A MULTI-LEVEL SCALABLE STARTUP FOR PARALLEL APPLICATIONS

BY

ABHISHEK GUPTA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Laxmikant V. Kalé

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 special purpose daemons which waste system resources or using a startup manager which becomes a scalability bottleneck. In this thesis, 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 still supports existing Charm++ runtime capabilities including process health monitoring, facilitation of recovery from failures and 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 MPICH2-1.3 startup (using Hydra as process manager) and over 800% compared to Open MPI 1.3.1 startup on Ranger.

To my parents, my wife and my newborn son.

Acknowledgements

I would first like to thank my advisor, Professor Kalé, for his invaluable guidance throughout this research. He has helped to guide this work from its conception and suggested ways to overcome the problems I encountered along the way.

Next, I would also like to thank Filippo Giaochin and Eric Bohm, my colleagues at the Parallel Programming Laboratory, for their insights into the design choices and implementation details in this work. Further, I thank them for answering even the most naive questions I had regarding parallel computing in general and Charm++ runtime and startup system in particular.

I would also like to thank Gengbin Zheng for his valuable insights into this work and helping me with the writing of a paper on this research. Also, Phil Miller helped me a lot during the process of getting my feet into the Linux environment. Further, the feedback obtained from the members of Charm++ core group during meetings has great influence in making this work possible.

Esteban Meneses helped in reviewing the thesis and provided his invaluable feedback. I would also like to thank these and all members of the Parallel Programming Laboratory not just for their help, but above all for their friendship.

Finally, I thank my parents for their unconditional love and understanding even though I am living so far from them. Further, my wife Kritika's incessant support, care and love throughout this work has been invaluable. Also, my newborn son has brought smile on my face in times of trouble.

Grants and other acknowledgements: 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.

Table of Contents

List of Tables

List of Figures

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 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, i.e. how to start the application on all the computation nodes (as an example, this is what `mpirun` does to start MPI [1] applications). On a large machine, there may be a significant delay between job allocation and application execution.[1]

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 launch-*

[1] Some portions reprinted, with permission, from "A Multi-level Scalable Startup for Parallel Applications" by Abhishek Gupta, Gengbin Zheng and Laxmikant V. Kale at the *1st International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2011) held in conjunction with ICS 2011*, ©2011 ACM

ing 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 many MPI implementations, 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 [2] and TakTuk [3], which are typically exploited for administrative purposes such as running updates and setting up configuration on all computation nodes. This thesis focuses on the startup process of parallel applications and scalable techniques to speed up the two components described above.

1.1 Motivation

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++ [4] and Open MPI [5] take 2 to 4 minutes for startup on 8K processors on Ranger [6] (at TACC) 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 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 purpose daemons running on compute nodes to facilitate the startup process [7, 8, 9]. An example of this is a system called Multi-purpose Daemons (MPD) [7] used for MPICH [10] 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 existing daemons such as `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 7.1).

1.2 Overview

In this thesis, we present a multi-level scalable startup method which is generic and can be applied to most parallel programming environments, including MPI and Charm++. 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 programming model for large scale scientific and engineering applications including NAnoscale Molecular Dynamics (NAMD) [11], which is a highly scalable molecular dynamics code used ubiquitously on the TeraGrid and other HPC systems. Multi-level startup can be enabled by providing a particular runtime flag at Charm++application launch command line. Starting up a Charm++ program on 16,384 cores of Ranger 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.

1.3 Contributions

The contributions of this thesis are the following:

- A multi-level startup mechanism for parallel applications, which can be used for any parallel programming environment.
- Concept of batching of remote shell sessions applied to parallel startup process.
- Theoretical analysis of the performance of different startup schemes and demonstration of their effectiveness by experimental results.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Related work is discussed in Chapter 2. Chapter 3 provides the background on parallel startup process and describes linear startup. Chapter 4 discusses the multi-level approach to parallel startup. In addition, this chapter introduces the concept of batching of remote shell sessions. A theoretical analysis of various startup schemes considered in this thesis is presented in Chapter 5. Next, the runtime capabilities supported by the startup system are discussed in Chapter 6. Chapter 7 presents performance evaluation of our multi-level startup scheme and compares them with startup schemes used by other prominent parallel programming systems. Finally, conclusions and future work are left for the final chapter. Portions of this thesis have been published by Gupta et. al. [12].

2 Related Work

The problem of scalable startup for parallel application has been studied by many researchers. Butler et al. [7] 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 special purpose 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. [13] 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 [8] 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 [9] is a similar application placement and launch system designed to address the needs of current and future Cray systems. STORM [14] is another resource management framework designed for scalability and performance and provides job-launch mechanisms. Similar to MPD, SLURM, ALPS and STORM use special purpose daemons running on each compute node for startup and monitoring.

Hydra [15] 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 [16] 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. Our SMP-aware startup (section 3.3) is similar to ScELA process launch 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 proposed approach (discussed in chapter 4) has child charmruns but they are only a few (e.g. $\sqrt{N}$ for 2-level startup on N nodes). Moreover, they are necessary because that provides I/O capabilities and scalable interaction with parallel application.

Brightwell et al. [17] 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 proposed approach makes that assumption; since in our experience, that is the common case in high performance systems.

Research has been done on concurrent launching strategies including tree-based launching. Claudel et al. [3] 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 [2] 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.

A comprehensive analysis and comparison of current parallel startup mechanisms with a modified Open MPI implementation is presented in [18]. They discuss the scalability of a parallel runtime system with focus on its four major functions - *launch, connect, control and io.* Further, they propose an all-gather algorithm based on flooding for the contact information exchange phase of startup. In addition, they discuss the basic requirements of a runtime system for launching and management of an exascale-level parallel application.

3 Background

3.1 CHARM++ and Charmrun

CHARM++ [4, 19, 20, 21, 22] is a parallel programming language based on the concept of object based over-decomposition and processor virtualization. The basic idea is that programmer decomposes the program into objects which can then be intelligently mapped by the runtime system on the set of available processors. This division of responsibility relieves the programmer from the burden of performing mapping and ensures best utilization of resources by the runtime. In addition, CHARM++ runtime system provides the benefits of adaptive overlap of computation and communication through the use of asynchronous message invocation using entry methods. Further, the runtime system provides additional benefits such as load balancing and fault tolerance.

