Overcoming Scalability Challenges for Tool Daemon Launching

Dong H. Ahn[1], Dorian C. Arnold[2], Bronis R. de Supinski[1],
Gregory L. Lee[1], Barton P. Miller[2], and Martin Schulz[1]
[1]Lawrence Livermore National Laboratory, Livermore, CA, USA, {ahn1,bronis,lee218,schulzm}@llnl.gov
[2]University of Wisconsin, Madison, WI, USA, {darnold, bart}@cs.wisc.edu

Abstract: *Many tools that target parallel and distributed environments must co-locate a set of daemons with the distributed processes of the target application. However, efficient and portable deployment of these daemons on large scale systems is an unsolved problem. We overcome this gap with LaunchMON, a scalable, robust, portable, secure, and general purpose infrastructure for launching tool daemons. Its API allows tool builders to identify all processes of a target job, launch daemons on the relevant nodes and control daemon interaction. Our results show that LaunchMON scales to very large daemon counts and substantially enhances performance over existing ad hoc mechanisms.*

1 Introduction

Runtime tools, such as debuggers and performance profilers, are essential to high performance computing. However, several obstacles limit the availability of such tools to one or few platforms and slow the development of new tools. The TDP project [20] noted that the myriad scheduler, job launch, and process control interfaces add severe costs to porting tools to high-end computing environments. On such systems, operating system services and the resource manager (RM) play a critical role in the launching of daemons and their interactions with system resources. Thus, we must port tools to local RMs for efficient execution or even just correct functionality. The variety of RMs, including Condor [23], LoadLeveler [16], LSF [19], SLURM [15], and YOD [6], exacerbate this effort. Without a common infrastructure for tool daemon launching, the task of porting m tools to n environments becomes an $m \times n$ effort. Tool developers must implement this capability for each tool/environment combination, often leading to difficult to maintain, ad hoc implementations.

TDP addressed portability but not the issue of *scale*. With the trend towards systems with 10^5 or 10^6 processors [2], efficient daemon launching is essential. Overheads that previously were merely discouraging are becoming prohibitive. As system scales increase, initialization activities can become bottlenecks that prevent user adoption, particularly for interactive tools like debuggers. Often, such tools must locate their daemons on the same compute nodes as the target application's processes [4, 5, 24, 12, 17, 20]. Further, tools for extreme scale jobs often leverage scalable communication infrastructures [5, 22] that require communication daemons on additional compute resources beyond the target program's allocation.

We seek to help tool developers create highly portable and scalable tools through a standard framework that leverages native system services for tool daemon launching. We present the key abstractions and mechanisms for a scalable and portable job (and tool) launch facility, and then describe the design and implementation of LaunchMON, a system that embodies these abstractions and mechanisms. LaunchMON can be viewed as a partial TDP implementation that abstracts native RM interfaces and services. Using LaunchMON, tools can automatically run on existing (or future) HPC systems that implement the widely available services upon which we base LaunchMON.

This work makes five primary contributions. First, we develop the essential abstractions for a scalable launch facility. Such abstractions must be useful to tool developers and compatible with the variety of facilities on existing systems, which we cannot assume will change to accommodate tool developers. Thus, LaunchMON provides the mapping to a uniform interface. Second, we present a design and implementation, for cluster and supercomputer environments, that embodies these abstractions. Third, several case studies show how LaunchMON supports existing tools and the creation of new ones. Fourth, one of our new tools is itself a contribution: Jobsnap provides the first portable and scalable mechanism for users to gather information for each MPI task normally captured through the */proc* [21] interface. Last, our experiments and LaunchMON performance model demonstrate that it achieves the goals of portable and scalable tool daemon launching, with overheads an order of magnitude less than existing ad hoc solutions.

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-401480-DRAFT).

LaunchMON builds on the concepts of previous tool and launch research. OpenRTE [8] proposed a common execution environment that allows the same source code to run on different HPC platforms. However, it primarily targets MPI applications and did not capture the distinct needs of HPC tools. For example, OpenRTE does not provide a mechanism to launch a tool daemon and attach it to an application process. The tool daemon protocol (TDP) [20] focused on codifying essential tool daemon activities into a standard API but did not address scalability. The message queue debugging interface [10] and the Automatic Process Acquisition Interface (APAI) [1, 6, 13] provide standard interfaces for debuggers to interact with MPI application tasks, but not for launching tool daemons.

