An Enhanced MPI-2 Dynamic Process Management Support for InfiniBand

Krishna Kandalla, Miao Luo, Sreeram Potluri, Tejus Gangadharappa, Matthew Koop
and Dhabaleswar K. Panda
Department of Computer Science and Engineering, The Ohio State University
Columbus, OH, U.S.A
{kandalla, luom, potluri, gangadha, koop, panda}@cse.ohio-state.edu

MVAPICH2 DPM DESIGN FOR IB

Krishna Kandalla
2015 Neil Avenue, Columbus, OH 43210-1277, Phone: (614) 292 - 5813 / Fax: (614) 292 - 2911
kandalla@cse.ohio-state.edu

Miao Luo
2015 Neil Avenue, Columbus, OH 43210-1277, Phone: (614) 292 - 5813 / Fax: (614) 292 - 2911
luom@cse.ohio-state.edu

Sreeram Potluri
2015 Neil Avenue, Columbus, OH 43210-1277, Phone: (614) 292 - 5813 / Fax: (614) 292 - 2911
potluri@cse.ohio-state.edu

Tejus Gangadharappa
gangadha@cse.ohio-state.edu

Matthew Koop
koop@cse.ohio-state.edu

Dhabaleswar K. Panda
785 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43210-1277, Phone: (614) 292 - 5199 / Fax: (614) 292 - 2911
panda@cse.ohio-state.edu

***Correspondence to Krishna Kandalla, 395 Dreese Laboratories, 2015 Neil Avenue, The Ohio State University, Columbus, OH 43210-1277**

Abstract

Dynamic process management is a feature of MPI-2 that allows an MPI process to create new processes and manage communication between processes belonging to both the groups - the original set and the newly spawned set. In this paper, we design an MPI-2 dynamic process management interface over InfiniBand. We study the Unreliable Datagram (UD) and Reliable Connection (RC) transport modes of InfiniBand for dynamic job startup. We also propose an improved inter-group point-to-point design that relies on a zero copy mechanism and an enhanced intergroup collective framework that leverages the performance benefits of using shared memory based collectives. We also propose a set of micro-benchmarks to evaluate these designs. Our studies revealed that the UD based-design allows for better spawns rates and there are significant improvements in the performance of communication operations with our proposed framework. Finally, we provide an evaluation of using a re-designed ray-tracing application.

Keywords : Message Passing Interface, Dynamic Process Management, InfiniBand, Collective communication.

1 Introduction

The Message Passing Interface (MPI) is currently the most dominant model for programming parallel computers today. The MPI specification defines a standard interface for communication, providing both point-to-point and collective communication primitives. The static model of MPI-1 mandates that the number of tasks is fixed at job launch time. This requirement restricts the applications from spawning additional tasks for portions of the application or to expand and contract with compute node availability. As a result, the MPI-2 specification added support for dynamic process management. This allows MPI applications to create and communicate with new processes, thus providing a new paradigm for programming MPI applications. Dynamic process management is an emerging concept that enables application designers to adopt the well-known master/worker paradigm to solve some of the complex real-world problems. Dynamic process management is being used in grid application design and some of the multi-scale applications[24]. Several popular MPI implementations (OpenMPI[17], MPICH2[16]) currently support this feature.

InfiniBand is commonly used as the high performance interconnect as it offers low communication latencies and high bandwidths. As many as 30% of the top500 supercomputers use InfiniBand [1]. It is necessary to have an efficient dynamic process interface that addresses the connection management and communication issues for MPI over InfiniBand to enable applications to fully leverage the performance benefits of using InfiniBand networks. This opens up important problems that need to be addressed while designing the dynamic process interface. The dynamic process management interface must use the most suitable transport offered by InfiniBand to effeciently manage the connections between all the processes involved. In a dynamic environment, processes communicate over an inter-communicator that comprises of both the parent communicator and the spawned communicator. Most of the communication operations in MPI are optimized for processes that are within the same communicator group. Designing and evaluating the communication operations between two or more groups of communicators is an important research problem that we have addressed in this paper. Unlike many other MPI operations, there are no standard benchmarks for evaluating the performance of the dynamic connection management framework and the inter-group communication framework.

To summarize, the following are the main problems that we have addressed in our paper :

- *InfiniBand offers various transport options. What are the performance implications of using the different transport methods for the dynamic process management interface?*
- *How do inter-communicator point-to-point operations comapre with their intra-communicator counterparts? Can we improve the inter-communicator point-to-point communication framework to achieve lower latencies?*
- *The performance of inter-communicator collective operations will strongly impact the application run-times. Is it possible to enhance the inter-communicator collective communication framework for the dynamic process interface?*
- *In the context of dynamic process interface, it is necessary to have a correctly benchmark the performance of the various dynamic process management functions and the communication operations. Is it possible to create a new benchmark suite to evaluate the performance of such functions?*

Unlike traditional Ethernet models, additional setup requirements are required for InfiniBand. We have explored using two of the common InfiniBand transports for the connection management framework and studied their performance implications. We have optimized the communication framework for the dynamic process interface, by proposing an enhanced inter-group point-to-point design that utilizes the kernel based zero-copy mechanism called LiMIC, which leads to lower latencies for larger messages exchanged between two process within the same node [9]. We also propose an improved inter-group collective framework that exploits the shared memory based collectives to achieve significant performance benefits for the collective operations [14, 19, 10, 20] in the dynamic process environment. We also propose a new set of benchmarks that evaluate the important dynamic process management functions and the various point-to-point and collective communication operations that are defined in the MPI standard. Our designs are implemented in MVAPICH2[18]: a MPI library for InfiniBand and iWARP. We evaluate our designs and compare our results against the dynamic management framework support available in OpenMPI. To model a real-world application, we evaluate a ray-tracing application that was re-designed to use the dynamic process model.

The rest of the paper is organized as follows: In Section 2 we present an overview of the dynamic process management interface. An introduction to InfiniBand and its capabilities are presented in Section 3. In Section 4, we speak about the inter and intra communicator collective operations and describe the high performance collective designs being used for intra-communicator collectives . Section 5 presents the various issues involved with designing a high-performance dynamic process management solution. In Section 6, we propose a number of new benchmarks to evaluate the performance of dynamic process management implementations. Section 7 provides an evaluation of the various design options proposed using benchmarks and Section 8 provides an application evaluation. Section 9 cites work related to dynamic process management and finally Section 10 provides conclusions and offers future work in this area.

2 MPI and Dynamic Process Management

This section provides a brief overview of MPI communicators and the MPI-2 dynamic process management interface. We also discuss an application use-case that uses the MPI dynamic process interface.

2.1 MPI Communicators

An MPI process is described by a (rank, process group) pair. The communicator encapsulates the ranks and the process group for which the ranks are described. All MPI communications are described in the context of a specific communicator. A communicator is a software construct that defines a group of processes and a context (tag/identifier) for communication within that group. MPI operations use the rank and communicator context information to decide the target rank within the process group. MPI_COMM_WORLD is a pre-defined communicator that allows for communication between all processes of the job. MPI allows programs to create new communicators that contains a specific set of processes. These communicators are refered to as intra-communicators as they are used for communication within that group of processes.

MPI defines another type of communicator called the inter-communicator. Inter-communicators have a local process group and a remote process group and all communication is always between process in the local group and a process in the remote

group.

