On Scalability for MPI Runtime Systems

George Bosilca
ICL, University of Tennessee Knoxville
bosilca@eecs.utk.edu

Thomas Herault
ICL, University of Tennessee Knoxville
herault@eecs.utk.edu

Ala Rezmerita
INRIA Saclay - Ile de France, LRI
rezmerit@lri.fr

Jack Dongarra
ICL, University of Tennessee Knoxville
dongarra@eecs.utk.edu

***Abstract*—The future of high performance computing, as being currently foretold, will gravitate toward hundreds of thousands to million node machines, harnessing the computing power of billions of cores. While the hardware part is well covered, the software infrastructure at that scale is vague. However, no matter what the infrastructure will be, efficiently running parallel applications on such large machines will require optimized runtime environments that are scalable and resilient. More particularly, considering a future where Message Passing Interface (MPI) remains a major programming paradigm, the MPI implementations will have to seamlessly adapt to launching and managing large scale applications on resources several levels of magnitude larger than today.**

In this paper, we present a modified version of the Open MPI runtime that has been adapted towards a scalability goal. We evaluate the performance and compare it with two widely used runtime systems: the default version of Open MPI and MPICH2; using various underlying launching systems. The performance evaluation demonstrates a significant improvement over the state of the art. We also discuss the basic requirements for an exascale-ready parallel runtime.

I. Introduction and Motivation

MPI implementations usually feature two separate components: the library, which implements the MPI standard, and an implementation-specific runtime. MPI users are most familiar with the runtime through its main command: `mpirun`. The major roles of the runtime are:

launch Launch the MPI application's processes. This role is shared between the implementation runtime and the parallel machine scheduling / launching mechanism.

connect Help the MPI library establish the necessary connections between the processes. Depending on the network used, hardware or configuration specific issues, the connection information (also known as the business card, or the URI of a process) may not be known before the MPI processes are launched. It is then necessary to distribute this information through an out-of-band messaging system. In most MPI implementations, this role is dedicated to the runtime.

control Control the MPI processes: ensure that in case of a crash the entire environment is gracefully cleaned; depending on the operating system and implementation, forward signals to the MPI processes; ensure that completion codes are returned to the user command: `mpirun`.

io Forward the standard input/output: users usually expect that the information printed on the standard output should appear on the standard output of the command they launched. Because the command they launched does not necessarily run on the same machine as where the print is issued in an MPI application, it is necessary to ensure the transmission of this information.

While each of these points introduces an overhead on the runtime system, they do not apply it in the same context. The roles *launch* and *connect* happen only during the startup step of the parallel application, while role *control* is a finalization or error correction step and role *io* is usually stretched out over the entire lifetime of the application. In this work, we will focus on roles *launch* and *connect*.

For a long time, simple strategies provided enough performance at the runtime level, to allow library implementors to focus their improvement efforts on the MPI library itself. Unfortunately, with the growth of the parallel machines, start-up and finalization times have become significantly more expensive. A direct consequence is a notable reduction in scientific outcome and an increase on the energy consumption per scientific result, directly proportional with the scale of the parallel run.

The general evolution of these supercomputers, as witnessed by the Top500[1], denotes a huge progression of the number of cores, simultaneously with the number of computing nodes. Vanilla versions of the two major open source MPI implementations (Open MPI [1], and MPICH [2]), although very versatile and adaptable to a large range of computing environments, can take minutes to launch a large scale run on the largest supercomputers.

In this paper, we present an adaptation of the Open MPI Runtime Environment (ORTE [3]) towards large scale systems. In this context, we implemented the Scalable Launch / Resource Reducing algorithm presented in Section III, and in Section V we evaluate it on a thousand nodes. Comparisons

[1] http://www.top500.org/

with vanilla runtimes of Open MPI and MPICH2 with different underlying launching mechanisms show a significant improvement in the startup performance. In Section VI, we discuss how this performance improvement can be projected on larger scale machines, and the key features that will be necessary from the launching systems to enable generic large scale runtime implementations.

II. Related Work

The scalability and performance of the parallel process startup have been studied extensively. Several projects have successfully decreased the launching time of the parallel application, addressing the role *launch* in Section I. Unfortunately, launching parallel applications involves multiple steps, including the role *connect*.

This role includes exchanging information with other peers. Peers must know each other, through contact information, which can only be obtained once a process is started. In this respect, the processes launched by the role *launch* are isolated, knowing only one process, the *mpirun*. Therefore, initial information exchanges are centralized, with limited chances to scale.

To spawn and manage parallel jobs, the MPICH2 project [2] provides several different internal process managers such as Hydra [4], [5], MPD [6], SMPD, Gforker and Remshell. Until the 1.2.x series of MPICH2, MPD was the default process manager. MPD is a ring of daemons on the machines, available to run MPI applications. However, this approach has proved to be non-scalable, and starting with the MPICH2 1.3.x series, Hydra becomes the default process manager. Designed as a response to the limitations (applied to modern HPC systems) of the first generic process management interface (PMI-1) [5], that is implemented in MPD, Hydra includes solutions related to the scalability for a large number of cores on a single node and efficient interaction with hybrid programming models that combine MPI and threads.