In the following section, we discuss typical strategies for tool daemon launching employed by current tools. Section 3 describes LaunchMON's design and implementation in detail. Next, we present and validate a detailed performance model of LaunchMON's daemon launching mechanism in Section 4. Section 5 covers the integration of LaunchMON with existing tools and its use to build Jobsnap, including its impact on the performance of those tools.

2 Ad Hoc Tool Daemon Launching Practices

Tool developers currently rely on ad hoc mechanisms for tool daemon launching. Most frequently, tools combine basic mechanisms like *ssh* or *rsh* with manual protocols to launch the necessary daemons. This approach generally fails to leverage the services of the RM, which can aid in the co-location of tool daemons. Thus, it requires significant porting efforts to systems that do not support *rsh*/*ssh* (e.g., BlueGene/L (BG/L) [3] or the Cray XT3 [25]). More importantly, these ad hoc solutions often have high overhead. While this does not restrict their functionality, which is often the core focus of the tool developer, it renders them impractical for realistic scenarios. For example, tools that use *rsh*-like methods must call a protocol to spawn each daemon sequentially, easily leading to an implementation prohibitively slow for launching hundreds of daemons and even worse for larger scales.

Instead of explicitly starting daemons during tool startup, some tools, like DPCL [11], use persistent daemons for all tool sessions, which requires that the daemons are always active on all nodes, which wastes system resources. The persistent daemons usually run as root, which makes them difficult to deploy, to install and to maintain in large scale production environments. They also represent a security risk, since they act as root on behalf of non-privileged users. Hackers routinely exploit such configurations, for example in FTP, Telnet, or RSH servers, even though these services are more thoroughly tested than most tool infrastructures. Better support for on-demand tool daemon launching would alleviate the need for persistent daemons.

In addition to co-located daemons, large scale tools increasingly rely on hierarchical infrastructures, such as Tree-Based Overlay Networks (TBŌNs) like MRNet [22], which use additional communication daemons. These additional daemons, which require separately allocated nodes, must also be launched efficiently. Current infrastructures manually allocate these nodes and then rely on ad hoc launching mechanisms. Efficient daemon launching is even more important since the front-end process deploys its children daemons, which in turn deploy their children and so on.

Most modern systems provide low-level components that eliminate many of these problems. RMs provide native interfaces and runtime services that can scalably launch tool daemons on a large number of nodes [6, 7, 15, 26]. Most RMs also provide a native *Automatic Process Acquisition Interface* (APAI) that debuggers use to acquire the necessary information about the parallel target application. APAI provides access to a Remote Process Descriptor Table (RPDTAB) that includes the host name, the executable name and the process ID of each MPI task.

These services, however, are insufficient to develop portable and scalable tools. The interfaces are usually platform and RM specific and are often difficult to use. For example, tools typically access APAIs as a debugger, which entails significant software complexity. Also, tools require other services that are not directly offered by the native interfaces, such as additional node allocations or inter-daemon communication. To overcome these problems and to fill this important tool development gap, we developed the LaunchMON infrastructure that abstracts the native, platform specific RM and APAI interfaces for general purpose tools and provides a new set of services needed to implement scalable, daemon-based tools.

3 Distributed LaunchMON Architecture

HPC tools are inherently distributed software systems that often include different front end, back end and intermediate communication components. Thus, we decompose LaunchMON to provide functionality for each tool component type. Figure 1 shows LaunchMON's four components: (1) the LaunchMON Engine; (2) the front end API (LaunchMON FE API); (3) the back end API (LaunchMON BE API); and (4) the middleware API (LaunchMON MW API). The LaunchMON FE and BE APIs provide information about application processes and scalably launch tool daemons on remote nodes. Similarly, the LaunchMON FE and MW APIs enable scalable launching and connection of TBŌN daemons. The LaunchMON Engine interacts with the RM to determine when, where and how to perform the services of the other components. A compact application layer network protocol, LMONP, enables interactions be-

