Virtual Machine Aware Communication Libraries for High Performance Computing

Wei Huang Matthew J. Koop Qi Gao Dhabaleswar K. Panda

Network-Based Computing Laboratory
The Ohio State University
Columbus, OH 43210
{huanwei, koop, gaoq, panda}@cse.ohio-state.edu

ABSTRACT

As the size and complexity of modern computing systems keep increasing to meet the demanding requirements of High Performance Computing (HPC) applications, manageability is becoming a critical concern to achieve both high performance and high productivity computing. Meanwhile, virtual machine (VM) technologies have become popular in both industry and academia due to various features designed to ease system management and administration. While a VM-based environment can greatly help manageability on large-scale computing systems, concerns over performance have largely blocked the HPC community from embracing VM technologies.

In this paper, we follow three steps to demonstrate the ability to achieve near-native performance in a VM-based environment for HPC. First, we propose Inter-VM Communication (IVC), a VM-aware communication library to support efficient shared memory communication among computing processes on the same physical host, even though they may be in different VMs. This is critical for multi-core systems, especially when individual computing processes are hosted on different VMs to achieve fine-grained control. Second, we design a VM-aware MPI library based on MVAPICH2 (a popular MPI library), called MVAPICH2-ivc, which allows HPC MPI applications to transparently benefit from IVC. Finally, we evaluate MVAPICH2-ivc on clusters featuring multi-core systems and high performance InfiniBand interconnects. Our evaluation demonstrates that MVAPICH2-ivc can improve NAS Parallel Benchmark performance by up to 11% in VM-based environment on eight-core Intel Clovertown systems, where each compute process is in a separate VM. A detailed performance evaluation for up to 128 processes (64 node dual-socket single-core systems) shows only a marginal performance overhead of MVAPICH2-ivc as compared with MVAPICH2 running in a native environment. This study indicates that performance should no longer be a barrier preventing HPC environments from taking advantage of the various features available through VM technologies.

SC 07 November 10-16, 2007, Reno, Nevada, USA.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management —*Message sending*; C.2.5 [**Computer Communication Networks**]: Local and Wide-Area Networks—*High-speed*

Keywords

Virtual Machines (VM), Inter-VM communication, MPI, Xen, VMM-bypass

1. INTRODUCTION

With ever-increasing computing power demands for ultra-scale applications, the High Performance Computing (HPC) community has been deploying modern computing systems with increasing size and complexity. These large scale systems require significantly more system management effort, including maintenance, reconfiguration, fault tolerance, and administration, which are not necessarily the concerns of earlier smaller scale systems. As a result, manageability is widely considered a critical requirement to achieve both high performance and high productivity computing.

Modern virtual machine (VM) technologies have emphasized ease of system management and administration. Through a Virtual Machine Monitor (VMM or hypervisor), which is implemented directly on hardware, VM technologies allow running multiple guest virtual machines (VMs) on a single physical node, with each guest VM possibly running a different OS instance. VM technologies separate hardware and software management and provide useful features including performance isolation, server consolidation and live migration [6]. For these reasons, VM technologies have already been widely adopted in industry computing environments, especially data-centers.

Many of the benefits provided by VM technologies are also applicable to HPC. For example, researchers have proposed proactive fault tolerance based on VM migration [27], which moves a running OS instance to another physical node when the original host is predicted to fail in the near future. HPC can also take advantage of online system maintenance, per-

formance isolation, or customized OSes [23], which are easily supported in VM-based environments.

Despite these promising features, VM technologies have not yet been widely deployed in HPC environments. Many in the HPC community are wary of virtualization technologies due to perceived performance overheads. These overheads, however, have been significantly reduced with recent VM technologies such as Xen [7] and I/O virtualization technologies like VMM-bypass I/O [21]. Two main challenges remain to be addressed to convince the HPC community of the low overhead of VM environments:

- **Lack of shared memory communication between VMs:** In VM environments, management activities such as migration occur at the level of entire VMs. Thus, all processes running in the same VM must be managed together. To achieve fine-grained process-level control, each computing process has to be hosted in a separate VM, thus, in a separate OS. This presents a problem, because processes in distinct OSes can no longer communicate via shared memory [5], even if they share a physical host. This restriction is undesirable because communication via shared memory is typically considered to be more efficient than network loopback and is being used in most MPI implementations [5, 11]. This inefficiency is magnified with the emergence of modern multi-core systems.
- **Lack of thorough performance evaluations:** As part of our earlier work, we have demonstrated the potential of a VM-based HPC environment[13, 21] with efficient network I/O virtualization. However, these studies have focused on prototype evaluations. In literature, there lacks a thorough performance evaluation published on VM-based computing environment with newer generation systems featuring multi-core platforms and recent high speed network interconnects.