CHARM++ uses a program called *Charmrun* for the parallel startup processes. Charmrun accepts various runtime input for launching a parallel application, Some important input parameters taken by charmrun are:

1. Number of parallel processes to be launched
2. Application executable
3. Nodelist file - a file containing the host names of nodes where the application will be launched
4. Run time parameters for the CHARM++ run time system

5. Run time parameters for the application

An example command for launching an application using charmrun is

```
./charmrun +p1024 ++nodelist hostfile ./pgm
```

This will perform the parallel startup for the executable “pgm” on the 1024 processors specified in the file “hostfile”.

3.2 Linear Startup

The basic tasks involved in startup of a parallel application are illustrated in Figure 3.1. Although the technique is generic and can be applied to most parallel programming environments such as MPI, we present the scheme in the context of Charm++. In this thesis, we will refer to the launcher as *charmrun* and the processes which constitute the parallel application as *clients*. 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, which is currently used in Charm++, is using a machine file, which we call the *nodelist* file. The functionality of charmrun is described next.

Remote process launch

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

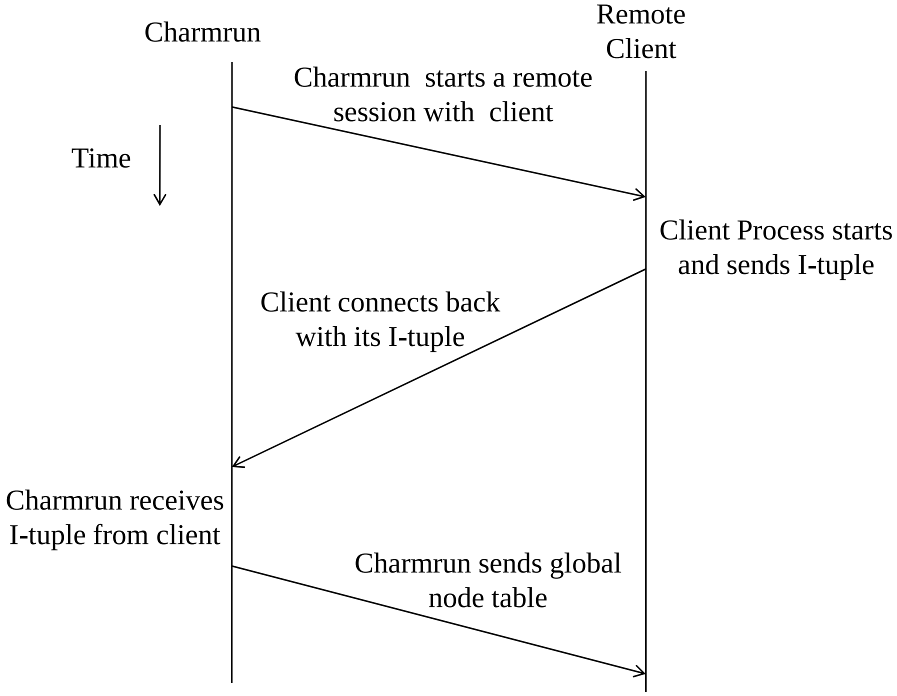

Charmrun
Remote Client
Charmrun starts a remote session with client
Time
Client Process starts and sends I-tuple
Client connects back with its I-tuple
Charmrun receives I-tuple from client
Charmrun sends global node table

Figure 3.1: Basic process of parallel startup

call. This task is performed for each processor in the nodelist file. We refer to this scheme as linear startup or centralized startup.

Establishment of communication channels

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 [23].

Polling mode

Charmrun is needed even after startup has completed since it acts as an interface between parallel application and the external world during execution. All input output and some additional features (discussed in chapter 6) such as parallel debugging, failure detection and process restart can utilize charmrun.

3.3 SMP-Aware Startup

There is an optimization which can lead to significant improvement in the performance of centralized startup. The scheme discussed in section 3.2 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, 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 linear startup. Each client sends an I-tuple to charmrun, which collects them and sends the node-table to every client.

4 Multi-level Startup

4.1 Approach

Even with the optimization regarding SMP-awareness, discussed in previous chapter, 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 current supercomputers, with hundreds of thousands of nodes, centralized startup becomes a bottleneck and the startup performance degrades significantly with increase in number of cores (see section 7.1). It is clear 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. Figure 4.1 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 nodes. 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 charmruns) which we call k. For simplicity, we keep the branching factor

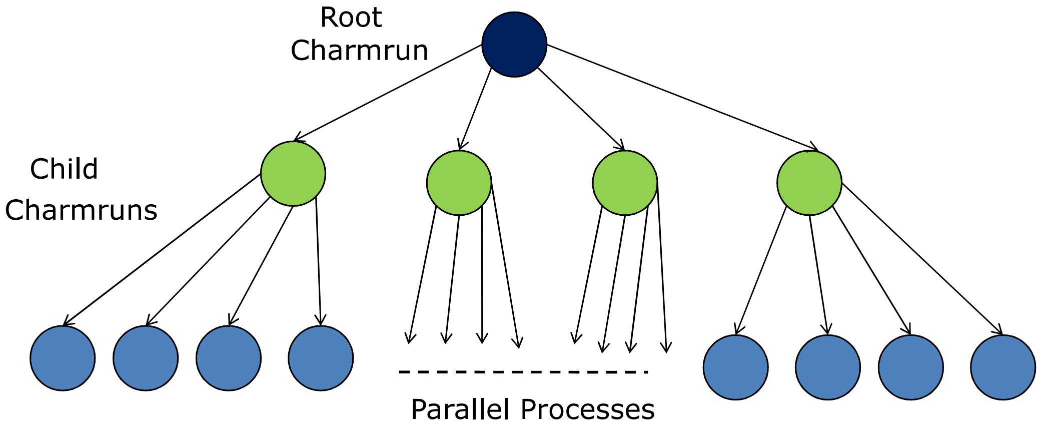


Figure 4.1: Multi-level startup scheme

as the square root of number of unique nodes where the application will be run. 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.

