

ScELA: Scalable and Extensible Launching Architecture for Clusters^{*}

Jaidev K. Sridhar, Matthew J. Koop, Jonathan L. Perkins,
and Dhableswar K. Panda

Network-Based Computing Laboratory
The Ohio State University
2015 Neil Ave., Columbus, OH 43210 USA
{sridharj,koop,perkinjo,panda}@cse.ohio-state.edu

Abstract. As cluster sizes head into tens of thousands, current job launch mechanisms do not scale as they are limited by resource constraints as well as performance bottlenecks. The job launch process includes two phases – spawning of processes on processors and information exchange between processes for job initialization. Implementations of various programming models follow distinct protocols for the information exchange phase. We present the design of a scalable, extensible and high-performance job launch architecture for very large scale parallel computing. We present implementations of this architecture which achieve a speedup of more than 700% in launching a simple *Hello World* MPI application on 10, 240 processor cores and also scale to more than 3 times the number of processor cores compared to prior solutions.

1 Introduction

Clusters continue to increase rapidly in size, fueled by the ever-increasing computing demands of applications. As an example of this trend we examine the Top500 list [1], a biannual list of the top 500 supercomputers in the World. In 2000 the largest cluster, ASCI White, had 8, 192 cores. By comparison, last year the top-ranked BlueGene/L had over 200,000 cores. Even as clusters increase in node counts, an emerging trend is increase in number of processing cores per node. For instance, the Sandia Thunderbird [2] cluster introduced in 2006 has 4K nodes – each with dual CPUs for a total of 8K processors, while the TACC Ranger cluster introduced in 2008 has 4K nodes – each with four quad-core CPUs for a total of 64K processors.

Programming models and their scalability have been a large focus as cluster size continues to increase. In addition to these concerns, other more basic concerns with regard to the system software must also be addressed. In particular,

^{*} This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342 and #CCF-0702675; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

the mechanism by which jobs are launched on these large-scale clusters must also be examined. All programming models require some executable to be started on each node in the cluster. Others, such as the Message Passing Interface (MPI) [3], may have multiple processes per node – one per core. Our work shows that current designs for launching of MPI jobs can take more than 3 minutes for 10,000 processes and have trouble scaling beyond that level.

In this paper we present a scalable and extensible launching architecture (ScELA) for clusters to address this need. We note that the initialization phase of most parallel programming models involve some form of communication to discover other processes in a parallel job and exchange initialization information. Our multi-core aware architecture provides two main components: a scalable spawning agent and a set of communication primitives. The spawning agent starts executables on target processors and the communication primitives are used within the executables to communicate necessary initialization information. As redundant information is exchanged on multi-core systems, we design a hierarchical cache to reduce the amount of communication.

To demonstrate the scalability and extensibility of the framework we redesign the launch mechanisms for both MVAPICH [4], a popular MPI library, and the Process Management Interface (PMI), a generic interface used by MPI libraries such as MPICH2 [5] and MVAPICH2 [6]. We show that ScELA is able to improve launch times at large cluster sizes by over 700%. Further, we demonstrate that our proposed framework is also able to scale to at least 32,000 cores, more than three times the scalability of the previous design.

Although our case studies use MPI, ScELA is agnostic as to the programming model or program being launched. We expect other models such as Unified Parallel C (UPC) [7] to be able to use this architecture as well. In addition, ScELA can be used to run commands remotely on other nodes in parallel, such as simple commands like ‘hostname’ or maintenance tasks. It is a generic launching framework for large-scale systems.

The remaining parts of this paper are organized as follows: In Section 2 we describe the goals and design issues of our launch framework. We use our framework to redesign two job launch protocols and present these case studies in Section 3. Section 4 contains a performance evaluation of the ScELA design. Related work is discussed in Section 5. We conclude and give future directions in Section 6.

2 Proposed Design

In this section we describe the ScELA framework. The main goals of the design are scalability towards a large number of processing cores, ease of extensibility and elimination of bottlenecks such as network congestion and resource limits. For ease of extensibility the various components of ScELA are divided into distinct layers. Figure 1 shows an overview of the framework. The following sections describe each of these layers in detail.

