LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-502951

An Optimal Process Launching Strategy for Extreme Scale Bootstrapping

J. D. Goehner, D. C. Arnold, D. H. Ahn, G. L. Lee

October 4, 2011

26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)
Shanhai, China
May 21, 2012 through May 25, 2012

Disclaimer

An Optimal Process Launching Strategy for Extreme Scale Bootstrapping

Joshua D. Goehner and Dorian C. Arnold
University of New Mexico
Albuquerque, New Mexico, U.S.A.
[jgoehner,darnold]@cs.unm.edu

Dong H. Ahn and Gregory L. Lee
Lawrence Livermore National Laboratory
Livermore, California, U.S.A.
[ahn1,lee218]@llnl.gov

***Abstract*—All software systems that run on high-end computing machines go through what we call a bootstrapping phase. These software systems are deployed onto a set of computing nodes and their initialization information is disseminated. Efficient bootstrapping is essential for extreme-scale machines. Many workloads on these machines depend on fast bootstrapping. In particular, this phase is increasingly on the critical path for interactive tools and newly emerging many-task computation models. Previously, we implemented the Lightweight Infrastructure-Bootstrapping Infrastructure (LIBI) to support extreme-scale software systems during their bootstrapping phase. In this paper, we propose a LIBI algorithm that creates an optimal process launching strategy. Our algorithm takes advantage of a novel performance model for process launching strategies, to find the optimal strategy given a specific computing system with a set of resources. Our proof shows that the recommended process launching strategy is optimal. Our performance evaluations demonstrate that our algorithm decreases the time for LIBI to bootstrap a software system by up to 50%.**

***Keywords*-Runtime systems; system software; large scale software systems**

I. Introduction

High-end or high performance computing (HPC) systems have grown tremendously in core counts over the past decades, and this trend is expected to continue in the years to come. On the most recent Top 500 list [1], a list released twice every year chronicling the fastest, publicly-known computing systems, 44.6% or almost half of the entries have more than 8,192 cores, compared to only 3.0% just 5 years ago. The list contains four systems with more than 200K cores, the largest of which has 548,352 cores. Sequoia, the first million core system, is expected to be delivered in 2012 with 1.6 million cores [2], and future extreme-scale systems are projected to have on the order of tens to hundreds of millions of cores by 2020 [3].

Most of the software-based applications, tools and system services to be deployed on these systems must scale to full system size to be effective. An important phase that affects scalability is the software startup or *bootstrapping* phase. As we depict in Figure 1, we define distributed-software infrastructure bootstrapping as the procedures of instantiating the infrastructure's processes on allocated computational nodes and delivering the initialization information necessary for these processes to complete their setup and to enter their primary operational phases[1].

An inefficient bootstrapping process can have a negative impact on any large-scale software system. There are several cases when this inefficiency can become an impediment to software deployment and usefulness. For example, there is a need for interactive, scalable HPC tools and services to help in the detection, analysis, diagnosis and remediation of functional and performance problems. It can be the case, where the time it takes to deploy a tool is several orders of magnitude longer than the time it takes for the tool to perform its key function. Our experiences with our own stack trace analysis tool (STAT) [4] demonstrated this problem: a full-scale instance of STAT on the Lawrence Livermore National Laboratory's BlueGene/L system could take minutes to start-up and subsequently, less than a single second to perform its analysis.

Another example of the impedance of inefficient bootstrapping is in the many-task computational model. In this model of computing, an application is decomposed into many (thousands and sometimes millions of) tasks and these tasks are mapped to processes; processes are continuously launched on the available computational resources throughout the application's entire execution. Therefore, process bootstrapping can have a big impact on the performance of the entire application run. Additionally, the more fine-grained the tasks, the greater the impact of inefficient bootstrapping.

In this paper, we focus on the process launching phase of bootstrapping. Indeed, there exist many resource managers (RMs) like LoadLeveler [5], LSF [6], PBS [7] and SLURM [8] for *bulk process launching*. such systems leverage ad hoc strategies for process launching. Our primary goal is to devise an optimal strategy for process launching and to understand the impact of finding and using an optimal strategy. In this context, our work makes several contributions:

1) We have created a performance model for process launching strategies;

[1]Technically, bootstrapping is not complete until every process has acted upon its setup information; this final activity is indeed infrastructure dependent.

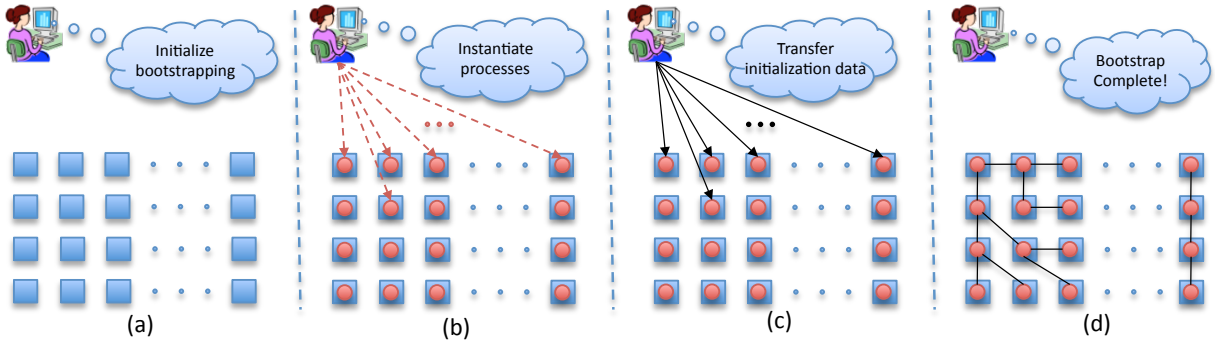


Figure 1. Distributed Application Bootstrapping: instantiating the distributed application's processes and propagating to these processes their requisite initialization information.