In this paper we address both of these challenges. We first propose *IVC*, an Inter-VM Communication library to support shared memory communication between distinct VMs on the same physical host in a Xen environment. *IVC* has a general socket-style API and can be used by any parallel application in a VM environment. To allow HPC applications to benefit without modification, we design a VM-aware MPI library, MVAPICH2-ivc. Our MPI library is a modified version of MVAPICH2 [26], a popular MPI-2 library over InfiniBand. MVAPICH2-ivc takes advantage of both IVC and VMM-bypass I/O [21]. VM migration is also supported, so shared memory communication through IVC can be established or torn-down at runtime as a VM migrates. As a result, MVAPICH2-ivc achieves near-native performance regardless of whether the communicating processes are on the same or different physical hosts. Finally, we conduct a thorough performance evaluation of MVAPICH2-ivc on modern clusters featuring multi-core systems and a high speed interconnect (PCI-Express InfiniBand HCAs [22]). We demonstrate that using MVAPICH2-ivc in a VM-based environment can deliver up to 11% better performance of NAS Parallel Benchmarks than running unmodified MVAPICH2 when hosting one process per VM. An integrated evaluation on an InfiniBand cluster with up to 128 processors shows that with the latest virtualization technologies, including Xen, VMM-bypass I/O and IVC, a VM-based environment achieves near-native performance. Our evaluation concludes that performance should no longer be a barrier to deploying VM environments for HPC.

The rest of the paper is organized as follows: Section 2 introduces background information on VM technologies and VMM-bypass I/O. In Section 3, we further motivate our work by discussing the value of VM technology to HPC and the importance of efficient shared memory communication between VMs. In Section 4, we present IVC, including its API, design and implementation. We describe the design of MVAPICH2-ivc in Section 5. Section 6 includes a detailed performance evaluation. Related work is discussed in Section 7 and we present our conclusions in Section 8.

2. BACKGROUND

In this section, we briefly describe Xen [7], a popular VM environment. We also introduce VMM-bypass I/O and InfiniBand, both of which play important roles in a VM-based HPC environment.

2.1 Xen Virtual Machine Environment

Xen is a popular VM technology originally developed at the University of Cambridge. Figure 1 (courtesy [34]) illustrates the structure of a physical machine hosting Xen. The Xen hypervisor (the VMM) is at the lowest level and has direct access to the hardware. Above the hypervisor are the Xen domains (VMs) running *guest OS* instances. Each guest OS uses a pre-configured share of physical memory. A privileged domain called *Domain0 (or Dom0)*, created at boot time, is allowed to access the control interface provided by the hypervisor and performs the tasks to create, terminate and migrate other guest VMs *(User Domain or DomU)* through the control interfaces.

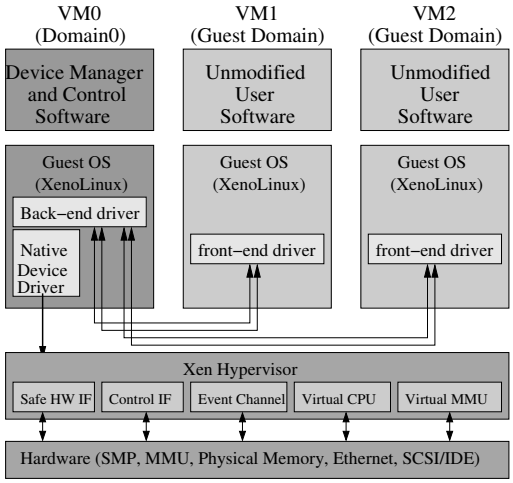


Figure 1: The structure of the Xen virtual machine monitor

I/O device virtualization in Xen follows a split device driver model [10]. The native device driver is expected to run in an *Isolated Device Domain* (or *IDD*). The IDD is typically the same as Dom0 and hosts a *backend* driver, running as a daemon and serving the access requests from each DomU. The