4.2 Design Issues and Alternatives

During the design of our multi-level startup system, we encountered several design choices, some of which are discussed here. The first issue is the option between uniform degree tree vs. non-uniform degree tree. The possible alternatives are:

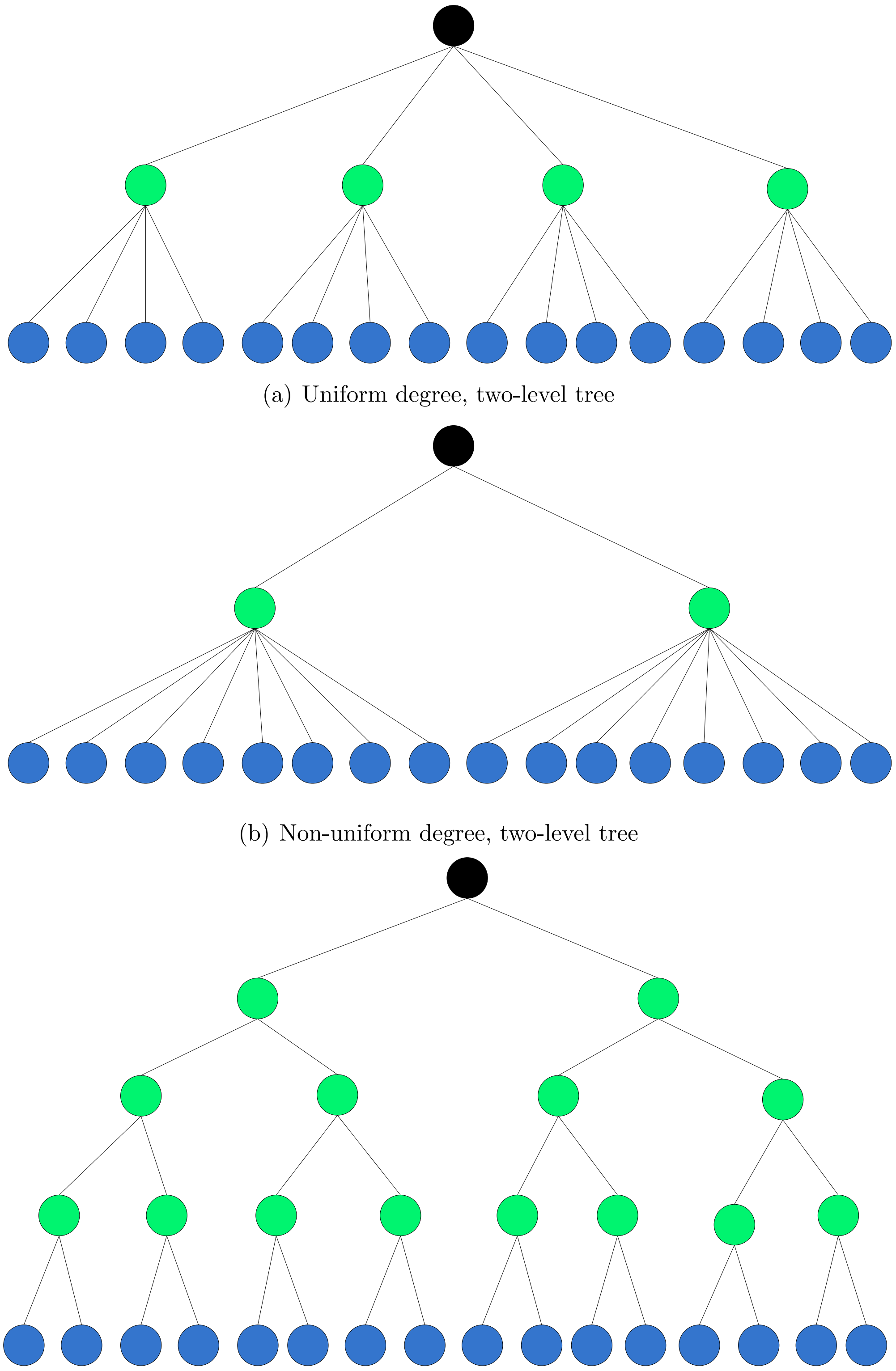
1. Uniform branching factor tree

2. Different branching factors at different levels of the tree

An example of a non-uniform degree tree is the case where there are two different branching factors - one when the children are charmruns and other when the children are clients (see figure 4.2(b)). We chose a uniform branching factor since the use of non-uniform branching factor would make our design and implementation complicated without much additional gains in terms of performance, especially since we are considering homogeneous clusters. Hence we concentrated our efforts on a k-ary d-level tree where k is the branching factor and d is the depth of tree.

Another design issue is the determination of suitable branching factor and tree depth (e.g. see figures 4.2(a) and 4.2(c)). Chapter 5 provides an analysis of the effect of branching factor and number of levels on performance. The basic idea is to determine the optimal point for the tradeoff between the parallelism resulting from presence of multiple charmruns and the overhead due to extra levels of tree. Moreover, larger number of levels result in higher levels of forwarding when charmrun is in polling mode after startup has finished. 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 [16] 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 ($d = 2$), $k = \sqrt{P}$.

The third design issue that we encountered was the option of terminating the charmrun tree after startup is finished. However, we realized that 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. Further, its useful for startup of a new process in case of recovery from failures. In addition, decentralized existence of charmruns makes parallel input output and user interaction with parallel application more scalable.



