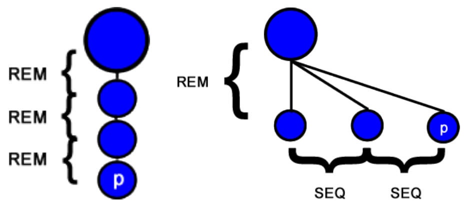
An Optimal Algorithm for Extreme Scale Job Launching

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

June 12, 2012

IEEE Cluster 2012
Bejing, China
September 24, 2012 through September 24, 2012



(a) REM (b) SEQ

Figure 1. Modeling the launch of process p in a chain and flat trees.

SEQ the constant amount of time required at a parent process between the instants that process can create two subsequent children processes.

For a flat tree of n processes (see Figure 1(b)), SEQ is repeated $n-2$ times, once for each child process node with exception to the final child launched (which incurs cost REM). Intuitively, the time it takes to launch a flat tree with i children is $SEQ*(i-1)+REM$.

C. Modeling Arbitrary Trees

For arbitrary trees, 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 build a recursive equation by observing that every child process in a given tree can be considered as the child of a flat tree rooted at the child's parent. 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. Formally, the launch time of a single process is:

$$\begin{aligned} launch(root) &= 0 \\ launch(child_i) &= launch(parent) + \\ &\quad SEQ*(i-1)+REM \end{aligned} \tag{1}$$

where i is the i^{th} process launched with respect to its parent.

If we execute this recursion up to the root of the entire tree (see Figure 2), we see that REM must be repeated for each level of the tree. We also observe that at each level of the tree, a process experiences a sequential delay SEQ proportional to the number of preceeding siblings.

Finally, we formalize the time required to launch a complete tree T as:

$$launch(T) = max_{p=1}^{n}\ launch(p) \tag{2}$$

IV. Optimal Process Launch Trees

We now present an algorithm that produces an optimal process launch tree for a given process launch problem. Optimal means that the output tree is guaranteed to launch the given processes on the given nodes in a minimal amount of time for a particular system. [1]

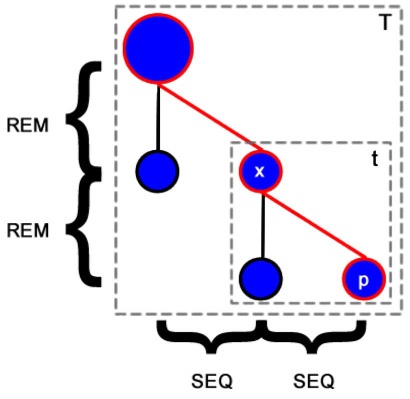


Figure 2. Modeling the launch of process p in an arbitrary tree of 5 nodes.

A. The Greedy Tree

We use a greedy algorithm to find an optimal process launch tree. We call the resulting process launch tree the *greedy tree*. Our algorithm is inspired by the construction of an optimal-multicast tree [7]. Based on a model that parameterizes the 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 uses similar parameters that have been adapted for process launching.

B. Greedy Algorithm

Figure 3 shows the pseudo code of the greedy algorithm, which produces a tree to be used for launching one process per node for a given set of nodes. 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. We start by placing the first node from the set in the root position. During each iteration of the algorithm, another node is removed from the input set and greedily placed in the position on the tree which has 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 newly available positions. Therefore, the greedy algorithm completes in $O(n\ log\ n)$ time. Due to the large number of positions with the same launch times, the algorithm is closer to $\Theta(n)$ in practice. Our testing shows that the additional overhead from generating the optimal tree is minimal, ranging from 7E-5 seconds for 100 processes to 0.09 seconds for 100,000 processes.

[1]Multiple trees may satisfy the minimum launch time requirement.

```
for (each process p)
  if (tree == empty)
    p = root
  else
    p = position of smallest launch time in heap
  add launch time for p's next sibling and first child to heap
```

Figure 3. Pseudo code for our greedy algorithm that creates an optimal process launch tree.

C. Proof of Optimality

Intuitively, the greedy algorithm should produce an optimal process launching tree. 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 and it will be assigned the next child.

Let us provide several definitions that we will use in our proof:

$launch(t)$
: the time taken to launch a tree t.

$launch(child_i)$
: the time taken to launch a particular node $child_i$.

$available(t)$
: returns the set of potential (unused) child positions in tree t with respect to a single insertion.

$time_i$
: the i^{th} lowest value of $range(launch(t))$

$nodes_i$
: the set of positions in a tree that will launch within $time_i$.

