# A Multi-level Scalable Startup for Parallel Applications

Abhishek Gupta
gupta59@illinois.edu

Gengbin Zheng
gzheng@illinois.edu

Laxmikant V. Kalé
kale@illinois.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

## ABSTRACT

High performance parallel machines with hundreds of thousands of processors and petascale performance are already in use, and even larger Exaflops scale computing systems which may have hundreds of millions of cores are planned. To run parallel applications on machines of such massive scale, one of the biggest challenges is the parallel startup process. This task involves two components: (1) parallel launching of appropriate processes on the given set of processors and (2) setting up communication channels to enable the processes to communicate with each other after process launching has completed. Most current startup mechanisms focus on either using daemons which waste system resources or using a startup manager which becomes a scalability bottleneck. In this paper, we investigate the design and scalability of a SMP-aware, multi-level startup scheme with batching of remote shell sessions, which provides a complete solution to startup of a parallel application and facilitates its management during execution. It monitors process health and can be used to support recovery from failures and provide scalable interaction with the application. We demonstrate the performance and scalability of this scheme by applying it to startup Charm++ applications. In particular, starting up a Charm++ program on 16,384 cores of Ranger (at TACC) with Ethernet as the underlying communication layer takes only 25 seconds and attains a speedup of over 400% compared to MPICH2 startup (using hydra as process manager) and over 800% compared to Open MPI startup on Ranger.

## 1. INTRODUCTION

High performance parallel machines with hundreds of thousands of processors and petascale performance are already in use, which provide unprecedented computing power to solve scientific and engineering problems. Even larger Exaflop/s scale computing systems which may have hundreds of millions of cores are planned. To run parallel applications on machines of such massive scale, one of the biggest challenges is the parallel startup process, i.e. how to start the application on all the computation nodes (as an example, this is what `mpirun` does to start MPI [6] applications). On a large machine, there may be a significant delay between job allocation and application execution.

Furthermore, as another important part of the startup process, all the processes on the computation nodes need to exchange information with each other to set up communication channels for inter-process communication during execution. This inter-process communication requires that each process knows about the existence of other processes and also where to send a message if it needs to communicate with a particular process. This information can be in the form of a socket address (in case of using TCP/UDP) consisting of IP address and a port. In general, each process can potentially communicate with any of the processes and hence should have information which enables it to send messages to them.

Hence, the task of parallel startup involves two components: (1) *parallel launching* of appropriate processes on the given set of processors and (2) *setting up communication channels* to enable the processes to communicate with each other after startup has completed. In case of MPI, the startup time would be from the time `mpirun` starts launching processes on computation nodes to the time `MPI_Init` finishes, at which point communication channels are setup and MPI processes are ready to communicate with each other. Note that our definition of the startup process is different with the ones that only consider parallel launching, e.g. remote execution tools such as GXP [25] and TakTuk [11], which are typically exploited for administrative purposes such as running updates and setting up configuration on all computation nodes. This paper focuses on the startup process of parallel applications and scalable techniques to speed up both components in the process.

The absence of fast startup mechanisms presents a major obstacle to the full utilization of high performance computing power by the research community. Users of supercomputers are charged in Service Units (SU) to run their experiments. One SU is equal to one core-hour of computations. Also, the typical allocation size for research groups is a few tens of thousands of SUs. Existing parallel startup mechanism such as those used by Charm++ and Open MPI [12] take 2 to 4 minutes for startup on 8K processors and perform even worse for higher core counts. This results in tremendous SU usage just to startup the application. As an example, startup time of 4 minutes for 16K processors would mean that a single experiment on 16K processors results in consumption of more than 1K SUs for application startup. This

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

results in limiting the number of experiments a researcher can perform given the fixed allocation size.

There are two types of approaches that have been adopted by researchers to address the problem of parallel startup. The first one assumes the presence of special daemons running on compute nodes to facilitate the startup process [9, 18, 22]. An example of this is a system called Multi-purpose Daemons (MPD) [9] used for MPICH [15] jobs. Here, when an application starts, the launcher contacts these daemons to start the processes on each compute node. The drawback of this approach is that these daemons keep running even when no MPI application is running and hence waste system resources. The second type of approach is to use a launcher which starts processes on compute nodes using `rsh` or `ssh` and then sets up communication channels among them. However, as we go up to high core counts, the centralized launcher becomes a bottleneck and imposes scalability limitations (demonstrated by Figure 4).

In this paper, we present a multi-level scalable startup method which is generic and can be applied to most parallel programming environments, including MPI and Charm++ [20]. The fundamental idea is to use multiple launchers which form a startup tree and reside on different processors. This makes the process of parallel application startup decentralized and hence it scales well with increasing number of processors. We also incorporate SMP-awareness in our approach to achieve faster startup. In addition, we introduce the concept of batching of remote shell sessions to make the parallel startup process fast on a consistent basis and discuss the trade-offs involved in parallel startup using a theoretical model. Moreover, our approach does not require presence of any special daemons (except `rsh` or `ssh` daemons) on parallel machines to startup the application. However, it can still be used to monitor process health and provide scalable interaction with parallel application after startup is complete.

