Designing and Evaluating MPI-2 Dynamic Process Management Support for InfiniBand

Tejus Gangadharappa, Matthew Koop, Dhabaleswar K. Panda
Department of Computer Science and Engineering, The Ohio State University
{gangadha, koop, panda}@cse.ohio-state.edu

Abstract

Dynamic process management is a feature of MPI-2 that allows an MPI process to create new processes and manage communication with these processes. The dynamic creation of processes allows application writers to develop multiscale applications or master/worker based programs. Although several MPI implementations support this feature we are not aware of any studies on the issues in designing the dynamic process management interface and benchmarking of dynamic process framework. In this paper we design a MPI-2 dynamic process management interface over InfiniBand. We consider two alternative designs using Unreliable Datagram (UD) and Reliable Connection (RC) transport modes of InfiniBand with two job startup models. In our evaluations we found that having an UD based-design allows for much higher spawns rates with existing job launch frameworks. We also design a set of micro-benchmarks to evaluate the performance of our design and other MPI libraries. Finally, we provide an evaluation of the dynamic process framework using a re-designed ray-tracing application.

Keywords: *MPI-2, Dynamic Process Management, InfiniBand*

I. Introduction

Large-scale deployments of clusters continue to achieve new heights in performance and scale. Two major components that affect performance are the interconnect between compute nodes and the application programming interface and implementation.

The Message Passing Interface (MPI) is currently the most dominant model for programming parallel computers today. The MPI standard was first defined in 1994 and was designed to include the attractive features of existing message passing systems such as PVM [6]. The MPI specification defined a standard interface for communication, providing both point-to-point and collective communication primitives in a static runtime environment.

The static model of MPI-1 means the number of tasks is fixed at job launch time. This restriction restricts the application ability to spawn 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 support. This allows MPI applications to create and communicate with new processes, thus providing a new paradigm for programming MPI applications. Unfortunately, unlike many other MPI operations, there are no standard mechanisms for determining the performance of implementations of dynamic process management.

Several popular MPI implementations (OpenMPI, MPICH2) currently support dynamic processes, however there had not been any investigation on efficient design of the MPI dynamic process interface and the benchmarking of the dynamic process interface. Our work explores the design issues of the dynamic process interface, introduces new microbenchmarks to evaluate performance of the dynamic process interface and finally presents an evaluation of our design on InfiniBand.

In this paper we design MPI-2 dynamic process support for MPI over InfiniBand. Unlike traditional Ethernet models, additional setup requirements are required for InfiniBand. Efficient design of dynamic process management over InfiniBand is important as it is a very popular interconnect in commodity clusters. Our designs were implemented in MVAPICH2: [13] a MPI library for InfiniBand and iWARP. We also address the lack of standardized metrics for determining the performance of dynamic process management by proposing a new set of benchmarks. We evaluate our design and the design of the OpenMPI dynamic management framework using these benchmarks. Dynamic process management is being used in grid application design and multi-scale applications[15]. To model a real-world application, we evaluate a ray-tracing application that was re-designed to use the dynamic process model versus the traditional MPI ray-tracing application.

The rest of the paper is organized as follows: In Section II we present an overview of the dynamic process management interface. An introduction to InfiniBand and its capabilities are presented in Section III. Section IV presents the various issues involved with designing a high-performance dynamic process management solution. In Section V we propose a number of new benchmarks to evaluate the performance of dynamic process management implementations. In Section VI provides an evaluation of the various design options proposed using benchmarks and Section VII provides an application evaluation. Section VIII cites work related to dynamic process management and

finally Section IX provides conclusions and offers future work in this area.

II. 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.

A. MPI Communicators

An MPI process is described by a (rank, process group) pair. An MPI communicator encapsulates the ranks and the process group for which the ranks are described. All MPI communications are described in the context of some communicator. A communicator is a software construct that defines a group of processes and an 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. The 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 to address a specific sets of processes. The above type of communicator is called intra-communicator: intra because they handle communication within the process group.