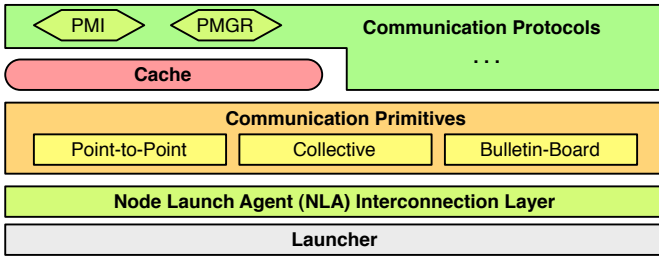


Fig. 1. ScELA Framework

2.1 Launcher

The launcher is the central manager of the framework. The job-launch process starts with the launcher and it is the only layer that has user interaction. The main task of the launcher is to identify target nodes, set up the runtime environment and launch processes on the target nodes.

Process Launching. Modern clusters deploy multi-core compute nodes that enable multiple processes to be launched on a node. On such systems, a launcher would have to duplicate effort to launch multiple processes on a node. ScELA has a Node Launch Agent (NLA) which is used to launch all processes on a particular node. The launcher establishes a connection to target nodes and sets up a NLA on each of them. This mechanism allows the Launcher to make progress on launching processes on other nodes while local NLAs handle node level process launching. The NLAs are active for the duration of the launched process after which they terminate, hence the framework is daemon-less.

Consider a cluster with n compute nodes and c processor cores per node. Table 1 shows a comparison of times taken to spawn $n \times c$ processes on such a cluster. T_{conn} is the time taken to establish a connection to a node, T_{launch} is the time taken to spawn a single process and T_{nla} is the time taken to setup a NLA. We see that as the number of cores per node increases, the time taken to start the job decreases with the NLA approach. Since the dominant factor on most clusters is T_{conn} (around 5 ms on our testbed), the use of NLAs on multi-core systems keeps the spawn time practically constant for a fixed number of nodes irrespective of the number of cores per node.

Process Health. An important task of job launchers is to handle process termination. When a process fails, a job launcher must clean up other processes. Failure to do so would impact performance of future processes. Having a node

Table 1. Time Taken to Spawn Processes With and Without NLAs

With NLAs	Without NLAs
$n \times (T_{conn} + T_{nla}) + c \times T_{launch}$	$(n \times c) \times (T_{conn} + T_{launch})$

level agent allows ScELA to handle monitoring of process health in parallel. The NLAs monitor the health of local processes. When a failure is observed the NLA sends a `PROCESS_FAIL` notification message to the central launcher. The Launcher then sends a `PROCESS_TERMINATE` message to all other NLAs which terminate all processes. User signals such as `SIGKILL` are handled similarly.

2.2 NLA Interconnection Layer

Many programming models require some form of information exchange and synchronization between processes before they complete initialization. For instance, MPI processes may need to discover other processes on the same node to utilize efficient shared memory communication channels or processes may need a barrier synchronization before they can enter a subsequent phase of initialization. Having a connection between every process does not scale for a large number of processes as the number of connections required is $O(n^2)$. Other approaches have all processes connect to a central controller which coordinates information exchange and synchronization. However, when a large number of processes initiate connections to a central controller, it becomes a bottleneck. The resultant network congestion causes TCP SYN packets being dropped. Since SYN retransmission timeouts increase with every attempt on most TCP implementations [8], this introduces a large delay in the overall launch process. In addition, most operating systems limit the number of connections that can be kept open which makes a central controller unfeasible.

We have designed a communication layer over the NLAs to facilitate communication and synchronization between processes. Each NLA aggregates initialization information from all processes on the node. This aggregation limits the total number of network connections needed per entity (process, NLA or the Launcher) on the system. NLAs from different nodes form a hierarchical k -ary tree [9] for communication of information between processes across nodes. The hierarchical tree improves overall parallelism in communication. A k -ary tree allows ScELA to launch processes over an arbitrary number of nodes while also keeping the number of steps required for synchronization and other collective operations such as broadcast or gather at a minimum at $\log_k(n)$ where n is the number of nodes. An example of a 3-ary tree of depth 3 is given in Figure 2.

