. In fact, the time required to launch any process in the Chain hierarchy is related to the number of ancestors it has. This idea is encapsulated into Equation 4.2, where $anc(x,h)$ is the set of ancestors of node x in hierarchy h and $chain_n$ is the Chain hierarchy with n nodes.

$$Lanch(x, chain_n) = (|anc(x, chain_n)| * REMOTE) \tag{4.2}$$

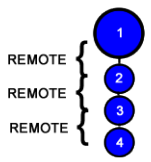


Figure 4.1: Modeling the Chain hierarchy at 4 nodes.

4.2.2 Modeling the Sequential Hierarchy

The next hierarchy that we will model is the Sequential hierarchy. As introduced in Section 2.2.1, in the Sequential hierarchy, the root launches every child by itself. This can be thought of as an k-ary tree when there are $k + 1$ nodes in the tree.

To model the Sequential Hierarchy, we need to label another relevant section of the algorithm in Listing 4.1. The section we are concerned with is executed by the root process. Let us label a single iteration of the "for(each child)" loop as *sequential_wait*. Let us label the time required by *sequential_wait* as SEQ.

For the Sequential hierarchy, *sequential_wait* is repeated $n - 2$ times, where n is the number of nodes in the hierarchy. The value $n-2$ also equates to the number of preceding siblings of the last node. In fact, the time required to launch any process in the Sequential hierarchy is related to the number of preceding siblings it has. This idea is encapsulated into Equation 4.3, where $presib(x, h)$ is the set of preceding siblings of node x in hierarchy h and $sequential_n$ is the Sequential hierarchy with n nodes.

$$Launch(x, sequential_n) = (|presib(x, sequential_n)| * SEQ) + REMOTE \quad (4.3)$$

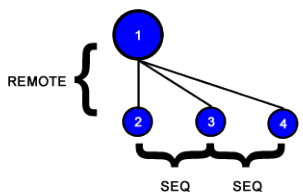


Figure 4.2: Modeling the Sequential hierarchy at 4 nodes.

4.2.3 Modeling Arbitrary Hierarchies

The last step is to model arbitrary hierarchies. Instead of limiting ourselves to a model of another rule based hierarchy, let us model how long it takes to launch a node within any given hierarchy. This model will encompass k-ary trees and recursive divide and conquer trees as well as the Chain and Sequential hierarchies.

For an arbitrary hierarchy we do not need to label any new sections of the algorithm; we still rely on *remote_launch* and *sequential_wait*. The difference is that the number of repetitions of each of these sections of the algorithm is not clearly identifiable.

To identify the number of repetitions of each section of the algorithm we recursively define the model. In the recursive definition, let us isolate each parent with its children. When isolated like this, the parent becomes the root and this sub-hierarchy becomes a sequential hierarchy. Let us label node p's sub-hierarchy as P. The modeled launch time of node x, which is a child of node p, can be defined as the time to launch node p plus the time to launch node x in sub-hierarchy P.

$$\begin{aligned} Launch(x,h) &= Launch(p,h) + Launch(x,P) \\ &= Launch(p,h) + (|presib(x,P)| * SEQ) + REMOTE \end{aligned}$$

If we follow the recursion to its conclusion, we can see that *remote_launch* is repeated for each ancestor of x. We also see that we count the preceding siblings of node x as well as the preceding siblings of each ancestor of node x. These ideas are encapsulated into Equation 4.4, where $psas(x,h)$ is the set of nodes which include the preceding siblings of node x as well as the preceding siblings of each ancestor of node x in hierarchy h.

$$Launch(x,h) = (|psas(x,h)| * SEQ) + (|anc(x,h)| * REMOTE) \tag{4.4}$$

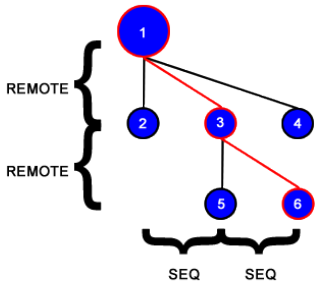


Figure 4.3: Modeling the launch of node 6 in an arbitrary hierarchy.

4.3 Greedy Hierarchy

For a given number of nodes there are a vast number of possible hierarchies. In this section we will outline an algorithm that creates a hierarchy of n nodes, which takes the least amount of time to launch out of any hierarchy of n nodes. Due to the use of a greedy algorithm we will call this hierarchy the Greedy hierarchy. We will also prove that the Greedy hierarchy is optimal, when using the three phases of process launching that were described in Section 4.2.

The Greedy hierarchy is inspired by the construction of optimal-multicast trees described by Park et al. [20]. As discussed in Section 2.2.1, Park et al. start by creating a parametrized model for multicast communication. This is similar to how we created a model of process launch time. They then use a dynamic programming model to create a tree which minimizes the multicast communication time. They combine smaller optimal trees to create larger optimal trees.

There are a couple of differences between the optimal-multicast tree and the Greedy hierarchy. First, the optimal-multicast tree was intended to be used for communication while the Greedy hierarchy is intended to be used for process launching. Second, the optimal-multicast tree parameters are machine specific, such as network latency. The Greedy hierarchy on the other hand relies on machine and application specific parame-

ters. For LIBI's individual launch version, REMOTE is dependent on numerous factors such as network latency, executable size, how the executable was compiled (compiler optimization flags, shared vs archived libraries, ...), as well as the location of the LIBI code within the executable (do other processes occur before LIBI initialization?). Finally, the optimal-multicast tree uses a dynamic programming algorithm while the Greedy hierarchy uses a greedy algorithm.

4.3.1 Greedy Algorithm

Listing 4.2: Greedy Algorithm Pseudo Code

```
1 for(each node){
2         Place node in position with lowest modeled time
3         Calculate modeled time of next sibling
4         Calculate modeled time of first child
5 }
```

Listing 4.2 shows the pseudo code of the greedy algorithm which creates the Greedy hierarchy. The algorithm takes a list of nodes as input and returns the Greedy hierarchy for those nodes. The first node of the list is placed in the root position of the hierarchy. Nodes are added to the hierarchy by greedily choosing the position which has the smallest modeled launch time.

To keep track of the available positions, we can use a heap data structure of <position,time> pairs. This allows us constant time lookup of the position with the smallest modeled launch time and $O(log\ n)$ time for the insertion of new available positions. This allows the greedy algorithm to complete in $O(n\ log\ n)$ time.

4.3.2 Proof of Optimality

Intuitively, the greedy algorithm should produce an optimal process launching hierarchy. Since we are placing new nodes in the position which results in the fastest launch time, none of the processes should ever be idle. If a parent process is idle for long, its child positions will eventually become the fastest in the hierarchy. At which point the parent node will be assigned a child node to launch. This should maximize the productivity of each process which should result in a minimal launch time.