guest OS in a DomU uses a *frontend* driver to communicate with the backend. To pass the I/O requests, domains communicate with each other through shared pages and *event channels*, which provide an asynchronous notification mechanism between domains. To send data to another domain, the typical protocol is for the source domain to grant the remote domain access to its local memory pages by first registering the pages to *grant tables*, which are managed by the hypervisor. After registration, a grant reference handle is created for each physical page. The remote domain then uses these reference handles to map the pages into its local address space. Our proposed inter-VM communication uses the same mechanism to create shared memory regions.

During VM migration, the front-end drivers receive a *suspend* callback when the VM is about to be suspended on the original host and a *resume* callback when the VM is resumed on the new host machine. This gives an opportunity for the device drivers in the VM to cleanup local resources before migration and re-initialize themselves after migration.

2.2 InfiniBand and VMM-Bypass I/O

InfiniBand [15] is a popular interconnect offering high performance through user level communication and OS-bypass. With OS-bypass, applications can directly initiate communication operations without the involvement of the operating system. This allows InfiniBand to achieve low latency and high bandwidth. High performance and other features such as Remote Direct Memory Access (RDMA) make InfiniBand a strong player in HPC cluster computing environments. User level communication of InfiniBand is through the user-level APIs. Currently, the most popular API is provided by the OpenFabrics stack [29].

InfiniBand can be supported in Xen through VMM-bypass I/O, which is proposed in our earlier work [21]. To achieve high performance, the Xen split device driver model is only used to setup necessary user access points to allow OS-bypass communication to take place. After the initial setup, data communication in the critical path bypasses both the guest OS and the VMM by taking advantage of OS-bypass. By avoiding the overhead caused by the Xen split driver model, which passes I/O requests between DomU and the privileged domain, VMM-bypass achieves near-native performance. Application-transparent migration support for VMM-bypass I/O is also feasible through namespace virtualization and coordination among the communicating peers, as proposed in [14].

3. MOTIVATION

Figure 2 illustrates a possible VM-based HPC environment deployment using Xen. Each computing node in the physical server pool is running a hypervisor that hosts one or more VMs. Because HPC applications are typically compute intensive, each active VM runs on one or more dedicated cores. A set of VMs across the system form a virtual cluster, on which users run their HPC applications. There can be a management console which performs all management activities, including launching, checkpointing or migrating VMs across the computing nodes.

System management capabilities are increased in a VM-based environment – including allowing special configurations and online maintenance. To prepare a virtual cluster with a special configuration, all that is required is to choose a set of physical nodes and launch VMs with the desired configurations on those nodes. This avoids the lengthy procedure to reset all the physical hosts. More importantly, this does not affect other users using the same physical nodes. Online maintenance is also simplified. To apply patches or update a physical node, system administrators no longer need to wait for application termination before taking them off-line. VMs on those nodes can be migrated to other available physical nodes, reducing administrative burden.

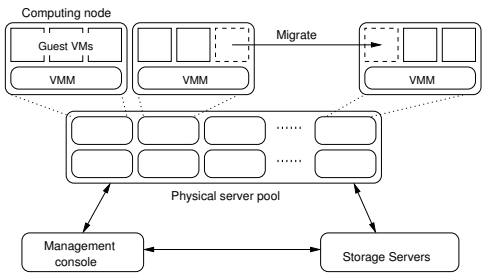


Figure 2: A possible deployment of HPC with virtual machines

Besides manageability, under certain circumstances VM environments can also lead to performance benefit. For example, with easy re-configuration it becomes much more practical to host HPC applications using customized, light-weight OSes. This concept of a customized OS is proven to be desirable for achieving increased performance in HPC [2, 8, 19, 23, 24]. VM migration can also help improving communication performance. For example, a parallel application may initially be scheduled onto several physical nodes far from each other in network topology due to node availability. But whenever possible, it is always desirable to schedule the processes onto physical nodes near each other, which improves communication efficiency. Through VM migration, it is also possible to alleviate resource contention and thus achieve better resource utilization. For instance, VMs hosting computing jobs with a high volume of network I/O can be relocated with other VMs hosting more CPU intensive applications with less I/O requirements.