The degree k of the k -ary tree determines the scalability and the performance of ScELA. An NLA in the hierarchical tree should be able to handle connection setup and communication from all processes on a node as well as the parent

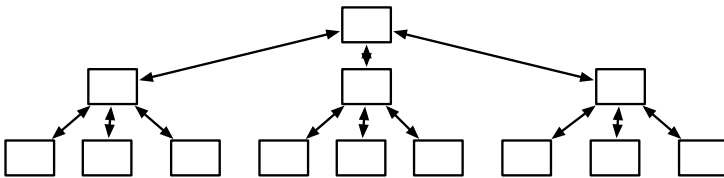


Fig. 2. Example 3-ary NLA Interconnection (with depth 3)

and children in the NLA tree. If the degree of the tree is too high, each NLA would have to process too many connections which would create further bottle-necks. If the degree is too low, the depth of the tree would result in too many communication hops.

We determine the degree k dynamically. If n is the number of nodes, we determine an ideal degree k such that the number of levels in the tree, $\log_k(n)$, is as follows: $\log_k(n) \leq MAX_DEPTH$. If c is the number of cores per node and $c + k + 1 \leq MAX_CONN$, then we select k as the degree. If not, we select $k = MAX_CONN - 1 - c$. The parameter MAX_CONN is the number of connections that an entity can process in parallel without performance degradation. From our experiments (Section 4.2) we have determined that a process can handle up to 128 connections with acceptable performance degradation on current generation systems.

2.3 Communication Primitives

The characteristics of the information exchange between processes depends on the programming model as well as specific implementations. The communication pattern could be point-to-point, collective communication such as broadcast, reduce, or a protocol such as a global bulletin board. We have designed the following communication primitives over the NLA Interconnection Layer for use by the processes for efficient communication.

Point-to-point Communication Primitives: Some initialization protocols have processes communicating directly with each other. For such protocols, we have designed two sided point-to-point communication primitives – `NLA_Send` and `NLA_Recv`.

The data from a sender is forwarded to the receiver over the NLA tree. Each process is assigned a unique identifier. During the setup of the NLA Interconnection Layer, every NLA discovers the location of each process. A process is either on the same node as the NLA, or it can be found in specific lower branch of the NLA tree or higher up the NLA tree.

Collective Communication Primitives: In most programming models, all processes go through identical initialization phases with identical communication patterns. These communication protocols resemble MPI-style collective communication. To support such protocols, we have designed MPI-style collective communication primitives `NLA_Gather`, `NLA_Broadcast`, `NLA_AllGather`, `NLA_Scatter` and `NLA_AllToAll` over the NLA tree.

Bulletin Board Primitives: Some communication protocols have processes publish information about themselves on a global bulletin board and processes needing that information read it off the bulletin board. To support such protocols over ScELA we have designed two primitives – `NLA_Put` and `NLA_Get`.

`NLA_Put` publishes data to all NLAs up the tree up to the root. When a process needs to read data, it invokes the `NLA_Get` primitive. When data is not available at a NLA, it forwards the request to the parent NLA. When data is found at a higher level NLA, it is sent down the tree to the requesting NLA.

Synchronization Primitive: In some programming models, the information exchange phase consists of smaller sub-phases with synchronization of the processes at the end of each sub-phase. For instance, in MVAPICH, processes cannot initiate InfiniBand [10] channels until all processes have pre-posted receive buffers on the NIC.

We have designed a synchronization primitive – `NLA_Barrier` which provides barrier synchronization over the NLA tree. Processes are released from an invocation of `NLA_Barrier` primitive only when all other processes have invoked the primitive. The `NLA_Barrier` primitive can be used in conjunction with `NLA_Send` and `NLA_Recv` to design other forms of communication required by a specific communication protocol.