We will prove that the greedy algorithm in Section 4.3.1 produces an optimal-process-launching hierarchy. We start by proving that the range of possible launch times is discrete (Lemma 2). Since there is a discrete range of possible launch times, these times can be ordered from lowest to highest (Definition 7). We then show that for each possible launch time, there is a finite number of nodes that require that amount of time to be launched (Lemma 3). Finally, we show that the Greedy hierarchy will saturate the lowest unsaturated launch time, before moving on to the next lowest unsaturated launch time. This will be used to prove that the Greedy hierarchy is optimal (Theorem 1).

Definition 1. Each node in a process launching hierarchy has one parent (except the root, which has none) and can have multiple children. A node's children are ordered, meaning child x has to exist in order for there to be a child $x+1$. The following is a list of functions which return a set of nodes that are related to node x in various ways.

Function	**Description**
$root(h)$	$root(h)$ returns the root of hierarchy h
$parent(x, h)$	$parent(x, h)$ returns the parent of node x in hierarchy h.
$anc(x, h)$	$anc(x, h)$ returns the set of ancestors of node x in hierarchy h. These are the nodes on the path between the root and node x, including the root but not node x. $anc(x, h) = \{parent(x, h),\ parent(parent(x, h), h),\ ...,\ root(h)\}$
$children(x, h)$	$children(x, h)$ returns the sequence of children of node x in hierarchy h.
$position(x, h)$	$position(x, h)$ returns the position of node x within $children(parent(x, h), h)$.
$siblings(x, h)$	$sibling(x, h)$ returns the set of siblings of node x in hierarchy h. $siblings(x, h) = \{y : y \in children(parent(x, h), h)\ and\ position(y, h) \neq position(x, h)\}$
$preSiblings(x, h)$	$preSiblings(x, h)$ returns the set of preceding siblings of node x in hierarchy h. $preSiblings(x, h) = \{y : y \in siblings(x, h)\ and\ position(y, h) < position(x, h))\}$
$psas(x, h)$	$psas(x, h)$ returns the set of preceding siblings of each ancestor of node x as well as the set of preceding siblings of node x in hierarchy h $psas(x, h) = \{y : y \in preSiblings(z, h)\ and\ z \in anc(x, h)\} \cup preSiblings(x, h)$
$available(h)$	$available(h)$ returns the set of nodes which is comprised of the next available child position of each node in hierarchy h. If node x has 3 children, the next available child of node x would be its 4^{th} child.

Definition 2. All process launching hierarchy implementations are based on three phases:

Preparation Phase: This phase can include such tasks as creating the hierarchy and validation of parameters. This phase occurs before the connection phase.

Connection Phase: During this phase, a parent node will sequentially contact each of its child nodes. Once a child node has been contacted by its parent, the child node will receive the information needed to contact its own children. The child node will then proceed to contact its own children.

Co-location Phase: During this phase, each node launches all of its local processes. This phase occurs after the connection phase.

Definition 3. Some abbreviations:

Name	**Definition**
PREP	PREP is the constant amount of time required by the preparation phase.
REMOTE	REMOTE is the constant amount of time required between when the parent starts to contact a child and when that child is ready to contact its children.
SEQ	SEQ is the constant amount of time required for a parent to contact one child and be ready to contact the next.
COLOC	COLOC is the constant amount of time required by the co-location phase.
H_n	H_n is the set of hierarchies containing n nodes.
inf	inf is the hierarchy where every node has an infinite number of children.

Definition 4. The time required to launch node x in hierarchy h is determined by the time required to launch each node along the path from the root to node x. There are $|psas(x,h)|$ preceding siblings along this path, each requiring SEQ time, and $|anc(x,h)|$ ancestors along this path, each requiring REMOTE time. The time required to launch node x in hierarchy h is defined as $Launch(x,h)$:
$Launch(x,h) = (|psas(x,h)| * SEQ) + (|anc(x,h)| * REMOTE)$

Definition 5. The time required to launch hierarchy $h \in H_n$ has to account for each phase of the implementation. The time required to launch hierarchy h is defined as $Launch(h)$:

$$\begin{aligned} Lanch(h) = \; & PREP \\ & + max_{x=1}^{n} \; Launch(x,h) \\ & + COLOC \end{aligned}$$

Definition 6. Let us create a new operation $A \oplus B$ where A and B are sets:

$$A \oplus B = \{a + b : (a,b) \in A \times B\}$$

Note: The Cartesian Product is defined as $A \times B = \{(a,b) : a \in A \; and \; b \in B\}$.

Lemma 1. *The range of $A \oplus B$ is discrete, when A and B are both countably infinite sets.*

Proof. The Cartesian product of two countably infinite sets is a countably infinite set [3]. $A \oplus B$ creates one value for each element of $A \times B$. The size of the range of $A \oplus B$ is bound between a finite number of values and a countably infinite number of values. At the lower bound, it is a finite set, and as such it is discrete. At the upper bound, a countably infinite set can be put into a one-to-one relationship with the natural numbers, which would make it discrete. Either way, The range of $A \oplus B$ is discrete. □

Lemma 2. *The range of Launch(h) is discrete.*

Proof. According to Definition 5, $Launch(h)$ relies on the maximum launch time of a single node in hierarchy h. Let us label the node that take the longest time as max.

$$Launch(h) = PREP + Launch(max, h) + COLOC \quad \text{Definition 5}$$

$$\begin{aligned} &= PREP \\ &\quad +(|psas(max, h)| * SEQ) \\ &\quad +(|anc(max, h)| * REMOTE) \\ &\quad +COLOC \end{aligned} \quad \text{Definition 4}$$

$$\begin{aligned} range(Launch(h)) = range(&PREP \\ &+ (|psas(max, h)| * SEQ) \\ &+ (|anc(max, h)| * REMOTE) \\ &+ COLOC) \end{aligned}$$

$$\begin{aligned} &= range(PREP) \\ &\quad \oplus range(|psas(max, h)| * SEQ) \\ &\quad \oplus range(|anc(max, h)| * REMOTE) \\ &\quad \oplus range(COLOC) \end{aligned}$$

$$\begin{aligned} &= \{PREP\} \\ &\quad \oplus \{nat * SEQ : nat \in \mathbb{N}\} \\ &\quad \oplus \{nat * REMOTE : nat \in \mathbb{N}\} \\ &\quad \oplus \{COLOC\} \end{aligned}$$

$$\begin{aligned} range(Launch(h)) = &\{(nat * SEQ) + PREP : nat \in \mathbb{N}\} \\ &\oplus \{(nat * REMOTE) + COLOC : nat \in \mathbb{N}\} \end{aligned} \quad \text{Definition 6}$$

$$range(Launch(h)) \; is \; discrete \quad \text{Lemma 1}$$

□

Definition 7. More abbreviations:

Name	**Definition**
$time[i]$	$time[i] =$ the i^{th} smallest value of $range(Launch(h))$.
$nodes[i]$	$nodes[i] = \{x : x \in inf \text{ and } Launch(x, inf) = time[i]\}$

Lemma 3. $\forall i \in \mathbb{N}, nodes[i]$ *is a finite set.*

Proof. A node can be encoded as a string of *s*'s and *c*'s. This encoding describes the path from the root to the node. If two nodes share the same encoding they occupy the same position in the hierarchy. An *s* indicates that you need to proceed to the following sibling of the current node. A *c* indicates that you need to proceed to the first child of the current node. Given these definitions of *s* and *c*, an encoding of node x in hierarchy h will have an *s* for each member of $psas(x, h)$ and a *c* for each member of $anc(x, h)$.