2) We have designed an algorithm for finding the optimal process launching strategy given a specific system and set of resources;
3) We present a proof that the recommended process launching strategy is optimal;
4) We evaluate our algorithm's costs for finding an optimal strategy; and
5) We evaluate the performance impact of using an optimal strategy for process launching versus an arbitrarily chosen strategy.

Our results show that the cost of using our algorithm to determine optimal process launch strategies is very cheap even for 100K node counts and can yield significant performance improvements over arbitrary or ad hoc strategies. Furthermore, the impact of ad hoc strategies can be exacerbated when process launch is based on a slow sequential mechanism, like `ssh` as compared to `rsh`. While this work has the greatest impact in the context of systems for which a bulk launching service that uses persistent daemons is unavailable or undesirable, this work can also help to inform the inter-connection topologies used for such bulk launchers.

The rest of this paper is outlined as follows. In Section II, we describe the bulk process launching concept and summarize related research and development projects. In Section III, we describe our process launch performance model. In Section IV, we also present our algorithm for finding an optimal process launch strategy and offer a proof of optimality. In Section V, we present our experimental results that validate our performance model, validate our algorithm's optimality and evaluates our process launch strategy using a real infrastructure. Finally, we conclude with a summary of the contributions and potential impact of our research.

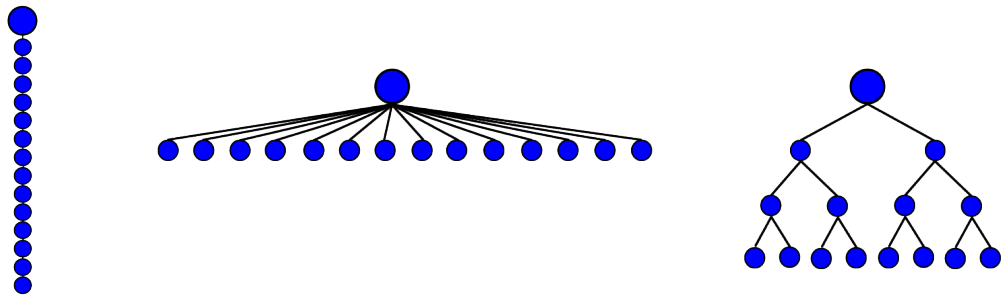
II. Bulk Process Launching

There are two models for process launching: *individual* and *bulk*. The difference between the individual and the bulk launch models is the number of processes that are capable of being launched in a single request. Individual launch mechanisms are capable of launching only a single process at a time while bulk launchers have the capability of launching multiple processes. The most popular individual launch mechanism is a basic remote process creation mechanism based on `rsh` or `ssh`. There are a myriad of bulk launch services with different launching strategies as we describe below. Indeed, some of these services may be built on top of individual launch mechanisms.

Formally, the bulk process launching problem can be defined as: given a set, P, of processes, a set, N, of nodes and a mapping, M, that maps every element of P to exactly one element of N, each process in P must be instantiated or launched on its corresponding node in N. In this work, we assume the mapping is *injective* and *surjective* or *one-to-one* and *onto*. In practice, the mapping function may be *non-injective*; that is, multiple processes may be mapped to the same node. However, in this work, we consider the problem of launching a single process onto each node: launching co-located processes can be done more efficiently by using the first launched process on each node as a local launching agent than by using a remotely located launching agent [9], [10]. A non-surjective mapping function, one in which not every node has a process mapped to it, is possible but does not make sense in practice. This would render unused nodes in an allocation.

A. Bulk Process Launching Trees

We define a bulk process launching tree, T, as a tree of processes such that a parent-child relationship in the tree denotes a launcher-launchee relationship in the bulk launch process. (The topology is a tree since each process can only have a single creator.) Examples of different process launching trees are shown in Figure 2. The topology of the tree inherently determines the potential efficiency of bulk process launching: disjoint branches of the tree can be launched concurrently; however, processes that share an ancestral relationships have inherent launching dependences.



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

Figure 2. Three distinct launch trees that start 15 processes.

B. Bulk Process Launching Frameworks

There are several frameworks that provide a bulk process launch service. Generally, high-end hardware vendors provide their own specialized resource management and control system which includes bulk-launch services. For example, IBM's BlueGene family implements its own control system with bulk-launch services via a set of commands such as `mpirun` [11], and Cray's XT and XE family also provides similar services through ALPS and `aprun` [12]. In addition, there exists a myriad of other, more generic resource management software stacks that provide bulk process launching capabilities: e.g., LoadLeveler [5], LSF [6], PBS [7] and SLURM [8]. We will discuss these frameworks with respect to their process launching strategies.

While some bulk launch services use less scalable topologies for process launch, like MPD, which employs a process ring [13], other bulk launch services have incorporated tree topologies into their process launching strategies [8], [9], [10], [12]. Some of these do leverage less scalable flat trees, but some do use a k-ary trees. In all cases, the choice of launching topology, even if configurable, typically is ad hoc.

III. Modeling Bulk Process Launching

We now construct a model for bulk process launch latency, the time that lapses between the initiation of the bulk process launch request and the creation of the last process. This model only considers launching a single process per node. Given a process launch tree and other relevant parameters, our model outputs launch latency. More specifically, our model assumes each parent sequentially creates its respective children and processes in disjoint branches create their children concurrently as outlined by the algorithm in Figure 3, which is executed by each process in the tree.

There are other aspects of bulk process launch latency that are outside of the scope of our model. For example, our model does not consider the time required to devise a process launch tree nor does it consider the time required to launch co-located processes. The repercussions of these exclusions are discussed in Section IV-D.

```
launch( children ):
SEQ:  for each child in children, do
REM:   create child on relevant node
```

Figure 3. Pseudo code for our bulk process launching model.