Another approach to handle the process management for parallel programming on large-scale systems involves external process managers that are usually coupled with a reservation system. This is the case of SLURM [7], LSF[2], Torque[3] or PBS[4].

They usually consist in a set of persistent daemons running on the machines and a centralized manager that allows to make reservations and monitors the resources/jobs. The demons, that are executed on the machines, are generally launched at boot time, and are connected to the centralized manager and to their peers based on a predefined topology using well-known ports, building the internal communication infrastructure. This infrastructure is used to forward commands for jobs initialization, to report system status, and to get information about jobs and job steps that are running or have completed.

While these external tools significantly improve the process management on large-scale systems, our work differs in that our implementation provides the necessary services to initialize MPI processes (role *connect*). By doing so, it may use the launching services of the aforementioned tools.

On large scale machines, like BlueGene/L [8] for example, the computing nodes can not run multiple processes simultaneously. Therefore, a specific execution environment is implemented for the machine and a control and I/O daemon (`ciod`) is executed on I/O nodes. The `ciod` implements system management services and supports file I/O operations by the compute nodes. All compute and I/O nodes of BlueGene/L are interconnected by a tree network that supports fast point-to-point, broadcast, and reduction operations on packets. The partitioning of the system into compute, I/O, and service nodes leads to a hierarchical system management software that provides a scalable application launch in the specific environment of the BlueGene/L.

The distributed approach of building a $k-$ary tree to implement the *launch* role of the runtime system has been extensively studied in systems, like taktuk [9], MRNet [10], and in the context of MPI, ScELA [11]. Tree-based parallel launching of a user command is also used, hidden from the caller, in most of the other large-scale runtime systems cited above. In this work, we advocate that the runtime cannot simply rely on an efficient launching system, but must ensure a better integration, to help the other roles, especially the *connect* role. In the case of ScELA, the $k-$ary tree is used as the single possible communication infrastructure for the runtime to support this role and the others. In this work, we use the underlying launching tree to exchange contact information at the runtime level, and let the runtime system build for itself an arbitrary communication infrastructure to support these roles. We demonstrate in the evaluation that building this infrastructure enables a higher flexibility (e.g. establishing redudant links to support fault-tolerance), and also helps provide better performance than the previous approaches.

III. Scalable Launch and Reduced Resource Consumption

Consider a distributed system where all computing nodes provide a remote execution service. Given a list of nodes, that is a subset of the nodes of the system, a message routing topology, and an application, the goal is twofold:

- Spawn the application processes on the list of nodes,
- establish a communication infrastructure that follows the routing topology.

We aim at attaining this in a scalable way, by minimizing both the launching time and the resource consumption. Scalability means that the time to deploy the set of processes should be logarithmic with the number of nodes in the

[2] http://www.platform.com/

[3] http://www.clusterresources.com/products/torque-resource-manager.php

[4] http://www.pbsgridworks.com/

```
Algorithm 1: Spawning Along a Tree (Phase 1)
  Inputs:
    $\alpha$: Identifier of the parent
    $\rho$: rank of this process
    $Nodes$: set of nodes (Only on the first process)
  Variables:
    $\tau$: local Identifier
    $Children$: set of couples $(node, rank)$
    $Desc$: set of couples $(node, rank)$
    $\eta$: integer (number of connected children)
  Functions:
    direct$(S, p)$ : subset of $S$ with the direct children of $p$
    offsprings$(S, p)$ : subset of $S$ with all descendants of $p$
1 - sub spawn(C)
      if $C = \emptyset$ then Phase 1 completed else
          foreach $(n, r) \in C$, do
              use the launch service of node $n$ to spawn
              the runtime process on $n$ with parameters
              $\tau, r$
      Start Algorithm 2 (phase 2)
2 - At Start
      $\eta = 0$
      $\tau =$ new Identifier
      if $\alpha \neq \perp$ then   Connect to $\alpha$
      else
          $Children =$ direct$(Nodes, \rho)$
          $spawn(Children)$
3 - Recv $NodeList$ from $\alpha$
      $Desc = NodeList$
      $Children =$ direct$(Desc, \rho)$
      $spawn(Children)$
4 - Accept connection from $Id$
      Let $d =$ offsprings$(Desc, \rho)$
      Send $d$ to $Id$
      $\eta = \eta + 1$
      if $\eta = |Children|$ then Phase 1 completed
```

subset, and reduced resource consumption means that the overhead in the amount of resources (number of established connections) should also be logarithmic in the number of nodes. To achieve this, we divide the goal in three overlapping phases:

1) Spawn the runtime daemons following a predefined spawning tree topology,
2) Exchange contact information between daemons, and establish connections to enable the routing topology,
3) Launch the application processes locally.