VM technology can also benefit HPC in many other aspects, including security, productivity or debugging. Mergen et al. [23] have discussed in more detail the value of a VM-based HPC environment.

To achieve efficient scheduling in VM-based environments, such as migration to avoid resource contention, it is desirable to schedule jobs at the granularity of individual processes. This is especially the case for multi-core systems which are capable of hosting many computing processes on the same physical host. Benefits of VMs are severely limited if all processes on one node are required to be managed at the same time. Since VM management, such as migration, is at the level of individual VMs, fine-grained control requires computing processes to be hosted in separate VMs. This separation, however, is not efficient for communication. Many MPI implementations communicate through user space shared memory between processes in the same OS [5, 11]. With processes running on separate VMs

(OSes), shared memory communication is no longer available and communication must go through network loopback. Since shared memory communication is typically more efficient than network loopback, hosting computing processes on separate VMs may lead to a significant performance gap as compared with native environments.

4. DESIGN FOR EFFICIENT INTER-VM COMMUNICATION

In this section we present our inter-VM communication (IVC) library design, which provides efficient shared memory communication between VMs on the same physical host. It allows the flexibility of hosting computing processes on separate VMs for fine-grained scheduling without sacrificing the efficiency of shared memory communication to peers sharing the same physical host.

Our design is based on the page sharing mechanisms through grant tables provided by Xen, as described in Section 2.1. There are still many challenges that need to be addressed, however. First, the initial purpose of grant table mechanism is for sharing data between device drivers in kernel space. Thus, we need to carefully design protocols to setup shared memory regions for user space processes and provide communication services through these shared regions. Second, we need to design an easy-to-use API to allow applications to use IVC. Further, IVC only allows communication between VMs on the same host, which leads to additional concerns for VM migration. We address these challenges in the following sections.

4.1 Setting up Shared Memory Regions

In this section we describe the key component of IVC: How to setup shared memory regions between two user processes when they are not hosted in the same OS?

IVC sets up shared memory regions through Xen *grant table* mechanisms. Figure 3 provides a high level architectural overview of IVC and illustrates how IVC sets up shared memory regions between two VMs. IVC consists of two parts: a user space communication library and a kernel driver. We use a client-server model to setup an IVC connection between two computing processes. One process (the client) calls the IVC user library to initiate the connection setup (step 1 in Figure 3). Internally, the IVC user library allocates a communication buffer, which consists of several memory pages and will be used later as the shared region. The user library then calls the kernel driver to grant the remote VM the right to access those pages and returns the grant table reference handles from Xen hypervisor (steps 2 and 3). The reference handles are sent to the IVC library on the destination VM (step 4). The IVC library on the destination VM then maps the communication buffer to its own address space through the kernel driver (steps 5 and 6). Finally, IVC notifies both computing processes that the IVC connection setup is finished, as in step 7 of Figure 3.

4.2 Communication through Shared Memory Regions

After creating a shared buffer between two IVC libraries, we design primitives for shared memory communication between computing processes. IVC provides a socket style