Let us examine node x where $x \in nodes[i]$. Every node which shares the same number of s's and c's will launch in the same amount of time. To count the number of nodes which share the same number of s's and c's as node x, we will analyze its encoding.

Since every encoding has to start with a c, we will remove the first c from the analysis. After removing the first c, the encoding has α s's and βc's, where $\alpha = |psas(x, inf)|$, $\beta = |anc(x, inf)| - 1$. Without the first c, the remaining encoding has a length of $(\alpha + \beta)$. To count the number of nodes which have the same number of s's and c's as node x, we start by counting the number of permutations of a set of $(\alpha + \beta)$ distinct values:

$$(\alpha + \beta)!$$

There are only two distinct values in the encoding of a node, s and c, so we need to remove some duplicate counts. The current count considers each instance of s to be a distinct value. So (s, s, s) is counted six times (one for each permutation), instead of once. This means there are $\alpha!$ duplicates for each legitimate ordering of s's. This also means

there are $\beta!$ duplicates for each legitimate ordering of c's. Since we are dealing with a factorial, we have to divide out these duplicates.

$$\frac{(\alpha+\beta)!}{\alpha! * \beta!}$$

The pair of of values, α and β, for node x may not be the only pair which results in a launch time of $time[i]$. To determine the cardinality of $nodes[i]$, we must account for the α and β values of all nodes whose launch time is $time[i]$.

Let $\epsilon[i]$ be the set of all pairs of α and β which make the following true:

$$PREP + (\alpha * SEQ) + ((\beta + 1) * REMOTE) + COLOC = time[i]$$

Let α_p^i and β_p^i denote the α and β values of the p^{th} pair of $\epsilon[i]$

$$\left.\begin{array}{l} |nodes[1]| = 1 \\ |nodes[i]| = \sum_{p=1}^{|\epsilon[i]|} \frac{(\alpha_p^i+\beta_p^i)!}{\alpha_p^i!*\beta_p^i!} \end{array}\right| \; nodes[1] = \{root\}$$

□

Definition 8. Let us label the Greedy hierarchy which contains n nodes as G_n. The Greedy hierarchy is defined recursively:

For $n = 1$, G_1 is the hierarchy which only contains the root node.
For $n > 1$, $G_n = G_{n-1} + x : x \in available(G_{n-1})$ *and*
$\forall y \in available(G_{n-1}), Launch(x, inf) \leq Launch(y, inf)$

Definition 9. Given that $op \in H_n$, *op* is optimal if $\forall h \in H_n, Launch(op) \leq Launch(h)$.

Theorem 1. *The greedy algorithm defined in Definition 8, will create an optimal hierarchy of n nodes.*

Proof. We prove it by induction:

For n = 1: G_1 is the hierarchy comprised of only the root. Since $|H_1| = 1$, $\forall h \in H_1, Launch(G_1) \leq Launch(h)$, so G_1 is optimal.

For n >1: G_n is created by starting with G_{n-1} and adding a node to the position which results in the lowest possible launch time. For increasing numbers of nodes, the greedy algorithm first adds all of the nodes from $nodes[x]$ to the hierarchy. Once all of the nodes in $nodes[x]$ are in the hierarchy, the greedy algorithm moves on to the nodes in $nodes[x+1]$.

The greedy algorithm is guaranteed to be able to add the nodes in $nodes[x+1]$ to its hierarchy because all of the nodes that preceed any of the nodes in $nodes[x+1]$ (ancestors, preceding siblings, ...), will require less time than the nodes in $nodes[x+1]$. As such they will be in $nodes[x]$ or $nodes[x-1]$ or ... , and are already in the Greedy hierarchy.

If G_n is not optimal, there would have to exist a hierarchy in H_n that launches faster than G_n. Let us label this faster hierarchy as $fast$. One way to define $fast$, is to describe how it is different from G_n. Let us create a function, $move(G_n)$, which will create the hierarchy $fast$ by moving a node in G_n to a new position (example: move a child node to a grand child position). If $Launch(G_n) = time[i]$, $move(G_n)$ can remove a node, g, from G_n where $Launch(g, G_n) \leq time[i]$, but the lowest place g can be moved to is a position in $nodes[i]$ or $nodes[i+1]$. If g is moved to a position in $nodes[i]$, $Launch(G_n) = Launch(fast)$. If g is moved to a position in $nodes[i+1]$, $Launch(G_n) < Launch(fast)$. Either way, $fast$ is not faster than G_n, so G_n is optimal.

□

4.3.3 Restrictions on Optimality

The Greedy hierarchy is only optimal under certain restrictions. The first restriction is that the Greedy hierarchy is only optimal when compared to hierarchies which are implemented using the three phases defined in Definition 2. There are ways to alter these three phases which could possibly result in faster launch times. For example, instead of sequentially creating separate processes to execute the individual launch commands,

a hierarchy of processes could be used. Additionally, the Greedy hierarchy launches co-located processes in the co-location phase. It might be possible to create a faster hierarchy, by intermixing the launch of remote and co-located processes.

A second restriction on the optimality of the Greedy hierarchy is that it assumes that REMOTE and SEQ are constant values throughout the launch. In reality, these values could increase as network and/or file system congestion increases. Furthermore, the physical network layout and machine differences may differentiate these values. Launching a child node that shares the same switch as its parent would be faster than launching a child that does not share its parent's switch. A hierarchy that accounts for these differences might prove to be faster.

4.4 Summary

In this Chapter we created a model of the time required to launch any process launching hierarchy. This model was used to create an optimal process launching hierarchy, the Greedy hierarchy. A proof to the Greedy hierarchy's optimality was presented and the restrictions of its optimality were discussed.

Chapter 5

Performance Analysis of LIBI

5.1 Overview

In this Chapter, we demonstrate the validity of the assertions made in previous Chapters. We perform tests that validate our model of bulk launch time. We provide evidence for the superior performance of the Greedy hierarchy. We use MRNet to show the performance gains of using LIBI to bootstrap distributed software.

5.1.1 Testing Platform

All of the experiments that are detailed in this Chapter were run on Lawrence Livermore National Laboratory's Atlas system. Atlas has 1,152 nodes, each of which contains 8 AMD Opteron 2.4 GHz CPUs. The nodes are interconnected via a double data rate InfiniBand network. The Atlas system is managed by the SLURM resource manager. The maximum jobs size is limited to 386 nodes.

5.2 Model Validation

Goals

The first experiment measures the actual performance of various launch hierarchies. There are three objectives of this experiment. First, it empirically tests the validity of the launch time model. Second, it empirically tests the optimality of the Greedy hierarchy. Third, it evaluates the performance of the LIBI group communication operations over varying connection hierarchies.