The dynamic process interface in MPI-2 allows a process that resides in a specific intra-communicator, to spawn a new set of processes, that are contained in a different intra-communicator. An inter-communicator that comprises of both the intra-communicators is created and facilitates communication between the two process groups. Further, MPI allows us to create a new intra-communicator that includes all the running processes in both the process groups - the parent communicator and the spawned communicator. In the Figure 1, the working of an inter-communicator has been illustrated

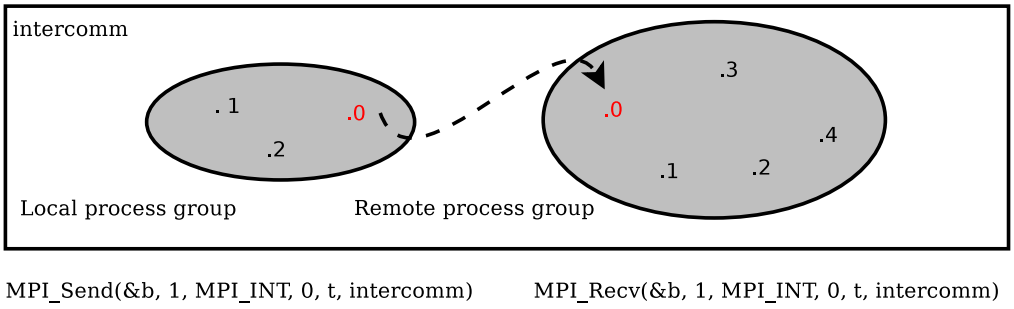

Figure 1: Inter-communicator

2.2 Dynamic Process API

The MPI standard defines three functions to create and join new processes into existing MPI jobs.

- **MPI_Comm_spawn** : This is a collective function called by all the processes in a communicator. When this function is called, the root process of the communicator spawns a set of child processes. All the processes return from the call only when the inter-communicator that connects the existing process group and the newly spawned process group has been created.
- **MPI_Comm_accept/MPI_Comm_connect** : These functions provide a client-server paradigm to facilitate the creation of an inter-communicator between the parent and child process groups.
- **MPI_Comm_join** : Using this function two processes with an existing TCP/IP connection can establish an inter-communicator and start MPI message exchange.

3 InfiniBand

InfiniBand is a popular processor and I/O interconnect that has become popular and is enjoying wide success due to low latency (1.0-3.0μsec), high bandwidth and other features. Over 30% of the Top500 fastest supercomputers show InfiniBand as the interconnect being used. The InfiniBand network device is also referred to as Host Channel Adapter (HCA).

3.1 Communication Model

The InfiniBand communication model uses two queues called the send queue and the receive queue, together called a queue pair (QP). Send and receive work requests (WRs) are posted in these queues and the completion of the request is indicated by putting a completion entry in the completion queue (CQ). Completion can be detected by polling the CQ. InfiniBand also supports an event-based completion model that can be used for asynchronous completion.

3.2 Transport Services

InfiniBand defines four transport modes: Reliable Connection (RC), Unreliable Datagram (UD), Reliable Datagram (RD) and Unreliable Connection (UC). Of these, RC and UD are required to be implemented in any InfiniBand-compliant HCA. RD is not required and is not implemented in any currently available devices.

The RC model is a connected reliable model and the most popular service. An RC QP can be used to communicate with another dedicated RC QP. Thus using RC for *n* peers requires each peer create at least *n-1* QPs to be fully-connected. RC provides most of the InfiniBand features such as RDMA and atomic operations.

The UD transport service is unconnected and unreliable. No message delivery guarantees exist. The main advantage of UD is that a single UD QP can communicate with any other UD QP in the system. They are not explicitly connected as in RC. Instead, to address a message to another QP in the system the Local Identifier (LID) and the QP Number (QPN) can be used. The LID can roughly be thought of as an IP address and QPN a port in InfiniBand terminology. The downsides of UD are that reliability must be taken care of in the application and only a single Maximum Tranfer Unit (MTU) of data (2KB on most HCAs) can be sent at a time. Thus, the software must perform the packetization. Despite these downsides, previous work has shown that MPI applications benefit from UD transports due to lower overheads [12].

4 Collective Communication

In this section, we give a brief overview of the intra and inter-communicator collective communication operations that are defined in the MPI Standard.

The collective communication for intra-communicators have been extensively studied. Profiling information gathered from common applications reveal that a significant amount of application run-time is spent in the collective calls. Since collective operations typically involve multiple processes exchanging messages simultaneously, it is necessary to design algorithms that are fully aware of the communication pattern, system and network characteristics to acheive high performance. InfiniBand offers RDMA support that enable a process to write directly into another process' memory and this feature has been leveraged for intra-communicator collectives [2, 21, 11, 25]. Multi-cores have become ubiquitous and the core density per node is constantly increasing. Owing to the common shared memory regions, such machines allow for faster communication between two processes that reside within the same node, when compared to communication done over the network. This concept has lead to several interesting design options that utilize the shared memory to optimize the intra-node exchange phases of the intra-communicator collective operations [14, 19, 10, 20]. These algorithms have resulted in significant improvements in application

runtime.

The inter-communicator collectives are commonly used in the context of dynamic process management. It is common for processes in the parent communicator to be involved in collective operations with the processes in the spawned communicator and this is done over the inter-communicator that comprises of both the communicator groups. Based on the message exchange pattern, the MPI standard categorizes the inter-communicator collective operations as *All-to-All*, *All-to-One*, *One-to-All* and *Other*. *All-to-One* and *One-to-All* collective operations are typically rooted operations and involve the movement of messages in a specific direction between the two groups of communicators. *All-to-All* operations involve messages being exchanged between processes belonging to both the groups simultaneously. Collective calls that belong to the *Other* category either have a communication pattern that do not fit into the above categories - such as MPI_SCAN or do not involve any explicit movement of messages such as MPI_Barrier.

The inter-communicator collectives involve slightly different message exchange patterns than their intra-communicator counterparts. Collective communication between processes that reside within the same communicator is a very well studied area and several optimal algorithms have already been proposed. However, in the context of collective communication in the dynamic process interface, optimizing the performance of inter-communicator collectives is an important research problem that we have addressed in this paper.

5 Design

In this section we describe our design for the dynamic process management framework. Figure 2 shows the architecture of the MPI-2 dynamic process management. The MPI application uses the API described in Section 2 to spawn new tasks. An MPI

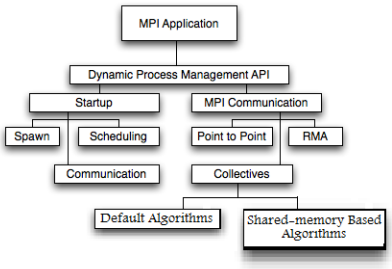

Figure 2: Dynamic Process Management framework

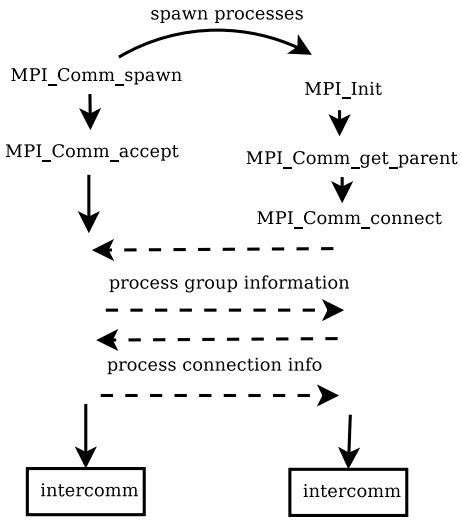

Figure 3: Flowchart of spawn

design has to handle the startup of the new tasks and the three parts of the startup are the spawn phase, scheduling phase and the communication phase.

5.1 Spawn phase

The spawn design requires that the MPI application talk to the job manager. This is accomplished using a common protocol between the dynamic process management API and the job launcher. This phase requires network communication on the management network (usually using TCP/IP). In our designs, we consider two job launchers, the Multi-Purpose Daemon (MPD), which is the default scheme in MPICH2 [16] and *mpirun_rsh*, a MVAPICH2 specific startup manager based on the ScELA [23] architecture. The job launcher interface defines a protocol that is used to propagate the parent's port information, size of the new job, command and arguments.