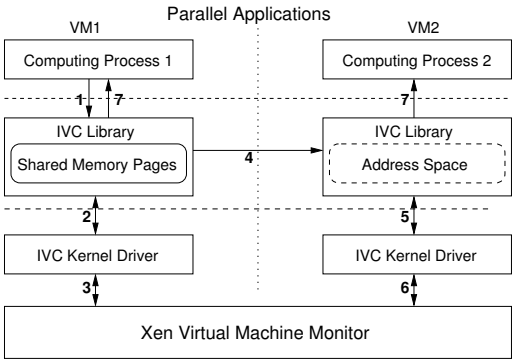


Figure 3: Mapping shared memory pages

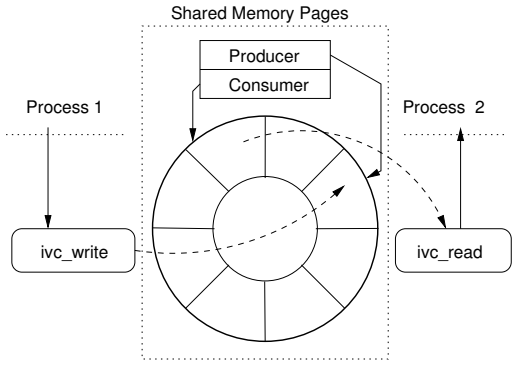


Figure 4: Communication through mapped shared memory regions

read/write interface and conducts shared memory communication through classic producer-consumer algorithms. As shown in Figure 4, the buffer is divided into send/receive rings containing multiple data segments and a pair of producer/consumer pointers. A call to `ivc_write` places the data on the producer segments and advances the producer pointer. And a call to `ivc_read` reads the data from the consumer segment and advances the consumer pointer. Both the sender and receiver check the producer and consumer pointers to determine if the buffer is full or for data arrival. Note that Figure 4 only shows the send ring for process 1 (receive ring for process 2). Total buffer size and the data segment size are tunable. In our implementation, each send/receive ring is 64KB and each data segment is 64 bytes. Thus, an `ivc_write` call can consume multiple segments. If there is not enough free data segments, `ivc_write` will simply return the number of bytes successfully sent, and the computing process is responsible to retry later. Similarly, `ivc_read` is also non-blocking and returns the number of bytes received.

Initialization of IVC also requires careful design. IVC sets up connections for shared memory communication between VMs only after receiving a connection request from each of the hosted processes. Computing processes, however, cannot be expected to know which peers share the same physical host. Such information is hidden to the VMs and thus has to be provided by IVC. To address this issue, an IVC backend driver is run in the privileged domain (Dom0). Each parallel job that intends to use IVC should have a unique

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, CLIENT_ID);
/* Get all Peers on the same physical node */
peers = ivc_get_hosts(ctx);
/* We assume only one peer now (the server).
 * Connect to server (peers[0]).
 * channel.id will be SERVER_ID upon success. */
channel = ivc_connect(ctx, peers[0]);
/* Send data to server, bytes will be @size upon
 * success. Otherwise, we will have to retry. */
bytes = ivc_write(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

(a) Communication client

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, SERVER_ID);
/* Block until an incomming connection.
 * Upon success, channel.id will be CLIENT_ID */
channel = ivc_accept(ctx);
/* Receive data from client. */
 * Upon success, bytes will be equal to @size.
 * We will have to retry other wise. */
bytes = ivc_read(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

(b) Communication server

Figure 5: A very brief client-server example for using IVC

magic id across the cluster. When a computing process initializes, it notifies the IVC library of the *magic id* for the parallel job through an `ivc_init` call. This process is then registered to the backend driver. All registered processes with the same *magic id* form a local communication group, in which processes can communicate through IVC. A computing process can obtain all peers in the local communication group through `ivc_get_hosts`. It can then connect to those peers through multiple `ivc_connect` calls, which initialize the IVC connection as mentioned above. Using a *magic id* allows multiple communication groups for different parallel jobs to co-exist within one physical node. Assignment of this unique *magic id* across the cluster could be provided by batch schedulers such as PBS [33], or other process manager such as MPD [1] or SLURM [35]. Additional details go beyond the scope of this paper.

Figure 5 shows a brief example how two processes communicate through IVC. As we can see, IVC provides socket style interfaces, and communication establishment follows a client-server model.

4.3 Virtual Machine Migration

As a VM-aware communication library, an additional challenge faced by IVC is handling VM migration. As a shared memory library, IVC can only be used to communicate between processes on the same physical host. Subsequently, once a VM migrates to another physical host, processes running on the migrating VM can no longer communicate with the same peers through IVC. They may instead be able to communicate through IVC with a new set of peers on the new physical host after migration.

IVC provides callback interfaces to assist applications in handling VM migration. Figure 6 illustrates the main flow of IVC in case of VM migration. First, the IVC kernel driver gets a callback from the Xen hypervisor once the VM is about to migrate (step 1). It notifies the IVC user library to stop writing data into the send ring to prevent data loss during migration; this is achieved by returning 0 on every attempt of `ivc_write`. After that, the IVC kernel notifies all other VMs on the same host through event channels that the VM is migrating. Correspondingly, the IVC libraries running in other VMs will stop their send operations and acknowledge the migration event (steps 2 and 3). IVC then gives user programs a callback, indicating that the IVC channel is no longer available due to migration. Meanwhile, there may be data remaining in the receive ring that has not been passed to the application. Thus, IVC also provides applications a data buffer containing all data in the receive ring that has not been read by the application through `ivc_read`(step 4). Finally, IVC unmaps the shared memory pages, frees local resources, and notifies the hypervisor that the device has been successfully suspended and can be migrated safely (step 5). After the VM is migrated to the remote host, the application will receive another callback indicating arrival on a new host, allowing connections to be setup to a new set of peers through similar steps as in Figure 5(a) (step 6).