Methodology

The general procedure for this experiment involves using LIBI to launch a test application numerous times using various launch hierarchies. Once the test application is launched, it performs a couple of LIBI's group communication operations and exits.

This experiment has two independent variables: process count and hierarchy type. Since, only one process is used per node the maximum number of processes is limited by the max job size of 386 nodes. The launch hierarchies used in the experiment are the Chain, Sequential, Greedy, 2-ary tree, 16-ary tree, and the 32-ary tree. The Chain and Sequential trees are included as degenerate cases. Each experimental condition was run 10 times and averaged.

All test runs were executed on the same allocation of nodes. There are a few reasons for this decision. The main reason is that it simplifies the batch scheduling requirements. The second reason is that it ensures that all of the test runs occur in the same time period. This helps keep the levels of network congestion as similar as possible for every test run. The third reason is that it ensures that test runs do not run concurrently. This eliminates the possibility that the network utilization of one test run would interfere with another.

There are a few consequences of this decision that affect the experiment. The main consequence is that the test executable is left in the file cache between test runs. The local file cache is only cleared between allocations. This makes us use different parameters for the launch time model, than if the executable had to be retrieved from a file server. Since the goal of the experiment is to validate the launch model and to test the optimality of the Greedy hierarchy, adjusting the parameters is acceptable.

The Greedy hierarchy requires a few parameters to create a model of launch time (see Chapter 4). These parameters are PREP, REMOTE, SEQ, and COLOC. These parameters represent the time required by the Preparation phase, PREP, the time required between when the parent process issues the individual launch command and the subsequent child process is ready to launch its children, REMOTE, the time required by the parent process to go through one iteration of the "for(each child)" loop, SEQ, and the time required by the co-location phase, COLOC. Since PREP and COLOC are both constants we can combine them, creating a 3-tuple to represent the parameters of the model: < PREP+COLOC, SEQ, REMOTE >.

To generate the parameters for the Greedy hierarchy, we performed a launch with estimated parameters and timed the relevant portions of code. We then averaged all of these times. In this manner we generated the parameters: <0.022s, 0.015s,0.227s>.

The application we used for this experiment is a small standalone application, comprised of two executables. Testmain is a 277 KB executable, which calls the LIBI launch function. Testmember is the 238K executable which is launched, and then performs the group communication operations. Each of these executables were compiled using the archived LIBI library (as opposed to the shared object).

For each test run, four performance metrics were measured: launch, barrier, broadcast followed by a gather, and scatter followed by a gather. Launch was measured in testmain by timing how long LIBI_fe_launch() took to return. Barrier was measured in testmember

by the master process. This is a valid measurement because the individual launch version of LIBI implements barrier as a one byte broadcast followed by a 1 byte gather. Broadcast starts at the master and gather ends at the master. The third metric, broadcast followed by a gather, used a similar setup. The broadcast sent a 4 byte message (the size of an int) with the gather returning the same 4 bytes back to the master. The fourth metric, scatter followed by a gather, used this method too. By performing a broadcast or scatter followed by a gather, the timing is able to take place at the master. This alleviates the need to synchronize the clocks on the given computers.

Data

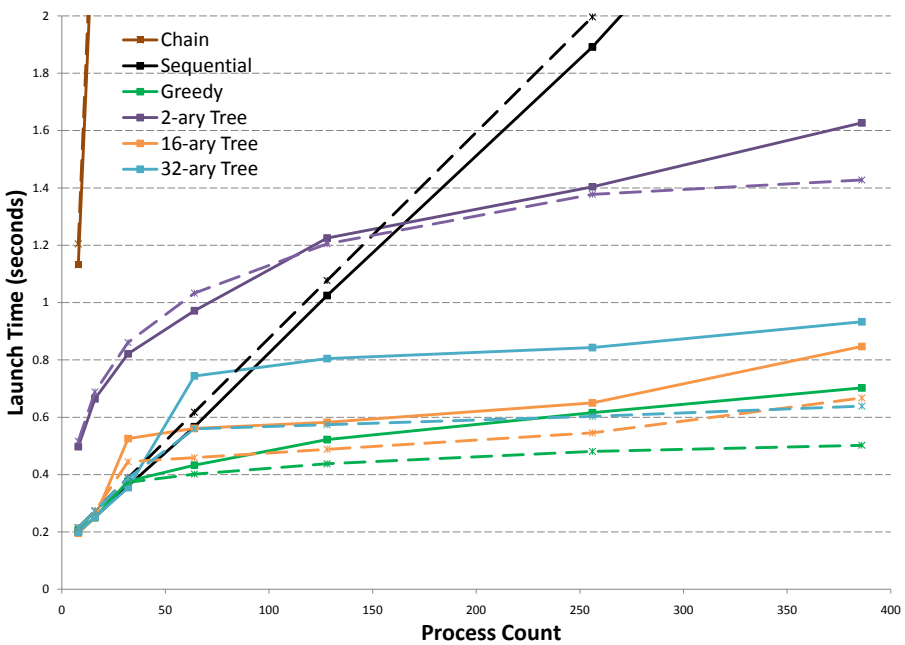
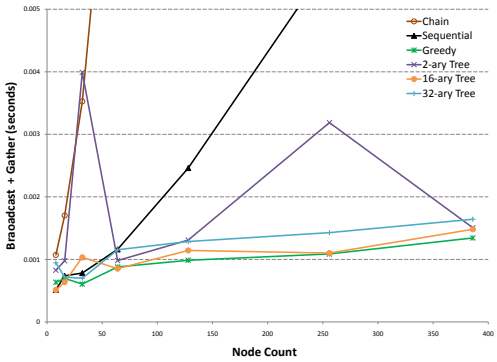
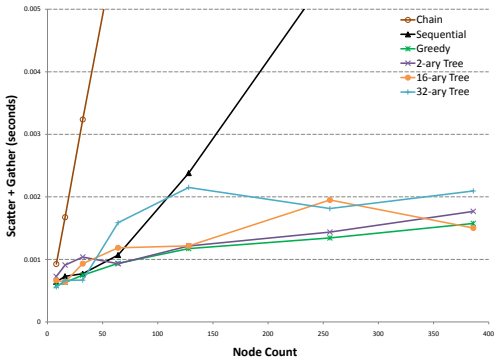


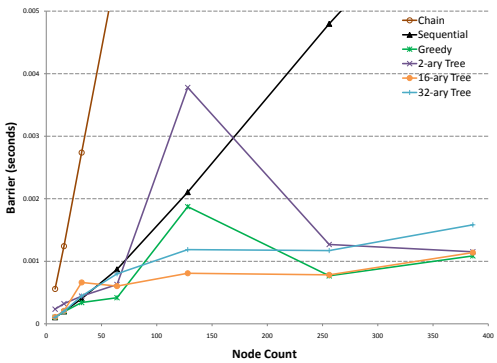
Figure 5.1: LIBI Launch Performance: The solid lines are measured values while the dashed lines represent the modeled launch time. The modeled launch times were created using the parameters: <0.013s, 0.007s, 0.172s>. This resulted in a coefficient of determination of $R^2 = 0.999$ between the modeled and the measured values.