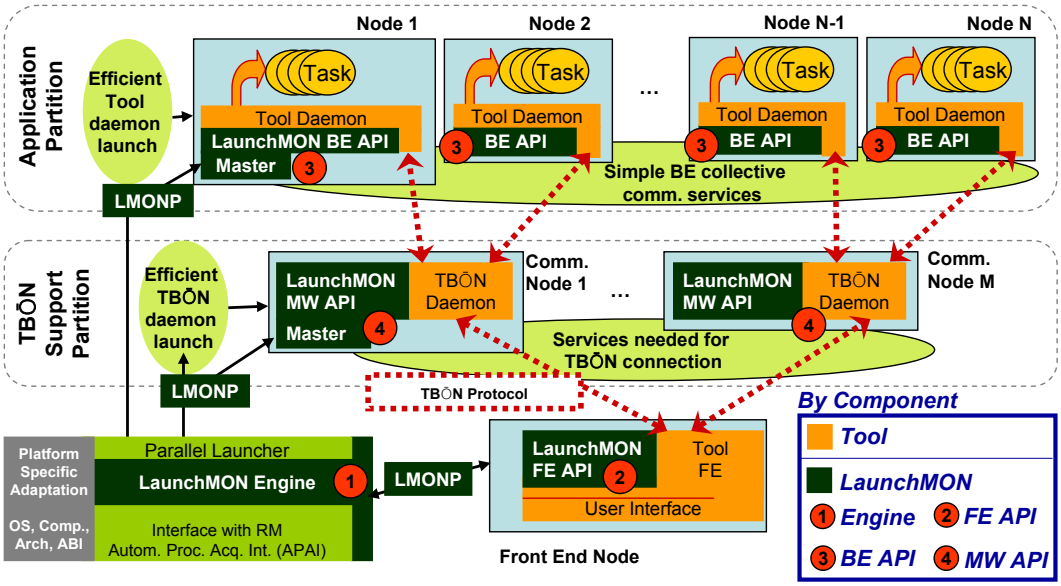

Figure 1. Architecture of LaunchMON

tween all of LaunchMON's components.

3.1 LaunchMON Engine

The essence of LaunchMON is its ability to interact with a wide array of RMs. To capture the required job information through APAI, the LaunchMON Engine, which provides this functionality, must trace the job's RM process. This typically requires debugger capabilities as well as a co-location with the target RM process. In addition, the LaunchMON Engine acts as a proxy for LaunchMON's other components, which generally cannot be co-located with the RM process, by translating a series of commands between them and the RM.

The LaunchMON engine is designed using a very modular class hierarchy that encapsulates all key components as separate abstract entities. We can use this to port it to new platforms by simply parameterizing and inheriting key abstract classes, filling in details of the computer architecture, the OS, and the RM of the new target machine, while keeping the core structure.

The central component is an independent *Driver* class that organizes its main operations: it first calls the *Event Manager*, which is responsible for polling the target RM process via an OS interface. Upon detecting a status update for this process, the *Event Manager* passes this native event back to the *Driver*, which then calls upon the *Event Decoder* to convert the event into a higher level LaunchMON event. The *Driver* next passes the LaunchMON event to the *LaunchMON Event Handler*, which invokes the handler matching the observed event.

A particularly important event is when the RM reports that the job is in a state where a tool can launch daemons co-located with the job. The handler associated with this event fetches the RPDTAB from the RM process address space, invokes an efficient daemon launch command and communicates the RPDTAB to the front end.

3.2 LaunchMON Front End

The engine's complexity arises from its interaction with RMs and other system software components. In contrast, the complexity of the LaunchMON FE API stems from the services that it provides to HPC tools. This subsection identifies the main functionalities required by HPC tool front ends. We then combine these requirements with technological constraints and performance optimization opportunities.

Our analysis determined seven main requirements for the FE API, consistent with previous studies of the interactions between tools and RMs [14, 20]. First, the FE API must launch or attach to an RM process, instructing it to prepare the job for tracing. Second, it must co-locate back end daemons with application tasks. Third, the FE API must launch middleware daemons. Fourth, it must fetch data, such as the RPDTAB, from the RM process. Fifth, it must transfer a tool's data between its front end and daemons. Sixth, the FE API must control a target job or daemons. Finally, it must bind its commands to a job or a group of daemons.

We easily map all but the last requirement to distinct procedure calls. We use a session, an abstraction for a group of daemons associated with a job, to provide the binding method. Most FE API procedures, which capture the other requirements, include a session parameter. In particular, our design requires users to create a session for each back end or middleware daemon launching and to pass that handle to

each FE API call. Internally, the front end runtime maintains a session resource descriptor table.

Our primary design goals of portability and performance, including scalability, led to FE API refinements. For example, on various platforms, the APAI tightly couples the operations of controlling and interacting with the RM and daemon co-location. Accordingly, our API combines these functionalities by supporting *attachAndSpawn* and *launchAndSpawn* calls but not calls that separate the actions. We also provide calls that allow a tool to register pack and unpack functions for enhanced performance. These calls can transfer the tool's data to and from either a back end or middleware master daemon. This enables piggybacking of the tool's data with the LaunchMON front end's handshaking exchanges with the daemons.