2.4 Hierarchical Cache

On multi-core nodes, with communication patterns such as the use of a bulletin board, many processes on a node may request the same information during initialization. To take advantage of such patterns, we have designed a NLA level cache for frequently accessed data. When a process posts information through `NLA_Put`, the data is sent up to the root of the NLA tree while also being cached at intermediate levels. When a process requests information through `NLA_Get`, the request is forwarded up the NLA tree until it is found at a NLA. The response gets cached at all intermediate levels of the tree. Hence subsequent processes requesting the same piece of information are served from a nearer cache. This reduces network traffic and improves the overall responsiveness of the information exchange.

Such a cache is advantageous even on non multi-core nodes or communication patterns without repeated access to common information because the caching mechanism propagates information down the NLA tree. Subsequent requests from other sub-branches of the tree may be served from an intermediate NLA and would not have to go up to the root. In Section 3.1 we describe an extension to the `PMI_Put` primitive that enables better utilization of the Hierarchical Cache.

2.5 Communication Protocols

As described in Section 2.3, the processes being launched may have their own protocol for communicating initialization information. We have designed the ScELA framework to be extensible so that various communication protocols can be developed over it by using the basic communication primitives provided. In Section 3 we describe two implementations of such protocols over the ScELA architecture.

3 Case Studies

In this section we describe implementations of two startup protocols over ScELA. We first describe an implementation of the Process Management Interface (PMI),

an information exchange protocol used by popular MPI libraries such as MPICH2 and MVAPICH2 over the ScELA framework. In addition, we describe an implementation of another startup protocol – PMGR used by MPI libraries such as MVICH [11] and MVAPICH.

3.1 Designing the PMI Bulletin Board with ScELA

When MPI processes start up, they invoke `MPI_Init` to set up the parallel environment. This phase involves discovery of other processes in the parallel job and exchange of information. The PMI protocol defines a *bulletin board* mechanism for information exchange. Processes do a `PMI_Put` operation on a (`key`, `value`) pair to publish information followed by a `PMI_Commit` to make the published information visible to all other processes. When other processes need to read information, they perform a `PMI_Get` operation by specifying a `key`. The PMI protocol also defines a barrier synchronization primitive `PMI_Barrier`.

To implement the PMI bulletin board over the ScELA framework, we utilized the `NLA_Put` and `NLA_Get` primitives. A `PMI_Put` by a process invokes a corresponding `NLA_Put` to propagate information over the NLA tree. When a process does a `PMI_Get`, a corresponding `NLA_Get` is invoked to search for information in the Hierarchical Cache. Since the `PMI_Puts` are propagated immediately, we ignore `PMI_Commit` operations.

We have observed that with the PMI protocol, information reuse is high for some information. In such cases it is beneficial to populate the node level caches even before the first `PMI_Get` request. We have designed an extension to the `NLA_Put` primitive that propagates information to all NLAs in the tree so that all `NLA_Gets` can be served from a local cache. To reduce the number of `NLA_Puts` active in the tree, we aggregate puts from all processes on a node before propagating this information over the tree. When processes invoke `PMI_Barrier`, we invoke the `NLA_Barrier` primitive to synchronize processes. We evaluate our design against the current startup mechanism in MVAPICH2 in Section 4.1.

3.2 Designing PMGR (Collective Startup) with ScELA

The PMGR protocol defines MPI style collectives for communication of initialization data during `MPI_Init`. These operations also act as implicit synchronization between processes. The PMGR interface defines a set of collective operations – `PMGR_Gather`, `PMGR_Broadcast`, `PMGR_AlltoAll`, `PMGR_AllGather` and `PMGR_Scatter` and an explicit synchronization operation `PMGR_Barrier`.

In our implementation when a process invokes a PMGR primitive, it is directly translated to an invocation of the corresponding collective communication primitive designed over the NLA tree. We evaluate our design against the current startup mechanism in MVAPICH in Section 4.2.

4 Performance Evaluation

In this section we evaluate the two case studies described in Section 3. We evaluate our designs against the previous launching mechanisms in MVAPICH2 and MVAPICH respectively. Our testbed is a 64 node InfiniBand Linux cluster. Each node has dual 2.33 GHz Intel Xeon “Clovertown” quad-core processor for a total of 8 cores per node. The nodes have a Gigabit Ethernet adapter for management traffic such as job launching. We represent cluster size as $n \times c$, where n is the number of nodes and c is the number of cores per node used.