In a process launch tree, creation of child nodes is ordered from left to right: for all x greater than or equal to 1, $child_x$ must exist in order before can be a $child_{x+1}$. We construct our model for arbitrary trees leveraging base models for the simple chain and flat trees. With the observation that generic trees are just recurrences of these basic trees, our model for an arbitrary tree is a recursive composition of these basic models.

A. Modeling Chain Trees

In a tree with a *chain* topology, every node except the last has exactly one child. This can be thought of as a 1-ary tree. To model a chain, we consider the line in Figure 3 labeled `REM` or *remote launch time*. This encompasses the time that lapses between the invocation of the creation command at a process' parent and the time the newly created child is ready to launch its first child.

As demonstrated in Figure 4, for a chain tree of n processes, `REM` is repeated $n-1$ times, the number of ancestors of the last node in the tree. Generally, the time it takes to launch a process p in a chain tree of n processes, $launch(p, chain_n)$ can be formalized as:

$$launch(p, chain_n) \quad = \quad |anc(p, chain_n)| * REM \quad (1)$$

where $anc(p, t)$ is the set of ancestors of process p in tree t, and $chain_n$ is a chain tree with n processes.

B. Modeling Flat Trees

In a tree with a *flat* topology, the root is the only parent node – for process launching, the root process launches every other process. To model a flat tree, we consider the line in Figure 3 labeled `SEQ` or *sequential wait time*.

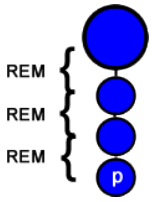


Figure 4. Modeling the launch of process p in a chain tree with 4 nodes.

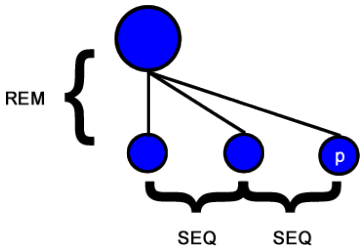


Figure 5. Modeling the launch of process p in a flat tree with 4 nodes.

This encompasses the necessary time delay between two subsequent launch commands.

As demonstrated in Figure 5, for a flat tree of n processes, SEQ is repeated $n-2$ times, the number of siblings of the last (rightmost) node in the tree assuming children are ordered from left to right. Intuitively, the time it takes to launch the last process p in a flat tree with n nodes is the delay for p's parent to launch all p's preceding siblings plus the remote launch time it takes p's parent to launch p. This can be formalized as:

$$\begin{aligned} launch(p, flat_n) &= (|presib(p, flat_n)| * SEQ) \\ &\quad + REM \end{aligned} \tag{2}$$

where $presib(p,t)$ is the set of preceding siblings of process p in tree t and $flat_n$ is a tree hierarchy with n processes.

C. Modeling Arbitrary Trees

Finally, to model the process launch of arbitrary trees, we construct a model for determining the time necessary to launch any process within any given tree. Therefore, this encompasses balanced trees and skewed trees as well as chain and flat trees. For an arbitrary tree, we still rely on *remote launch time*, REM and *sequential wait time*, SEQ. However, we must identify the number of repetitions of these components for any given process in any given tree.

To identify the number of repetitions of REM and SEQ, we define a recursive model. To build this recursive model, we observe that every child process in a given tree can be considered as the child of a flat tree rooted at the child's parent. So intuitively, the time to launch that child would be the sum of the time required to launch the child's parent and the time to launch the child in the flat tree rooted by the parent. Generally, the launch time of process p, a child of process x in tree T is:

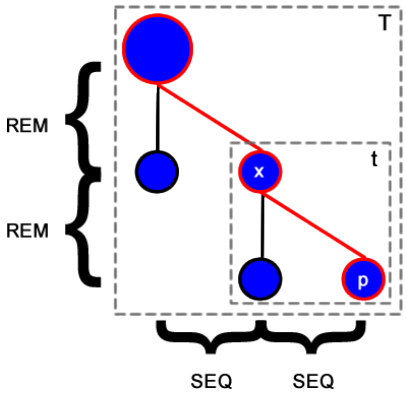


Figure 6. Modeling the launch of process p in an arbitrary tree of 5 nodes.

$$\begin{aligned} launch(p, T) &= launch(x, T) + launch(p, t) \\ &= launch(x, T) + \\ &\quad (|presib(p, t)| * SEQ) + \\ &\quad REM \end{aligned} \tag{3}$$

where t is the flat tree rooted at process x.

As demonstrated by Figure 6, if we execute this recursion up to the root of the entire tree, we see that REM must be repeated for each ancestor of p in T: a process cannot be created until its parent process is created; a process' parent cannot be created until the parent of the process' parent has been created and so on. We also observe that the creation of the chain of processes on the path from x to the root of the tree T are all delayed by the sequential delay, SEQ each parent on the path must incur while creating the preceding siblings of process p and the preceding siblings of each ancestor of process p. Generally, the time to launch any process p in a tree T is:

$$\begin{aligned} launch(p, T) &= (|psas(p, T)| * SEQ) \\ &\quad (|anc(p, T)| * REM) \end{aligned} \tag{4}$$

where $psas(p,T)$[2] is the set that includes the preceding siblings of process p and the preceding siblings of each ancestor of process p in T.

Finally, we define T_n to be the set of all trees of n nodes. The time required to launch tree $t \in T_n$ is defined as:

$$launch(t) = max_{p=1}^{n}\ launch(p, t) \tag{5}$$

IV. Optimal Process Launch Trees

For a given number of nodes, there are a vast number of possible process launch trees. In this section, we present an algorithm that inputs a process launching problem and outputs an optimal process launch tree. By optimal, we mean that the output tree is guaranteed to launch the given

[2]"psas" is shorthand for preceding siblings of self and ancestors.