(a) Uniform degree, two-level tree

(b) Non-uniform degree, two-level tree

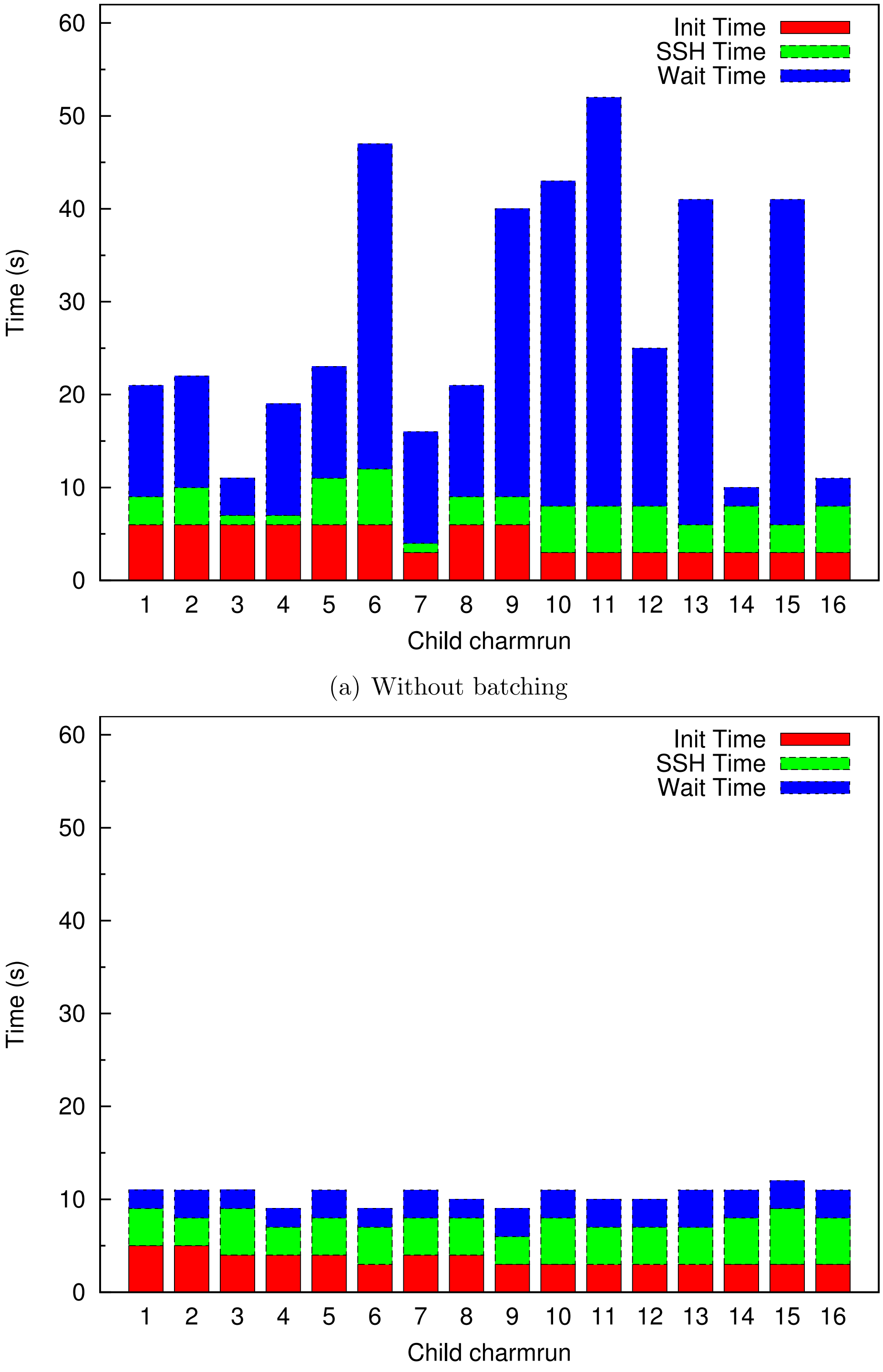
(c) Binary tree

Figure 4.2: Variations in startup tree - figure shows some of the possible startup trees for launching 16 clients

4.3 Inconsistent Performance

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 4.3(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 for 2-level startup. So, there are 16 child charmruns. *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.

One hypothesis was that the unreasonable delay in client connect could be due to the synergistic effect of packet loss and TCP Nagle's algorithm [24]. Nagle's algorithm aims to solve the "small packet problem" by automatically limiting the transmission of unnecessarily small packets if there is an outstanding acknowledgement. Hence, in the presence of packet losses, it could potentially introduce message delays. However, disabling Nagle's algorithm did not solve the problem.



(a) Without batching

(b) With batching using batch size of 8

Figure 4.3: Breakup of time spent in parallel startup using multi-level scheme for 4K processors

4.4 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 number 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 4.3(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 4.3(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 7.2.

4.5 Runtime Flags

Because of these tradeoffs involved in the different startup schemes, we provide runtime options to the application users which enable them to choose the method for starting their parallel application. Table 4.1 describes the various flags which can be provided at execution time.

Charmrun Command	Description
`./charmrun +p numProc ./exec`	Linear startup

`./charmrun +p numProc ./exec`
`+batch batchsize`
	Linear startup with batching of remote shell sessions, *batchsize* sessions will be started in one phase

`./charmrun +p numProc ./exec`
`++scalable-start`
	SMP-Aware startup

`./charmrun +p numProc ./exec`
`++scalable-start`
`+batch batchsize`
	SMP-Aware startup with batching of remote shell sessions, *batchsize* sessions will be started in one phase

`./charmrun +p numProc ./exec`
`++hierarchical-start`
	Multi-Level startup, a 2 level tree is the default

`./charmrun +p numProc ./exec`
`++hierarchical-start`
`+batch batchsize`
	Multi-Level startup with batching of remote shell sessions, *batchsize* sessions will be started in one phase, batching is used by for leaf-level charmruns. Multi-level startup is also SMP-aware.

Table 4.1: Various flags available with charmrun regarding the startup method to be used

5 Theoretical Analysis

In this chapter, we present a theoretical analysis of the different startup schemes discussed in this thesis. 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 linear startup scheme (T_{linear}) can be modeled as:

$$\begin{aligned} T_{linear} = T_{init} + P \times T_{ssh} + T_{client} \\ + T_{send} + T_{nw} + P \times T_{recv} \end{aligned} \tag{5.1}$$

where T_{init} is the charmrun initialization time (which includes getting the list of nodes to start by 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 consider the total overhead due to T_{init}, T_{client}, T_{send} and T_{nw} as constant and represent that by T_c to keep the analysis readable.[1] Hence, we have

$$T_{linear} = T_c + P \times (T_{ssh} + T_{recv}) \tag{5.2}$$

[1] T_{nw} is the maximum network latency of a short message between any two processors of the supercomputer. Note that with increase in the number of processors, T_{nw} (and hence T_c) could increase due to network contention. This would depend on the network architecture and should be considered for a more accurate analysis.