We demonstrate the performance and scalability of multi-level startup method by applying it to Charm++ run-time system. Charm++ is a widely used programming model for large scale scientific and engineering applications including NAnoscale Molecular Dynamics (NAMD) [7], which is a highly scalable molecular dynamics code used ubiquitously on the TeraGrid and other HPC systems. Starting up a Charm++ program on 16,384 cores of Ranger [3] (at TACC) with Ethernet as the underlying communication layer now takes only 25 seconds. This results in the SU consumption getting reduced by an order of magnitude compared with the centralized startup. Moreover, our scheme outperforms Open MPI startup by a factor of over 8 and MPICH2 startup (using Hydra) by a factor of 4 for 16K cores on Ranger.

The remainder of the paper is organized as follows: Section 2 describes the process of startup of parallel application. Section 3 discusses the multi-level approach to parallel startup. Section 4 presents performance evaluation of our multi-level startup scheme. Related work is discussed in Section 5. Finally, conclusions are left for the final section.

## 2. BASIC METHOD

The techniques presented here are generic and can be applied to most parallel programming environments such as MPI, Charm++, etc. However, in this paper, we will discuss the schemes with focus on their implementation for Charm++. We will refer to the launcher as *charmrun* and the processes which constitute the parallel application as *clients*.

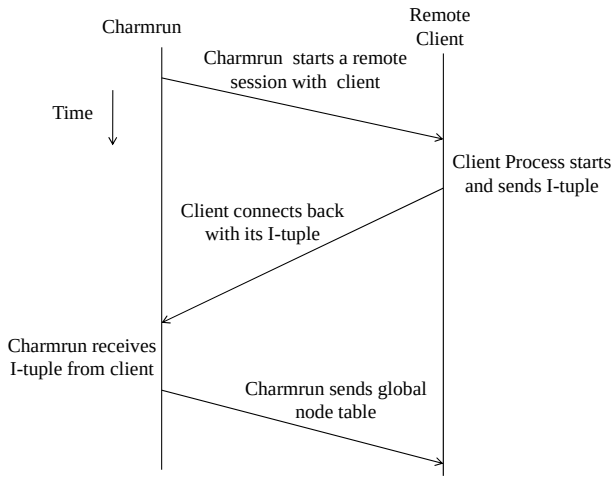



**Figure 1: Basic process of parallel startup**

Figure 1 discusses the basic startup scheme. For simplicity, we assume processes on remote processors communicate via UDP sockets. Charmrun needs to know the set of processors where the parallel application will be run. One way of providing this information is using a machine file, which we call the *nodelist* file. Charmrun starts a remote shell session with each processor where the application will be run. Standard remote execution shell facilities (such as `rsh` or `ssh`) can be used for this purpose. We chose to use `ssh` since it provides strong authentication and is more secure compared to `rsh`. After starting a remote shell session, charmrun sets up some environment variables, creates a process on the remote processor using the `fork()` system call and loads the application executable using the `exec()` system call. This task is performed for each processor in the nodelist file.

During the execution of a parallel application, the remote clients need to be able to communicate with each other. This inter-process communication requires establishment of communication channels among clients. The second component of startup performs information collection and dissemination to enable the clients to communicate with each other after startup has completed. Each client sends some information to charmrun which is used to set up communication channels between the clients. We refer to this information as an *I-tuple*. In the case of TCP or UDP as the underlying communication layer, an I-tuple consists of a socket address comprising IP address and dataport. The dataport is the port where a client will listen for any incoming message from other clients during execution of the application. Charmrun receives I-tuples from all clients and collects them to form a table of I-tuples which we call the *node-table*. The node-table is sent to every process. After receiving the node-table, clients can communicate with each other without any need of charmrun and startup is complete. As another example, this component of startup would involve the process of establishing queue pairs on an Infiniband based communication network [17].

Charmrun is needed even after startup has completed

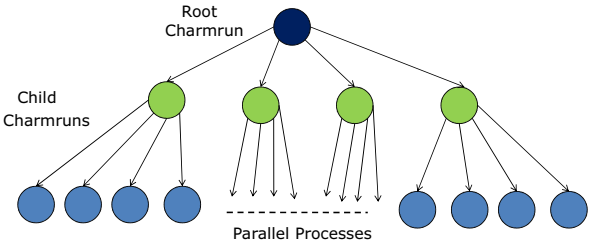



**Figure 2: Multi-level startup scheme**

since it acts as an interface between parallel application and the external world during execution. All input output and some additional features (discussed in section 3.5) such as parallel debugging can utilize charmrun.

# 3. MULTI-LEVEL STARTUP

In this section, we discuss the multi-level startup technique. There is an optimization which can lead to significant improvement in the performance of centralized startup. We discuss that before describing the multi-level approach.