(a) Broadcast + Gather



(b) Scatter + Gather



(c) Barrier

Figure 5.2: Comparing the performance of group communication operations on different connection hierarchies.

Analysis

In Figure 5.1, both the Sequential hierarchy and the Chain hierarchy went off of the graph. For 386 nodes, the Sequential hierarchy took 2.92 seconds, while the Chain hierarchy took 67.12 seconds.

In Figure 5.1, the parameters used to create the Greedy hierarchy were not the same parameters used to create the modeled launch times. The Greedy hierarchy used the parameters that were created based on the average times measured from a single test run.

When the modeled launch times that were created from these parameters were placed next to the actual launch times, it was clear that the model parameters were an over estimate. Even though they were an overestimate, the model produced from these values had a coefficient of determination of $R^2 = 0.886$. The parameters used in Figure 5.1 are the result of a least-squares fit of Equation 4.4 to the actual data.

Figure 5.1 shows the same relative performance of each launch hierarchy as their modeled launch time. Even with overestimated model parameters, the Greedy hierarchy performed the best, followed by the 16-ary tree, the 32-ary tree, the 2-ary tree, the Sequential hierarchy, and the Chain hierarchy. The only caveat occurs at smaller node counts. Due to the low node count, the Sequential hierarchy out performs each of the k-ary trees, up until a certain node count.

As mentioned in Chapter 3, the launch hierarchy pulls double duty as the communication hierarchy. Figure 5.2 depicts the performance of each communication hierarchy under group communication operations. The main observation is that the Greedy hierarchy performs about the same as all of the other hierarchies. The performance in this regard is based off of the values chosen for REMOTE and SEQ. They were chosen to optimize the launch hierarchy, not the communication hierarchy. Even still, if the ratio between REMOTE and SEQ was the same for launch as it was for communication, one could expect the group communication operations to be optimized under the Greedy hierarchy as well.

5.3 Modeled Launch Topology Performance

Goals

This experiment examines the effect of launch hierarchy on modeled launch time. We examine the modeled launch time because it is easier and faster to compute than timing the actual launch event. The purpose is to see a wide range of modeled launch times using

a large array of launch hierarchies. These modeled times will be used to guide the selection of the parameters for the MRNet case study.

Methodology

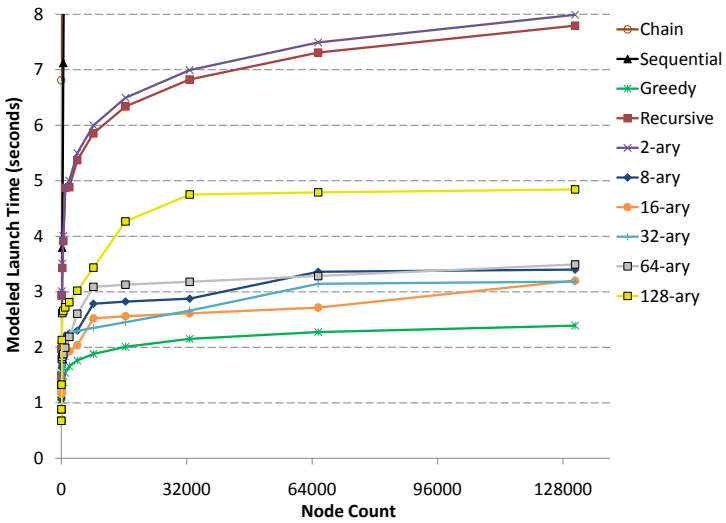
To create a single data point, we first create a launch hierarchy using the specified parameters. Using Equation 4.4 from Chapter 4, we model the time required to launch each process in the tree. The process requiring the most time to launch is used as the modeled launch time of the specified hierarchy.

The first independent variable is the node count. Since these are only modeled launch hierarchies, the node counts varied from $(2^4 - 1)$ to $(2^{17} - 1)$. These values correspond to the number of nodes in a full 2-ary tree of increasing depth.

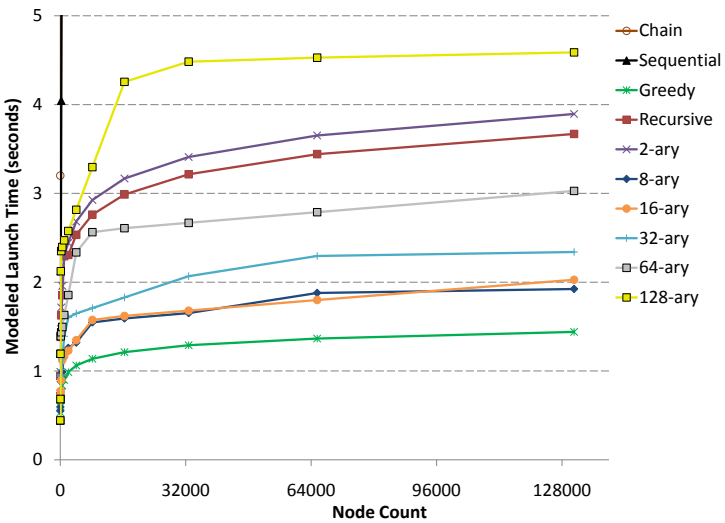
The second independent variable is the type of hierarchy. We covered most of the hierarchies discussed in previous Chapters. These hierarchies include Chain, Sequential, Greedy, Recursive Divide and Conquer, and a handful of k-ary trees. The k-values used were 2, 8, 16, 32, 64, and 128.

The final independent variable is the set of parameters used by Equation 4.4, <PREP+COLOC, SEQ, REMOTE>. The values that we used reflect two different launch environments. The first set <0.022s, 0.013s, 0.485s>, reflects launching a 1.6M executable when it is on an NFS server. The second pair <0.022s, 0.015s, 0.227s>, reflects launching a 155K executable when it is in the local file cache. These values were created by performing a single test run at 386 nodes, in each environment, timing the relevant portions of code, and taking the average.

Data



(a) parameters: <0.022s, 0.013s, 0.485s>



(b) parameters: <0.022s, 0.015s, 0.227s>

Figure 5.3: Modeled launch time with the executable on the server (a) and in the local cache (b).

Analysis

There are several observations worth noting in Figure 5.3. The first of which is the performance of the various hierarchies. The Greedy hierarchy was the best under both sets of parameters while the Chain and Sequential were the worst (both hierarchies hug the y-axis).

The second observation worth noting is how the performance of the k-ary trees change with the decrease of REMOTE. The smaller values of k change more than the larger values. This is readily apparent when comparing the 2-ary tree with the 128-ary tree. In Figure 5.3(a) the 2-ary tree takes almost twice as long to launch a hierarchy at any node count. In contrast, Figure 5.3(b) shows the 2-ary tree always launching faster than the 128-ary tree. The same relative-performance reversal can be seen between the 8-ary tree and the 32-ary tree. This is due to the fact that lower k-values have taller trees. This means their launch time is dominated by REMOTE more than higher k-values.