Definition 1. 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.

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

Definition 2. Let T_n be the set of all possible trees with n nodes. 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 1, 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 $nodes_x$ 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 than or equal time than the nodes in $nodes_{x+1}$. By way of contradiction, they will be in $nodes_x$, $nodes_{x-1}$ or $nodes_{...}$ and are already in the Greedy tree.

If G_n is not optimal, there would have to exist a tree $t \in T_n$, such that $launch(t) \leq launch(G_n)$.

First let us consider the case where an alternative tree of equal launch time exists. We label this tree $equal$, such that $launch(equal) = launch(G_n)$, and $available(equal) \neq available(G_n)$. One way to define $equal$, is to describe how it is different from G_n. We can describe this difference in terms of a function, $move(G_n)$, which will create the tree $equal$ by moving a node x in G_n to a new position y such that $launch(y) = launch(x)$. In the case that we add a new node to G_n the smallest position in $available(G_n)$ is y, which will not increase $launch(G_n)$ since $x = y$. A similar argument can be made for adding a node to the tree $equal$. Assuming the trees have only two points x and y that are equivalent, we see that they converge to the same tree after adding a single node to each. A proof by induction shows that if trees have K positions of equivalent launch time, after K addtions the possible trees converge into a single identical tree. For any addition less than K the trees have equivalent performance since $launch(x) = launch(y), \forall x, y \in K$.

Let us consider the case in which G_n is not optimal such that there is a tree with faster launch time. We label this faster tree as $fast$. If $launch(G_n) = time_i$, $move(G_n)$ can remove a node, g, from G_n where $launch(g) \leq time_i$, 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

This approach assumes a hierarchical process launching strategy that follows the algorithm outlined in Section III. It also assumes that the parameters `REM` and `SEQ` are constant values. These conditions are reasonable but they are not absolute. For example, a process launching strategy could replace the sequential operation, of a parent launching its children, with a non-sequential operation. This could potentially be accomplished by intermixing the launch of remote and co-located processes, instead of launching co-located processes at the end. The co-located process could then be used to launch the remaining remote processes, in

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

Table I
OVERHEAD (SECONDS) TO DESIGN THE OPTIMAL TREE VS A 16-ARY TREE.

parallel. Additionally, a better performance model might be created by modeling the variability of REM and SEQ throughout the launch.

Constant values for REM and SEQ do not necessarily capture physical environments in which these values are not constant in practice, for example environments with non-uniform network latencies. In future work, we plan to evaluate the impact of accounting for hardware topologies, however, our current experiments in the next section demonstrate that despite this imprecise assumption, our model can still be very accurate.

V. EXPERIMENTS AND RESULTS

Our experimental goals were: (1) to validate our process launching performance model; (2) to validate our optimal (greedy) tree empirically; (3) to demonstrate the impact of choosing an optimal process launch tree versus an arbitrary one, and (4) to evaluate the cost of determining the optimal tree using our greedy algorithm. Table I summarizes the latter result and shows that a process launch tree for 100,000 processes can be generated in less than 0.3 seconds. We now present the experiments and results for the first three goals.

A. 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 cores. The nodes are interconnected via a double data rate InfiniBand network. Atlas is managed by the SLURM resource manager. Maximum job size is limited to 386 nodes.

We used LIBI, a scalable bootstrapping framework, in our experiments [10]. LIBI launches one process per node using `rsh` in a manner dictated by a process launch tree. Once one process exists on each of the requested nodes, then co-located processes are launched locally.

B. Validating our Process Launch Performance Model

To validate the performance model outlined in Section III-C, we used LIBI to launch a test application. We then compared the measured launch times with the projected launch times of our performance model.

All tests were executed sequentially on the same allocation of nodes. This simplified batch scheduling, minimized the environmental variations between test run and eliminated inter-test interferences and contentions. Consequently, this left the test executable in each node's local file cache