## 3.1 SMP-Aware Startup

The scheme discussed in section 2 requires charmrun to perform a remote shell login to each processor. Most supercomputers and even desktop systems today have multi-core chips where each node has many processor cores. 8-core, 16-core and 32-core nodes are not uncommon. Consider a node with 16 cores; charmrun would create a `ssh` session with each of the 16 cores. An optimization to this is to create only one `ssh` session per node and spawn all clients from the same `ssh` session. We call this *SMP-aware startup*. With the trend towards clusters with increasing number of cores per node, such as Intel's Single-chip Cloud Computer [16], this optimization is extremely useful. In addition, this is useful when multiple processes need to be launched on a single processor, such as in parallel application testing and debugging. The second phase of startup remains the same as the basic scheme. Each client sends an I-tuple to charmrun, which collects them and sends the node-table to every client.

## 3.2 Multi-level Startup

Even with the optimization discussed in previous section, the startup is inherently serial. Charmrun starts `ssh` sessions sequentially and waits for all the clients to connect back. Charmrun becomes a bottleneck in a few ways:

1. Charmrun has to start an `ssh` session with each node.
2. Charmrun has to receive a message containing an I-tuple from each of the clients.

In todays supercomputers, with hundred of thousands of nodes, centralized startup becomes a bottleneck and the startup performance degrades significantly with increase in number of cores (See section 4.1). We can conclude that we need to explore a distributed startup scheme to prevent the central charmrun process from becoming a bottleneck.

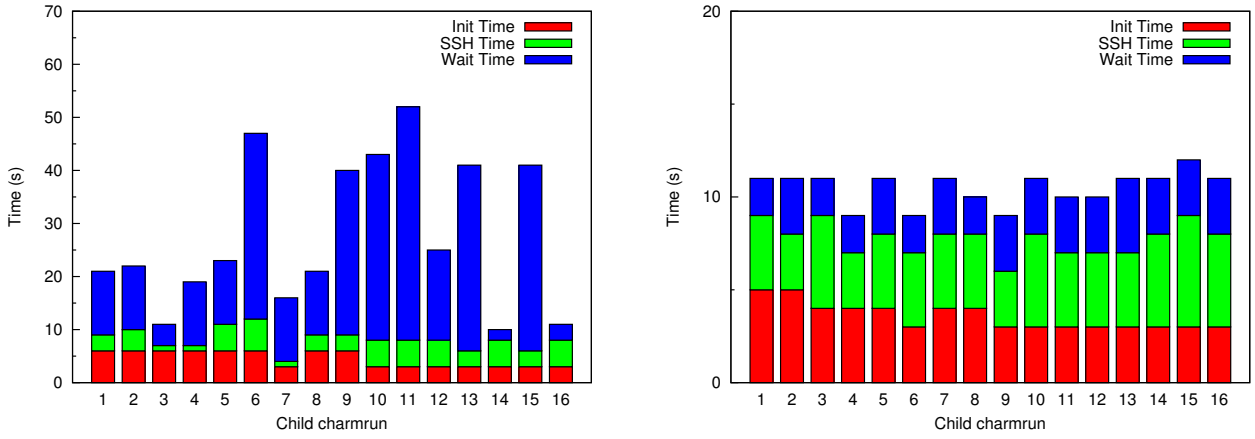
We propose a multi-level startup to overcome the problems discussed so far. Fig 2 illustrates 2-level startup scheme. Here we have a master charmrun process, which we call the *root charmrun*, and second level charmrun processes, which we will refer to as *child charmruns*. Child charmruns reside on different processors. Each child charmrun is assigned a subset of unique nodes for which it acts as a manager. The root charmrun acts as a top-level manager that coordinates the startup process between child charmruns. It decides the branching factor (number of child charmrun) which we call $k$. For simplicity, we keep the branching factor as the square root of number of unique nodes where the application will be run. (Section 3.4 provides an analysis of the effect of branching factor and number of levels on performance.) Each child charmrun gets approximately $k$ nodes assigned to it and acts in a similar manner to the charmrun of centralized startup method. It starts processes on its node set and waits for clients to connect back. After receiving I-tuple, it forwards that to root charmrun. Root charmrun receives all the I-tuples and disseminates node-table to child charmruns, which in turn, forward that to their respective set of clients. We note that multi-level startup uses SMP-aware startup at leaf charmrun level.

Child charmruns must continue to exist after the startup is complete. A client process is aware of only its manager and is oblivious to the presence of other child charmruns and the root charmrun. All input-output and any communication for additional supported features must go through it. The decentralized existence of charmruns makes parallel input output and user interaction with parallel application more scalable.