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 a process in the local group and a process in the remote group. Figure 1 illustrates the working of an inter-communicator. The data is being sent from rank 0 on the left group to the rank 0 on the right group.

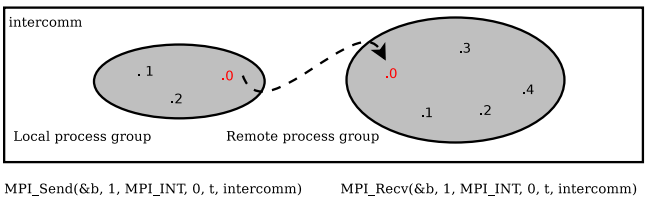

Fig. 1. Inter-communicator

The dynamic process interface in MPI-2 uses inter-communicators to connect together two existing intra-communicators and facilitate communication between the local and remote process groups. This allows an existing MPI application with a local process group to spawn a new set of processes; which now forms the remote group. The local and remote groups form an inter-communicator and both the groups can now exchange messages. Further, MPI allows us to create a new intra-communicator that includes all the running processes (local and remote).

B. Dynamic Process API

The MPI standard defines three ways of creating or joining new processes into existing MPI jobs. In our description of the interface we call the spawning process the parent-root and the root of the spawns as the child-root.

- **MPI_Comm_spawn**

```
int MPI_Comm_spawn(char *command, char *argv[],
     int maxprocs, MPI_Info info,
     int root, MPI_Comm comm, MPI_Comm *intercomm,
     int array_of_errcodes[])
```

  The function starts *maxprocs* copies of *command* process. The function is collective over the communicator *comm*, i.e. the function does not complete until all processes in the communicator have created the inter-communicator and the *root* process performs the role of the parent-root. The newly created set of processes form an MPI_COMM_WORLD of their own. The root of the spawned process group uses the function *MPI_Comm_get_parent* to discover if it was spawned from an existing MPI process. The API uses the accept/connect interface between the parent-root and child-root to exchange rank and process group information. The function returns *intercomm*, an inter-communicator which contains the spawns in the remote group. The MPI application can now exchange messages with the newly created processes using *intercomm*. The MPI standard does not specify where and how the processes were started and leaves it to the job scheduling infrastructure to manage. It only provides a information structure *info* to propagate any hints to the job scheduling framework.

- **MPI_Comm_accept / MPI_Comm_connect:**

```
int MPI_Comm_accept(char *port_name,
     MPI_Info info, int root,
     MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_connect(char *port_name,
     MPI_Info info, int root,
     MPI_Comm comm, MPI_Comm *newcomm)
```

  This API facilitates a client-server computing model to MPI processes with the server process using the *MPI_Comm_accept* to wait for incoming connection on *port_name*. The client uses *MPI_Comm_connect* to connect to the *port_name*. *port_name* is an implementation and interconnect specific string that identifies a process. The resulting inter-communicator *intercomm* now allows the client process group to exchange messages with the server process group. As with MPI_Comm_spawn, the accept/connect calls are collective over the communicator *comm* and the *root* act as the root ranks in the connection establishment. Once the connection is created, both process groups can communicate with the remote groups using the inter-communicator.

- **MPI_Comm_Join:**

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

  Using this interface, two processes with an existing TCP/IP connection described by the socket *sockfd* establish an inter-communicator and start MPI message exchange. The inter-communicator describes a

singleton local group and a remote group in this case. The socket is used to exchange MPI port information, followed by an MPI connection creation using the accept/connect interface. The socket is never used for MPI communication.

III. 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 25% of the Top500 [1] fastest supercomputers show InfiniBand as the interconnect being used. The InfiniBand network device is also referred to as Host Channel Adapter (HCA).