We have written an MPI microbenchmark to measure the time taken to launch MPI processes and time spent in `MPI_Init`, which represents the information exchange phase. For the purpose of these microbenchmark level tests, we disable all optional features that impact job initialization.

4.1 PMI over ScELA

In this section, we compare the performance of our implementation of PMI over ScELA (ScELA-PMI) against the default launch framework in MVAPICH2 (MVAPICH2-PMI). The default startup mechanism of MVAPICH2 utilizes a ring of daemons – MPD [12] on the target nodes. The launcher – `mpiexec` identifies target nodes and instructs the MPD ring to launch processes on them. PMI information exchange is done over the MPD ring. Figure 3 shows the time taken to establish the initial ring. We observe a linear increase which is not scalable over larger number of nodes. We have also observed that the MPD ring cannot be setup on larger sizes such as thousands of nodes. While a MPD ring can be reused for launching subsequent MPI jobs, most job schedulers elect to establish a separate ring as both target nodes and job sizes may be different. Figure 4 shows a comparison of the launch times for various system sizes. On ScELA-PMI, the spawn phase represents the time taken for the Launcher to setup

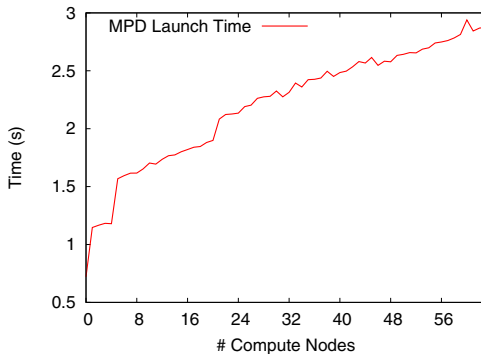


Fig. 3. Time to Setup MPD Ring with MVAPICH2

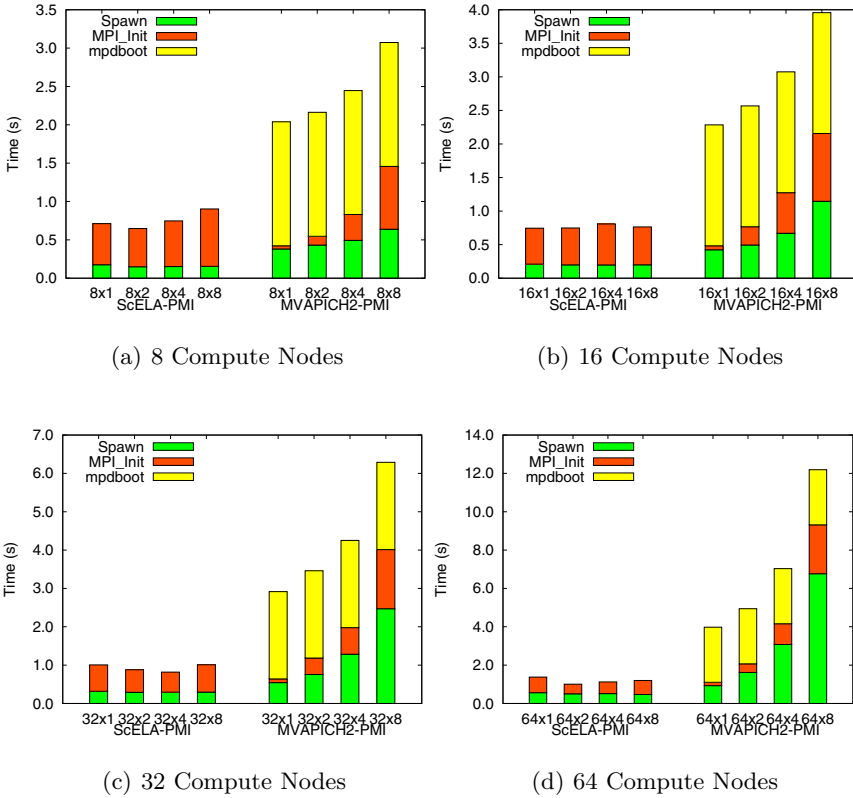


Fig. 4. Comparison of Startup Time on MVAPICH2

NLAs on the target nodes and for the NLAs to launch the MPI processes. The MPI_Init phase represents the time taken to establish the NLA Interconnection Layer and for PMI information exchange. On MVAPICH2-PMI the mpdboot phase represents the time taken to establish the ring of MPD daemons. The spawn phase represents time taken to launch MPI processes over the MPD ring and the MPI_Init phase represents the time taken for information exchange.