The multi-level startup technique distributes the task of performing remote shell sessions and receiving I-tuples to a set of processors and is intuitively more scalable. However, while evaluating startup performance we discovered that there was a huge variation in the startup time between different runs. As an example, for 4096 cores on TACC's Ranger cluster with 16 cores per node, startup time using multi-level startup varied from 10 to 60 seconds. To discover the cause of this unexpected behavior, an analysis of the breakdown of time taken by startup was done. Figure 3(a) shows the breakdown of the time spent by different child charmruns in startup. With 4096 processors and 16 cores per node, there are 256 unique nodes and hence the branching factor is kept 16. So, there are 16 child charmrun. *Init time* is the initialization time taken by each child charmrun. It comprises the time taken by a child charmrun to connect to root charmrun and receive its set of nodes. *SSH time* is the time spent in creating remote login sessions and launching clients on remote processors. *Wait time* is the time spent in waiting for their set of clients to connect back. It can be observed that Init time and SSH time is small and does not vary a lot across charmruns. The main component where a large fraction of time is spent is Wait time. Moreover, Wait time varies from 2 seconds to 49 seconds across charmruns resulting in inconsistent performance. If there is even a single outlier, it delays the whole startup process.

## 3.3 Multi-level Startup with Batching

It was observed that the variation in startup time is small for less number of cores and becomes worse as we increase number of cores. Although multi-level startup makes the startup process distributed, it does not reduce the total num-




(a) Without batching

(b) With batching using batch size of 8

**Figure 3: Breakup of time spent in parallel startup using multi-level scheme for 4K processors**

ber of I-tuple messages that can be present in the network at any time. This is equal to the number of clients. If there are tens of thousands of clients, this can lead to congestion in network. To overcome this problem, we introduce the concept of *batching* of remote shell sessions. In this strategy, the nodes assigned to a leaf charmrun are divided into sets of fixed size. Each child charmrun performs `ssh` to the nodes in the current set, waits for the clients to connect back and then performs `ssh` on the next set. We call the number of nodes in one `ssh` set as batch size. Batching reduces the total number of messages at any time and hence leads to better scalability. Figure 3(b) shows the breakdown of the time spent by different child charmruns in multi-level startup with batch size of 8 for 4096 processors on Ranger. Comparing with Figure 3(a), we observe that the wait time is consistent across all charmruns and is small. This leads to a faster and more scalable startup process.

However, batching introduces some serialization; only after the clients in the first set are launched and I-tuples are received from those, next set of clients can be launched. We discuss the effect of batch size on performance in section 4.2

## 3.4 Analysis

In this section, we present a theoretical analysis of the different startup schemes discussed in this paper. Consider a supercomputer with $P$ processor cores and $N$ nodes. Let $c = P/N$ be the number of cores per node. Parallel startup time for our basic scheme ($T_{basic}$) can be modeled as:

$$\begin{aligned} T_{basic} = T_{init} + P \times T_{ssh} + T_{client} \\ + T_{send} + T_{nw} + P \times T_{recv} \end{aligned} \quad (1)$$

where $T_{init}$ is the charmrun initialization time (which includes getting the list of nodes to start e.g. reading nodelist file, starting a server port where clients can send I-tuples etc), $T_{ssh}$ is the time taken by charmrun to start a `rsh` or `ssh` session with a remote node, $T_{client}$ is the time taken by the remote shell to create a new process at the remote processor and load the program executable, $T_{send}$ is the processor sending overhead at a client, $T_{nw}$ is the network latency for a message, $T_{recv}$ is the message receiving overhead incurred by charmrun.

We regard the total overhead due to $T_{init}$, $T_{client}$, $T_{send}$ and $T_{nw}$ as constant, regardless of the number of processors, and we represent them by $T_c$ to keep the analysis readable. Hence, we have

$$T_{basic} = T_c + P \times (T_{ssh} + T_{recv}) \quad (2)$$

where

$$T_c = T_{init} + T_{client} + T_{send} + T_{nw} \quad (3)$$

SMP-aware startup starts only one `ssh` process per node and hence it reduces the total ssh time. However, it still incurs the receive overhead for all clients. so, the time taken by SMP-aware startup ($T_{SMP}$) can be modeled as:

$$T_{SMP} = T_c + N \times (T_{ssh} + c \times T_{recv}) \quad (4)$$

Hence, both $T_{basic}$ and $T_{SMP}$ grow as $\theta(P)$ since charmrun has to start a `ssh` process for each node and incur a receiving overhead for each client. This becomes to a scalability bottleneck.

Now consider our multi-level startup. Let $k$ be the branching factor and $d$ be the depth of the startup tree. We assume the branching factor is kept same across the levels of the tree. In a $k$-ary, $d$-level startup tree, a charmrun at $level \neq d$ is responsible for acting as a manager for its $k$ child charmruns and a charmrun at $level = d$ acts as a manager for $k$ nodes ($k \times c$ clients). This scheme emulates the basic startup scheme at each level of the startup tree. Hence, the startup time for a $d$-level SMP-aware startup ($T_{d-level}$) can be modeled as:

$$\begin{aligned} T_{d-level} = d \times (T_c + k \times (T_{ssh} + T_{recv})) \\ + k \times (c-1) \times T_{recv} \end{aligned} \quad (5)$$

The crucial parameters here are $k$ and $d$, which related by:

$$d = \log_k(N) \quad (6)$$

Multi-level startup increases the overhead due to $T_c$ by a factor of d. However, it reduces the effect of ssh time and receive overhead by a factor of $P/(ck\log_k(P/c))$. We note that $T_{d-level}$ grows asymptotically as $\theta(k\log_k(P/c))$

instead of $\theta(P)$ for basic startup. An optimal value of the number of levels can be obtained by minimizing $T_{d-level}$ using equation 5, with k given by equation 6. Determining the exact expression for this optimal value is beyond the scope of this paper.