5.2 Scheduling the tasks

MPI-2 standard does not define a way to do task placement. The task scheduling is performed by the startup agent or a job management system. Scheduling of dynamic tasks requires the job manager to maintain global history of dynamic tasks and place tasks based on this history. Our implementation uses MPD or *mpirun_rsh* to schedule tasks. MPD is discussed in more detail in [3]. *mpirun_rsh* framework is based on the ScELA architecture and is discussed in detail in [23].

Both tools place tasks in a round-robin manner, but suffer from the drawback that multiple spawns are scheduled to the same nodes resulting in imbalance. The studies in [4] have addressed this issue in LAM-MPI by suggesting various task placement mechanisms to maintain load balance. The job launchers schedule the tasks and perform the final *exec* of the processes. Finally, the child processes sychronize with the parent process group to complete the dynamic process launch.

5.3 Communication phase

To design the spawn interface we require the parent to request a spawn and wait to sychronize with the child processes to establish the inter-communicator. The communication phase begins with the child-root of the spawned process group connecting back to the parent to exchange process group information. To establish the inter-communicator the processes need to know the process group ID, the size of the remote process group and the context ID to be used. Additionally, implementations may require a way to identify each remote rank independently to exchange messages. In our design each rank is uniquely identified by their UD queue pair numbers and the LID. This information is exchanged between the root processes and broadcast within their local groups. Figure 3 shows the flow of information required to implement the spawn interface.

5.3.1 Communication methods

Every spawn requests results in the child-root connecting to the parent process to exchange information. An application that spawns tasks frequently will incur the overhead of this connection establishment and communication for every spawn. Thus, to efficiently design the spawn interface we need lightweight connection establishment protocols. As noted in Section 3 there are two different transport modes for InfiniBand: RC and UD, that we can use for this designing this phase.

5.3.2 MPI_Comm_spawn

To perform the spawn, we first create the connection information of the parent that is passed to environment of the spawned children. This is managed via environment variables and propagated by the job manager. The parent process advertises a port in the form of an LID and two UD queue pair numbers. One of the UD queue pair numbers is utilized for the accept/connect interface. The other UD queue pair number is used for RC QP connection establishment [26]. Once the processes are spawned, the parent process waits for the child-root of the remote group to connect back.

5.3.3 MPI_Comm_connect

The spawned process group collectively performs the connect. Only the child-root connects to the parent process, while the other ranks wait for remote group information. We have two possible designs at this point, using RC for message exchange versus using UD.

- UD: If the amount of data to be exchanged with remote root is small then it is more efficient to use a direct UD exchange. In this mode, the child-root sends the process group size, process group ID and context ID for the communicator in a single UD message. The parent-root acknowledges the exchange and sends its process group ID, group size and context ID. Both the ranks broadcast the remote group information within their own MPI_COMM_WORLD. In the next step, both root processes exchange the connection information within their local groups. In our design the connection information consists of the LID and UD QPN. In applications that spawn often and spawn few processes the UD direct exchange model is more scalable and quicker than creating short-lived RC connections.
- RC: If an application spawns large jobs and spawns are infrequent, the connect API uses the second UD QP number to establish an RC connection with the remote root. This connection establishment is according to the algorithm defined in [26]. Following the message exchange, the two root ranks establish a RC connection that is used to exchange process group information.

At the end of the above stage each process has the information required to independently create the inter-communicator to communicate with the remote group. The inter-communicator can now use regular MPI communication using the point-to-point, remote memory access (RMA) or collectives.

5.4 Enhanced inter-communicator point-to-point operations

Multi-core computing has now become ubiquitous and the number of processing units within a compute node is constantly increasing. For most message sizes, the communication between processes that reside within the same compute node is faster than communication between processes that are on different nodes. This is because the processes use the common shared memory for exchanging the messages. However, this approach involves an extra-copy of the message being made in the shared memory region. For larger messages, the overhead introduced by the extra-copy is undesirable. In [9], authors have proposed utilizing a portable zero-copy based, kernel assisted design called LiMIC, for optimizing the intra-node communication. In this

work, we have studied the advantages of using the LiMIC module for optimizing the intra-node communication, in the context of dynamic process management.

5.5 Shared Memory based inter-communicator collectives

In the dynamic process framework, collective operations are performed over processes that are in both the communicator groups - the parent communicator and the spawned communicator. These collective operations are essentially inter-communicator collective operations done on the communicator that comprises of the parent and the spawned communicators. In MVAPICH2, the inter-communicator collective operations are designed on top of the intra-communicator collective calls. Most of the inter-communicator collectives can be viewed as a combination of two communication phases. One phase involves an intra-communicator collective call done locally on either of the two groups. In this phase, the MPI library creates a new intra-communicator from the inter-communicator to include all processes that are local to a particular group, either the parent communicator or the spawned communicator. The intra-communicator collective phase is done in the context of the new intra-communicator. The other phase involves communication between the local roots of the two groups. The inter-communicator MPI_Alltoall operation is one of the exceptions to such a generalization. By default, all inter-communicator collective operations utilize the non-optimized versions of the intra-communicator collectives. This is mainly due to the fact that that new intra-communicator is a single-level communicator that does not consider the topology of the processes. As explained in Section 4, the shared memory based collective algorithms rely on grouping processes that are within the same compute node within a hidden 2-level communicator. In this paper, we have re-designed the inter-communicator collective operations to utilize the optimized shared memory based collective algorithms. In our proposed shared memory based inter-communicator collective framework, the MPI library creates a 2-level intra-communicator from the inter-communicator. This allows the inter-communicator collective operations to make full use of the high performance designs that have been proposed for intra-communicator collective operations.

6 Designing Benchmarks for Dynamic Process Management

To the best of our knowledge, there are currently no metrics or standard applications to benchmark various designs and implementations of MPI-2 dynamic process management. To address this need we design a set of benchmarks that are useful to measure performance of a MPI-2 library. The benchmarks are similar to the existing OSU Benchmark suite [15] released with the MVAPICH/MVAPICH2 software.

6.1 Spawn Latency

The spawn latency benchmark measures the time taken to perform the MPI_Comm_spawn routine. We time the execution of this function in the parent-root process. MPI_Comm_spawn is a collective operation over the spawning communicator and hence the benchmark can vary the size of spawning parent-communicator and the size of spawned child-communicator. The time to spawn is an important metric as it is the measure of the overhead in using dynamic process management. Minimizing

this overhead is vital if dynamic processes are to be used in MPI applications. Due to involvement of system resources and job manager framework, the measured values of the latency has significant variation. The benchmark averages the latency over a large number of runs. The time to disconnect is not considered in the measurement of the latency. The work done by the spawned processes does not affect the measurement as the latency measured is a part of the MPI_Init time on the child group.

6.2 Spawn Rate

The spawn rate benchmark measures the rate at which an implementation is able to perform the MPI_Comm_spawn routine. It is calculated by spawning jobs continuously and finding the rate at which the implementation is able to create new MPI jobs. The benchmark does not consider the time for disconnecting the inter-communicator. Spawn Rate is an important metric as it can estimate the scalability of our design. As the spawn rate benchmark oversubscribes the processors we try to minimize the effect of spawned jobs on the spawn rate by putting the spawned process to sleep until the benchmark is complete. This is required as multiple jobs will be scheduled to the same cores as the benchmark progresses, this being the default behavior of the job launchers considered.