	100	1000	10,000	100,000
16-ary tree	1.8E-4	1.8E-3	1.8E-2	1.9E-1
greedy tree	2.5E-4	2.1E-3	2.3E-2	2.8E-1

Table I
TIME (SECONDS) TO CREATE A PROCESS LAUNCH TREE OF A GIVEN NODE COUNT.

processes on the given nodes in a minimal amount of time for a particular system. We also present a proof of our algorithm's optimality.

A. The Greedy Tree

We use a greedy algorithm to find an optimal process launch tree[3]. Therefore, we call the resulting process launch tree the *greedy tree*. Our greedy algorithm is inspired by the construction of optimal-multicast trees described by Park et al. [14]. Based on a model that incorporates the necessary delay between subsequent transmissions from a single process and inter-process communication latency, Park et al used a dynamic-programming algorithm to create optimal-multicast trees by combining smaller optimal trees into larger optimal ones. Our greedy algorithm is based on similar parameters that have been adapted to process launching.

B. Greedy Algorithm

Figure 7 shows the pseudo code of the greedy algorithm that creates the greedy tree. The algorithm takes a set of nodes as input and returns a greedy tree to be used for launching a set of processes, one per node, on each node in the input set. As we discuss in Section IV-D, we assume that the time for a process on any node to launch a process on any other node is constant. This means we can treat all nodes in the input set equally, and the order in which we process nodes from the set does not matter. The first node from the set is placed in the root position of the tree. At each iteration of the algorithm, the launch time for the next child of every node in the tree is modeled. Subsequent nodes are processed iteratively from the input set and are added to the tree by greedily choosing the position in the tree with the smallest modeled launch time.

To keep track of the available positions, we 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. Due to the large number of positions with non-unique launch times, the algorithm is closer to $\Theta(n)$ in practice. Table I gives a comparison of creation times between a greedy tree and a 16-ary tree. In absolute values, the cost of executing the greedy algorithm on large numbers of nodes is small, particularly when compared to the actual times to launch processes on these large numbers of processes, as we see in Section V.

[3]There may be multiple process launch trees that satisfy the minimum launch time requirement.

C. Proof of Optimality

Intuitively, the greedy algorithm should produce an optimal process launching tree, since, based on our assumptions, all nodes are equal, and the algorithm places new nodes in the tree in a position that results in the fastest launch time. The algorithm tries to maximize the productivity of each process, which results in a minimal launch time. If a parent process is idle for long, its next child position eventually will become the best next child position in the tree. If there remain child processes to be created, at that point the parent process will be assigned a child process to launch.

Our proof that the greedy algorithm produces an optimal process launch tree is as follows: first, we prove that given a set of nodes on which to launch processes, the range of possible launch times is discrete (Lemma 2). Since there is a discrete range of possible launch times, there is a total ordering of these times, lowest to highest. Finally, we show that the greedy algorithm will saturate the lowest unsaturated launch time, before moving on to the next lowest unsaturated launch time.

Operator	Description
REM	REM is the constant amount of time required between the instant a parent process begins to launch a child process and the instant the created child process is ready to create its first child.
SEQ	SEQ is the constant amount of time required at a parent process between the instants that process can create two subsequent children processes.
T_n	T_n is the set of all trees containing n nodes.
$anc(p,t)$	$anc(x,t)$ returns the set of ancestors of node x in tree t. These are the set of nodes on the path from the root to x, including the root but not x.
$psas(x,t)$	$psas(x,t)$ returns the set of nodes that includes all the preceding siblings of each ancestor of node x and all the preceding siblings of node x in tree t
$available(t)$	$available(t)$ returns the set of nodes that includes the next available child position of each node in tree t. If node x has 3 children, node x's next available child position is 4.
inf	inf is the *infinite tree*, in which every node has an infinite number of children. This tree is larger than any other tree that can exist.
$time[i]$	$time[i]$ is the i^{th} lowest value of $range(launch(inf))$.
$nodes[i]$	$nodes[i] = \{x : launch(x,inf) = time[i]$ and $x \in inf\}$ $nodes[i]$ is the set of positions in the infinite tree that will launch in $time[i]$.

Definition 1. 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 $A \times B = \{(a,b) : a \in A\ and\ b \in B\}$.

```
for( each node ):
  Place node in position with the smallest modeled time
  Calculate the modeled time of the next sibling
  Calculate the modeled time of the first child
```

Figure 7. Pseudo code for our greedy algorithm that creates an optimal process launch tree.

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 [15]. $A \oplus B$ 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, $launch(t)$ equals the maximum launch time of the nodes in tree t. Let us label the node that takes the longest time as max.

$$
\begin{aligned}
& range(launch(t)) \\
& \quad = range(launch(max, t)) && \text{(Eqn. 5)} \\
& \quad = range(\ |psas(max, t)| * SEQ \\
& \qquad + |anc(max, t)| * REM\ ) && \text{(Eqn. 4)} \\
& \quad = range(|psas(max, t)| * SEQ) \\
& \qquad \oplus range(|anc(max, t)| * REM) && \text{(Defn. 1)} \\
& \quad = \{nat * SEQ : nat \in \mathbb{N}\} \\
& \qquad \oplus \{nat * REM : nat \in \mathbb{N}\} \\
& range(launch(t))\ is\ discrete && \text{(Lemma 1)}
\end{aligned}
$$

■

Definition 2. Let us label the Greedy tree which contains n nodes as G_n. The Greedy tree is defined recursively:

For $n = 1$, G_1 is the tree which only contains the root node.