A. 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.

B. 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 [9].

IV. 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

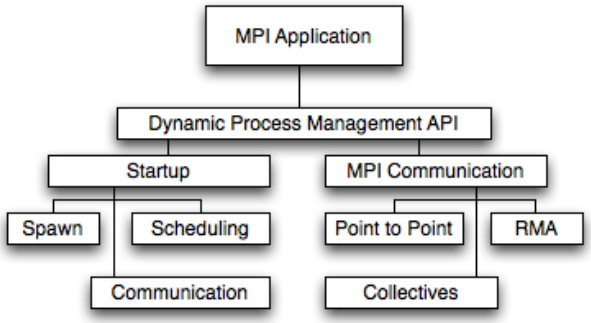

Fig. 2. Dynamic Process Management framework

MPI application uses the API described in Section II to spawn new tasks. An MPI 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.

A. Spawn phase

The spawn design requires 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 [12] and *mpirun_rsh*, a MVAPICH2 specific startup manager based on the ScELA [14] 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.

B. 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 [2]. *mpirun_rsh* is based on the ScELA architecture and is discussed in detail in [14]

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 [3] 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. Once the child processes they sychronize with the parent process group to complete the dynamic process launch.

C. 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.

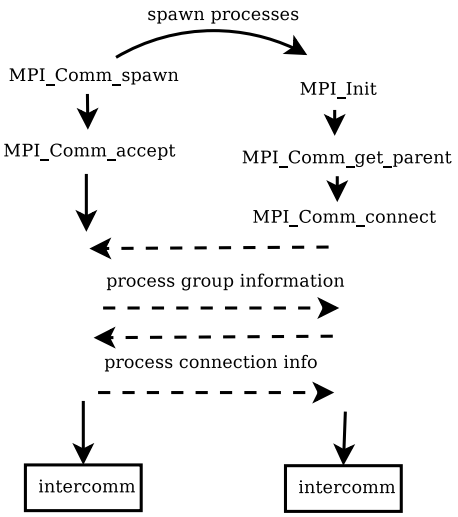

Fig. 3. Flowchart of spawn

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 and as noted in Section III there are different transport modes for InfiniBand that we can use for this designing this phase:

- Reliable Connection (RC): Reliable but connection-oriented. There is significant overhead to communicate with a new process.
- Unreliable Datagram (UD): Unreliable and connection-less. There is a very low overhead to communicate with a new process. Software must perform segmentation of messages over the MTU (often 2KB).

If the amount of data to be exchanged between the local and remote roots is small then using UD provides benefits. Since the data size is small providing segmentation is cheap and there is no connection overhead. As the number of spawned process in a group goes up, the data size will increase. In this case using the RC may provide a benefit.

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 [16]. Once the processes are spawned, the parent process waits for the child-root of the remote group to connect back.

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 [16]. 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.

V. 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 [11] released with the MVAPICH/MVAPICH2 software.

A. 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.

B. 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 of 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.

C. Inter-communicator point-to-point latency

Sending point-to-point MPI data across an inter-communicator requires us to send 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. The benchmark thus measures the effective latency due to the connection establishment and the data transfer. The inter-group latency calculates an average latency for a range of data sizes, between two ranks. Other inter-communicator operations like merge and collectives are not considered here and left as a part of our future work.

VI. Benchmark Evaluation

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 MVAPICH-2 library. We evaluate our design in MVAPICH2 as well as OpenMPI, another popular MPI library.

A. Spawn Latency

Figure 4 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 a 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. MPD, however has higher startup latency as number of ranks grows. *mpirun_rsh* is a daemon-less startup manager based

on ScELA architecture[14]. It incurs higher overhead for small job launches, but it is highly scalable and provides very low latency for higher job sizes.