3.3 LaunchMON Back End

As with the FE API, our analysis of HPC tool requirements led to a procedural form for the LaunchMON BE API. However, rigorous scalability requirements also guide our software design for this module: we need basic collective communications for back end daemons to propagate and to gather launch and setup information. Since these collective services are useful for other tool functionality, the BE API makes them available for general use. We leverage native communication subsystems that the RM sets up if possible; our layered approach encapsulates interactions with native communication subsystems in the Internal Collective Communication Layer (ICCL). ICCL maps native interfaces to our back end collective calls; hence it is the only layer with significant platform dependencies.

Our collective services are not intended to replace TBŌN infrastructures but instead provide the minimal services needed for daemon launching. Tools that require extreme scalability and more flexibility should leverage a more advanced communication infrastructure, like MRNet. Supporting minimal services reduces porting efforts when building on top of native services and facilitates providing efficient generic implementations. Their use beyond LaunchMON's primary goals is only intended to provide tools with rudimentary inter-daemon communication services. For these reasons, we only support simple barriers, broadcasts, gathers and scatters.

3.4 LaunchMON Middleware

Our analysis showed requirements for the LaunchMON MW API are similar to those for the BE API: once launched into a set of newly allocated nodes, each TBŌN daemon must set up the TBŌN based on information that LaunchMON scalably distributes to it. Specifically, the MW API assigns to each simultaneously launched TBŌN daemon a unique personality handle that is similar to an MPI rank. It also sets up a simple network fabric based on a native communication subsystem such that a TBŌN daemon can send data to and receive data from other daemons collectively or individually using the personality handles, which TBŌN daemons can use to bootstrap their own network.

LaunchMON's middleware initialization also distributes the RPDTAB to the TBŌN daemons. The RPDTAB allows TBŌN daemons to locate the target program and the back end daemons. Some TBŌN implementations need other data in order to bootstrap their networks completely. Thus, our design allows piggybacking of tool specific data with LaunchMON's handshaking data exchanges between a front end and a TBŌN master daemon.

3.5 LMONP Communication Protocol

We provide a simple inter-component communication protocol, LMONP, built on TCP/IP that facilitates component communication. However, LMONP only supports communication between pairs of representatives, one representative per component, to achieve high performance and scalability. Our protocol must support the communications needed for daemon launching and initialization, such as distribution of the RPDTAB, and allows bundling of client tool data for efficient overall tool startup. LMONP compactly encodes these functionalities and offers a straightforward path for extension if more complex requirements emerge as LaunchMON is adopted more widely.

LMONP has a 16 Byte header and two variably sized payload sections: one for LauchMON data and one for user data. Besides a message tag and payload attributes, such as length, the header also includes a three bit *msg class* field that encodes a communication pair. Three of the eight possible pairs are currently used for (front end, LaunchMON Engine), (front end, back end), and (front end, middleware) connections. We could use the remaining pairs for a (middleware, middleware) connection that would support the use of multiple communication infrastructures or spreading a single communication infrastructure across multiple resource allocations.

4 Evaluating Overhead and Scalability

We construct an analytic model of *launchAndSpawn*, a key LaunchMON service that launches a parallel job under a client tool's control, in order to study the overhead and scalability properties of our infrastructure. We analyze the events on the critical path of this service and investigate their component costs. We denote the events as e_i with the duration between two events e_i and e_j (with $i < j$) as $T(e_i, e_j)$. The total time for *launchAndSpawn* is $\sum T(e_i, e_{i+1})$ of these critical path events .