The first phase relies on the distributed launch service. This algorithm, presented in Algorithm 1, follows a simple strategy of deployment along a spanning tree. Supported trees include $\delta-$ary tree (independent of the routing topology), or the spanning tree of the routing topology. The first process is launched by the user, initializes its α variable to $\perp$, and gets the list of nodes from the user parameters. It then extracts the list of its direct children from this list, and spawns the corresponding runtime processes.

Every runtime process when launched, first connects to its parent in the spawning tree, based on the contact information of its parent provided as a launching argument. When connected with the parent, it identifies itself using its newly allocated contact information. The parent then extracts the subset of nodes located under this child process in the spawning tree, and provides it with this list.

When receiving the list of nodes, the child divides it into two sets: it's direct descendants, and the other nodes. Direct descendants are spawned by contacting the remote launch service of the appropriate node, and providing as the parent argument its newly created contact information. The list of other nodes will be transmitted recursively when the children contact this process. Once all children have contacted this process, the phase one is locally completed, and although all processes in the system have not yet been launched, it enters the second phase.

```
Algorithm 2: Sharing Contact Information (Phase 2)
  Inputs:
  All inputs of Algorithm 1 $(\alpha, \rho, Nodes)$
  and the following variables of Algorithm 1:
   $Children$: set of $(node, rank)$ for each children
   $\tau$: local Identifier
  Variables:
   $OldChildren$: set of $identifier$
   $KnownURI$: set of $(identifier, rank)$
   $ChildrenURI$: set of $(identifier, rank)$
1 - At Start  – from Algorithm 1
      $OldChildren = \emptyset$
      $KnownURI = \{(\rho, \tau)\}$
      $ChildrenURI = \emptyset$
      if $|Children| = 0 \wedge \alpha \neq \perp$ then
          Send $KnownURI$ to $\alpha$
2 - Recv $URIList$ from $p$
      foreach $o \in OldChildren$ do
          Send $URIList$ to $o$
      if $p \neq \alpha$ then
          Send $ChildrenURI$ to $p$
          $OldChildren = OldChildren \bigcup \{p\}$
      $ChildrenURI = ChildrenURI \bigcup URIList$
      if $|OldChildren| = |Children|$ then
          Send $ChildrenURI$ to $\alpha$
     else
          $KnownURI = KnownURI \bigcup URIList$
```

The second phase consists of exchanging the contact information to build a global knowledge of all contact information in the system. This knowledge will enable an arbitrary routing topology to establish all needed connections for the remainder of the run. The algorithm, presented formally in Figure 2, works as follows: each process keeps a list of children that have already entered the phase 2 algorithm. At the beginning, this list is empty, and the only known identifier is it's own. This identifier is sent to the parent if the process is a leaf. Non-leaf processes accumulate this contact information in the $KnownURI$ variable. Each time they learn a new one, they forward it to all the older children. When all of them are known, the accumulated buffer is sent to the parent. When such a contact information buffer is received by any other process (parent or child), and all children are connected, it is forwarded to all the children. Thus, the algorithm is an all-gather algorithm based on flooding. However, to avoid sending too many small messages, like the classical flooding algorithm would do, it accumulates child information into a single message to the parent, thus favoring a smaller number of larger messages (there are multiple identifiers in the messages that are simply forwarded). When the algorithm completes, $ChildrenURI$ contains the identifiers of all the processes.

Phase 3 executes simultaneously with phase 2: when all remote processes have been spawned during phase 1, it spawns the local application, and contributes to a simple non blocking broadcast algorithm to signal the start of the target application. This broadcast algorithm follows the user-defined routing topology. If the routing decides to communicate directly with one of the runtime processes whose identifier is not yet known, the communication is delayed until phase 2 algorithm discovers the missing identifier. Thus, the broadcast will proceed at worst when all identifiers have been collected by the phase 2 algorithm.

A. Analysis of the algorithms

We present here the cost analysis of the proposed algorithms, considering n as being the number of nodes, and a $\delta-$ary tree as the spawning topology.

Number of connections: To evaluate the Reduced Resource Consumption requirement of the algorithm, we estimate how many connections are created during the three phases. In our system, all messages of phase 3 or later are routed by the user-defined topology. So, we consider only messages of phase 1 and 2, and evaluate how many additional connections our algorithm initiates.

Both phase 1 and 2 algorithms introduce only communications between a node and its parent (α) or its children. The number of children a node has is $|\text{direct}(Nodes, \rho)|$ which is bounded by δ. Thus, this algorithm expects any runtime process to handle at most $\delta + 1 + k$ connections, where k is the number of connections required by the chosen topology.

Completion time: To evaluate the completion time, we consider a synchronous system, where each communication takes a single time step. Section V presents a more practical evaluation of the implementation. The phase 1 converges in $O(\log_\delta n)$ time, since its behavior is similar to a simple broadcast along the spawning tree. This is identical for the phase 3. During the phase 2, one can distinguish two kinds of messages: some messages go up on the spawning tree, accumulated by the nodes, others flow down on the tree. The duration of the algorithm is evaluated by considering the highest leaf f in the tree: let h be the height of f, it takes $\max(h, \delta - 1)$ emissions to send the messages and accumulate them at the root of the tree. Then, it takes at most h phases to let this information flow down if there is a branch of depth h that does not hold the leaf f. Thus, the time of phase 2, and the whole algorithm is $O(\max(\log_\delta n, \delta))$.