A simpler method of obtaining $d$ is to bound $k$ so that charmrun can handle the given number of clients without significant performance degradation. Past research [24] suggests that a process can handle 128 simultaneous connections with acceptable performance. Hence, $d = 2$ should be adequate to handle up to 16384 processes. For a 2-level startup ($d = 2$), $k = \sqrt{P/c}$.

There is one factor which we have ignored so far. As we scale to high core counts, the number of messages in the network sent by clients to connect back to the respective charmruns increase. This can make the network congested and degrade performance significantly. Batching overcomes this problem and makes startup time consistent. However, batching comes with the cost of increasing the best case (no congestion) startup time. Startup time for SMP-aware multi-level scheme with batching can be modeled as:

$$T_{\text{batched d-level}} = (d-1) \times (T_c + k \times (T_{ssh} + T_{recv})) + (k/b) \times T_c + k \times (T_{ssh} + c \times T_{recv}) \quad (7)$$

where $b$ is the batch size. Since batching is only done at the last level of the tree, it does not degrade the performance of starting charmrun tree itself. However, it affects the time taken by last level charmruns to startup the clients. The overhead $T_c$ is now incurred for every batch phase of a last-level charmrun.

## 3.5 Runtime Capabilities

The startup system presented in this paper can also be used to interact with a parallel application after startup is finished. After startup of the application is complete, each charmrun acts as a manager for its set of clients. We discuss two aspects of this:

### 3.5.1 *Process Health and Recovery from Failures*

After startup of the application is complete, each charmrun is accountable for monitoring the status of its clients. Each charmrun enters into a polling mode where it monitors process health. If a process fails, charmrun is responsible to terminate the whole application if there is no support for fault tolerance. Charmrun system has also been used to facilitate the design of fault tolerance protocols for Charm++ applications [10]. If a process fails, appropriate charmrun restarts the failed process using restart protocol. In this protocol, execution of the parallel application is suspended by the charmruns till the failed process is restarted. A new process is launched and node-table is modified to use the I-Tuple received from the restarted process. This new node-table is communicated to all the clients and execution can resume. Multi-level charmrun system makes process health monitoring and failure recovery a decentralized process and hence more scalable.

### 3.5.2 *Support for Scalable Interaction with Parallel Application*

The multi-level charmrun system can be used to provide scalable interaction with the parallel application. Such interaction can be extremely useful for providing parallel debugging, online performance analysis and simulation visualization [14]. Developers and end-users of parallel application can greatly benefit from these capabilities provided by a runtime system. We have used Charmdebug [19] with our multi-level charmrun system to debug parallel application. Charmdebug is a graphical tool that allows programmers to debug large scale parallel programs. It uses Converse Client Server (CCS) [23] which is a communication protocol that allows parallel applications to act as remote server that receives and serves requests from remote clients. Multi-level charmrun system makes this process more scalable by distributing the task of managing the client processes among multiple charmruns and hence avoiding the single charmrun from becoming a bottleneck. CCS can also be used to provide simulation visualization using tools such as liveViz [1]. During execution charmrun also acts as a means of any communication between the client process and the outside world. An example of this is console input output.

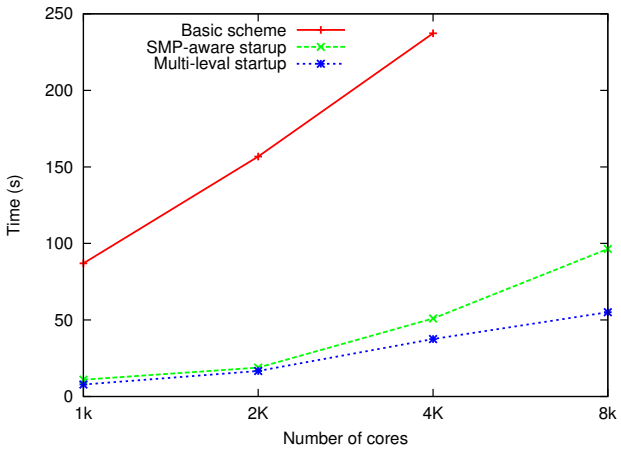



**Figure 4: Startup time: Comparison among 3 startup schemes**

# 4. PERFORMANCE RESULTS

We ran performance tests on TACC's Ranger Cluster which is one of the largest computational resource in the world (Ranked 15 in the November, 2010 top500 list [4] of supercomputers). The Ranger system has 3,936 16-way SMP compute nodes providing 15,744 AMD Opteron processors for a total of 62,976 compute cores [3]. It has a theoretical peak performance of 579 TFLOPS. All nodes are interconnected using InfiniBand technology in a full-CLOS topology and provide a 1GB/sec point-to-point bandwidth. For core counts above 4K, the executable was cached in each node's memory immediately before launching the parallel application to avoid any inconsistencies caused by disk to memory transfers. The following subsections discuss various performance results we got on this supercomputer.