For $n > 1$, $G_n = G_{n-1} + x$ where $x \in available(G_{n-1})$ and $\forall y \in available(G_{n-1})$, $launch(x, inf) \leq launch(y, inf)$

Definition 3. Given that $op \in T_n$, op is optimal if $\forall t \in T_n, launch(op) \leq launch(t)$.

Theorem 1. *The greedy algorithm defined in Definition 2, will create an optimal tree of n nodes.*

Proof: By induction:

For $n = 1$: G_1 is the tree comprised of only the root. Since $|T_1| = 1$, $\forall t \in T_1, launch(G_1) \leq launch(t)$, 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 tree. Once all of the nodes in $nodes[x]$ are in the tree, 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 tree because all of the nodes that precede 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 tree.

If G_n is not optimal, there would have to exist a tree in T_n that launches faster than G_n. Let us label this faster tree 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 tree $fast$ 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.

■

D. Discussion

The greedy tree is only optimal under certain conditions. The first condition for optimality is the use of the pseudo code in Figure 3. There are ways to alter this pseudo code which could possibly result in faster launch times. For example, instead of sequentially creating separate processes to execute the individual launch commands, a tree of processes could be used. Additionally, this pseudo code launches co-located processes in a later phase. It might be possible to create a faster tree, by intermixing the launch of remote and co-located processes.

A second condition for the optimality of the greedy tree is the assertion that the parameters `REM` and `SEQ` are constant values. In reality, these values could increase as network and/or file system congestion increases. Furthermore, the physical network layout and machine differences may skew 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 tree that accounts for these differences might prove to be faster.

V. Experiments and Results

We had several motivational goals for the experiments we performed: (1) we wish to validate our performance model for tree-based process launching; (2) we wish to validate empirically that our optimal greedy tree performs better than others; (3) we wish to demonstrate the impact of choosing an optimal process launch tree versus an arbitrary one, and (4) we wish to evaluate the cost of executing our greedy algorithm to determine an optimal tree.

A. LIBI: A Framework for Scalable, Flexible Process Launching

LIBI provides a set of launch and communication abstractions that are sufficient for bootstrapping and creates a framework for implementing these abstractions [16]. It is designed to sit between software systems and the underlying launch and communication mechanisms. In this manner, LIBI provides portability to the software system while retaining the performance benefits of using the native bulk-launch and communication services.

The version of LIBI which we used for this paper, relies on individual launch services, namely `rsh`. The individual launch version of LIBI, takes a host list as input, converts the host list into a process launch tree, and launches one process per node using `rsh` in a manner dictated by the process launch tree. Once one process exists on each of the requested nodes any co-located processes are then launched locally.

B. Experimental Environment

All experiments 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 job size is limited to 386 nodes.

C. Validating our Process Launch Performance Model

Recalling our process launch performance model from Section III-C, the time required to launch tree t is:

$$launch(t) = max_{p=1}^{n}\ launch(p,t)$$

where $launch(p,t)$ is defined as:

$$\begin{aligned} launch(p,t) \quad = \quad & (|psas(p,t)| * SEQ) \\ & (|anc(p,t)| * REM) \end{aligned}$$

For our model validation experiments, the general procedure involves using LIBI to launch a test application numerous times and compare the measured launch times with the projected launch time according to our performance model.

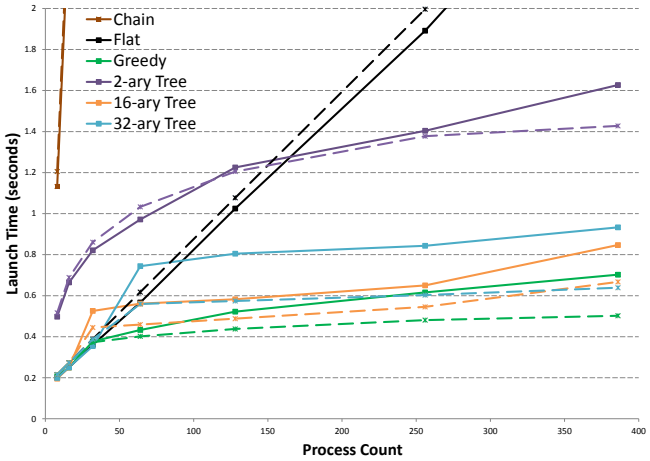


Figure 8. LIBI Launch Performance: The solid lines are measured values while the dashed lines represent the modeled launch time. The modeled launch times were created using the parameters: $\{0.007s, 0.172s\}$. This resulted in a coefficient of determination of $R^2 = 0.999$ between the modeled and the measured values.

All experimental runs were executed on the same allocation of nodes. Primarily, this strategy simplifies the batch scheduling requirements of our managed cluster environment. A second reason is that executing all tests in roughly the same time frame and on the same resources helps to ensure minimal environmental variations, for example due to varied levels of network congestion. Lastly, this strategy ensures that test runs do not occur concurrently, eliminating possible inter-test interferences and contentions, for example, contending for the same executable files on the same file system.

One consequence of this strategy is that the test executable is left in each node's local file cache between test runs. Local file caches are only cleared between separate allocations. This forces us to adjust the parameters of our launch model. Since the goal of the experiment is to validate the launch model and its adaptability to different environmental conditions, such adjustments are at the very least acceptable, if not desirable.