Communications: We now consider the amount of information that is exchanged during the three phases of the algorithm, and the number of messages. There are different kinds of data transmitted: nodes names, identifiers, and rank lists. Additionally, phase 1 will undergo n spawn commands, which will transmit a constant-size information (the command to launch, the rank and the identifier of the parent node, as well as the node on which to launch the command). Each node that is not a leaf will execute at most δ spawn operations.

Phase 3 depends on the user-command to launch, and on the routing topology chosen by the user. So, we consider the messages transmitted during phases 1 and 2 only. In phase 1, when a process connects to its parent, the parent sends the list of pairs $(node, rank)$ belonging to offsprings$(Desc, \rho)$. Each node that is not a leaf will thus send $O(\delta)$ messages of size at most $O(n/\delta)$. During phase 2, each node that is not a leaf nor the root will send one message to its parent, $O(\sum_{k=1}^{\delta-1} k) = O(\delta^2)$ messages to its children, and any message received from it's parent to its children. These messages consists of aggregated contact information from each process at distance 1 from one of its ancestors, but not its parent. Information coming from processes at distance more than one has been accumulated. Let h be the height of the process, there are $O(h \times (\delta - 1))$ such messages.

In total, in the worst case, this consists of $O(\delta^2 + \log_\delta n \times (\delta - 1))$ messages. These messages hold the contact information of the processes; each message contains information that is complemental to the other messages that passed through this process, and at the end of phase 3, all processes have *all* the information, thus the accumulated size of the information sent by a process that is not a leaf during this phase is $O(n)$.

IV. Background: ORTE and Open MPI

In this section, we briefly present how the runtime of Open MPI, ORTE, coordinates the deployment, control and monitoring of an application. We discuss the role of each software component that is significant for this work, and how they interact with each other. Then, we will present the implementation of the Scalable Launch / Resource Reducing Algorithm in the next section.

A. Process Architecture

One can distinguish three kinds of processes that interact together to implement the runtime environment of Open

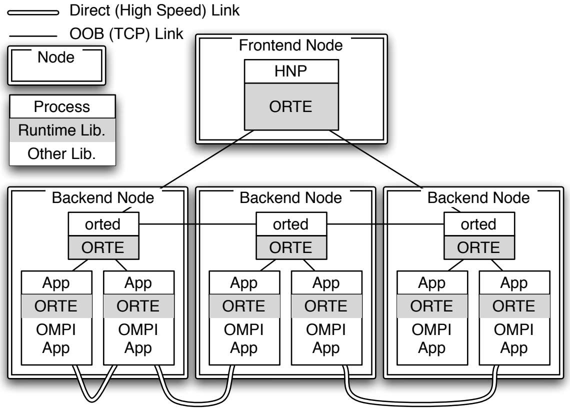

Figure 1. Open MPI processes and software stack

MPI. Figure 1 gives a representation of these processes and their software stack.

HNP The Head Node Process consists solely of runtime environment: it is usually aliased to the `mpirun` command, and its main role is to deploy and coordinate orte daemons (`orted`) on the parallel machine. This process is usually launched by the user or the batch scheduler on the front-end machine with a list of computing machines to serve as support for the parallel application.

orted ORTE Daemons are runtime environment processes: they are the link between the HNP and the application processes. One of such orted is launched per computing node that belongs to the run.

App MPI Applications are processes that are linked with the Open MPI library (which implements the MPI standard), and the ORTE library. For MPI messages, they directly communicate with each other. For the out-of-band messages, they relay information through the ORTE layer with the local orted process (e.g. forwarding a standard output message to the `mpirun` command, or getting the port on which a direct MPI connection can be established). There are as many `mpiapp` processes as necessary on each node, to fulfill the users requirements. All `mpiapp` processes on a node that belong to the same MPI application are connected to the same orted process.

B. Code Architecture

The Open MPI RunTime Environment (ORTE), as well as the Open MPI library, are designed in a modular way using the framework of the Modular Component Architecture (MCA). Modules developed in the MCA framework follow an object-oriented approach, with well defined public interfaces, and potentially multiple alternative implementations of the interface. Which module is used for a specific setup is decided at runtime, based on a selection process that involves the hardware capabilities and user preferences.

ORTE includes 14 different components, which together define its general behavior. Examples of such components are `errmgr` to handle errors, `iof` to handle Input/Output forwarding, etc... A detailed presentation of ORTE is outside the scope of this paper. At the implementation level, most of the modifications have been encapsulated in a new Process Launching Module (PLM), prsh, presented below. In addition, the Scalable Launch / Resource Reducing algorithms rely on a routing topology, defined in the context of ORTE by a *routed* module. The routing module implements different strategies to route messages between daemons, and between daemons and the Head Node Process.

C. PLM prsh

As described previously in Algorithms 1 and 2, the algorithms internally rely on two topologies: one for spawning the daemons and one for exchanging the messages during phase 3 and after. The message routing topology is defined by the choice of the *routed* MCA module.