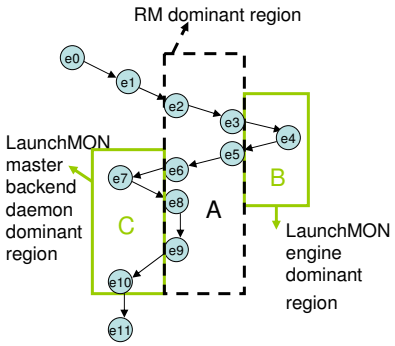

Figure 2. Critical Path of *launchAndSpawn*

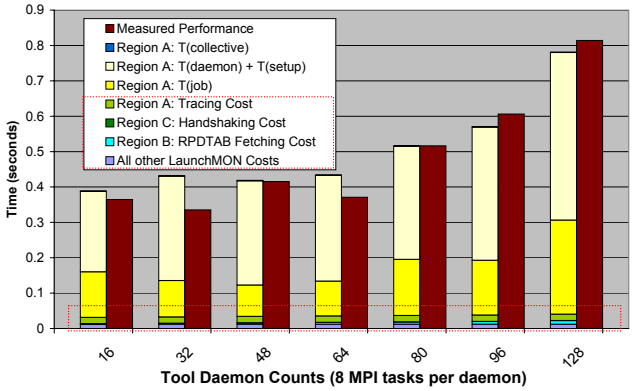

Figure 3. Modeled vs. Measured Performance

Figure 2 shows that our *launchAndSpawn* model includes eleven critical path events. The client initiates this service by calling the corresponding FE API function in e_0, which invokes the LaunchMON engine at e_1. The engine then executes the RM's job launcher under its control at e_2. The RM prepares to launch the job, including allocating a set of nodes for the job, after which it calls the *MPIR_Breakpoint* function and stops at e_3. The engine then fetches the RPDTAB from the RM process at e_4. Next, the engine either invokes another RM launcher or simply instructs the existing launcher to spawn tool daemons at e_5. The RM finishes spawning the daemons at e_6, and LaunchMON begins its handshaking process that establishes daemon input parameters at e_7. As part of this process, the master back end daemon begins coordinating the inter-daemon network setup with the RM-provided communication fabric at e_8. The network setup completes at e_9 and the master daemon finishes the handshaking process by sending a *ready* message to the front end at e_{10}. The *launchAndSpawn* service completes when it returns control to the client tool at e_{11}.

We group critical path events into regions based on their dominant contributors as shown in Figure 2 and separately analyze their costs as we increase job size. $T(e_0, e_2)$ and $T(e_{10}, e_{11})$ are local operations and do not impact scaling. Region A is dominated by the RM performance, while Regions B and C are LaunchMON overheads.

We model Region A's cost as the sum of several RM activities, specifically the spawning of the job tasks and the tool daemons, the inter-daemon communication fabric setup, and the handshake message distribution. We denote the costs of these activities as $T(op)$ where *op* is *job* for spawning the job tasks, *daemon* for spawning the tool daemons, *setup* for the communication fabric setup, and *collective* for broadcasting, gathering and scattering handshake messages. Our analysis shows that LaunchMON's only contribution to the overhead of Region A is the cost of tracing the RM process in $T(e_2, e_3)$. We model this cost with the number of RM debug events times LaunchMON's average event handler cost. If the RM debug events increase with job size then this cost can increase with the job size; our experience shows that a well designed RM does not have this property.

Region B and C have limited costs that are impacted by job scale. The size of the RPDTAB, which is fetched during Region B, is linear in the number of job tasks. Similarly, the handshake messages between the front end and the master back end daemon in region C include data on all daemons so their size grows linearly with scale.

We validate this model on *Atlas*, a 1,152 node/9,216 core Linux cluster with four-way, dual-core 2.4 GHz AMD Opteron nodes, 16 GB of main memory per node and connected with 4x DDR Infiniband. The hardware, as well as the software stack, of its front end nodes is identical to that of its compute node. All nodes run a full-featured Linux using the CHAOS distribution [18] version 3.3, a derivation of Red Hat Enterprise Linux (RHEL) 4 [21]. This machine runs Moab [9] for job scheduling, which leverages SLURM, the Simple Linux Utility for Resource Management [15], for job scheduling and remote process management.

We empirically build functions for $T(op)$ functions with a simple benchmark and gather LaunchMON overhead metrics by running an instrumented version of LaunchMON. The experiments with the instrumented version reveal that SLURM currently has no events that occur more frequently with increasing scale, a change that arose due to our interactions with SLURM developers. Thus, LaunchMON's contribution to Region A, the tracing cost, is 18 ms, regardless of scale. The other costs that are independent of scale, including those not in Regions A, B and C, amount to 12 ms. We measured other costs at small scales and then fit models for them. Figure 3 compares the performance predicted by our model (with various breakdowns) to the measured performance from 16 nodes to 128 nodes with the interval of 16 nodes (eight MPI tasks and one daemon per node). Both the model and the measured data demonstrate that *launchAndSpawn* scales well, taking less than one sec-