6.3 Inter-communicator Merge Latency

MPI specification provides four intercommunicator operations, including MPI_INTERCOMM_CREATE, MPI_INTERCOMM_MERGE, MPI_COMM_DUP and MPI_COMM_FREE. Since MPI_INTERCOMM_CREATE has been implemented in MPI_Comm_spawn, we choose MPI_INTERCOMM_MERGE funcion as a representive of intercommunicator operations. MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator, which provides users more flexible options on intercommunicator operations, for example, applying appropriate topology on merged intra-communicator. The latency for intercommunicator merging is increased along the number of processes in local and remote process group. In our micro-benchmark, we spawn n processes in remote process group, where n is equal to the number of processes in local process group. To get a reliable result, the benchmark calls the function in a loop. In each loop, the new intra-communicator is first established through MPI_Intercomm_merge function, then it's freed by MPI_Comm_free function. Only the time spent on merge function is accumulated. An average result is calculated in order to get more precise numbers.

6.4 Inter-communicator point-to-point latency

Point-to-point inter-communicator operations involves the movement of data from a local group to a remote group. This inter-group message latency is an important metric as designs may have better optimizations for intra-communicators than inter-communicators. With inter-communicators, message delivery has an additional overhead of mapping from the (local process group, rank) to the (remote process group, rank). In some designs, such as ours, no connections are setup between ranks of the local and remote process groups. Connections are setup on-demand, when the ranks really need to communicate. This connection establishment time can be excluded from measurement by using a warmup loop. Our benchmark calculates an average latency between two processes from different groups but are on the same node.

6.5 Inter-communicator collectives latency

As explained in Section 4, the time taken spent in the collective operations significantly impacts the overall performance of the applications. Applications that utilize the dynamic process management features perform collective operations that are done over an inter-communicator that comprises of the parent and the spawned communicators. In Section 5, we proposed an inter-communicator collective framework that utilizes the optimized shared memory based collective algorithms. It is necessary to design micro-benchmarks that measure the performance of the various inter-communicator collective algorithms. To the best of our knowledge, there is no benchmark suite that is also designed to measure the performance of inter-communicator collective operations. In this paper, we have designed a benchmark suite to measure the performance of a few of the inter-communicator collective operations including MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Reduce_Scatter, MPI_Allgather, MPI_Alltoall and MPI_Barrier. For each of these operations, we have designed the benchmarks to adhere to the inter-communicator collective specifications defined in the MPI-2 Standard. Since the connections between processes are set up only when the they first start sending messages to each other, it is essential to design the benchmarks to amortize the cost of the connection setup phase. In our benchmarks, for each message size, we call the inter-communicator collective operations several times and collect the average time each process takes to complete one collective operation. For *rooted* collective calls, it is also important to measure the minimum and maximum amount of time spent within the collective call, across the set of all the processes in the inter-communicator. This is because of the assymetric nature of the message exchanges for such collective calls. In MVAPICH2, we have shared memory based collective algorithms for MPI_Bcast, MPI_Reduce, MPI_Allreduce and MPI_Barrier. In the following section we have demonstrated the benefits of using these algorithms for inter-communicator collectives in the context of communication in a dynamic environment. However, owing to a limitation in our shared memory design, at this point, we are limited to having only one process in the parent communicator for the benchmarks involving the inter-communicator counterparts of these operations. However, we believe that the results are still representative of the overall performance and can be used to infer the benefits of using shared memory based designs for inter-communicator collective operations. For the final version of the paper, we intend to have this minor issue resolved to allow for a broader set of results.

7 Performance Evaluation

7.1 Experimental Platform

We use a 64-node Xeon cluster with each node having 8 cores and 6 GB RAM. The nodes are equipped with InfiniBand DDR HCAs and 1 GigE NICs. All the nodes are connected to a single file server accessed by the NFS (Network File System) protocol. We present results using a 64x8 layout, which uses all 512 cores, with cyclic allocation of ranks. We also present a result with block allocation of ranks. Our designs were implemented in the MVAPICH2 library. We evaluate our design in MVAPICH2 as well as OpenMPI, another popular MPI library.

7.2 Spawn Latency

Figure 4 (a) shows the results of running the spawn latency benchmark. The latency experiments are run on nodes without any busy tasks to yield the lowest possible results. We present five results in the graph, *mvapich2-MPD-RC*: which uses only RC connections and MPD for startup, *mvapich2-mpirun_rsh*-RC: which uses RC connections and *mpirun_rsh* for startup, *mvapich2-MPD-UD*, which uses UD for initial information exchange and MPD for startup and *mvapich2-mpirun_rsh*-UD which uses UD for initial information exchange, *mpirun_rsh* for startup and *OpenMPI*: which shows the latency results for the OpenMPI library. As seen in Figure 4, the RC and UD implementations perform almost equally when MPD is used for very small job sizes. For job size of 32 and beyond the UD design shows a slight benefit. With *mpirun_rsh* we see that the UD design provides a lower spawn latency. The *mvapich2-mpirun_rsh-RC* and *OpenMPI* perform similarly (up to 128 processors) as both use a connection based startup model with similar job launch mechanism. However, for 512 processes, *mvapich2-mpirun_rsh* designs perform better than OpenMPI. On the job startup angle, we find the MPD startup mechanism is faster than *mpirun_rsh* for small job size, however for larger jobs *mpirun_rsh* is more scalable. This is due to the fact that MPD maintains a ring-of-daemons on all nodes, spawning a new job on a node just requires a TCP/IP message to be sent to the daemon. The MPD mechanism, however has higher startup latency as number of ranks grows. *mpirun_rsh* framework is a daemon-less startup manager based on ScELA architecture[23]. It incurs higher overhead for small job launches, but it is highly scalable and provides very low latency for higher jobs sizes.

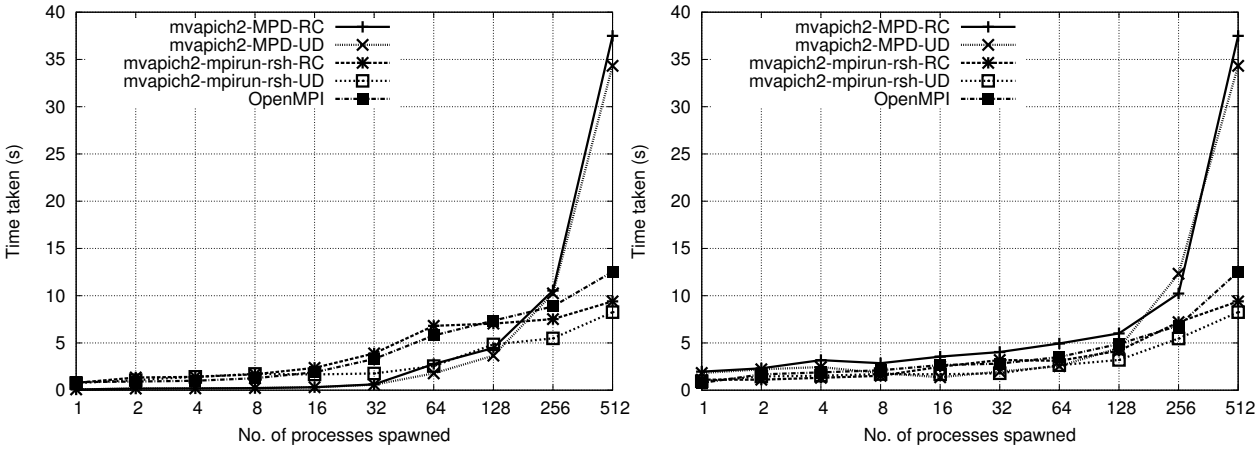

Figure 4: Latency - 512 cores: (a) Cyclic rank allocation and (b) Block rank allocation

The second set of results we present in Figure 4 (b) are the the spawn latency with block allocation of ranks. This is an important result as it shows the effect of HCA contention on the spawn time. As seen in the graph, when there are multiple jobs per node, the UD spawn design performs better than the RC design, as the UD model has lesser startup overhead. The UD design is more relevant here as job allocation is generally block distributed. The UD design is simpler and lightweight. OpenMPI performs very similar to *mvapich2-mpirun_rsh-RC* in this benchmark for up to 256 processes. However, for 512 processes *mvapich2-mpirun_rsh* designs perform the best.