The spawning topology is defined by the PLM prsh optional argument, `prsh_spawning_degree`, which determines the value of the δ parameter in the algorithms of Section III. It uses a global ORTE parameter to define its underlying launching mechanism, and assumes that any node is capable of issuing commands given by this parameter to launch an orted process. In the special case of δ being null, the spawning tree used is identical to the message routing tree defined by the selected routing topology (argument of the `routed` MCA parameter).

D. Vanilla version of ORTE

To help understand the performance of the prsh PLM, we describe here the vanilla version of ORTE based on the rsh PLM and the slurm PLM. In the vanilla version of ORTE, all orted are launched by a PLM with the same identifier: the HNP identifier. As a consequence, all orted will first connect to the HNP (in a star network), and when this is done, will receive from the HNP the full system map (which includes the identifiers of all nodes that are registered to the HNP). Once this map is known, the selected routing topology can be activated, since all orted know how to connect to any other orted or HNP. From this point on, all messages are routed following the selected routing topology. Then, the application is started by broadcasting the launch command on the routing tree, as described above.

We compare with two PLMs: slurm, and rsh. The first uses the Simple Linux Utility for Resource Management [7]. Through an interactive SLURM allocation (the `salloc` command), HNP obtains the list of hosts and the number of processes to be launched. Then, in a forked process from the HNP, it makes a call to the `srun` command that launches all orted on the set of all machines. Such an approach relies on the capabilities of the SLURM environment to launch the runtime in a efficient and scalable manner.

The second, rsh, implements a simple loop of rsh commands, forking new rsh processes, in the main loop of the PLM. A user-level parameter defines how many of these children co-exists simultaneously, as a way to control the rate of connections, and the amount of processes on the head node. A producer / consumer algorithm continue to fork new rsh processes, until all of them have been launched. Then, the PLM waits until all orted have connected to the HNP, and proceeds with exchanging the node map.

V. Performance measurements

In this section we present the performance measurements of the Scalable Launch / Resource Reducing implementation, and compare it with other MPI runtime environments. All the experiments were conducted on Grid'5000.

A. Experimental setup

Grid'5000 [12] is a French national testbed, dedicated to large scale distributed system experiments. The platform features 13 clusters distributed over 9 sites in France, each with 90 to 340 PCs. The platform gathers approximately 7,000 cores featuring two architectures (Xeon and Opteron).

Within each cluster, the nodes communicate through Gigabit Ethernet links and the communications between the clusters are made through 10GB dedicated dark fibers of the Renater French Education and Research Network.

One of the major features of the Grid'5000 testbed is the ability for the user to boot all reserved nodes with his own environment (including the operating system, distribution, libraries...). We used this feature to run all our measurements in a homogeneous environment. All the nodes were booted under Linux 2.6.26, all the MPI implementations were installed in the same environment, and all the tests were run in a dedicated mode; process and network wise, no other users were running any experiments on the same machines, or on the same networks during the evaluation.

We compare our implementation, that we call ORTE PLM prsh, with four other setups: the vanilla implementation of ORTE from the Open MPI trunk, checkout 24321, using the PLMs rsh and slurm, MPICH2 version 1.3.2p1 using Hydra with rsh and SLURM launchers, and MVAPICH version TODO:1.2?, using the ScELA launcher. All versions are compiled in optimized mode. For the experiments using SLURM, we used SLURM version 1.3.6, as distributed by Debian version 5.0.8 Lenny. Experiments based on rsh were using ssh as a remote shell system.

B. Methodology for performance measurements

To highlight each feature of an execution environment we defined two testbeds. The first one, using /bin/true application, underscores the time of deployment of the runtime environment. We launch, in parallel, one /bin/true process per node, for a varying number of nodes, using the MPI runtime. Both MPI runtimes that we use (Open MPI and MPICH2) are able to launch non-MPI commands. The main interest of this testbed is to evaluate solely the operating costs due to the runtime: the MPI initialization and finalization overheads are absent from this picture.

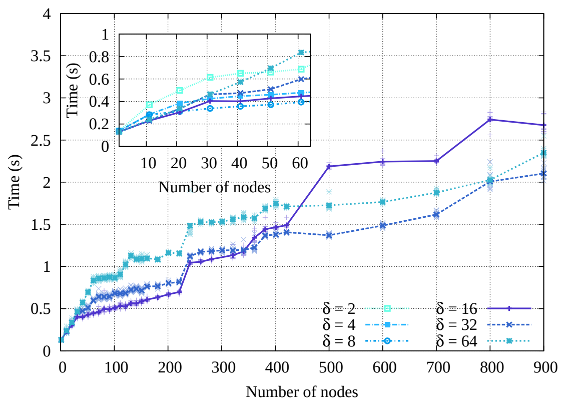

Figure 2. Execution time of /bin/true with the PLM prsh at various δ parameters (routed $\delta-$ary tree)

The second one, using an empty MPI application, highlights the initialization and finalization time of an MPI application. During this initialization, through the MPI routine MPI_Init, the exchange of contact information takes place, enabling communication between MPI processes. After the MPI_Init follows immediately the MPI routine MPI_Finalize which finalizes the MPI application and tears down existing connections. This testbed evaluates the overheads due to runtime in a parallel machine: everything between the MPI_Init and the MPI_Finalize calls can be considered as useful work for the user.