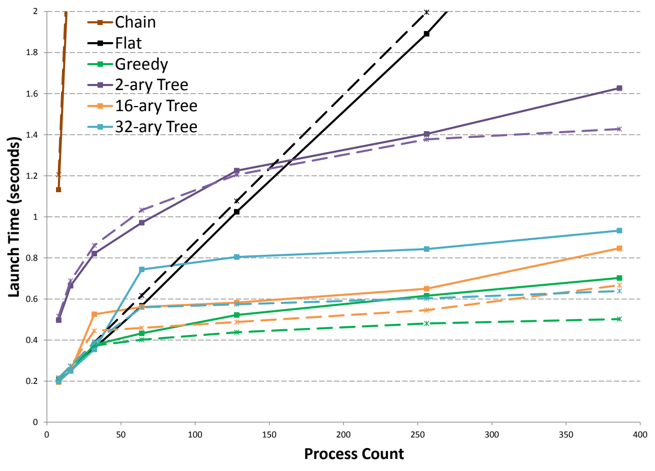


Figure 4. LIBI Launch Performance: Solid: measured; Dashed: modeled. The modeled launch times were created using the parameters: {0.007s, 0.172s}. The coefficient of determination between the modeled and the measured curves range from $R^2 = 0.59$ to $R^2 = 0.99$.

between test runs. We accounted for this by adjusting our performance model's parameters.

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 statically-linked. We deploy a single process per node, varying the *process count* and *tree topology*. Each test condition was executed ten times and averaged.

As described in Section IV-B, the greedy algorithm requires two parameters. We use the 2-tuple, {SEQ, REM}, to represent these values. We obtained values for these parameters by averaging these metrics from a small number of test runs. The values obtained were {0.015s, 0.227s}.

Figure 4 shows that **our model very accurately matches observed performance**. In this figure, solid lines represent measured performance and dashed lines represent modeled performance. The parameters of the modeled launch times are the result of a least-squares fit of Equation 2 to the actual data, giving the parameters, {0.007s, 0.172s}. These parameters differ from the parameters used to create our greedy tree. We suspect the differences stem from differing levels of system noise (from resource contentions). Despite the differences, the modeled data created from the overestimated values had a coefficient of determination of $R^2 = 0.886$ to the measured data, an indication that our model can tolerate system noise to some extent.

C. 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

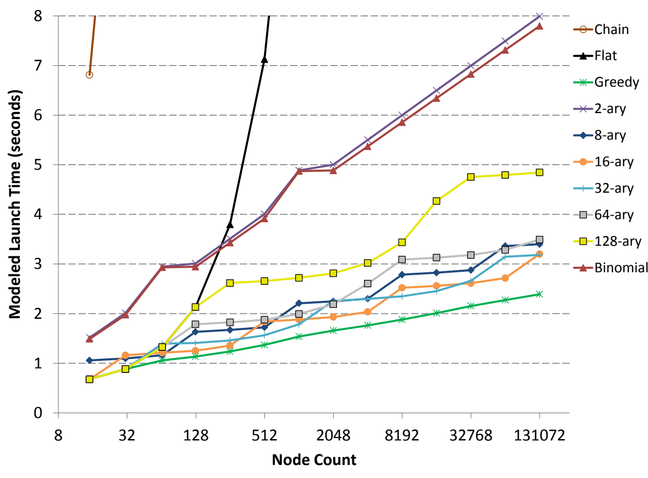


Figure 5. Modeled launch time with the executable on the NFS server.

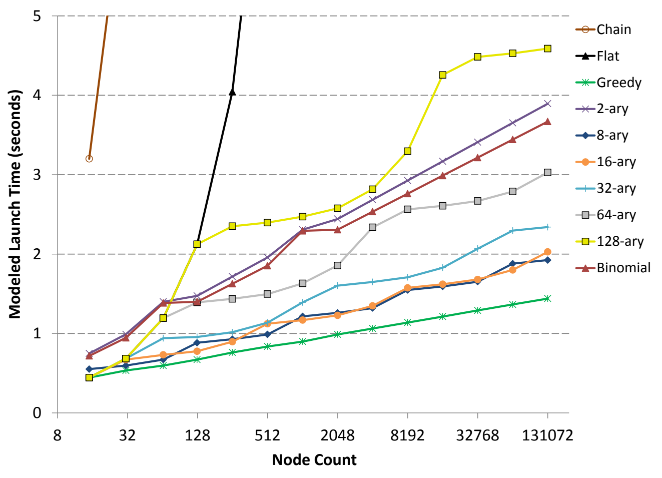


Figure 6. Modeled launch time with the executable in the local file cache.

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 this experiment we apply our performance model to a varying set of topologies at varying node counts. However, unlike in the previous experiment that needed to be validated empirically, we can vary process counts to much greater extents, 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 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 5 and 6. The first rather obvious observation is that both the chain and flat tree topologies are poor performers and must be avoided at large scale. 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. **At the largest process count, the differences range from 70% to 360% better.** The absolute differences are on the order of a few seconds; however, the absolute differences increase with a larger REM parameter, for example, when `ssh` must be used instead of `rsh`.