We observe that as we increase the number of processes per node, ScELA-PMI demonstrates better scalability. For a fixed node count, the duration of the spawn phase in ScELA-PMI is constant due to parallelism achieved through NLAs. In Figure 4(d) we see the spawn time for MVAPICH2-PMI increase from around 1s to 6.7s when the number of cores used per node is increased from 1 to 8 but ScELA-PMI is able to keep spawn time constant at around 0.5s. At larger job sizes, for instance 512 processes on 64 nodes (64×8 in Figure 4(d)), we see an improvement in the MPI_Init phase from around 2.5s to 0.7s due to the better response times of communication over the NLA Interconnection Layer and due to reduced network communication due to NLA cache hits.

4.2 PMGR over ScELA

In this section we compare our design of PMGR over ScELA (ScELA-PMGR) against the default startup mechanism in MVAPICH (MVAPICH-PMGR). The default startup mechanism in MVAPICH has a central launcher that establishes a connection to target nodes and launches each process individually. On multi-core systems, this launcher needs several connections to each node. Also, each MPI process establishes a connection to the central controller which facilitates the PMGR information exchange. As the number of processes increase, this causes a flood of incoming connections at the central controller, which leads to delays due to serialization of handling these requests and network congestion. The number of MPI processes that can be handled simultaneously is also limited by resource constraints such as open file descriptor limits, which is typically 1024.

Figure 5 shows a comparison of the launch times. With ScELA-PMGR, the spawn phase represents the time taken to setup NLAs on the target nodes and for the NLAs to launch MPI processes on the node. The MPI_Init phase represents the time taken to setup the NLA Interconnection Layer and the PMGR information exchange between MPI processes. With MVAPICH-PMGR, the spawn