The third observations worth noting is that the performance of the recursive-divide-and-conquer hierarchy is incredibly similar to that of the 2-ary tree. This is because they share the same height. The only thing the recursive-divide-and-conquer hierarchy does better is that it increases the breadth of the tree, earlier than the 2-ary tree. This spreads the children out across more nodes, resulting in fewer along the critical path.

5.4 Case Study of MRNet

Goals

To further evaluate the bootstrapping performance of LIBI, we have implemented a version of MRNet that uses LIBI. Section 3.2.3 explains the detail of this implementation. We will compare MRNet's current bootstrapping mechanism to that of LIBI. The purpose of this

experiment is to evaluate LIBI under real world conditions.

Methodology

The general strategy of this experiment is to bootstrap MRNet numerous times, under varying conditions. There are three independent variables: process count, bootstrap mechanism, and MRNet fanout. The bootstrap mechanisms include the current version of MRNet and MRNet over LIBI. The MRNet Over LIBI case used both the LaunchMon version of LIBI and the individual launch version of LIBI. The LaunchMon version of LIBI used SLURM's bulk launch service, `srun`, in conjunction with COBO, a group communication service designed for bootstrap communication. The individual launch version of LIBI is further differentiated by using the Greedy, the 2-ary tree, the 16-ary tree and the Sequential hierarchies. The parameters used to create the Greedy hierarchy were given in Section 5.3 for the NFS server condition, <0.022s, 0.013s, 0.485s>.

Each test run was given its own allocation of 386 nodes regardless of the actual number of nodes needed. This accomplished several things. First, this means that each test run occupies $\frac{1}{3}$ of Atlas. This reduces the network congestion caused by other users. Second, this makes it unlikely that two test runs will run concurrently. Atlas is a fully-utilized machine, even at night and on the weekends. Third, the separate allocations mean that the file cache is cleared between each test run. There could still be some caching that occurs on the server, but this cannot be avoided.

The executables being launched include MRNet's communication daemon, mrnet_commnode, and our back-end daemon, test_launch_be. A distinct executable was compiled for the Current MRNet, the MRNet over LIBI, and the MRNet over LIBI over LaunchMon conditions. All of the executables were compiled using the archived version of the needed libraries. The size of each executable is listed in Table 5.1.

We ran two tests using MRNet. The first test kept MRNet's fanout constant at 16, while

	mrnet_commnode	test_launch_be
Current MRNet	1.5M	1.5M
MRNet over LIBI	1.6M	1.6M
MRNet over LIBI over LaunchMon	3.9M	3.9M

Table 5.1: Executable sizes of each test condition.

varying the process count. The process count includes MRNet's communication daemons as well as our back-end daemons. Varying the process count will show the scalability of each launching mechanism, when bootstrapping MRNet. The second test kept the process count constant at 3080, while varying MRNet's fanout. This will show the affect of MRNet's fanout on it's bootstrapping performance.

To mitigate the affect of intermittent network congestion on a single test case, each bootstrapping condition was executed in order, for each test condition. This sequence of test runs was repeated three times, and averaged per test case. To increase the process counts, all test were ran with 8 processes per node.

Analysis

Figure 5.4 (page 58) shows the scalability of each bootstrapping condition when MRNet has a fanout of 16. MRNet over LIBI performs the best, for all launch hierarchies, 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.

Figure 5.5 shows the breakdown of the MRNet bootstrap timings for the current version of MRNet, MRNet over LIBI using the Greedy hierarchy, and MRNet over LIBI over LaunchMon. The functionality related to each Section is described in Table 5.2. The cause of the linear scaling is apparent when viewing the timing breakdown. The largest portion of MRNet's bootstrapping is the "Parse MRNet Topology" component and the "MRNet

TBON Formation". Both of these tasks scale linearly, each taking approximately 0.009 seconds per node.

As for LIBI, the biggest component involves translating the MRNet topology back into a host list. This is the "Preparation for LIBI" component, which scales linearly, taking about 0.004 seconds per node. The "LIBI Greedy Launch" component scales less than linearly and the "LIBI Communication" component never took more than an eighth of a second.

The most surprising result is the "LaunchMon over SLURM Launch" component. This time was a lot higher than expected. Considering that SLURM uses persistent daemons, one would expect that the SLURM launch time would be smaller than the LIBI 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 (see Table 5.1). 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 would require some extra communication. However, considering that the "LIBI Communication" component is insignificant, any communication costs incurred by these plugins would probably also be insignificant. Third, LaunchMon is using SLURM for bulk launching and COBO for communication. During the launch phase COBO has to connect all of the daemons. The launch and connection tasks are seperate under LaunchMon while they are intermixed for the individual launch version of LIBI.

Figure 5.6 shows the results of changing MRNet's fanout, while holding the process count constant at 3080. Here we see that the bootstrapping time of the current version of MRNet changes with the fanout, but MRNet over LIBI remains relatively constant. This alleviates most of the bootstrap performance concerns when choosing an MRNet topology.

Figure 5.7 shows the break down of the Figure 5.6 timings. The "Parse MRNet Topol-

ogy" component was removed because it was the same for both the current version of MRNet and MRNet over LIBI. This also serves to highlight the comparison between the relevant portions of code.

The only significant difference in MRNet over LIBI performance occurs with an MRNet fanout of two. Here, both the "Preparation for LIBI" and "MRNet TBON Formation" are smaller than the other fanouts, while the "LIBI Greedy Launch" is just slightly larger. One potential reason for why the "MRNet TBON Formation" component is smaller is because each parent only has to accept two child connections. This means there is a reduced chance of network resource contention. The "LIBI Greedy Launch" is slightly larger due to the ratio between mrnet_commnode and test_launch_be processes. With a fanout of two, $\frac{1}{2}$ of MRNet's total processes are mrnet_commnode. With a fanout of eight, only $\frac{1}{8}$ of MRNet's total processes are mrnet_commnode. Due to the scalability of the Greedy hierarchy, launching a large topology and a small topology is faster than launching two equal sized topologies.

5.5 Summary

In this Chapter, we validated our model of bulk launch time. We then demonstrated the superior performance of the Greedy hierarchy. We then demonstrated the increased bootstrapping performance of MRNet when using LIBI.