C. Evaluation

Launching overhead: The first experiment aims at evaluating the impact of the value of the prsh_spawning_degree parameter on the performance of the PLM prsh. As described previously, this parameter sets the degree of the spawning tree, δ. We measure the overall time of mpirun /bin/true on a varying number of nodes (one process per node), for different values of δ, and present the measures in Figure 2. Light color points represent the different measures, and darker lines represent the mean values for these measures. As expected when the number of processes to launch is smaller than δ, the execution time progresses linearly.

The slope changes when the δ first processes take charge of a significant part of the launch, exhibiting a nearly logarithmic progression. However, communications induced by the phase 2 algorithm, to enable arbitrary routing, introduce a measurable linear component on the performance. After 220 nodes, the execution time increases by approximately half a second for all δ values. This is mainly due to the experimental platform configuration: nodes are allocated

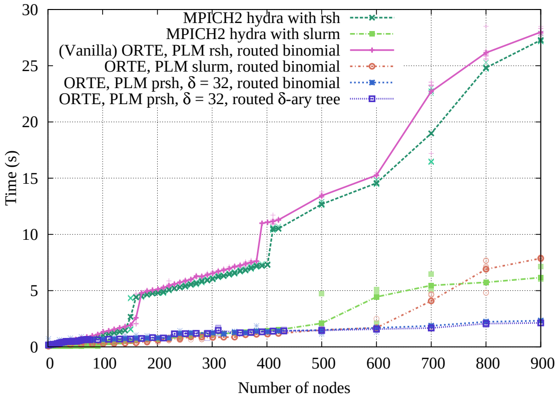

Figure 3. Execution time of `/bin/true` with the PLM prsh, and comparison with other runtimes

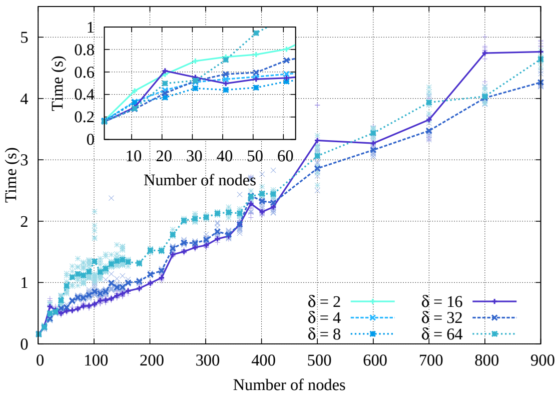

Figure 4. Execution time of an empty MPI application with the PLM prsh at various δ parameters (routed $\delta-$ary tree)

on remote clusters, with variable response times for the `rsh/ssh` service.

With $\delta = 64$, the overall execution time is consistently higher than with $\delta = 32$. It is thus more efficient to distribute the launching load on more processes than to execute the loop 64 times. This threshold depends mostly on the underlying remote service mechanism: for a service faster than `rsh/ssh`, a larger δ might be more efficient. Experiments with smaller values of δ at smaller scale (see zoomed-in part of the figure) demonstrate that distributing the load on too many nodes (using small δ values) introduces a larger overhead, due to the latency of the ssh operation.

Launching overhead comparison: We compare the PLM prsh with state of the art runtime systems in Figure 3. The experimental setup is identical to the preceding experiment. We used $\delta = 32$ for the PLM prsh, as it was providing the most consistently performant results. Both other rsh-based runtimes (MPICH2 Hydra and vanilla Open MPI) present an execution time significantly higher than prsh. This remains true independently of the message routing topology used (binomial tree and $\delta = 32-$ary tree). The shape of both vanilla ORTE and MPICH2 curves are similar: it is close to a linear progression, with gaps at 128 and 390-400 nodes. In ORTE, we were able to trace the cause of these gaps: they are due to a connection storm happening at the HNP level (all the launched processes are calling back to the HNP as it is their only point of contact). When the connection queue in the kernel becomes overloaded, connection packets are dropped by the operating system hosting the HNP, and the TCP protocol reemits the packets after a delay of three seconds. Based on the nearly identical observations of the vanilla ORTE and MPICH2 Hydra with rsh launcher, we suspect Hydra is afflicted by the same behavior.

Using a launcher dedicated to clusters management, SLURM, both vanilla ORTE and MPICH2 Hydra achieve a notably higher performance: not being responsible to launch each and every rsh command, they remain in the main loop to accept incoming connections, and avoid the connection re-emission penalty. However, because the only contact information that can be passed to the launcher is the contact information of the only existing process at the time of the launch (*mpirun*), the approach remains centralized. Therefore, every daemon has to connect to *mpirun* to bootstrap the contact information exchange. At larger scales (above 500 nodes), the overhead of handling an increasing number of connections becomes significant, and the distributed approach of the PLM prsh is able to outperform even a scalable launcher.