Technique	Startup Time
Linear startup	$T_{linear} = T_c + P \times (T_{ssh} + T_{recv})$
SMP-Aware startup	$T_{SMP} = T_c + (c-1) \times T_{client} + N \times (T_{ssh} + c \times T_{recv})$
Multi-level startup	$T_{d-level} = d \times (T_c + k \times (T_{ssh} + T_{recv})) + k \times (c - 1) \times (T_{recv} + T_{client})$
Multi-level startup with batching	$T_{\text{batched d}-\text{level}} = (d-1) \times (T_c + k \times (T_{ssh} + T_{recv})) + (k/b) \times T_c + k \times (T_{ssh} + c \times T_{recv})$

Table 5.1: Theoretical Analysis of startup time for various schemes

where

$$T_c = T_{init} + T_{client} + T_{send} + T_{nw} \tag{5.3}$$

SMP-aware startup starts only one `ssh` process per node and hence it reduces the total ssh time. However, after each node-process is started, the node-process has to start $c-1$ other clients, In addition, it still incurs the receive overhead for all clients. Hence, the time taken by SMP-aware startup (T_{SMP}) can be modeled as:

$$T_{SMP} = T_c + (c-1) \times T_{client} + N \times (T_{ssh} + c \times T_{recv}) \tag{5.4}$$

Hence, both T_{linear} 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 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 linear 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:

$$\begin{aligned} T_{d-level} &= d \times (T_c + k \times (T_{ssh} + T_{recv})) \\ &+ k \times (c-1) \times (T_{recv} + T_{client}) \end{aligned} \tag{5.5}$$

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

$$d = \log_k N \tag{5.6}$$

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 $N/kd = N/(k \log_k N)$. We note that $T_{d-level}$ grows asymptotically as $\theta(k \log_k N)$ instead of $\theta(P)$ for linear startup. An optimal value of the number of levels can be obtained by minimizing $T_{d-level}$ using equation 5.5, with k given by equation 5.6. Determining the exact expression for this optimal value is beyond the scope of this thesis.

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 [16] 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 ($d = 2$), $k = \sqrt{P/c} = \sqrt{N}$.

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:

$$\begin{aligned} T_{\text{batched d}-\text{level}} &= (d-1) \times (T_c + k \times (T_{ssh} + T_{recv})) \\ &+ (k/b) \times T_c + k \times (T_{ssh} + c \times T_{recv}) \end{aligned} \tag{5.7}$$

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. Table 5.1 summarizes the results of our analysis.

6 Continued Support for Existing Runtime Capabilities

The startup system presented in this thesis 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 multiple capabilities which exist in the basic charmrun system but needed modification for correct functioning with the multi-level system.

6.1 Process Health Monitoring

Charmrun monitors process health and provides for complete application termination (which involves termination of processes on all processors) when a process exits or crashes. In the absence of proper parallel exit mechanisms, processes can continue to execute and consume CPU cycles even after the user thinks that application has exited. This can interfere with the performance of other application which are run after the terminated application. With our modification to make charmrun multi-level, there is division of responsibility among various charmruns. Each leaf level 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, charmruns are responsible to terminate the whole application if there is no support for fault tolerance. This involves forwarding the information that the process has failed to upper levels of startup tree. Once the root charmrun discovers the failure of a client, it broadcasts it to its children which in turn, forward it to their children till it reaches the clients. A second type of failure consists of the cases where a child charmrun fails. This can happen if the process crashes or the node where the charmrun resides crashes. The second case would involve the failure of both - one or more clients and a charmrun. We also detect these failures to provide a clean application exit. In addition, multi-level charmrun system makes process health monitoring and failure recovery a decentralized process and hence more scalable.

6.2 Fault Tolerance - Process Restart

Charmrun system has also been used to facilitate the design of fault tolerance protocols for Charm++ applications [25, 26, 27, 28, 29]. 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. Use of multi-level charmrun required proper coordination between different charmruns to ensure proper restart protocol. Presence of multiple charmruns presents multiple complicated failure scenarios. Consider the three cases for a two-level charmrun:

1. Failure occurs on a processor that does not have any second-level charmrun process. We ensure that this case is handled well with our system.

2. General case, where the failure not only kills one application process but also makes one second level charmrun node to crash. This requires that the charmrun at the root of the tree remember the IPs and ports of the failed node's children and raises the question that what happens to the other children of the failed charmrun (since they are still alive)? Techniques for supporting such scenarios is open to exploration.

3. Failure kills the root charmrun process - in this case recovery is not supported by our system.

It is clear that having a hierarchical process launcher makes things more scalable, but less resilient. A failure in one of the intermediate nodes of the startup tree will shut off a portion of the nodes.

6.3 Request Handling from Clients

During execution charmrun also acts as a means of any communication between the client process and the outside world. Client provide different kinds of requests to charmrun which it serves. Some examples of these are the following.

```
req_handle_print
req_handle_printerr
req_handle_scanf
req_handle_barrier
req_handle_ending
req_handle_abort
```

In case of multi-level startup, client is only aware of its manager charmrun and not the root charmrun, so the request need to be handled by its manager. Handling these in multi-level startup case presents us with two design choices that we considered:

- Forward each request to parent charmrun as soon as received.
- Collect a fixed number of requests at each charmrun before forwarding them to parent.

The first approach ensures prompt response for an input output request whereas the second approach aims for minimizing network traffic by reducing the number of messages sent between charmruns. We chose the first approach based on the observation that most input/output is performed for application debugging and understanding application execution and for this purpose, prompt response is desired. Child charmruns forward the request to the root charmrun where it gets serviced and the response is then sent to the client through the charmrun tree.

6.4 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 [30]. Developers and end-users of parallel applications can greatly benefit from these capabilities provided by a runtime system. These capabilities are discussed in following sections.