## 4.1 Performance of Different Schemes

In this section, we discuss the performance of our startup schemes. We ran all our experiments using all the 16 cores per node and used Ethernet as the underlying communication network. Figure 4 compares three schemes - the basic scheme, SMP-aware startup and multi-level startup without batching. The effect of batching is discussed in section 4.2. We can observe that the basic scheme does not scale beyond

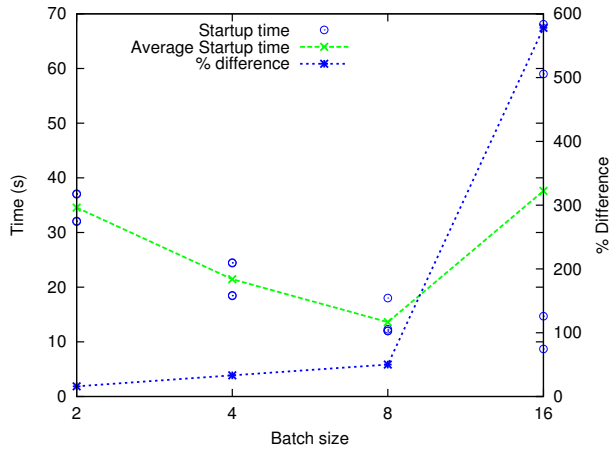



**Figure 5: Variation in startup time with batch size for 4K processors**

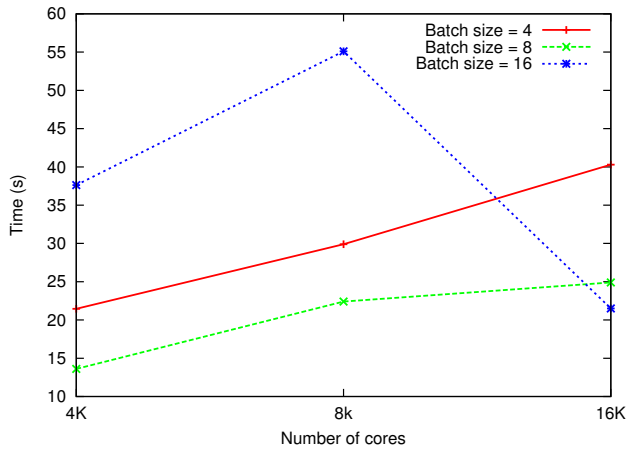



**Figure 6: Startup time vs number of cores for different batch sizes**

4K processors. For 8K cores, we waited 8 minutes for startup to finish using basic scheme; our allocation on Ranger did not allow us to wait indefinitely. For 4K processor cores (256 nodes), the basic startup scheme takes 237 seconds to finish startup, SMP-aware startup takes 51 seconds whereas multi-level startup without batching takes 37 seconds on average giving a speedup of 6.4X over the basic scheme and 1.4X over SMP-aware startup. Moreover, the slopes of the three lines in the figure show that multi-level startup scales better with increasing number of processors as compared to other schemes. With a branching factor equal to $\sqrt{N}$ where N is the number of nodes, we expect the time taken by `ssh` phase of startup to grow as $\theta(\sqrt{N})$ instead of $\theta(P)$ in the base scheme where P is the number of processors. If we use more levels of charmrun, we get a tree startup scheme that can scale as $\theta(log(N))$. However, in our experiments, 2-level scheme was sufficient since Ssh time is not the main bottleneck as discussed in section 3.3. The main bottleneck is the Wait time which is reduced further by using batching. We discuss improvement in performance and scalability due to batching in section 4.2.

## 4.2 Effect of Batching

We experimented with different batch sizes to find out the optimal batch size. Figure 5 shows the effect of batch size on startup time for 4K processors (256 nodes) on Ranger. For 256 nodes, branching factor is kept 16. So, batch size of 16 is identical to starting all nodes (no batching). For each batch size on the x-axis, circles represent the startup time of a particular run. Each point on the lower line shows the % difference between the maximum and minimum startup time for a fixed batch size. The graph also shows average startup time for different batch sizes. We can make two important observations from this figure. First, we see that the variation in startup time increases with batch size. The variation is as high as 600% for no batching. Second, average startup time initially decreases with batch size, become lowest for batch size of 8 and increases again. The reason for that is the trade-off between the slowdown caused due to inherent serialization introduced by batching and the speedup caused due to avoiding congestion by reducing the total number of message in the network at any time.