One can also see by comparing both prsh measurements that the message routing topology, has no measurable effect on the performance of a non-MPI application. When launching such an application, the only impact of this topology is on the phase 3 of the launch process: different broadcast trees are used. However, both broadcast trees (binomial tree and $\delta-$ary tree) appear to provide similar performance.

MPI overhead: Figure 4 is similar to Figure 2: it presents the evaluation of the impact of the δ parameter, on an empty MPI application. The behavior is significantly different in this case: all versions keep a consistent behavior, and the progression of the execution time becomes linear.

The Open MPI library, when it enters its `MPI_Init` routine, starts by exchanging a significant amount of information above the out-of-band messaging system of ORTE, during an operation called the modex. This operation consists of an all-gather of the contact information of the MPI processes themselves (including low level communication device connection information). This modex operation dominates the launching time, and introduces a linear progression. To tackle this overhead, it will be necessary to adapt the modex operation, which is left for future work.

MPI overhead comparaison: As illustrated by Figures 5 and 6, when compared to the state of the art run-

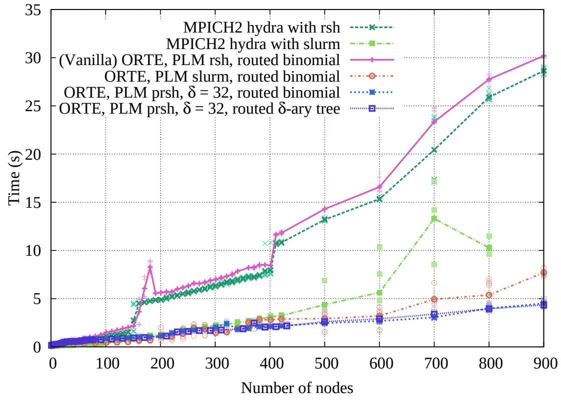

Figure 5. Execution time of an empty MPI application with the PLM prsh, and comparison with other runtimes

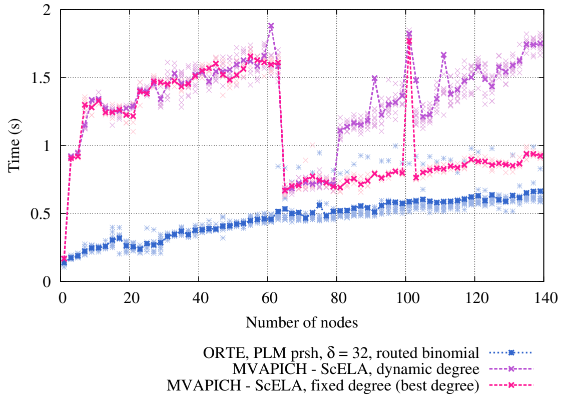

Figure 6. Execution time of an empty MPI application with the PLM prsh, and comparison with ScELA

time environments, the PLM prsh still provides significant reductions of the Init / Finalize overhead. Based on the data in Figure 5, at 900 nodes the PLM prsh provides a factor 5 speedup when compared to other rsh-based launchers. In the case of SLURM-based launcher, even if this factor is reduced, it remains apparent starting from 500 nodes and going up to about 2 at 900. All runtimes, including the SLURM-based launchers, exhibit linear behaviors when launching MPI applications. However, the slope of the PLM prsh is the smallest one, potentially offering the most scalable approach.

When comparing the two PLM prsh curves, one using the $\delta-$ary tree the other a binomial tree as the underling routing topology, one can see small variations in the execution time. The modex operation is done using a all-gather above the routing tree. Using different trees to complete this operation does not introduce a significant performance difference; however another implementation for the all-gather could take advantage of the routing information to improve the dissemination of information.

ScELA is capable to launch any kind of application, but in our experiments using the `/bin/true` benchmark, it was failing to establish a connection with the application processes. This does not prevent the run of the benchmark, but the exceptions raised to manage the absence of connection seem to introduce a significant overhead, making the overall runtime of the `/bin/true` benchmark significantly higher than the MPI benchmark. As a consequence, we focus our comparison with the MPI benchmark. Among the MPI applications, ScELA is also dedicated to launch MVAPICH applications, as ORTE is dedicated to launch Open MPI applications, and Hydra, MPICH applications. Since MVAPICH is designed to work on Infiniband clusters, we limit our comparison to the largest cluster featuring infiniband system at our disposal: the 140-nodes Graphene cluster hosted at the Nancy site of Grid'5000.

Figure 6 thus compares the performance of the empty MPI benchmark, with the prsh and ScELA, from 1 to 140 nodes, using the two available strategies for ScELA: dynamic selection of the degree, and fixed degree. For the fixed degree strategy, we tuned for each number of nodes the best value for the degree, and represent only the measurements of the best value. One can see that the prsh implementation of the $\delta-$ary tree approach outperforms the ScELA implementation of the same idea at small scale (below 64 nodes). At 80 nodes and above, the fixed degree strategy performs better than the dynamic degree for ScELA, but the fixed degree implementation of the prsh continue to provide better performance. At this scale, the exchange of all contact information, that is done in prsh to enable an arbitrary communication infrastructure, but not in ScELA, that uses the spawning tree to provide all communications, does not impact negatively the performance.