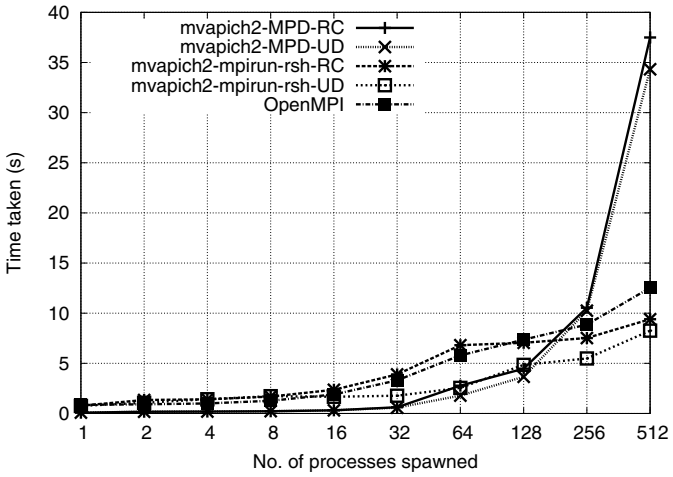

Fig. 4. Latency- 512 cores: Cyclic rank allocation

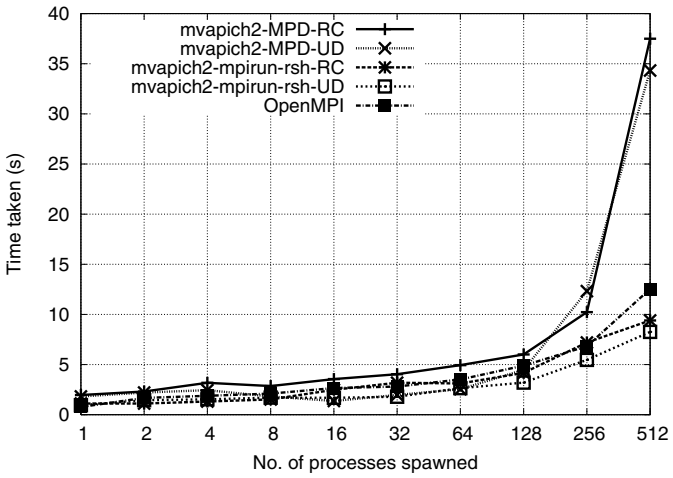

Fig. 5. Latency - 512 cores: Block rank allocation

The second set of results we present in Figure 5 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.

B. 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 6 shows the results of the benchmark running on our design. We see that the UD

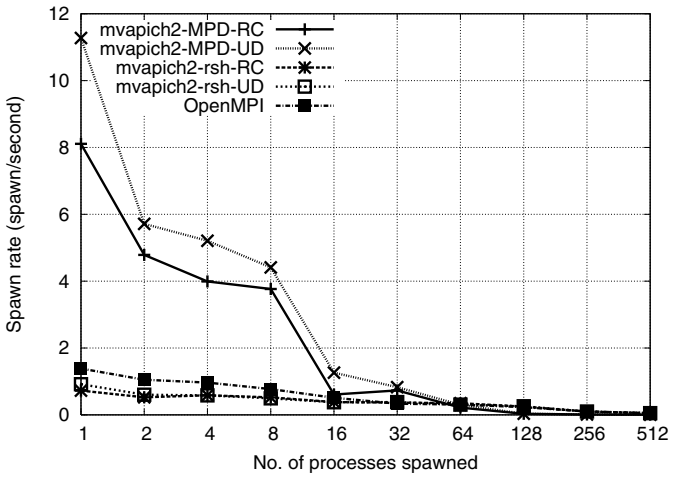

Fig. 6. Spawn rate

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.