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

1) The Tests: We use this methodology to test two independent variables: *process count* and *tree topology*. Since we deploy a single process per node, the maximum process count is limited by the system's maximum job size limit: 386 nodes. We test chain, flat, greedy, 2-ary (or binary), 16-ary and 32-ary trees. Each experimental scenario was executed

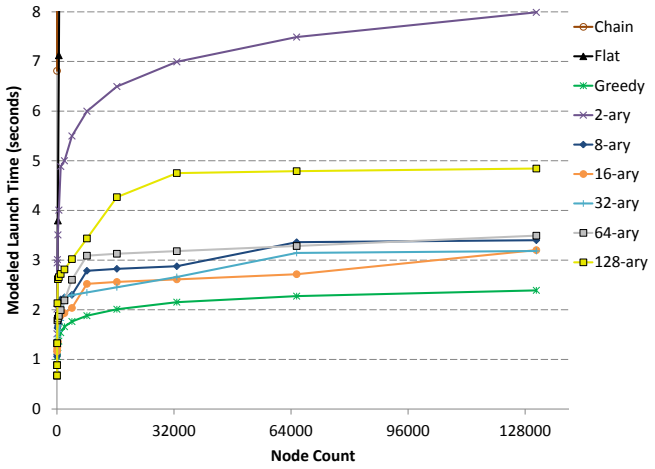


Figure 9. Modeled launch time with the executable on the server.

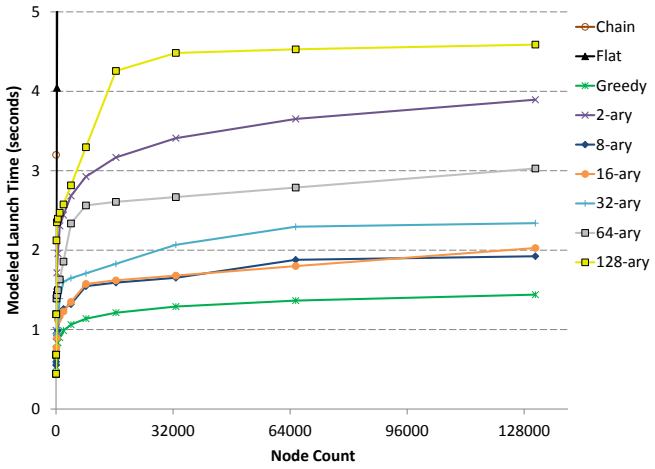


Figure 10. Modeled launch time with the executable in the local cache.

ten times and averaged.

As described in Section IV-B, the greedy algorithm requires us to set the SEQ and REM parameter values. We use the 2-tuple, {SEQ, REM}, to represent these values. We obtained values for these parameters by measuring and averaging these metrics from a small number of simple process launching experiments using the LIBI framework. The specific values obtained were {0.015s, 0.227s}.

2) The Results: The results of these experiments are shown in Figure 8. In this figure, solid lines represent measured performance and dashed line represents modeled performance. Our modeled performance tracts very precisely to our measured performance data. In fact, the parameters used to create our greedy tree differed from the parameters used in the launch time performance models.The greedy algorithm used the parameter values, {0.015s, 0.227s}, obtained from averages from a single test run of multiple launches. These turned out to be overestimates of the values, {0.007s, 0.172s}, from the actual experiments. We suspect the differences be caused by different levels of noise in the system at the time. In spite of the differences, the model produced from these values had a coefficient of determination of $R^2 = 0.886$, an indication that our model can tolerate system noise to some extent. The parameters used in Figure 8 are the result of a least-squares fit of Equation 4 to the actual data.

D. Evaluating Process Launch Tree Topologies

We now use our validated performance model to evaluate the impact that a process launch tree topology has on the bulk process launch performance. Using modeled launch times allow us to execute a larger, more comprehensive suite of experiments in an easier and faster manner. Additionally, we can project results for system scales orders of magnitude larger than the test machines.

1) The Tests: For each test in this experiment, we first create a process launch tree of a specified topology. Then we use our performance model to project the time to launch that tree. Once again, the independent variables we vary are process count and tree topology. However, unlike in the previous experiments that needed to be validated empirically, we can vary process counts to much greater extents, from $\left(2^4 - 1\right)$ to $\left(2^{17} - 1\right)$. These values correspond to the number of nodes in a full 2-ary tree of increasing depth. We also expanded the coverage of topologies that we test to include 64-ary and 128-ary trees.

For these experiments, the values used for {SEQ, REM} reflected two different launch environments. The first set {0.013s, 0.485s}, reflects launching a 1.6M executable from an NFS server. The second pair {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.

2) The Results: The results of these experiments are shown in Figures 9 and 10. The first rather-obvious observation is that both the chain and flat tree topologies are poor performers and must be avoided at large scale. Indeed, existing infrastructures that use these strategies for simplicity demonstrate poor scaling behavior.

Secondly, it shows the greedy tree outperforms all other trees in all scenarios, corroborating our proof. Thirdly, while the relative performance improvements of our greedy algorithm over other techniques are dramatic, the absolute differences are not as impactful. For example, at the largest process count, the differences range from 70% better than the second best to 360% better than the worse, non-degenerate case, the absolute differences are only on the order of a few seconds. We attribute that to the performance of the underlying sequential launch mechanism being

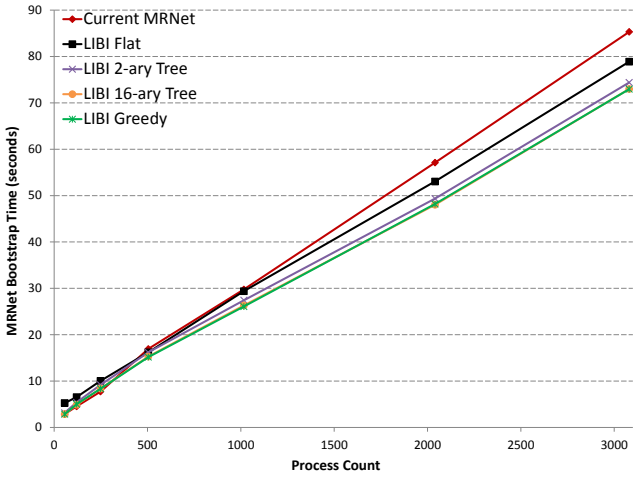


Figure 11. MRNet Bootstrap Time vs. Process Count. MRNet has a fanout of 16.