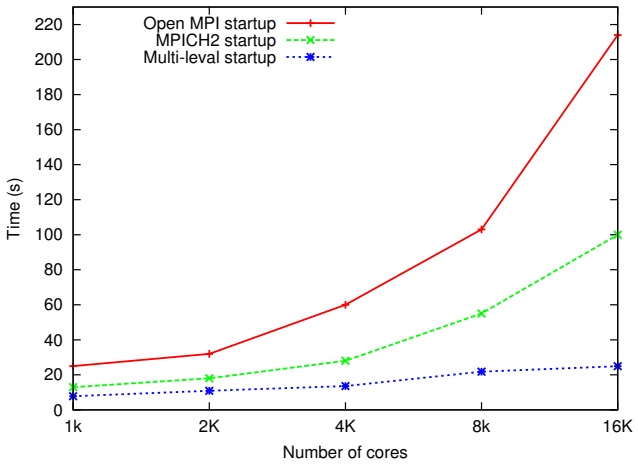



**Figure 7: Startup time on Ranger: Open MPI vs MPICH2 (Hydra) vs Multi-level Startup**

The results for scaling of startup using multi-level startup with batching with different batch sizes are shown in Figure 6. We see that the batch size of 8 performs the best as we increase the number of processors. Smaller batch size does not scale well with the number of processors due to the serialization introduced. A batch size of 16 does not perform well for 4K and 8K processors because it is close to no batching for these number of processors. It performs well for 16K processors (1024 nodes). We expect a batch size of 16 to be good for even higher number of cores.

## 4.3 Comparison with Open MPI and MPICH2 (Hydra) startup

Open MPI and MPICH2 are two of the most prominent free implementations of MPI standard. Open MPI [12] is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. It is used by many TOP500 supercomputers including Ranger. Ranger uses SGE (Sun Grid Engine) [13] parallel environment to launch and manage Open

MPI processes. MPICH2 [2] is a high performance and widely portable implementation of MPI standard. We installed MPICH2-1.3 on Ranger to compare our approach with hydra [5] which is the default process management framework for starting MPI processes for MPICH2-1.3 onwards. Hydra uses existing daemons such as ssh, rsh, pbs, slurm and sge to start MPI processes. We compared the startup time taken by our multi-level startup with batching with Open MPI and MPICH2 startup time on Ranger. MPI startup time was measured by calculating the difference between the time measured using Linux `date` command from the job script just before starting the parallel program and a timer call after MPI initialization. For our scheme, the startup time includes the time of the two phases of startup - parallel process launch and establishment of communication channels between parallel processes. Figure 7 shows the comparison between the startup time with varying number of processors. We can note that our startup scheme outperforms Open MPI startup by a factor of 8 and MPICH2 by a factor of 4 for 16K processors. Also, we see that our scheme scales very well with the number of cores. Startup for 16K cores on Ranger using multi-level startup takes only 25 seconds.

# 5. RELATED WORK

The problem of scalable startup for parallel application has been studied by many researchers. Butler et al. [9] presented a scalable process management system called MPD (for Multi-purpose Daemon) for parallel programs such as those written using MPI. The main idea is the presence of persistent daemons, typically one instance per host in a TCP-connected network. The daemons are connected in a ring. Manager processes are started by the daemons to control the application processes (clients) of a single parallel program and provide most of the MPD features. To run an MPI program, `mpirun` first connects to the daemon ring in order to start the parallel program and then switches to manager ring in order to control the program. Our approach does not assume presence of any daemons and provides fast startup and application management capabilities without needing any daemons. Yu et al. [26] have done research on startup of MPI programs on InfiniBand clusters. They use MPD for process spawn and focus on reducing the data volume exchanged during information exchange.

SLURM [18] is a fault-tolerant and highly scalable cluster management and job scheduling system for large and small Linux clusters. It allocates resources (compute nodes) to users for some duration of time and also provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. ALPS [22] is another similar application placement and launch system. Similar to MPD, both SLURM and ALPS use persistent daemons running on each compute node for startup and monitoring.

Hydra [5] is the default process management framework for starting MPI processes for MPICH2-1.3 onwards. It uses existing daemons such as ssh, rsh, pbs, slurm and sge to start MPI processes. ScELA [24] is a job launch mechanism which targets multi-core clusters. It decouples the two phases in a parallel application launch - spawning of processes and information exchange between processes to complete initialization. It comprises a spawning agent which starts executables on target processors and the communication primitives are used within the executables to communicate necessary initialization information. ScELA process launch is similar to our SMP-aware startup since ScELA has a Node Level Agent (NLA) for every node. An NLA is used to launch all processes on a node. NLAs are active only for the duration of launch, hence the framework is daemonless. However, since there is an NLA per node, there is an extra process per node consuming processor cycles. Our approach has child charmruns but they are only a few ($\sqrt{N}$ for 2-level startup on $N$ nodes). Moreover, they are necessary because that provides I/O capabilities and scalable interaction with parallel application.

Research has been done on concurrent launching strategies including tree-based launching. Claudel et al. [11] study the performance of standard remote execution protocols and explore various concurrent launching strategies. Also, they propose work-stealing method to balance the tasks of deployment to child nodes. They present TakTuk, a remote execution deployment system which can be used for fast and scalable distributed machine administration and parallel application development. Their work focuses on the execution of same process on a set of nodes. Another such parallel shell tool is GXP [25] which facilitates running an identical or a similar command line to many machines in parallel and getting results back interactively. In both TakTuk and GXP, the processes do not need to communicate with each other and hence the second phase of parallel startup - setting up communication channels is not needed.