6.4.1 Converse Client Server (CCS)

Converse Client Server (CCS) [31] is a communication protocol that allows parallel applications to act as remote server that receives and serves requests from remote clients. In CCS protocol, charmrun listens to remote requests at a specific port and forwards them to appropriate clients. When it receives a response from application client for a CCS request, it sends the CCS response to the remote requesting client. In multi-level scheme, only the root charmrun opens the CCS port during application startup. Requests received from remote clients by root charmrun follow the same forwarding mechanism as in section 6.3 although in opposite direction. Multi-level charmrun system makes this process more scalable by distributing the task of managing the client processes among multiple charmruns and hence preventing the single charmrun from becoming a bottleneck.

6.4.2 CharmDebug

We have used CharmDebug [32, 33] with our multi-level charmrun system to debug parallel application. CharmDebug is a graphical tool that allows programmers to debug large scale parallel programs. CharmDebug uses CCS for interacting with the parallel application. It can be used on large number of processors, displaying information about objects, breakpoint, and the content of each processor. Furthermore, it can also be used for applications written in AMPI [34]. Moreover, CharmDebug can be used in a virtualized environment to reproduce and debug scaling bugs using small number of processors [33].

6.4.3 Online Visualization

CCS can also be used to provide simulation visualization using tools such as LiveViz [35]. LiveViz is a library which enables a Charm++ application to present output to the user in a graphical form as the program continues to execute. The live stream of images can be used by remote client to provide online visualization of the scientific phenomena or simulation performed by application. Online visualization allows the application user to understand the nature of the scientific process without waiting for it to finish execution. User can determine at runtime to stop or continue the simulation by observing the visual results obtained so far in simulation.

7 Performance Results

We ran performance tests on TACC's Ranger Cluster which is one of the largest computational resources in the world (Ranked 25 in the November, 2011 top500 list [36] 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 [6]. 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 sections discuss various performance results we got on this supercomputer.

7.1 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 7.1 compares three schemes - linear startup, SMP-aware startup and multi-level startup without batching. The effect of batching is discussed in section 7.2. We can observe that linear startup does not scale beyond 4K processors. For 8K cores, we waited 8 minutes for startup to finish using linear startup; our allocation on Ranger did not allow us to wait indefinitely. For 4K processor cores (256 nodes), linear startup takes 237 seconds to finish startup, SMP-aware startup takes 51 seconds whereas multi-level startup without batching takes

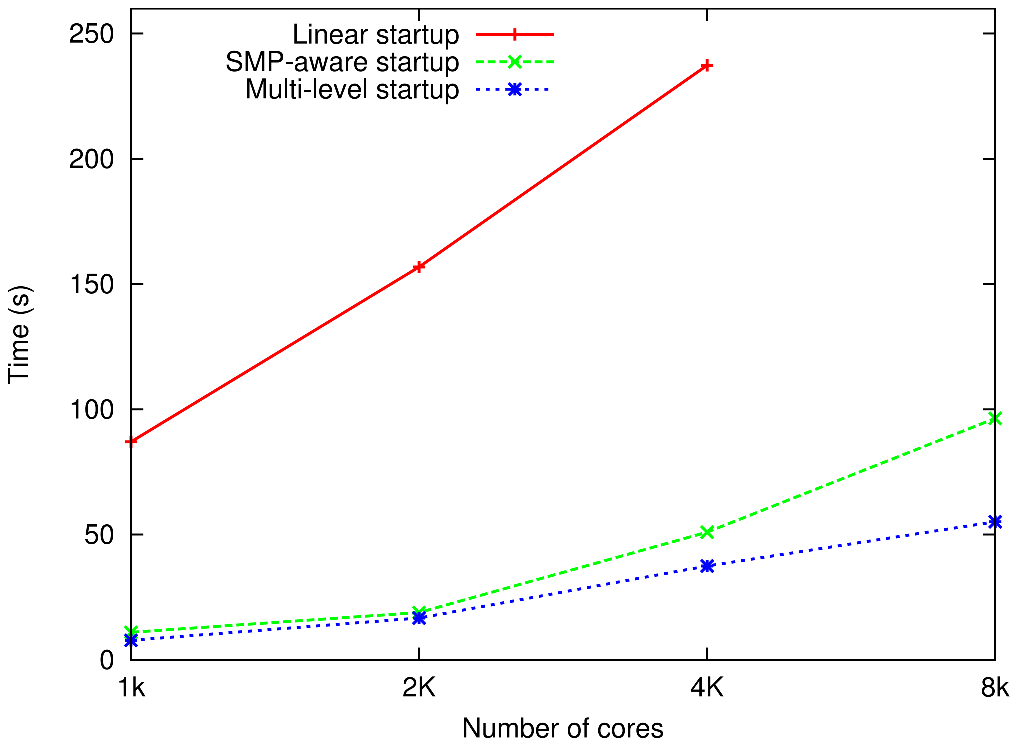


Figure 7.1: Startup time: Comparison among three startup schemes

37 seconds on average giving a speedup of 6.4X over linear startup 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 4.4. 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 7.2.

7.2 Effect of Batching

We experimented with different batch sizes to find out the optimal batch size. Figure 7.2 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.

The results for scaling of startup using multi-level startup with batching with different batch sizes are shown in Figure 7.3. 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.