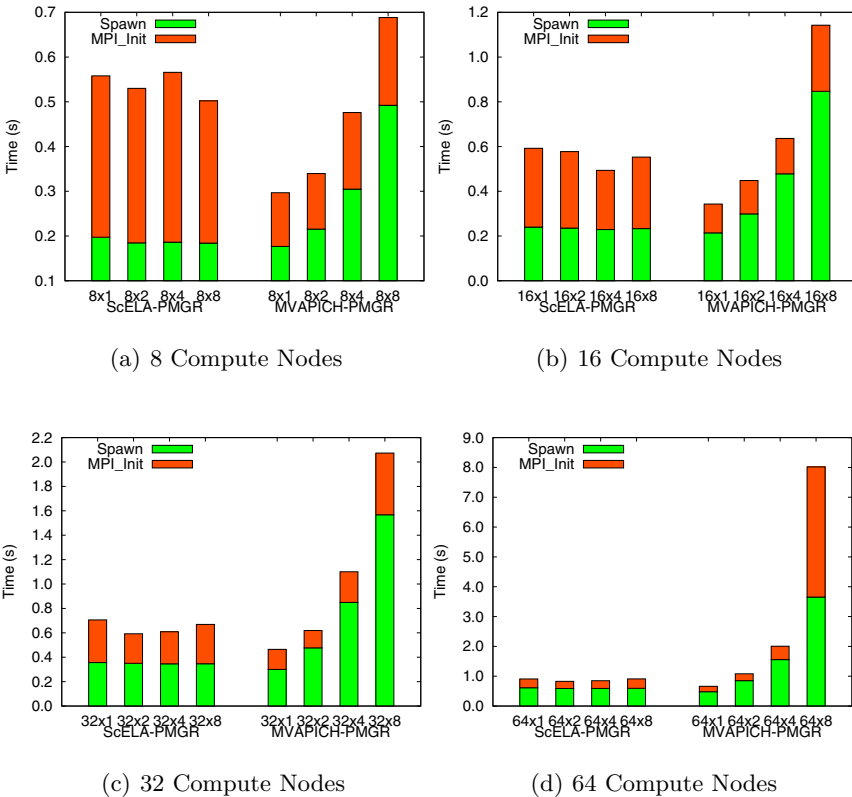


Fig. 5. Comparison of Startup Time on MVAPICH

phase represents the time taken for the central controller to launch each MPI process on target nodes. In the `MPI_Init` phase, the MPI processes establish connections to the central controller and exchange information over the PMGR protocol. We see that for a fixed node count, ScELA-PMGR takes constant time for the spawn phase as it benefits from having NLAs while the spawn phase with MVAPICH-PMGR grows with increase in number of processes per node. For instance in 5(d), we see that ScELA-PMGR is able to keep spawn time constant at 0.6s, but on MVAPICH-PMGR the spawn phase increases from 0.5s to 3.6 as we increase the number of cores used per node from 1 to 8. When the overall job size is small, the central controller in the MVAPICH startup mechanism is not inundated by a large number of connections. We see that the central controller is able to handle connections from up to 128 processes with little performance degradation in the `MPI_Init` phase. Hence the MVAPICH startup performs better at a small scale, but as the job sizes increase we observe larger delays in the `MPI_Init` phase. From figure 5(d) we see that for 512 processes (64×8), the `MPI_Init` phase takes 4.3s on MVAPICH-PMGR, but on ScELA-PMGR it takes around 0.3s. For 512 processes we see an improvement of 800% in the overall launch time.

Figure 6 shows a comparison of ScELA-PMGR and MVAPICH-PMGR on a large scale cluster – the TACC Ranger [13]. The TACC Ranger is an Infini-Band cluster with 3,936 nodes with four 2.0 GHz Quad-Core AMD “Barcelona” Opteron processors making a total of 16 processing cores per node. The Figure shows the runtime of a simple *hello world* MPI program that initializes the MPI environment and terminates immediately. In terms of number of processing cores, ScELA-PMGR scales up to at least three times more than MVAPICH-PMGR (based on MVAPICH version 0.9.9). On 10,240 cores, we observe that MVAPICH-PMGR takes around 185s while ScELA-PMGR takes around 25s which represents a speedup of more than 700%. We also see that MVAPICH-PMGR is unable to scale beyond 10,240 cores, while ScELA-PMGR is able to scale to at least 3 times that number.

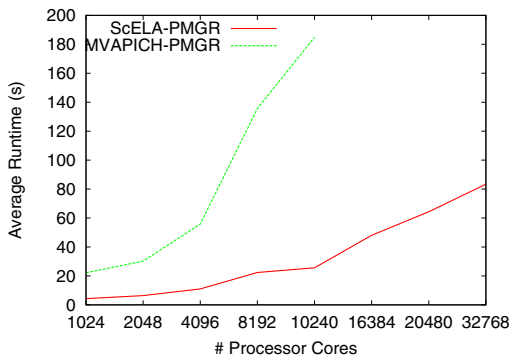


Fig. 6. Runtime of Hello World Program on a Large Scale Cluster (Courtesy TACC)

5 Related Work

The scalability and performance of job startup mechanisms in clusters have been studied in depth before. Yu, et. al. [14] have previously explored reducing the volume of data exchanged during initialization of MPI programs in InfiniBand clusters.

In our work, we have assumed availability of executable files on target nodes through network based storage as this is a common model on modern clusters. Brightwell, et. al. [15] have proposed a job-startup mechanism where network storage is not available.

SLURM [16] is a resource manager for Linux clusters that implements various interfaces such as PMI and PMGR for starting and monitoring parallel jobs. Unlike ScELA, SLURM has persistent daemons on all nodes through which it starts and monitors processes.

6 Conclusion and Future Work

Clusters continue to scale in core counts. Node counts are increasing significantly, but much of the growth in core counts is coming from multi-core clusters. In this paper we have demonstrated a scalable launching architecture that improves the launch performance on multi-core clusters by more than an order of magnitude than previous solutions. Although our case studies have been with two MPI libraries, we have presented an architecture extensible to any cluster launching requirements. For launching parallel jobs, we provide scalable and efficient communication primitives for job initialization.