7.3 Spawn Rate

The spawn rate benchmark is evaluated with 16-nodes of the cluster, for a total of 128 cores. The benchmark measures the rate of sustained spawn supported by our design. The reported value is the number of spawns/second with increasing job sizes. Figure 5 shows the results of the benchmark running on our design.

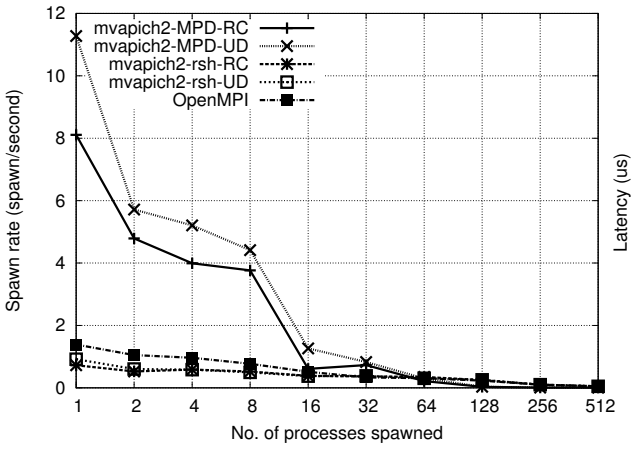

Figure 5: Spawn rate

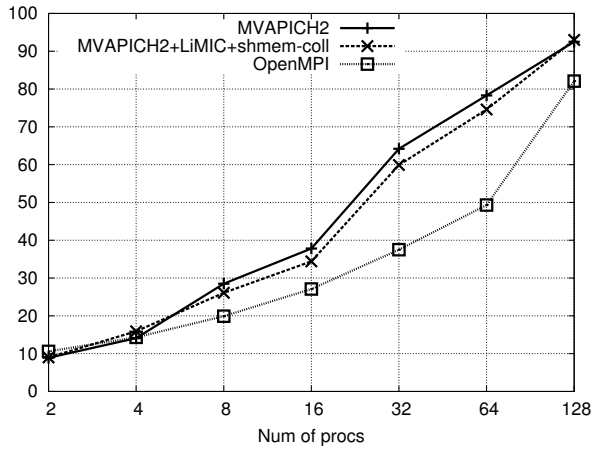

Figure 6: Inter-communicator merge latency

We see that the UD design using MPD job manager provides the best spawn rate. The relatively higher cost of creating and destroying RC queue pairs leads to a slower spawn rate with RC. As we have seen *mpirun_rsh* startup has a higher initial overhead and results in a lower spawn rate, however it scales very well and maintains a steady spawn rate with increasing job size. OpenMPI performs similar to *mpirun_rsh* and has a low spawn rate for small jobs. Only mvapich2-MPD designs are able to provide a high spawn rate for small jobs. The spawn rate is an important metric to consider when designing an MPI application with frequent job spawns. The benchmark clearly shows that to have a high spawn rate we need a low-overhead connection mode (like UD) and an MPD-like startup framework.

7.4 Inter-communicator Merge Latency

The inter-communicator merge latency benchmark involves up to 16 nodes with 128 cores. On each node, half of the cores are occupied by processes from local progress group while the other half of cores are used by spawned processes. Then the benchmark creates a new intra-communicator which includes all the processes in both local and remote process groups. In our mvapich2 implementation, we use point-to-point functions to exchange information that is used to determine which group is ordered first in the generated communicator, while in OpenMPI, this is done by allgather operation. The difference in the algorithm might result in the better performance of OpenMPI on merge latency, according to Fig 6. Also after the new intra-communicater established, an intra-communicator broadcast is called in order to update local information, where the shared memory collective design helps to reduce the latency slightly. We will have more insight in inter-communicator collective operations in Section 7.6.

7.5 Inter-group Latency

The inter-group latency is a basic latency test to measure the difference between intra-communicator latency and inter-communicator latency.

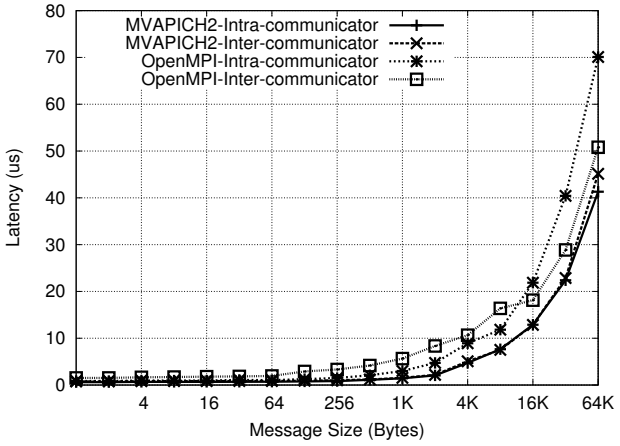

Figure 7: Inter-group vs Intra-group latency

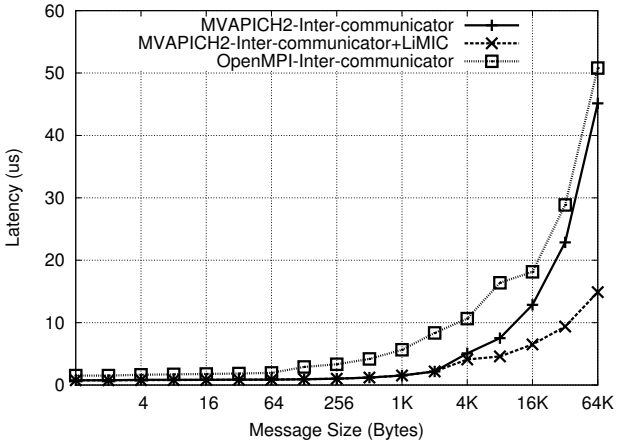

Figure 8: Inter-group latency

In Figure 7 we compare the latency for inter-communicator and intra communicator point-to-point operations for default MVAPICH2 and OpenMPI. For the default MVAPICH2 version, the inter-communicator latency is slightly higher than the intra communicator exchange latency. This is due to searching of process group and managing the translation from local group to remote group. In OpenMPI, intra-communicator latency is better than inter-communicator latency for messages smaller than 16K. However, for larger messages, inter-communicator point-to-point operations shows better results.

In Figure 8, we compare the inter-communicator latency across OpenMPI, the default version of MVAPICH2 and our proposed design that utilizes the LiMIC module. We can see that our proposed design performs significantly better than either of the other two inter-communicator versions. The performance improvement is about 67% for 64KB message size.

7.6 Inter-communicator collectives latency

7.6.1 MPI_Alltoall

In an MPI_AlltoAll operation done over an intra-communicator, each process sends a distinct message to every other process and receives a distinct message from every other process. In case of an inter-communicator, each process in one group sends a distinct message to every process in the other group and vice versa and there is no exchange of data between processes within a single intra-communicator. Since MVAPICH2 does not have a shared memory based algorithm for MPI_Alltoall, in our benchmark, we have equal number of processes in the parent process group and the the dynamically spawned process group. As seen in Fig 9, the current implementation of MVAPICH2 performs better than OpenMPI for all the message sizes. It is imporant to observe that the current algorithm for inter-communicator MPI_Alltoall uses a pair-wise exchange scheme for all messages which results in poor performance for small messages. However, utilizing the zero-copy based mechanism for point-to-point calls, does not lead to any gains for this collective operation. This can be attributed to the communication pattern of

the pair-wise exchange scheme. In a given iteration, a process might be involved in message exchange between two processes, one of them is within the same node and the other is in a different node. The time required to complete this iteration depends on the inter-node communication time, irrespective of how quickly the intra-node operation completes.