C. Inter-group Latency

The inter-group latency is a basic latency test to measure the difference between intra-communicator latency and inter-communicator latency. As we see in Figure 7 for small message sizes, the mvapich2 inter-communicator exchange has a slightly higher latency. This higher latency is due to searching of process group and managing the translation from local group to remote group. For large messages, the latency of both the message exchanges are almost equal with very little variation and the data transfer component dominates and the process group translation cost does not affect overall latency. OpenMPI does not

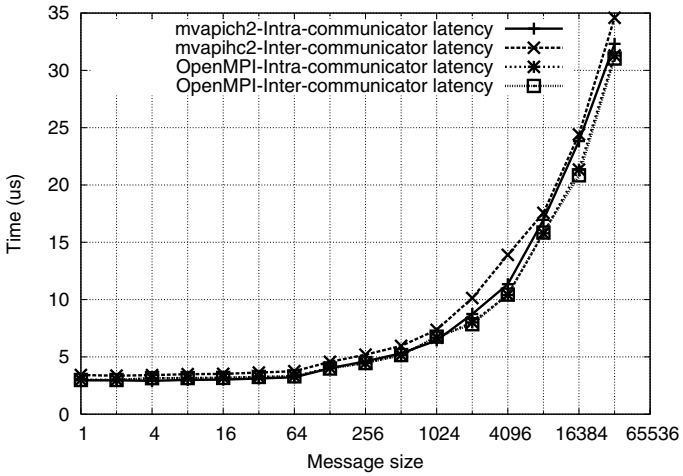

Fig. 7. Inter-group latency

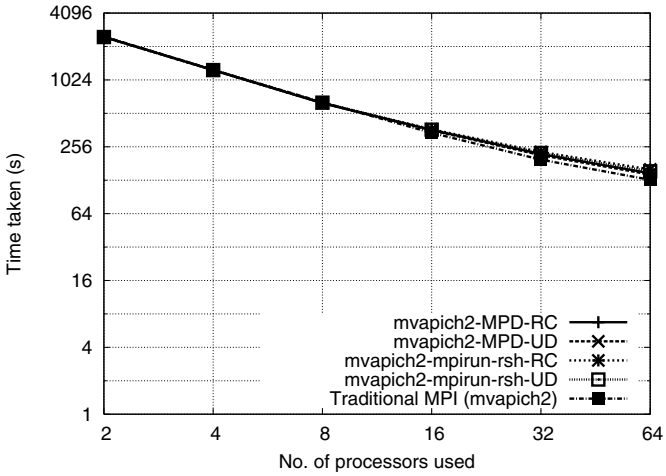

Fig. 8. Parallel POV-Ray evaluation

show any difference between inter-communicator and intra-communiators. However, OpenMPI does perform slightly better than MVAPICH2 for larger messages. This is due to the higher rendezvous threshold utilized by the OpenMPI library compared to MVAPICH2.

VII. Application Evaluation

A. Distributed Rendering with Dynamic Process Management

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 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 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 [5], but these implementations use a static runtime environment.The dynamic process interface can be programmed to have an 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. 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 8 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.

VIII. Related Work

The architecture of a dynamic process creation framework for MPI was described by Gropp and Lusk [7]. The MPI-2 standard [6] 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 [4] 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 [3] 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 [10]. Kim et al. [8] 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.

IX. Conclusions

With increasing clusters and multi-cores, 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. We implemented our designs and evaluated them on MVAPICH2, a popular MPI implementation for InfiniBand. The lack of benchmarks in this area was addressed and we designed new benchmarks to evaluate our designs. Our study draws the following conclusions on designing and evaluating the dynamic process framework.

- 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.
- MPD suffers from very high latency for large job sizes. For very large job launches, the ScELA [14] 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.
- 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.
- A high-performance implementation of the dynamic process interface will require both a good process manager and a fast implementation of the child-parent synchronization mechanisms.

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 area of collectives for dynamic processes and job scheduling for dynamic tasks in more detail.

X. Software Distribution

The dynamic process management designs, discussed in this paper, will be available with the upcoming MVAPICH2 release. The new benchmarks designed for evaluating dynamic process management interface of MPI-2 libraries will be integrated with the standard OSU benchmarks [11] and made available to the community in the near future.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Sangbum Kim, Namyoon Woo, and Heon Y. Yeom. Design and Implementation of Dynamic Process Management for Grid-Enabled MPICH.

[9] 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.

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

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

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

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

[14] 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.

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

[16] 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.