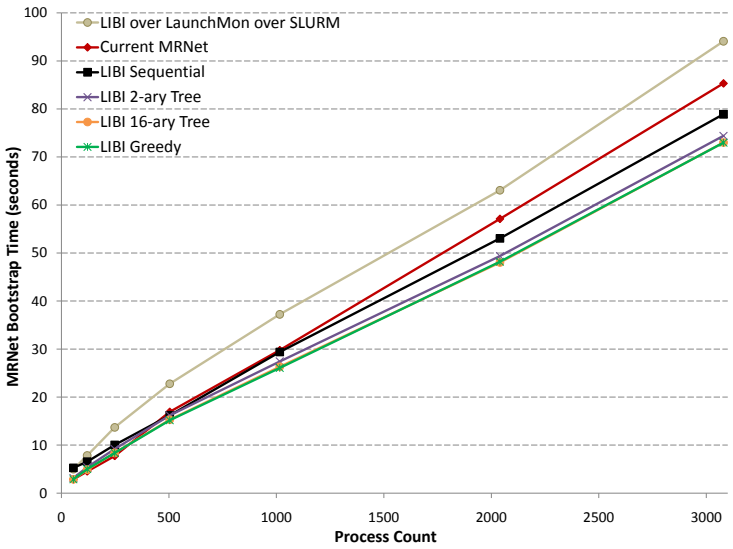


Figure 5.4: MRNet Bootstrap Time vs. Process Count. MRNet has a fanout of 16.

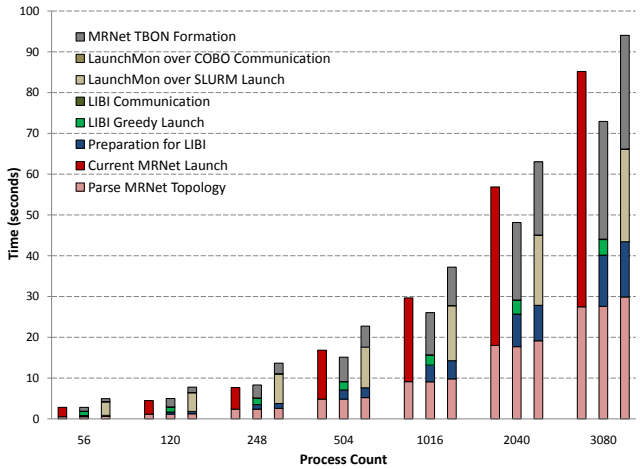


Figure 5.5: A Breakdown of MRNet's Bootstrap Time vs. Process Count. MRNet has a fanout of 16. From left to right the columns are Current MRNet, MRNet over LIBI, and MRNet over LIBI over LaunchMon.

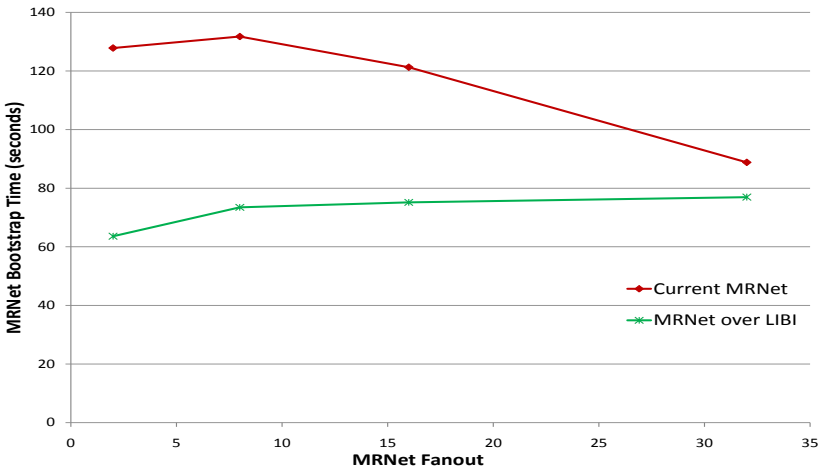


Figure 5.6: MRNet Bootstrap Time vs. MRNet Fanout. Using a total of 3080 processes.

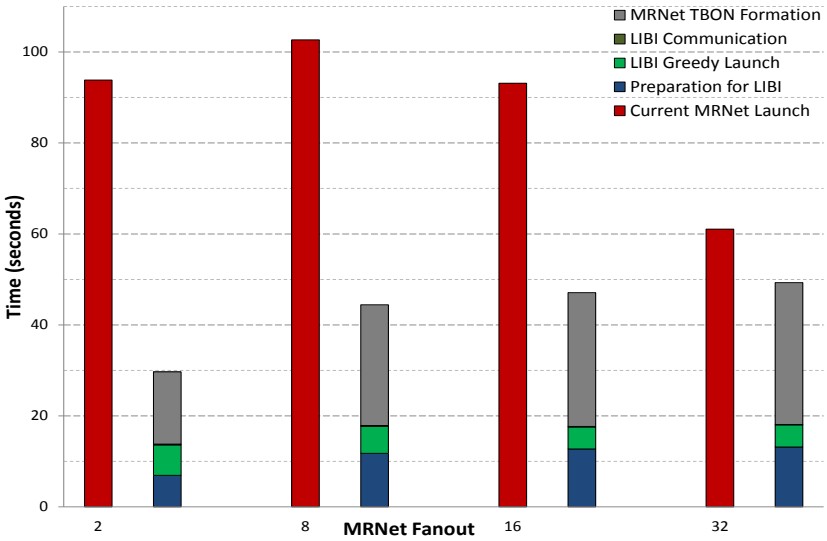


Figure 5.7: A Breakdown of MRNet's Bootstrap Time vs. MRNet Fanout. Using a total of 3080 processes. Left columns are the current MRNet version while the right are MRNet over LIBI.

Name	Description
MRNet TBON Formation	Connect MRNet child processes to their parent.
LaunchMon over COBO Communication	Distribute the connection setup information to all of the nodes, using COBO.
LaunchMon over SLURM Launch	Launch the mrnet_commnode and test_launch_be processes on the requested nodes, using `srun`.
LIBI Communication	Distribute the connection setup information to all of the nodes, using LIBI's Greedy hierarchy.
LIBI Greedy Launch	Launch the mrnet_commnode and test_launch_be processes on the requested nodes, using LIBI's Greedy hierarchy.
Preparation for LIBI	Translate MRNet's topology representation into a host list for LIBI.
Current MRNet Launch	The current version of MRNet intermixes the launch, communication, and TBON formation.
Parse MRNet Topology	Parse MRNet's topology file.

Table 5.2: Description of the components in Figure 5.5 and Figure 5.7.

Chapter 6

Conclusions

The goal of this thesis was to improve the current state of software-system bootstrapping. Our approach was to create a separate entity which could facilitate bootstrapping portability for software-systems as well as provide the best performance available on a given platform. In this final chapter, we summarize the contributions of this thesis and describe some future directions of this work.

6.1 Contributions

In this thesis, we created a system for classifying bulk-launch strategies. A bulk-launch strategy can be classified based on the framework persistence and the type of connection hierarchy used between the daemons. This classification system allows for the exploration of new bulk-launch strategies.

In this thesis, we created an optimal-process-launching hierarchy. We started by creating a model of launch time. Using this model we constructed a greedy algorithm that produced the Greedy hierarchy. We then proved that the Greedy hierarchy is optimal under certain conditions.