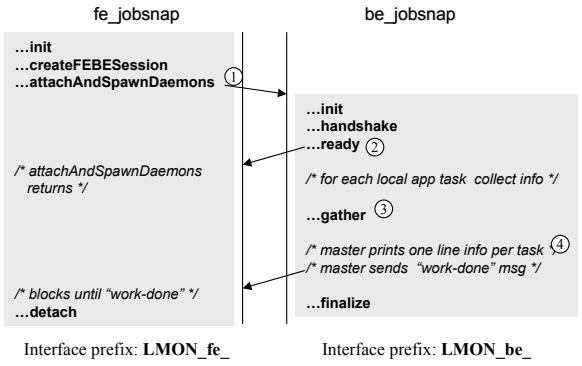

Figure 4. Jobsnap Operations

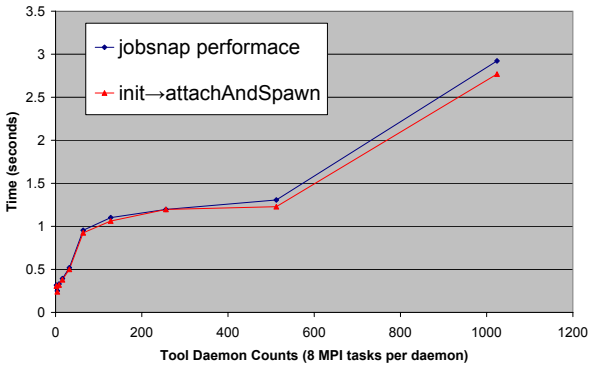

Figure 5. Jobsnap Performance

ond at 128 nodes (1024 MPI tasks). Further, the portions due to LaunchMON constitute only about 5.2% of that total time.

We have also ported LaunchMON to BlueGene/L. Our experiments on that platform demonstrate that LaunchMON has similar overheads on it. However, we found that the time for spawning the job tasks and tool daemons (i.e., $T(job)$ and $T(daemon)$) by *mpirun*, the RM on that system, were significantly higher. We are currently working with IBM to reduce these RM costs.

5 Tool Creation with LaunchMON

We demonstrate that LaunchMON supports a wide variety of HPC tools by implementing a new tool, Jobsnap, on top of it as well as integrating it into two existing tools, STAT and Open|SpeedShop .

5.1 Fast, Scalable Tool Creation: Jobsnap

Jobsnap gathers the distributed state of a parallel application including the task's personality (such as its rank and executable name), state (process state, program counter value and the number of active threads) and various memory statistics on both virtual and physical memory (i.e., high watermark and locked memory size). It then concisely presents this information as well as simple performance metrics including user time, system time and the number of major page faults to the user.

Figure 4 illustrates Jobsnap's use of LaunchMON. The Jobsnap front end launches lightweight back end daemons on each node being used by a running parallel job (step 1). Each daemon collects process snapshots for its local tasks identified in the RPDTAB (step 2). A master daemon then uses ICCL to collect the data from each back end daemon (step 3), which it merges and writes into a text file, one line per task (step 4). LaunchMON allowed us to implement this tool with about 100 lines of the front end code and 500 lines for the back end.

Figure 5 shows Jobsnap total performance on *Atlas*, as well as the time spent in the LaunchMON functionality that starts the tool daemons. Jobsnap scales well up to 4096 tasks, taking under 1.5 seconds total and requiring less than an additional half second compared to 512 tasks. At 8192 tasks, the maximum process count in our test, the total time is 2.92 seconds, of which 2.76 seconds are spent in the LaunchMON functionality. We speculate that the additional cost for the last processor count doubling is mostly due to the sub-optimal scaling characteristics of the RM functionality at this scale. In addition, we are considering a TBŌN architecture that would reduce the impact of collecting and printing information from each back end daemon.

5.2 Improving STAT Start-up

The Stack Trace Analysis Tool (STAT) [4] is a highly scalable, lightweight debugging tool. It gathers and merges multiple stack traces from a parallel application's processes to form a call graph prefix tree that identifies process equivalence classes (i.e., similarly behaving processes). A full featured debugger can attach to equivalence class representatives to perform root cause analysis at a manageable scale.

STAT uses a TBŌN model to gather its stack trace data; specifically, it uses MRNet for scalable tool communication and data collection and reduction. MRNet itself relies on a manual process to specify the target nodes and uses remote access protocols like *ssh* or *rsh*, which reduces the usage threshold of STAT as well as its portability.