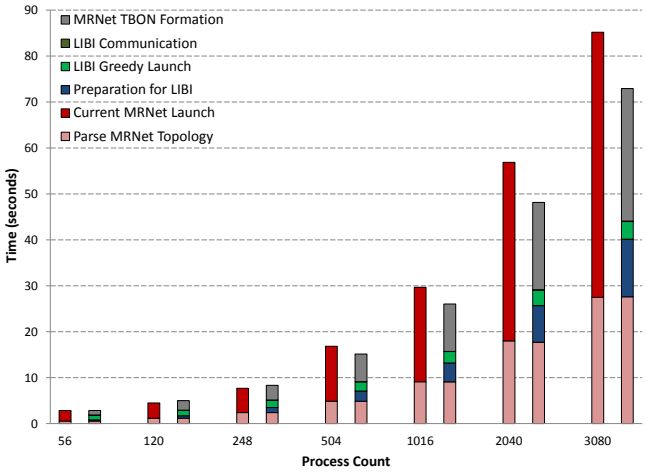


Figure 12. A Breakdown of MRNet's Bootstrap Time vs. Process Count. MRNet has a fanout of 16. Left columns are the current MRNet version while the right are MRNet over LIBI.

used. While `rsh` can be extremely fast, `ssh` relies on authentication and authorization mechanisms that can make even single `ssh` connections take seconds to establish. The poorer the performance of the underlying sequential launch mechanism, the greater the absolute impact of choosing an optimal launch topology will be.

Our final observation from these experiments is that the performance of the k-ary trees changes dramatically with the the value of `REM`. This is most readily apparent when comparing the 2-ary tree and the 128-ary tree. In Figure 9 the 2-ary tree takes almost twice as long to launch a tree at any node count. In contrast, Figure 10 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, and therefore their launch time is dominated more by `REM`. This means that the best k-values are system and environment dependent, so if arbitrary k-values are chosen for a launch strategy that always uses a k-ary tree, launch performance may suffer dramatically.

E. A Real Case Study: Improving MRNet Startup

Our final evaluation of process launch involves integrating our framework into a real infrastructure. For these experiments, we integrated our LIBI framework into MRNet [17]. MRNet is a software overlay network that provides efficient data multicast and reduction communications for distributed software systems. MRNet uses a tree of processes between the application's front-end and back-ends to improve group communication performance. The tree also is used to distribute important activities, like data reductions and data analysis, keeping front-end loads manageable.

In the old version, MRNet used a bootstrap mechanism in which parent processes create their children processes using `rsh` in as concurrent a fashion as possible. We modified MRNet to use LIBI for creating the tree processes and for disseminating the topology information needed for children processes to form the tree-based overlay network by establishing connections with their parents. Previously, MRNet's start-up process integrated process launch and information dissemination: when a parent created its children, it passed on the command line the necessary port information the children needed to establish a connection with the parent. LIBI completely separates the process launch and information dissemination interactions. In the new LIBI-based MRNet the LIBI session master gathers the relevant start-up information and then scatters it to relevant session members.

1) The Tests: The general strategy of this experiment is to evaluate the time it takes to bootstrap MRNet under varying conditions. There are three independent variables: process count, bootstrap mechanism, and MRNet fanout. The first variation in bootstrap mechanism is the current version of MRNet versus the new MRNet over LIBI. MRNet over LIBI is further varied by using different process launch tree topologies – flat, 2-ary, 16-ary and greedy. The parameters, $\{0.013s, 0.485s\}$, used to create the greedy tree were from the NFS server scenario in the previous section.

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 a third 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 easily avoided.

The executables being launched include the program for

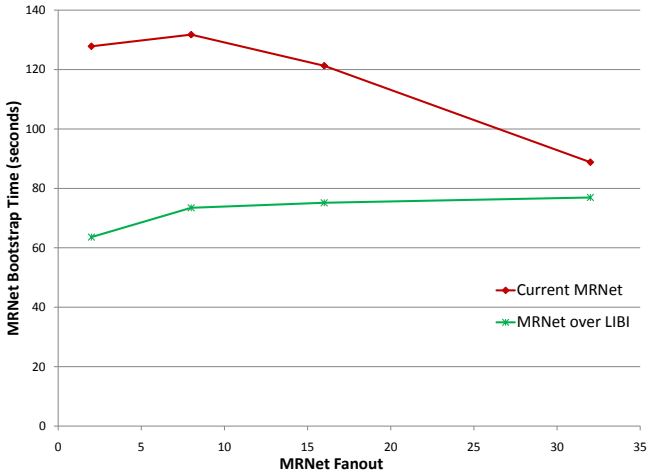


Figure 13. MRNet Bootstrap Time vs. MRNet Fanout. Using a total of 3080 processes.

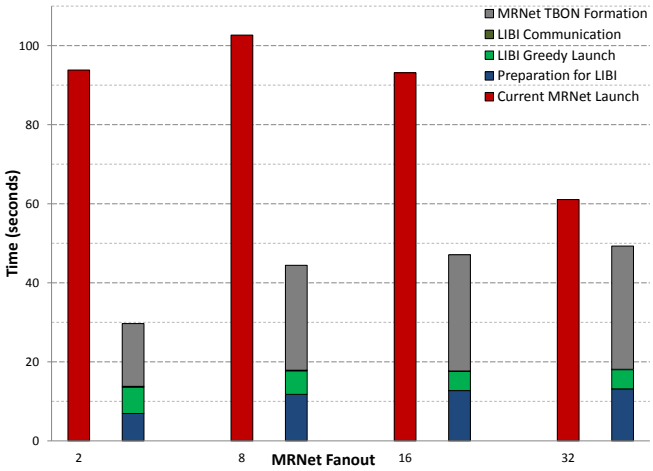


Figure 14. A Breakdown of MRNet's Bootstrap Time vs. MRNet Fanout. Using a total of 3080 processes. Left columns are the current MRNet version while the right are MRNet over LIBI.