In this thesis, we have demonstrably improved the bootstrapping performance of MR-

Net through the use of LIBI. This is the culmination of all the contributions of this thesis. MRNet gained portability as well as performance through the use of LIBI.

6.2 Future Work

Our goal of improving distributed software bootstrapping is not complete. There are still some areas which can be improved.

Automatically Generating Model Parameters

Currently, LIBI relies on user supplied parameters when creating the Greedy hierarchy. To create the parameters for the tests in Chapter 5, we had to perform an unoptimized test run first, timing the relevant portions of code. Judging by the disparity between these estimated parameters and the least squares fitted parameters in Section 5.2, this approach was not accurate.

A better approach would be to generate the REMOTE and SEQ parameters during the actual launch. Before creating the launch hierarchy, LIBI could launch a single process and time the relevant portions of code. With these application specific values, the performance of the Greedy hierarchy should improve. This would also remove from the user, the burden of choosing model parameters.

Large Multi-Core Computer Optimization

One corollary of the trend of ever increasing processors counts in extreme-scale systems, is the trend of increasing processor counts in individual computers. The platform that we tested on only had 8 cores per node. It doesn't take a huge leap of imagination to envision nodes which have 10s or 100s of cores themselves.

This trend of increasing numbers of processing cores in a single computer, opens up an avenue of optimization for the Greedy hierarchy. As discussed in Section 4.3.3, the Greedy hierarchy's current approach is to *sequentially* create separate processes to execute the individual launches for a node's children. The scalability of this approach should mimic that of the Sequential process launching hierarchy. At small scales sequentially creating new local processes is relatively fast. However at larger scales, creating a hierarchy of local processes will be faster. This approach should prove to be faster, up to the point where the network interface is saturated.

Minimizing the Impact of the File System

One aspect of process launching that we did not address in this thesis, is file system resource contention. During the process launching phase, every node in the hierarchy requests an executable from the file system. This can create a bottleneck at the network file server.

There are several ways to address the scalability of the file system. The Lustre File System uses a concept called "collaborative caching" [24]. Once a node has received a file, it can then be used to service I/O requests for that file. Another approach is to bypass the file system entirely. The bulk launching service on CPlant, Yod, does this [7]. Yod, establishes connections to the persistent daemons on each node, and then uses these connections to scalably disseminate the executable.

In a similar fashion LIBI could be combined with the Scalable Binary Relocation Service (SBRS) to minimize the impact of the file system [16]. LIBI could launch its own daemon on each node, from an optimized executable. Since this executable would be used by multiple jobs, it is more likely to be cached closer to the nodes. The LIBI daemon could then be used to disseminate the larger user applications and libraries.

Further MRNet Bootstrap Optimizations

As noted in Chapter 5, there are still some large bottlenecks in MRNet's bootstrapping process. The largest of which is parsing the MRNet topology. During this portion of bootstrapping, MRNet converts the topology file to a class-based topology definition.

Instead of starting with a class-based topology definition and then converting this into a host lists for LIBI, MRNet could directly parse the topology file into a host list. At the end of bootstrapping, the class based topology definition could be assembled from the MRNet TBON. This would turn a linear process into a parallel one, alleviating the bottleneck.

Refactoring LaunchMon

Currently LIBI sits on top of LaunchMon. We believe this relationship should change. As LIBI is improved through additional optimizations of its individual launch version and the inclusion of HPC vendor and resource manager specific versions, LaunchMon may benefit from sitting on top of LIBI.

References

[1] D. H. Ahn, D. C. Arnold, B. R. d. Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *37th International Conference on Parallel Processing*, ICPP '08, pages 578–585, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Advanced Institute for Computational Science. http://www.aics.riken.jp/index_e.html (visited June 2011).

[3] B. Bacarisse. The Cartesian Product of a Finite Number of Countable Sets is Countable. http://planetmath.org/?op=getobj&from=objects&id=7142 (visited July 2011).

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: a scalable parallel process-management interface for extreme-scale systems. *Recent Advances in the Message Passing Interface*, pages 31–41, 2010.

[5] N. M. J. Barraz, E. Salcedo, and M. C. Barbosa. Thermodynamic, Dynamic, Structural and Excess Entropy Anomalies for core-softened potentials. *The Journal of Chemical Physics*, 135(10):1–11, 2011.

[6] BlueGene/L. https://asc.llnl.gov/computing_resources/bluegenel/ (visited June 2011).

[7] R. Brightwell and L. A. Fisk. Scalable parallel application launch on Cplant. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[8] 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.

[9] COBO. http://sourceforge.net/projects/launchmon/ (visited July 2011).

[10] M. Fan, J. Stallaert, and A. B. Whinston. A web-based financial trading system. *Computer*, pages 64–70, Apr. 1999.

[11] J. Goehner, D. Arnold, D. Ahn, G. Lee, B. de Supinski, M. LeGendre, M. Schulz, and B. Miller. A Framework for Bootstrapping Extreme Scale Software Systems. In *Workshop on High-performance Infrastructure for Scalable Tools*, Tucson, Arizona, 2011.

[12] S. Gooding, L. Arns, P. Smith, and J. Tillotson. Implementation of a Distributed Rendering Environment for the TeraGrid. In *2006 IEEE Challenges of Large Applications in Distributed Environments*, pages 13–22. IEEE, 2006.

[13] A. Gupta, G. Zheng, and 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*, pages 41–48, New York, New York, USA, 2011. ACM Press.

[14] M. A. Jette and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *ClusterWorld Conference and Expo*, San Jose, California, June 2003.

[15] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing. The Application Level Placement Scheduler. In *Cray User Group*, pages 1–7, 2006.

[16] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208K: Towards debugging millions of cores. *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, 27344:1–9, Nov. 2008.

[17] E. Lusk. Integrating Scalable process management into component-based Systems Software. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 16–22, 2003.

[18] R. Morishima, J. Stadel, and B. Moore. From planetesimals to terrestrial planets: N-body simulations including the effects of nebular gas and giant planets. *Icarus*, 207(2):517–535, June 2010.

[19] Message Passing Interface Forum. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html (visited June 2011).

[20] J. Park, H. Choi, N. Nupairoj, and L. Ni. Construction of optimal multicast trees based on the parameterized communication model. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 180–187. IEEE Comput. Soc. Press.

[21] PMGR_COLLECTIVE. http://sourceforge.net/projects/pmgrcollective/ (visited June 2011).

[22] PMI-2 API. http://wiki.mcs.anl.gov/mpich2/index.php/PMI_v2_API (visited June 2011).

[23] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 21, Phoenix, AZ, November 2003. IEEE Computer Society.

[24] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, pages 380–386. Citeseer, 2003.

[25] J. Sridhar and D. Panda. Impact of Node Level Caching in MPI Job Launch Mechanisms. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 230–239, 2009.

[26] J. K. Sridhar, M. J. Koop, J. L. Perkins, and D. K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *15th International Conference on High performance Computing*, HiPC'08, pages 323–335, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Top 500 Supercomputer Sites. http://www.top500.org/ (visited June 2011).