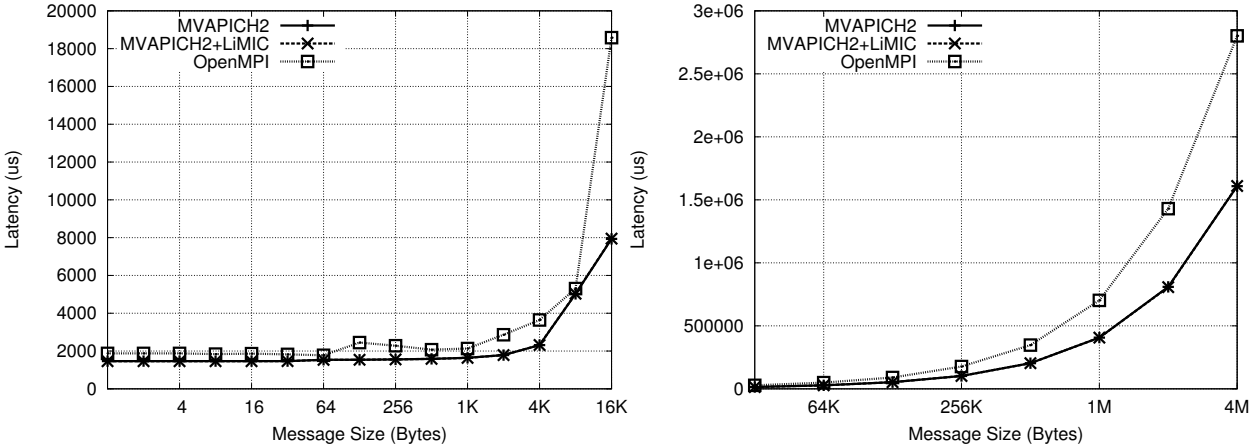

Figure 9: Inter-communicator Alltoall Latency :(a) Small Messages and (b) Large Messages

7.6.2 MPI_Allreduce

This benchmark measures the time taken for an inter-communicator MPI_Allreduce operation to complete. In an inter-communicator MPI_Allreduce operation between two communicators A and B, the result of the reduction of the data provided by processes in group B will be stored at all the processe in group A and vice versa. Since MVAPICH2 offers an effecient shared memory based algorithm for MPI_Allreduce, we have explored the performance implications of having a shared-memory enabled inter-communication MPI_Allreduce operation. However, we are constrained to have one process in the parent communicator. In this test the parent processes spawns 63 child processes uniformly across 8 nodes. From the results in Figure 10 we see that the current version of MVAPICH2 consistently shows lower latencies when compared to OpenMPI. We can also see performance benefits of using shared memory based inter-communicator MPI_Allreduce algorithm along with the kernel-based zero copy mechanism being used for all intra-node large message exchanges. The improvement is about 36% at 512KB message size.

7.6.3 MPI_Bcast

In an inter-communicator broadcast, one of the groups defines the root and data is sent from the root to all the processes in the other group. In MVAPICH2, we perform this operation by having one point-to-point exchange between the root of the parent communicator and the child-root of the spawned communicator. The child-root then does an intra-communicator MPI_Bcast. In our proposed design, the intra-communicator MPI_Bcast phase uses the effecient shared memory broadcast algorithm. Our benchmark involves one process in the parent communicator and it spawns 63 processes and performs MPI_Bcast over this inter-communicator. The results in Figure 10(a) show that MVAPICH2 performs comparably or better than OpenMPI for

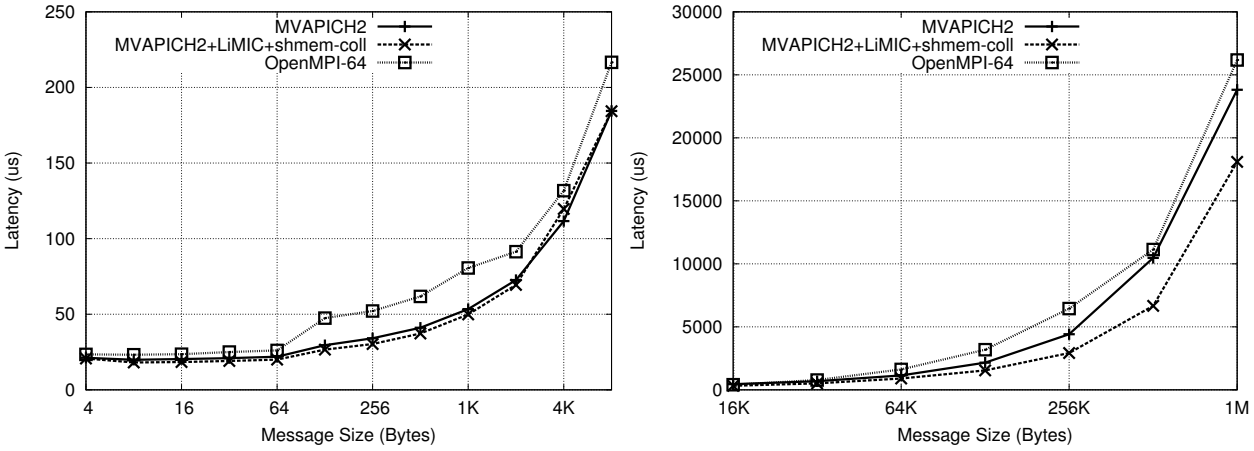

Figure 10: Allreduce Latency for 63 Spawned Processes: (a) Small Messages and (b) Large Messages

messages sizes upto 8KBytes. For 16KByte messages OpenMPI performs better. From Figure 10(b) we see that MVAPICH2 consistently performs better than OpenMPI for large message sizes. The proposed schemes show notable effect on performance for message sizes beyond 16KBytes. We see a 56% improvement in latency for 512KB message size and a 22% improvement for 2MB message size.

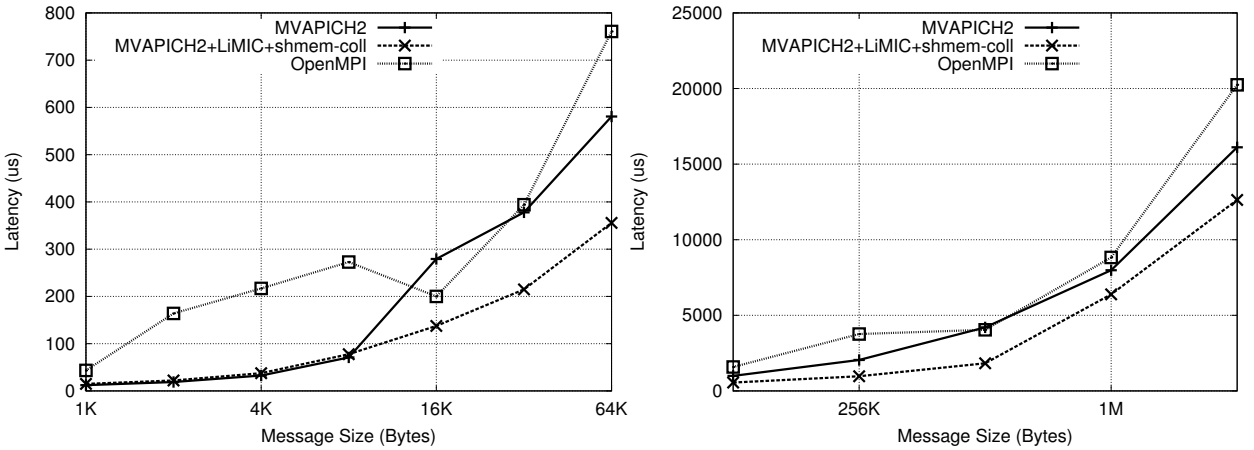

Figure 11: Broadcast Latency for 63 Spawned Processes: (a) Small Messages and (b) Large Messages