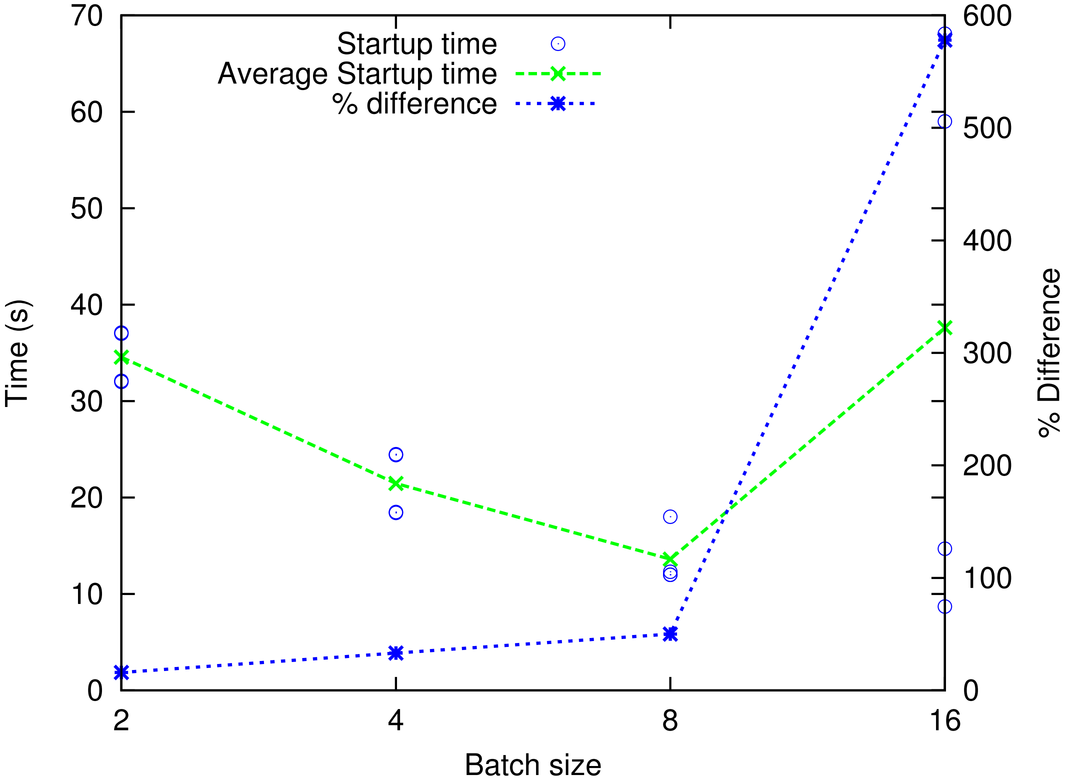


Figure 7.2: Variation in startup time with batch size for 4K processors

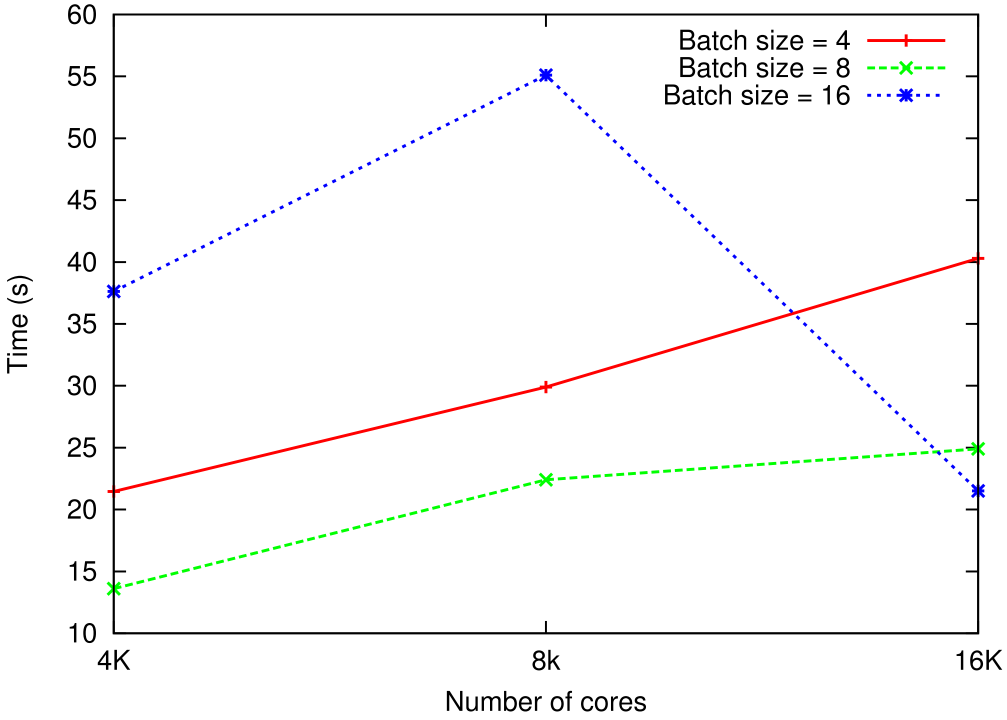


Figure 7.3: Startup time vs number of cores for different batch sizes

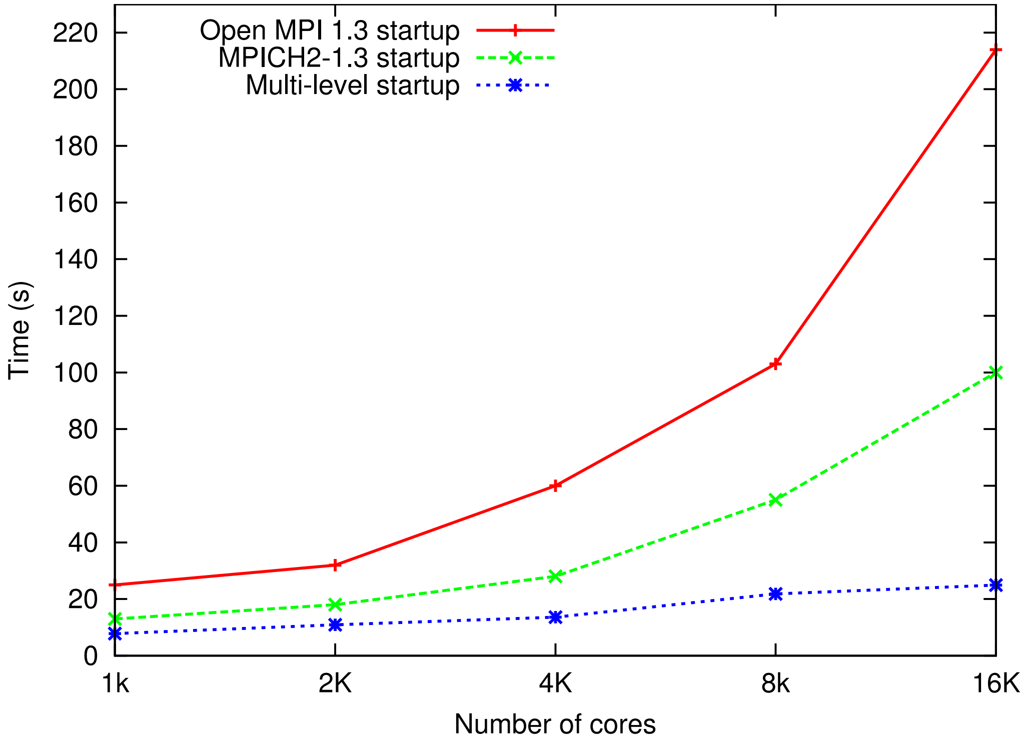


Figure 7.4: Startup time on Ranger: Open MPI vs MPICH2 (Hydra) vs Multi-level Startup