To overcome these problems, we integrated STAT with LaunchMON, which identifies all applications tasks using the RM's RPDTAB and launches STAT's stack sampling daemons co-located with the application tasks. STAT also uses LMONP to broadcast MRNet communication tree information from the front end to the daemons. This infor-

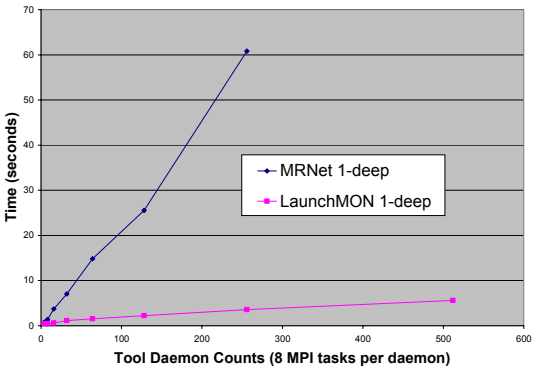

Figure 6. STAT Start-up Performance

mation was previously communicated through less scalable methods such as command line arguments or through a single file accessed by all daemons.

Integrating LaunchMON into STAT not only improves STAT's portability—by removing its dependence on system specific interfaces–but also enhances its performance. Figure 6 compares the performance of the two STAT versions using the native MRNet startup mechanism and using LaunchMON. Both tests were run on *Atlas* using the same tree topology (1-to-N).

Even with just four compute nodes, LaunchMON improves STAT daemon launch from 0.77 seconds to 0.46. At 256 compute nodes, the MRNet launching mechanism takes 60.8 seconds. In comparison, the LaunchMON implementation provides an improvement of over an order of magnitude, requiring only 3.57 seconds, of which 0.77 seconds occur in MRNet's handshaking protocol. At 512 compute nodes, the ad hoc approach consistently fails when forking an rsh process. If it had succeeded at that scale, it would take approximately two minutes, assuming the clear linear scaling trend (it could be worse). At that scale, LaunchMON launches all daemons in only 5.6 seconds.

5.3 Eliminating Root Daemons in Open|SpeedShop

Open|SpeedShop ($O|SS$) [17] is a parallel performance analysis toolset that includes support for PC sampling, hardware counter analysis as well as MPI and I/O tracing. It builds on DPCL's [11] binary instrumentation functionality. For parallel applications it uses that instrumentation mechanism to interface with the system's native APAI to locate all target application processes. However, DPCL does not contain any mechanisms to start its daemons along with the application: it either relies on a set of preinstalled root daemons, which is infeasible in production or security-sensitive environments, or requires a cumbersome manual launch of

Number of Nodes	2	4	8	16	32
DPCL	33.77s	34.27s	34.31s	34.32s	34. 66s
LaunchMON	0.606s	0.627s	0.604s	0.617s	0.626s

Table 1. $O|SS$ APAI Access Times

the daemons. The $O|SS$ approach also treats the RM process in the same way as the target application, including parsing its binary fully, which entails unnecessary overhead.

We integrated LaunchMON into $O|SS$ by replacing its central *Instrumentor* class, which encapsulates all interactions between the tool and the target application. Our new version uses LaunchMON to acquire RPDTAB instead of DPCL and then passes this information to the DPCL startup routines to connect to all target processes. We augmented the DPCL daemons so the front end can directly start them instead of a system daemon and added capabilities to connect back to the main tool. The only change necessary for this mode of operation was the inclusion of the appropriate back end calls inside the DPCL daemon startup routines.

These changes significantly improve the usability of Open|SpeedShop. In its current version users must start the daemons manually, including explicit identification of the application partition. Even worse, users must explicitly verify launch completion before using the tool daemons or the performance experiment would fail. The LaunchMON integration removes both usability issues to provide a fully transparent and automatic solution.

From a performance and scalability perspective, however, we did not expect any significant improvements since $O|SS$ already used an efficient launching mechanism. However, LaunchMON is specifically designed to read the information from APAI efficiently, unlike the general purpose remote instrumentation infrastructure of DPCL. Thus, the changes result in a roughly constant reduction in APAI access time. We see this effect in Table 1, which compares the time between initiating a performance experiment and when $O|SS$ has acquired all APAI information.

6 Conclusions

Many parallel tools, including debuggers and performance analyzers, must launch and control tool daemons. Large scale tools also often use additional middleware daemons for scalable communication. Launching and controlling these daemons are non trivial tasks that require an efficient, portable, secure solution that can be reused across a wide set of tools.