7.6.4 MPI_Reduce

The inter-communicator MPI_Reduce operation involves all the processes specifying the root of the operation. Suppose the root is in the intra-communicator A, the data from all the processes in the intra-communicator B is ultimately accumulated at the root. In MVAPICH2, this operation involves a local intra-communicator MPI_Reduce performed on the remote communicator and this data is sent to the root through a point-to-point call. MVAPICH2 supports an optimized shared memory based algorithm for MPI_Reduce. In our proposed design, we have used the optimized intra-node MPI_Reduce algorithm for the reduction phase on the remote communicator. We also observed that process skew that originated during back-to-back calls to MPI_Reduce resulted

in widely varying timing results. To address this issue, in our benchmark, we call MPI_Barrier, before each call to MPI_Reduce. However, the time spent in MPI_Barrier is not considered for our analysis. In Figure 12, we can see that for medium and larger messages, our proposed framework delivers better performance than OpenMPI and the default inter-communicator schemes in MVAPICH2. Compared to the default MVAPICH2 performance the proposed framework has a 30% improvement at 512KB message size.

7.6.5 MPI_Barrier

Finally, we also evaluate the performance of the inter-communicator MPI_Barrier operation. In MVAPICH2, the inter-communicator barrier operation has been designed on top of a pair of inter-communicator broadcast operations. As explained earlier, the inter-communicator broadcast operation has been designed on top of intra-communicator MPI_Bcast calls. Since the MPI_Barrier involves the movement of a small message, we use the point-to-point based binomial algorithm during the broadcast operation. Owing to this, we do not expect to see any performance improvements in our proposed inter-communicator framework. We noticed that the inter-communicator MPI_Barrier operation has been implemented on top of a call to inter-communicator MPI_Allreduce, in OpenMPI. In Table **??**, we can see that MPI_Barrier designs used in OpenMPI provide lower latencies.

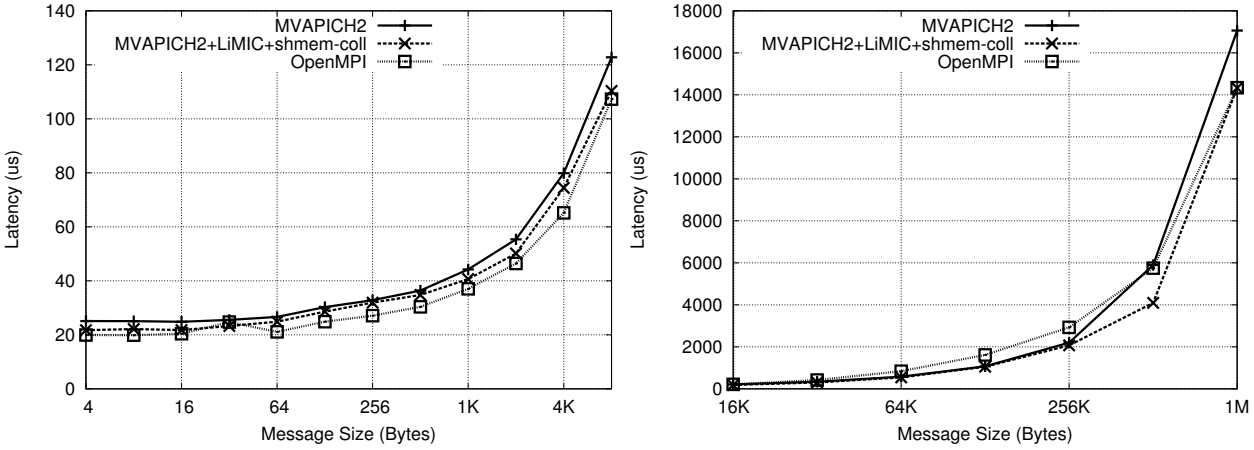

Figure 12: Reduce latency for 63 spawned processes: (a) Small Messages and (b) Large Messages

8 Application (Distributed Rendering) Evaluation

Graphics rendering is a highly parallelizable activity. Distributed rendering works by distributing each frame to be rendered to the compute nodes of a cluster. A frame can usually be rendered independently of other frames and the only communications involved are the initial frame data distribution and final collection of rendered images. Rendering can be programmed easily using a master-slave model. Render farms are common in Computer Graphics Imagery (CGI) industry, with the farms hosting several render servers that can be used by clients.

To demonstrate the feasibility and real-world application of the dynamic process interface we designed a dynamic process version of POV-Ray, a popular, open-source ray-tracing application. Using our design, a graphics programmer can decide at

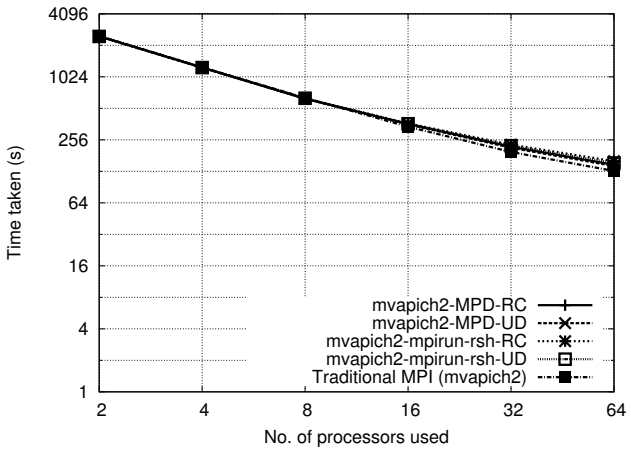

Figure 13: Parallel POV-Ray evaluation

execution time the optimal number of compute nodes required for the job and spawn the rendering on the nodes. There have been MPI parallelization efforts on POV-Ray [6], but these implementations use a static runtime environment. The dynamic process interface can be programmed to have a changing environment in which we can expand or contract the available slave resources. This is similar in concept to a render farm and this paradigm can be programmed using the MPI-2 interface. We present our evaluation of the POV-Ray in the following paragraph.

We implemented a parallel version of POV-Ray to use the MPI-2 dynamic process interface. However as of now, all message transfers are based on point-to-point operations. So, we do not expect to see any improvements through our proposed collective framework. The timing data for the application with our proposed framework that uses LiMIC could not be obtained for this version of the paper. We compare the results of using our RC design, UD design and traditional static runtime parallel POV-Ray. For our evaluation we render a 3000x3000 *glass chess board* with global illumination. Figure 13 shows the results of our evaluation.

As seen in the graph, the dynamic process framework adds very little overhead to the overall execution of the application. Until 32 processors the speedup factor is almost the same for all three designs. Beyond 32 processors, the cost of startup and parallelization starts to accumulate and the dynamic version incurs some slowdown.

Evaluating real-world problems clearly shows the feasibility of the dynamic process framework. Moreover, using dynamic processes gives more control to the application programmer who can intelligently decide the parallelization factor and placement of the jobs at run-time. Additionally, using the dynamic process framework applications can dynamically change size and scale of the application which is a key benefit.

9 Related Work

The architecture of a dynamic process creation framework for MPI was described by Gropp and Lusk [8]. The MPI-2 standard [7] defined the process creation and management interface. The standard defined only the process creation interface leaving the job scheduling to the MPI implementation. Gabriel et al [5] evaluated the dynamic process interfaces of several MPI-2

implementations. They measure the bandwidth achieved in point-to-point message exchange over traditional versus dynamic communicators, however they do not provide any mechanism for implementing the interfaces.