7.3 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 [5] 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. The available installation of Open MPI on Ranger when performing experiments was Open MPI 1.3.1. Ranger uses SGE (Sun Grid Engine) [37] parallel environment to launch and manage Open MPI processes. MPICH2 [38] is a high performance and widely portable implementation of MPI standard. We installed MPICH2-1.3 on Ranger to compare our approach with Hydra [15] 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 7.4 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. We acknowledge that at the time of writing this thesis, later versions of these MPI implementations have started using tree-based parallel launch mechanisms similar to our multi-level startup.

8 Conclusions and Future Work

8.1 Summary

This thesis 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 thesis 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 16K cores of Ranger [6] 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 1.3 startup by a factor of over 8 and MPICH2-1.3 startup by a factor of 4 for 16K cores.

The multi-level startup system presented in this thesis is a complete solution to the startup of a parallel application and its management during execution. It continues to support existing capabilities of charmrun such as process health monitoring, support for recovery from failures and scalable interaction with the application.

8.2 Future Work

In future, we plan to extend this work to startup of parallel application using underlying high-performance interconnects such as Infiniband. Also, we plan to explore parallel startup techniques that involve lazy establishment of communication channels in the second phase of startup, which requires on-demand connection establishment for the remainder during execution. In our current startup mechanisms, communication channels are set up between every two processes making it a $\theta(P^2)$ operation and hence a scalability bottleneck. Moreover, communication graphs in applications are very sparse. Rarely a node needs to communicate with more than 8 other nodes. Investigations on the scalability of eager vs. lazy connection establishment have been performed in [39] and further analysis needs to be done to examine its effect on application runtime performance.

Another future direction of research is hierarchical fault tolerant job launch mechanism. Currently, if a node containing a charmrun process crashes, failure recovery poses significant challenges since this failure results in termination of a subset of clients. Addition of extra links (redundancy) in the startup tree would help in tolerating failures. Assuming nodes rarely fail concurrently [40], one extra link per node is sufficient. Further, that extra link would add an additional route for sharing information.

References

[1] MPI: A Message Passing Interface Standard. In *M. P. I. Forum*, 1994.

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

[3] Benoit Claudel, Guillaume Huard, and Olivier 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.

[4] L.V. Kalé and Sanjeev Krishnan. 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.

[5] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 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.

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

[7] Ralph Butler, William Gropp, and Ewing Lusk. A Scalable Process-Management Environment for Parallel Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes in Computer Science*, pages 168–175. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-45255-9_25.

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

[9] Michael Karo, Richard Lagerstrom, Marlys Kohnke, and Carl Albing. The Application Level Placement Scheduler. In *Cray User Group May 2006*.

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

[11] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, pages 1–12, April 2008.

[12] Abhishek Gupta, Gengbin Zheng, and Laxmikant V. Kalé. A Multi-Level Scalable Startup for Parallel Applications. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 41–48, New York, NY, USA, 2011. ACM.

[13] Weikuan Yu, Jiesheng Wu, and Dhabaleswar K. P. Fast and Scalable Startup of MPI Programs in InfiniBand Clusters.

[14] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *In Supercomputing 2002*, 2002.

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

[16] Jaidev K. Sridhar, Matthew J. Koop, Jonathan L. Perkins, and Dhabaleswar K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *Proceedings of the 15th international conference on High performance computing*, HiPC'08, pages 323–335, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Ron Brightwell and Lee Ann Fisk. Scalable Parallel Application Launch on Cplant TM. In *In Proceedings of the SC2001 Conference on High Performance Networking and Computing*, 2001.

[18] George Bosilca, Thomas Hérault, Ala Rezmerita, and Jack Dongarra. On Scalability for MPI Runtime Systems. In *CLUSTER*, pages 187–195, 2011.

[19] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *Parallel Programming Laboratory Technical Report #95-02*, 1994.

[20] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[21] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[22] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[23] Infiniband Trade Association. Infiniband Architecture Specification, Release 1.0. Technical Report RC23077, October (2004).

[24] John Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comput. Commun. Rev.*, 14:11–17, October 1984.

[25] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Dieago, CA, September 2004.

[26] Chao Huang. System Support for Checkpoint and Restart of Charm++ and AMPI Applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[27] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive Fault Tolerance in Large Systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

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

[29] Esteban Meneses, Greg Bronevetsky, and Laxmikant V. Kale. Evaluation of Simple Causal Message Logging for Large-Scale Fault Tolerant HPC Systems. In *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[30] Filippo Gioachin, Chee Wai Lee, and Laxmikant V. Kalé. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[31] Parallel Programming Laboratory. *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.

[32] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[33] Filippo Gioachin, Gengbin Zheng, and Laxmikant V. Kalé. Debugging Large Scale Applications in a Virtualized Environment. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, number 10-11, Huston, TX (USA), October 2010.

[34] Chao Huang, Gengbin Zheng, and Laxmikant V. Kalé. Supporting Adaptivity in MPI for Dynamic Parallel Applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

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

[36] Top500 Supercomputing Sites. `http://top500.org`.

[37] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35 –36, 2001.

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

[39] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Trff. MPI on a Million Processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 20–30. Springer Berlin / Heidelberg, 2009.

[40] Esteban Meneses, Xiang Ni, and Laxmikant V. Kale. Design and Analysis of a Message Logging Protocol for Fault Tolerant Multicore Systems. Technical Report 11-30, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2011.