Our final observation from these experiments is that the performance of the k-ary trees changes dramatically with the the value of `REM`. In Figure 5 the 2-ary tree takes almost twice as long to launch a tree at any node count. In contrast, Figure 6 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 means that **arbitrarily chosen k-values for k-ary launching strategies can lead to dramatically poor launch performance.** We discuss this point further in Section V-E.

D. A Real Case Study: Improving MRNet Startup

We also evaluate our process launch strategy by incorporating it into a real infrastructure. For these experiments, we integrated our LIBI framework into MRNet [15]. MRNet is a software overlay network that provides efficient data multicast and reduction communications for distributed software systems. MRNet improves group communication performance using a tree-based overlay network (TBON) of processes between the application's front-end and back-ends.

Currently, MRNet's start-up process integrates process launch and information dissemination: when a child is launched with `rsh`, the child must connect to its parent and receive topology information before it can launch its own children. We modified MRNet to use LIBI for both the launching of the TBON processes and the the dissemination of topology information. LIBI completely separates these two tasks. Once the processes are launched, the LIBI session master gathers then scatters the relevant setup information.

1) The Tests: This experiment is designed to evaluate the time required 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. The parameters, {0.013s, 0.485s}, used to create the greedy tree were from the NFS server scenario in Section V-C.

Each test run was given its own allocation of 386 nodes regardless of the number of nodes needed. The relatively

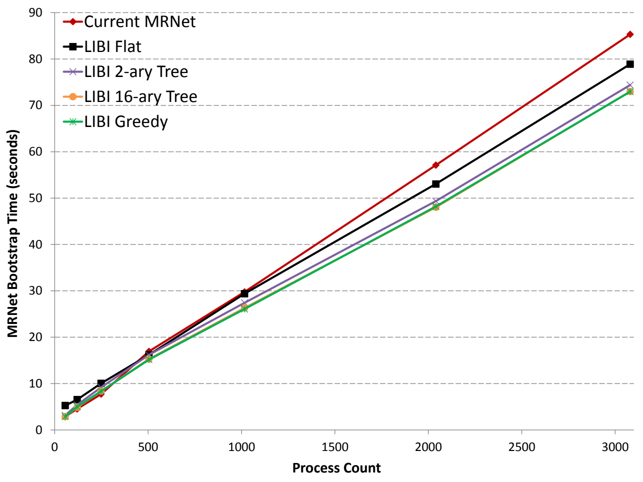


Figure 7. MRNet Bootstrap Time vs. Process Count using an MRNet fanout of 16.

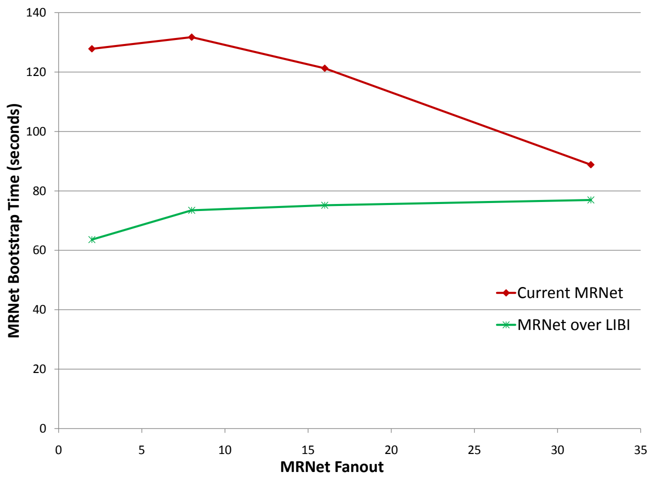


Figure 8. MRNet Bootstrap Time vs. MRNet Fanout using a total of 3080 processes.

large allocation size reduces the network congestion from other users and the likelihood of running tests concurrently. We also cleared the local file cache between test runs.

The executables being launched include the program for MRNet's communication (intermediary) processes and the program for our MRNet back-end (leaf) processes. All executables were compiled to be statically-linked.

We ran two sets of tests: first we held MRNet's fanout constant at 16, while varying the total process count. The process count includes MRNet's communication daemons as well as our back-end daemons. Then we held the process count constant at 3080, while varying MRNet's fanout. Each test condition was repeated three times, and all tests were run with 8 processes per node.