Marcia Cera et al [4] have explored the issue of improving scheduling of dynamic processes. Their solutions are aimed at load balancing jobs across nodes of a cluster and providing novel ways of scheduling dynamic processes across a cluster.

Several researchers have explored using dynamic processes for fault-tolerance in MPI applications [13]. Kim et al. [22] explored the design and implementation of dynamic process management for grid-enabled MPICH. However, their work did not explore the design of the MPI-2 dynamic process interface, but implemented a new MPI interface *MPI_Rejoin* that allows processes to join existing process groups.

Several designs that utilize shared memory and RDMA features to acheive high performance for intra-communicator collective operations have been proposed. Some of the notable works include [14, 19, 10, 2, 21, 11, 20, 25]. Effecient intra-node point-to-point designs have been proposed in [9].

10 Conclusions

With increasing popularity of multi-core processors and commodity clusters, MPI has become the dominant parallel programming model. However, several large applications have traditionally used the master/slave programming model. The MPI-2 dynamic process interface can be used in the master/slave model. Additionally, MPI-2 dynamic process primitives provide a client-server API as well. In this paper we have addressed the design perspective of an efficient dynamic process interface.

MPI Communication operations over intra-communicators are well studied and optimized but optimizing point-to-point and collective communication operations over inter-communicators is an important issue that needs to be addressed in the context of dynamic process interface. In this paper, we have proposed a new framweork for inter-communicator point-to-point and collective operations that leverage the use of kernel based zero copy intra-node message transfers and effecient shared memory based collective algorithms to achieve high performance. We implemented our designs and evaluated them on MVAPICH2, a popular MPI implementation for InfiniBand. We have also designed a new benchmark suite to evaluate our designs and compare them with the dynamic process framework available in Open-MPI. Our study draws the following conclusions on designing and evaluating the dynamic process framework and inter-communicator communication.

- An MPD-like daemon based startup model is required for supporting frequent task spawning. The spawn rate benchmark clearly shows the superiority of the daemon-based startup model. But at the same time MPD suffers from very high latency for large job sizes. For very large job launches, the ScELA [23] architecture has proved to be highly scalable and reliable. Thus, *mpirun_rsh* based startup models are required for managing large jobs.

- Lightweight communication primitives are better for the task startup phase. The benchmarks show the advantage of using a UD model for InfiniBand. Similar lightweight transport schemes (such as UDP) should apply in other environments (such as 10GigE). Ideally, we need a hybrid design that can switch from low-overhead mode to a higher-overhead high-performance mode based on job sizes. Our current designs do not perform this switch and this is a area of future work we plan to explore.

- Evaluation of our designs shows that Kernel-based zero copy mechanism (LiMIC) and shared-memory based collective algorithms can be exploited to achieve considerable performance gains in inter-communicator point-to-point and collective communciation operations in the dynamic process management framework.
- MPI applications don't incur heavy overhead in using the dynamic process framework. The evaluation of the ray-tracing application clearly demonstrates the feasibility of the dynamic process paradigm with the benefits of dynamically growing or shrinking jobs.

In the future we hope to explore designing applications with non-static job sizes using the MPI-2 inter-communicator merge operations. We also hope to explore the job scheduling for dynamic tasks in more detail.

11 Software Distribution

Currently, in MVAPICH2, we use mpirun_rsh and mpd for spawning jobs in the dynamic process management interface. For establishing connections between processes and for transfering messages between processes, we use the RC transport mode offered by InfiniBand. In the future, We plan to incorporate the designs for UD based connection establishment, enhanced inter-communicator point-to-point and collective communication operations into our next releases . The new benchmarks designed for evaluating dynamic process management interface of MPI-2 libraries will be integrated with the standard OSU benchmarks [15] and made available to the community in the near future.

Acknowledgments

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, IBM, Appro, QLogic, and Sun Microsystems. We would like to thank Hari Subramoni, whose help has been invaluable in bringing out this paper.

References

[1] TOP 500 Supercomputer Sites. http://www.top500.org.

[2] H.-W. Jin A. Mamidala, L. Chai and D. K. Panda. Efficient smp-aware mpi-level broadcast over infiniband's hardware multicast. In *IPDPS*. IEEE, 2006.

[3] Ralph Butler, William Gropp, and Ewing Lusk. Components and Interfaces of a Process Management System for Parallel Programs. *Parallel Computing*, 27:2001, 2001.

[4] Márcia C. Cera, Guilherme P. Pezzi, Elton N. Mathias, Nicolas Maillard, and Philippe Olivier Alexandre Navaux. Improving The Dynamic Creation of Processes in MPI-2. In *PVM/MPI*, pages 247–255, 2006.

[5] Graham E. Fagg Edgar Gabriel and Jack J. Dongarra. Evaluating the Performance of MPI-2 Dynamic Communicators and One-Sided Communication. In *Lecture Notes in Computer Science*, pages 88–97, 2003.

[6] Alessandro Fava, Emanuele Fava, and Massimo Bertozzi. MPIPOV: A Parallel Implementation of POV-Ray Based on MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 426–433, London, UK, 1999. Springer-Verlag.

[7] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing*, 1993.

[8] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 530, Washington, DC, USA, 1995. IEEE Computer Society.

[9] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems. In *IEEE International Conference on Cluster Computing*, 2007.

[10] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda. Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters. In *IPDPS*. IEEE, 2009.

[11] Sushmitha P. Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff, Dhabaleswar K. Panda, Dhabaleswar K. P, S. P. Kini, J. Liu, J. Wu, and P. Wyckoff. Fast and scalable barrier using rdma and multicast mechanisms for infiniband-based clusters. In *In EuroPVM/MPI*, pages 369–378, 2003.

[12] M. Koop, T. Jones, and D. K. Panda. MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008.

[13] Ewing Lusk. Fault Tolerance in MPI Programs. *Special issue of the Journal High Performance Computing Applications*, 18:363–372, 2002.

[14] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid), Lyon, France*, pages 130–137.

[15] OSU Microbenchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[16] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[17] OpenMPI. http://www.open-mpi.org/.

[18] MVAPICH2: High Performance MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

[19] Amith R. Mamidala Rahul Kumar and Dhabaleswar K. Panda. Scaling alltoall collective on multi-core systems. In *IPDPS*, pages 1–8. IEEE, 2008.

[20] Richard L. Graham and Galen M. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *EuroPVM/MPI*, 2008.

[21] A. Mamidala H.-W. Jin S. Sur, U. Bondhugula and D. K. Panda. High performance rdma based all-to-all broadcast for infiniband clusters. In *HiPC*. IEEE, 2005.

[22] Young Yeom Taesoon Park Sangbum Kim, Namyoon Woo and Hyoung-Woo Park. Design and Implementation of Dynamic Process Management for Grid-Enabled MPICH. In *PVM/MPI*, pages 653–656, 2003.

[23] J. Sridhar, M. Koop, J. Perkins, and D. K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *International Conference in High Performance Computing (HiPC08)*, December 2008.

[24] Angela Violi and Gregory A. Voth. A Multi-scale Computational Approach for Nanoparticle Growth in Combustion Environments. In *HPCC*, pages 938–947, 2005.

[25] Qian Ying and Afsahi Ahmad. Efficient shared memory and rdma based collectives on multi-rail qsnetii smp clusters. volume 11, pages 341–354, December 2008.

[26] Weikuan Yu, Qi Gao, and D.K. Panda. Adaptive connection management for scalable MPI over InfiniBand. *Parallel and Distributed Processing Symposium, International*, 0:81, 2006.