Brightwell et al. [8] present the components of the runtime system for parallel application launch on Cplant project. They do not assume that the executable to be launched is available on a global file system. Our approach makes that assumption; since in our experience, that is the common case in high performance systems. Also, they do not discuss the issues in the connection establishment phase of parallel startup.

# 6. CONCLUSION

This paper presents a scalable multi-level approach for startup of parallel applications on large systems. Parallel startup consists of two phases - parallel launching of appropriate processes on the given set of processors and setting up communication channels to enable the processes to communicate with each other after startup has completed. We explored techniques to speed up both of these components. We also introduced the concept of batching of remote shell sessions and incorporated SMP-awareness to further improve scalability. We analyzed the performance of different startup techniques presented in this paper using a theoretical model and also evaluated their performance on TACC's Ranger cluster using Charm++. Our scheme was able to startup a Charm++ program on 16,384 cores of Ranger [3] with Ethernet as the underlying communication layer in only 25 seconds. We also compared the performance with Open MPI and MPICH2 (with hydra as the process manager) startup and our scheme outperformed Open MPI startup by a factor of over 8 and MPICH2 startup by a factor of 4 for 16K cores.

The multi-level startup system presented in this paper is a complete solution to the startup of a parallel application and its management during execution. It monitors process health and can be used to support recovery from failures and provide scalable interaction with the application.

## Acknowledgments

This work was supported in part by NSF grant OCI-0725070 for Blue Waters deployment, by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign, and by Department of Energy grant DE-SC0001845. We used machine resources on the Ranger cluster (TACC), under TeraGrid allocation grant TG-ASC050039N supported by NSF.

## 7. REFERENCES

[1] LiveViz Library. `http://charm.cs.uiuc.edu/manuals/html/libraries/6.html`.

[2] MPICH2:High-Performance and Widely Portable MPI. `http://www.mcs.anl.gov/mpi/mpich`.

[3] Ranger User Guide. `http://services.tacc.utexas.edu/index.php/ranger-user-guide`.

[4] Top500 supercomputing sites. `http://top500.org`.

[5] Using the Hydra Process Manager. `http://wiki.mcs.anl.gov/mpich2/index.php/Using_the_Hydra_Process_Manager`.

[6] Mpi: A message passing interface standard. In *M. P. I. Forum* (1994).

[7] Bhatele, A., Kumar, S., Mei, C., Phillips, J. C., Zheng, G., and Kale, L. V. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008* (April 2008), pp. 1–12.

[8] Brightwell, R., and Fisk, L. A. Scalable Parallel Application Launch on Cplant TM. In *In Proceedings of the SC2001 Conference on High Performance Networking and Computing* (2001).

[9] Butler, R., Gropp, W., and Lusk, E. A Scalable Process-Management Environment for Parallel Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds., vol. 1908 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2000, pp. 168–175. 10.1007/3-540-45255-9_25.

[10] Chakravorty, S. *A Fault Tolerance Protocol for Fast Recovery*. PhD thesis, Dept. of Computer Science, University of Illinois, 2008.

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

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

[13] Gentzsch, W. Sun Grid Engine: towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on* (2001), pp. 35 –36.

[14] Gioachin, F., Lee, C. W., and Kalé, L. V. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09* (Arlington, VA, USA, June 2009).

[15] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22*, 6 (September 1996), 789–828.

[16] Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., Pailet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R., and Mattson, T. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International* (February 2010), pp. 108 –109.

[17] Infiniband Trade Association. Infiniband Architecture Specification, Release 1.0. Tech. Rep. RC23077, October (2004).

[18] Jette, M. A., Yoo, A. B., and Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[19] Jyothi, R., Lawlor, O. S., and Kale, L. V. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004* (2004), IEEE Press, p. 294.

[20] Kalé, L., and Krishnan, S. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications* (September 1993).

[21] Kale, L. V., and Zheng, G. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[22] Karo, M., Lagerstrom, R., Kohnke, M., and Albing, C. The Application Level Placement Scheduler. In *Cray User Group May 2006*.

[23] Laboratory, P. P. *Converse Programming Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. http://charm.cs.uiuc.edu/manuals/html/converse/manual.html.

[24] Sridhar, J. K., Koop, M. J., Perkins, J. L., and Panda, D. K. ScELA: scalable and extensible launching architecture for clusters. In *Proceedings of the 15th international conference on High performance computing* (Berlin, Heidelberg, 2008), HiPC'08, Springer-Verlag, pp. 323–335.

[25] Taura, K. GXP : An Interactive Shell for the Grid Environment. *Innovative Architecture for Future Generation High-Performance Processors and Systems, International Workshop on 0* (2004), 59–67.

[26] Yu, W., Wu, J., and P, D. K. Fast and Scalable Startup of MPI Programs in InfiniBand Clusters.