With an implementation of our architecture, we have achieved a speedup of 700% in MPI job launch time on a very large scale cluster at 10,240 processing cores by taking advantage of multi-core nodes. We have demonstrated scalability up to at least 32,768 cores.

These solutions are being used by several large scale clusters running MVA-PICH such as the TACC Ranger – currently the largest InfiniBand based computing system for open research.

In our solution, the Launcher launches Node Launch Agents serially. As node counts increase, this could be a potential bottleneck. However, this can be easily extended so that NLAs are launched hierarchically by other previously launched NLAs. We plan to explore this mechanism in the future.

With the recent demonstration of a 80 core processor by Intel, the number of cores per node on large scale clusters is projected to increase further. We can use more efficient communication channels such as UDP or shared memory for communication between processes and the NLA on a node so that the degree of the NLA tree can be decoupled from the number of cores on a node.

Software Distribution: Our implementation of PMGR over ScELA (ScELA-PMGR) is integrated with the 1.0 release of MVAPICH. The PMI implementation over ScELA (ScELA-PMI) will be available with the upcoming 1.2 release of MVAPICH2 [17].

Acknowledgment

We would like to thank Dr. Karl W. Schulz from TACC for helping us evaluate our designs on the TACC Ranger system. We would also like to thank Mr. Adam Moody from LLNL for providing the PMGR framework in MVAPICH.

References

1. TOP 500 Project: Top 500 Supercomputer Sites, <http://www.top500.org>
2. Sandia National Laboratories: Thunderbird Linux Cluster, <http://www.cs.sandia.gov/platforms/Thunderbird.html>
3. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1994)
4. Network-Based Computing Laboratory: MVAPICH: MPI-1 over InfiniBand, <http://mvapich.cse.ohio-state.edu/overview/mvapich>
5. Argonne National Laboratory: MPICH2 : High-performance and Widely Portable MPI, <http://www.mcs.anl.gov/research/projects/mpich2/>
6. Huang, W., Santhanaraman, G., Jin, H.-W., Gao, Q., Panda, D.K.: Design of high performance mvapich2: Mpi2 over infiniband. In: Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2006) (2006)
7. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to upc and language specification. IDA Center for Computing Sciences (1999)
8. Shukla, A., Brecht, T.: Tcp connection management mechanisms for improving internet server performance. In: 1st IEEE Workshop on Hot Topics in Web Systems and Technologies, 2006. HOTWEB 2006, November 13-14, 2006, pp. 1–12 (2006)
9. Moody, A., Fernandez, J., Petrini, F., Panda, D.: Scalable NIC-based Reduction on Large-scale Clusters. In: Supercomputing, 2003 ACM/IEEE Conference (2003)
10. InfiniBand Trade Association: InfiniBand Architecture Specification, <http://www.infinibandta.com>
11. Lawrence Berkeley National Laboratory: MVICH: MPI for Virtual Interface Architecture (2001), <http://www.nersc.gov/research/FTG/mvich/index.html>
12. Butler, R., Gropp, W., Lusk, E.: Components and interfaces of a process management system for parallel programs. In: Parallel Computing (2001)
13. Texas Advanced Computing Center: HPC Systems, <http://www.tacc.utexas.edu/resources/hpcsystems/>
14. Yu, W., Wu, J., Panda, D.K.: Scalable startup of parallel programs over infiniband. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296. Springer, Heidelberg (2004)
15. Brightwell, R., Fisk, L.: Scalable parallel application launch on cplant. In: Supercomputing, ACM/IEEE 2001 Conference, November 10-16 (2001)
16. Lawrence Livermore National Laboratory and Hewlett Packard and Bull and Linux NetworX: Simple Linux Utility for Resource Management, <https://computing.llnl.gov/linux/slurm/>
17. Network-based Computing Laboratory: (MVAPICH: MPI over InfiniBband and iWARP), <http://mvapich.cse.ohio-state.edu>