2) The Results: As we keep fan-out constant and vary process count, Figure 7 shows how the **LIBI-based MRNet bootstrapping outperforms the original procedure**. Likewise, Figure 8 shows the results of changing MRNet's fanout, while holding the process count at 3080. Here we see that the bootstrapping time of the current version of MRNet changes with the fanout while **LIBI-based bootstrapping performance remains relatively constant**. Many overlay networks like MRNet use their preferred run-time topology as their bootstrapping topology. The important corollary is that in many instances, **the optimal topology for steady-state overlay network execution may not be the optimal topology for overlay network deployment.** Therefore, it is important to use a strategy like the one we devised to determine optimal deployment strategy independent of the desired runtime topology.

E. The Impact of Arbitrary Topologies

Finally, we use our model to evaluate the impact that arbitrary launch topology choices can have on launch performance. We model the launching of 1000 processes under three different scenarios. The first scenario, $S1$, represents an HPC system with a high speed inter-connection fabric with $\{SEQ, REM\}$ set to $\{0.007, 0.172\}$. The second scenario, $S2$ is more representative of wide area network connectivity with parameters $\{0.007, 2\}$. The third scenario, $S3$ represents a wide area networked system in which remote job launch also entails transferring an executable program to the destination node with parameters $\{0.007, 10\}$. These roughly map to different scenarios of the *many task computing model* often used in the volunteer computing paradigm or in uncertainty quantification as described in Section I.

The results, shown in Table II, demonstrate the potential impact of always using a fixed process launch topology independent of the features of the target environment. For each of the above scenarios, this table ranks the process launch topologies based on their resulting launch times. If, for example, an arbitrary 4-ary launch topology is chosen, this would perform favorably for scenario $S1$ but suffer more than 100% slowdown compared to optimal for scenario $S2$ and almost 200% slowdown for scenario $S3$. Also, contrary to intuition, the flat tree is the best topology for scenario $S3$, but performs abysmally for the other scenarios. In short, **no arbitrary topology is good for all scenarios**. Of course, as the number of jobs launched increases, the absolute penalty for poor topology choices also increases.

VI. Conclusion

The main impact of this work is our efficient algorithm for finding an optimal way to launch jobs comprised of large numbers of processes on extreme-scale systems. Efficient process launching is becoming increasingly important for emerging computational models such as many-task computing and uncertainty quantification. Moreover, it becomes more expensive as the scales of HPC, clustered, Grid and cloud systems increase. We show performance benefits in

	S1: SEQ=0.03, REM=0.172		S2: SEQ=0.007, REM=2		S3: SEQ=0.007, REM=10	
Rank	**Topology**	**Launch Time (s)**	**Topology**	**Launch Time (s)**	**Topology**	**Launch Time (s)**
1	greedy	0.609	greedy	4.272	greedy	17.006
2	16	0.753	32	4.44	flat tree	17.006
3	32	0.784	64	4.552	32	20.44
4	8	0.841	128	4.944	64	20.552
5	64	0.896	256	5.812	128	20.944
6	4	0.971	16	6.237	256	21.812
7	128	1.288	512	7.422	512	23.422
8	2	1.624	8	8.153	16	30.237
9	256	2.156	flat tree	9.006	8	40.153
10	512	3.769	4	10.111	4	50.111
11	flat tree	7.178	2	18.076	2	90.076

Table II
PROCESS LAUNCH TOPOLOGIES RANKED BY LAUNCH TIME FOR 1000 PROCESSES UNDER VARIOUS SCENARIOS.

today's environments and project even more value when mapped to future system requirements.

In addition to highlighting the performance benefits of our launching algorithm over ad-hoc strategies, this work provides a quantitative framework for performance evaluations of resource managers. Additionally, resource managers and tools can leverage this work to launch processes optimally, as we demonstrated in our MRNet case study.

Currently, determining the greedy topology relies on user-supplied parameters. We are looking into automatically generating these parameters during application configuration or installation. Additionally, the greedy tree *sequentially* executes the individual launches that originate from the same node. Multicore systems provide an opportunity for a single node to concurrently execute multiple remote launches. Finally, adding physical machine topology awareness may provide even further potential for improvements.

REFERENCES

[1] "Top 500 Supercomputer Sites," http://www.top500.org/ (visited March 2013).

[2] 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.

[3] 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," in *Supercomputing 2008 (SC2008)*, Austin, TX, November 2008.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] B. Claudel, G. Huard, and O. Richard, "TakTuk, adaptive deployment of remote executions," in *HPDC 09 Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 91–100.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.

[15] 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.

Prepared by LLNL under Contract DE-AC52-07NA27344.