MRNet's communication (intermediary) processes and the program for our MRNet back-end (leaf) processes. All executables were compiled using the archived version of the needed libraries.

We ran two tests using MRNet. The first test kept MRNet's fanout constant at 16, while 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 its 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 run with 8 processes per node.

2) The Results: We ran two tests using MRNet. The first test kept MRNet's fanout constant at 16, while 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 its 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 run with 8 processes per node.

Analysis: Figure 11 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. That being said, all of the bootstrapping conditions appear to scale linearly.

Figure 12 shows the breakdown of the MRNet bootstrap timings for the current version of MRNet and MRNet over LIBI using the greedy tree. The functionality related to each Section is described in Table II. 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.

Figure 13 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 14 shows the breakdown of the Figure 13 timings. The "Parse MRNet Topology" 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 per-

Name	Description
MRNet TBON Formation	Connect MRNet child processes to their parent.
LIBI Communication	Distribute the connection setup information to all of the nodes, using LIBI's greedy tree.
LIBI Greedy Launch	Launch the communication and back-end processes on the requested nodes, using LIBI's greedy tree.
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 II
DESCRIPTION OF THE COMPONENTS IN FIGURE 12 AND FIGURE 14.

formance 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 communication and back-end processes. With a fanout of two, $\frac{1}{2}$ of MRNet's total processes are communication processes. With a fanout of eight, only $\frac{1}{8}$ of MRNet's total processes are communication processes. Due to the scalability of the greedy tree, launching a large topology and a small topology is faster than launching two equal-sized topologies.

VI. CONCLUSION

Process launching is part of the bootstrapping phase, which is on the critical path of many HPC applications and tools. In this paper, we designed an algorithm for creating an optimal process launching strategy. Our process launching strategy uses a greedy algorithm and thus the resulting process launch tree is called the greedy tree. We proved that the greedy tree was indeed optimal for process launch under certain conditions, all of which are reasonable in practice.

The main impact of this work is that we have devised a cheap, efficient strategy for determining the optimal way for launching large numbers of processes in large scale computing systems. This strategy is particularly useful in computing environments that are not managed by a resource manager. Additionally, these research concepts can be used by resource managers to influence the way they launch processes. As we continue to look for ways to improve application and tool performance on large scale systems, there are a variety of related ways in which we can extend this work.

Future Research Directions

Currently, the greedy tree relies on user supplied parameters. As evidenced by the discussion in Section V-C, user supplied parameters are not likely to be accurate. A better approach would be to generate the REM and SEQ parameters during the configuration of the application. Since these parameters are application and system dependent, the correct values should remain constant once the application has been installed on the system. With the correct application specific parameters, the performance of the greedy tree should improve. This would also remove the burden from the user, 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. Future systems are expected to have 100s if not 1000s of cores.

This trend of increasing numbers of processing cores on a single node, may yield additional opportunities for optimizing process launch. As discussed in Section IV, the greedy tree's current approach is to *sequentially* execute the individual launches that originate from the same node. Multicore systems provide an opportunity for a single node to concurrently execute multiple remote launches. This approach should prove to be faster, up to the point where the network interface becomes saturated.

ACKNOWLEDGEMENTS

This work was supported in part by Lawrence Livermore National Security, LLC subcontract B590510. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL- CONF-485781).

REFERENCES

[1] "Top 500 Supercomputer Sites," http://www.top500.org/ (visited September 2011). [Online]. Available: http://www.top500.org/

[2] "Sequoia," https://asc.llnl.gov/computing_resources/sequoia/ (visited May 2011). [Online]. Available: https://asc.llnl.gov/computing_resources/sequoia/

[3] P. Kogge, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep., September 2008.

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Applications," in *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS '07)*, Long Beach, CA, March 2007.

[5] "IBM Tivoli Workload Scheduler LoadLeveler," http://www-03.ibm.com/systems/software/loadleveler (visited May 2011). [Online]. Available: http://www-03.ibm.com/systems/software/loadleveler/

[6] "Platform LSF," http://www.platform.com/workload-management/high-performance-computing (visited May 2011). [Online]. Available: http://www.platform.com/workload-management/high-performance-computing

[7] "PBS," http://www.pbsworks.com/ProductPBSWorks.aspx (visited May 2011). [Online]. Available: http://www.pbsworks.com/ProductPBSWorks.aspx

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

[9] 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*, ser. HiPC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 323–335. [Online]. Available: http://portal.acm.org/citation.cfm?id=1791889.1791925

[10] 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*. New York, New York, USA: ACM Press, 2011, pp. 41–48. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1988796.1988803

[11] N. Adiga, G. Almási, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, and Others, "An overview of the BlueGene/L supercomputer," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, vol. 00, no. c. Los Alamitos, CA: IEEE Computer Society, 2002, pp. 1–22.

[12] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, "The Application Level Placement Scheduler," in *Cray User Group*, 2006, pp. 1–7.

[13] R. Butler, W. Gropp, and E. Lusk, "Components and interfaces of a process management system for parallel programs," *Parallel Computing*, vol. 27, no. 11, pp. 1417–1429, 2001.

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

[15] 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). [Online]. Available: http://planetmath.org/?op=getobj&from=objects&id=7142

[16] 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. [Online]. Available: http://ics11.cs.arizona.edu/workshops/whist-2011/papers/whist-2011-goehner.pdf

[17] 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)*. Phoenix, AZ: IEEE Computer Society, November 2003, p. 21.