LaunchMON fills this gap using a general purpose, distributed infrastructure. Its main component, the LaunchMON Engine, directly leverages the services offered by the underlying resource manager on the target platform. Thus,

it uses efficient platform specific mechanisms to launch and to manage daemons that have accepted security properties. Tool developers can control this process with a set of libraries and APIs from the front end tool, the tool daemons running on the application nodes and, if applicable, middleware nodes. The latter two APIs include simple communication mechanisms that are useful for tool coordination.

We used LaunchMON in three tools: *Jobsnap*, a tool that gathers a distributed application's state; *STAT*, a tool to analyze distributed stack traces; and Open|SpeedShop, a general purpose performance analysis tool. In all three cases, LaunchMON provides significant advantages in terms of performance, efficiency, and usability as well as guaranteed scalability for launching and controlling daemons. Our case studies clearly demonstrate that LaunchMON's flexibility successfully provides the commonly needed mechanisms for scalable tool daemon management. Further, our LaunchMON performance model not only demonstrates the efficiency of LaunchMON functions but can also guide improvements in standard system services such as RM application launch.

References

[1] mpi-debug: Finding Processes. http://www-unix.mcs.anl.gov/mpi/mpi-debug/.

[2] Top 500 Supercomputer Sites. http://www.top500.org/.

[3] G. Almási, C. Archer, J. G. Castaños, J. A. . Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. M. K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and and B. Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3), 2005.

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Long Beach, CA, 2007.

[5] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a Traditional Debugger to Debug Massively Parallel Applications. *Journal of Parallel and Distributed Computing*, 64(5):617–628, 2004.

[6] R. Brightwell and L. A. Fisk. Scalable Parallel Application Launch on Cplant. In *Proceedings of Supercomputing*, Denver, Colorado, 2001.

[7] 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.

[8] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting*, Italy, 2005.

[9] Cluster Resources . Moab Workload Manager. http://www.clusterresources.com/.

[10] J. Cownie and W. Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, 1999.

[11] L. DeRose, T. Hoover, and J. Hollingsworth. The Dynamic Probe Class Library — An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2001.

[12] M. Gerndt, K. Fürlinger, and E. Kereku. Periscope: Advanced Techniques for Performance Analysis. In *Proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005)*, pages 15–26, Malaga, Spain, 2005.

[13] IBM. Parallel Environment for AIX 5L and Linux V4.3.1 MPI Programming Guide. http://publib.boulder.ibm.com/epubs/pdf/am106m00.pdf.

[14] V. Ivars, A. Cortés, and M. A. Senar. TDP_SHELL: An Interoperability Framework for Resource Management Systems and Run-time Monitoring Tools. In *Proceedings of Euro-Par*, Dresden, Germany, 2006.

[15] M. Jette and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of ClusterWorld Conference and Expo*, San Jose, CA, 2003.

[16] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovlra. *Workload Management with Load Leveler*. IBM Redbooks, 2001.

[17] Krell Institute. $Open|SpeedShop$. http://www.openspeedshop.org/.

[18] Livermore Computing. CHAOS: Linux from Liremore. http://www.llnl.gov/LCdocs/chaos/.

[19] R. McDougall, A. Cockcroft, E. Hoogendoorm, E. Vargas, and T. Bialaski. *Resource Management*. Prentice Hall, 1999.

[20] B. Miller, A. Cortés, M. A. Senar, and M. Livny. The Tool Daemon Protocol (TDP). In *Proceedings of Supercomputing*, Phoenix, Arizona, 2003.

[21] Red Hat. Red Hat Enterprise Linux 4.5.0 Reference Guide. http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manul/pdf/Reference_Guide450.pdf.

[22] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of Supercomputing*, Phoenix, AZ, 2003.

[23] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[24] TotalView Technologies, LLC. TotalView. http://www.totalviewtech.com/TotalView/.

[25] J. S. Vetter, S. R. Alam, T. H. D. Jr. M. R. Fahey, P. C. Roth, and P. H. Worley. Early evaluation of the Cray XT3. In *Proceedings of the 20th IEEE International Parallel and Distributed Pro cessing Symposium*, Rhodes Island, Greece, 2006.

[26] W. Yu, J. Wu, and D. K. Panda. Scalable Startup of Parallel Programs over InfiniBand. In *Proceedings of the International Conference on High Performance Computing*, Bangalore, India, 2004.