VI. Discussion

A linear regression of the empty MPI applications launching times for both rsh-based frameworks, and for the PLM prsh, estimates that the PLM prsh has a progression slope an order of magnitude smaller than the others rsh-based launchers. The estimation forecasts that up to 20,000 computing nodes can be launched in less than a minute. It is thus reasonable to consider it for current supercomputers. Improvement in the MPI library and the routing systems will need to be considered to prepare for larger scales. From the experience harnessed while developing the PLM prsh, we isolate two major features that we think crucial to obtain a reasonable launching / managing overhead:

1) Parallel Launching: Parallel launching remains a key component to reach a lower launching time, one cannot afford to iterate over a set of launching commands. Using a more efficient approach is required, such as a recursive launching process as we did with the PLM prsh, or such as a dedicated launching system, more integrated with the machine scheduler, like SLURM.

2) Distributed Management: This is a key point to keep a low overhead at larger scale, as illustrated by Figures 5 and 3. Requesting all processes of the runtime to connect back to a single process creates an obvious bottleneck at scale. This bottleneck remains even if an efficient routing strategy is applied after the initial storm.

However, simultaneously achieving points 1 and 2 is often made difficult by the launching systems available on parallel machines: on most parallel launchers, only a single node can issue launch commands. In other words, even the most scalable launcher only provides a single point of contact, and thus annihilates the benefit of the parallel launching. To avoid the single point of contact, it is imperative that as processes are launched, they publish their contact information, and that other processes, launched afterwards, use this new contact information to connect to the runtime infrastructure. To the best of our knowledge, most of the popular launching systems don't allow for this kind of interaction between the MPI runtime and the underlying launching system. As a result, they are able to launch in a very short time large number of processes, but they have to pay the expensive cost of building a communication infrastructure, on their own, from scratch. It seems beneficial to build this runtime communication infrastructure in cooperation with the launching system, directly during the launch progress.

VII. Conclusion

In this paper, we presented a Scalable Launch / Resource Reducing algorithm to enable the launch of large scale MPI applications. The algorithm comes in three phases: first a deployment phase along a $\delta-$ary tree; second a contact information exchange phase, which implements an all-gather operation; and last an application launch phase that runs on top of the runtime-specific routing. We analyzed the phases of this algorithm, and highlighted their scalable properties: logarithmic launch time and number of messages per nodes. The algorithm has been implemented in the PLM prsh, a module for the Open MPI Runtime Environment (ORTE). We evaluated the performance of this implementation on a thousand node platform, demonstrating a significant improvement over the state of the art runtime environments for MPI: Open MPI, MPICH2, and MVAPICH.

While the PLM prsh presents a certain factor of improvement for applications compared with others runtimes, the most significant feature is the slope of it's linear behavior which is the smallest of all launchers we compare with (including SLURM based), up to an order of magnitude smaller than all other rsh-based launchers.

We contend that even higher factor of improvement can be achieved for parallel applications with a tighter interaction between the launching system and the parallel runtime.

References

[1] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[2] Mathematics and Argonne National Laboratory Computer Science Division. MPICH-2, implementation of MPI 2 standard. http://www-unix.mcs.anl.gov/mpi/mpich2/, 2006.

[3] 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, 12th European PVM/MPI Users' Group Meeting*, Sorrento, It., Sep. 2005.

[4] Mathematics and Argonne National Laboratory Computer Science Division. Hydra process management framework. http://wiki.mcs.anl.gov/mpich2/index.php/Hydra Process Management Framework, 2009.

[5] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: a scalable parallel process-management interface for extreme-scale systems. In *Proceedings of the 17th European MPI users' group meeting*, pages 31–41, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In *Euro PVM/MPI*, pages 168–175. Springer-Verlag, 2000.

[7] M. Jette, M. Jette, and M. Grondona. SLURM: Simple linux utility for resource management. In *LNCS: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[8] C. Archer, M. Gupta, X. Martorell, J. E. Moreira, S. Rus, and B. Toonen. MPI on BlueGene/L: Designing an efficient general purpose messaging solution for a large cellular system. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, pages 352–361. Springer, 2003.

[9] B. Claudel, G. Huard, and O. Richard. TakTuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM.

[10] D.C. Arnold, G.D. Pack, and B.P. Miller. Tree-based overlay networks for scalable applications. *Parallel and Distributed Processing Symposium, International*, 0:236, 2006.

[11] J.K. Sridhar, M.J. Koop, J.L. Perkins, and D.K. Panda. ScELA: Scalable and extensible launching architecture for clusters. In *HiPC*, volume 5374 of *Lecture Notes in Computer Science*, pages 323–335, 2008.

[12] F. Cappello *et al.* Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.