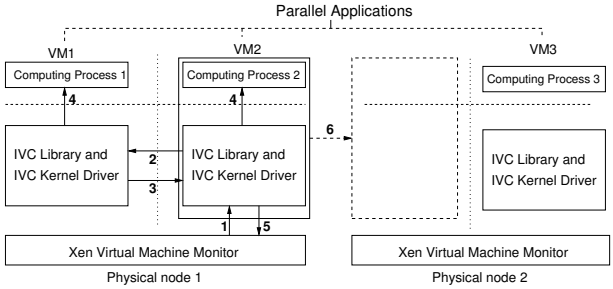
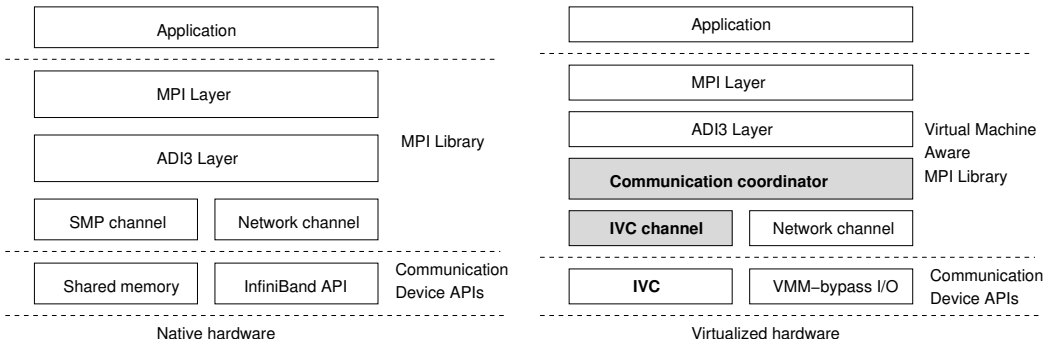


Figure 6: Migrating one VM (VM2) of a three process parallel job hosted on three VMs

5. MVAPICH2-IVC: VIRTUAL MACHINE AWARE MPI OVER IVC

IVC has defined its own APIs. Thus, to benefit HPC applications transparently, we design MVAPICH2-ivc, a VM-aware MPI library modified from MVAPICH2. MVAPICH2-ivc is able to communicate through IVC with peers on the same physical host and over InfiniBand when communication is inter-node. MVAPICH2-ivc is also able to intelligently switch between IVC and network communication as VMs migrate. By using MVAPICH2-ivc in VM environments, MPI applications can benefit from IVC without modification.



(a) Native environment (b) VM environment

Figure 7: MVAPICH2 running in different environments

5.1 Design Overview

MVAPICH2-ivc is modified from MVAPICH2 [26], a popular multi-method MPI-2 implementation over InfiniBand based on MPICH2 from Argonne National Laboratory [1]. For portability reasons, MVAPICH2/MPICH2 follows a layered approach, as shown in Figure 7(a). The Abstract Device Interface V3 (ADI3) layer implements all MPI-level primitives. Multiple communication channels, which in turn provide basic message delivery functionalities on top of communication device APIs, implement the ADI3 interface. There are two communication channels available in MVAPICH2: a shared memory channel communicating over user space shared memory [5] to peers hosted in the same OS and a network channel communicating over InfiniBand user-level APIs to other peers.

An unmodified MVAPICH2 can also run in VM environments; however, its default shared memory communication channel can no longer be used if computing processes are hosted on different VMs (OSes). By using IVC, MVAPICH2-ivc is able to communicate via shared memory between processes on the same physical node, regardless of whether they are in the same VM or not. Figure 7(b) illustrates the overall design of MVAPICH2-ivc running in a VM environment. Compared with MVAPICH2 in a native environment, there are three important changes. First, we replace the original shared memory communication channel with an IVC channel, which performs shared memory communication through IVC primitives. Second, the network channel is running on top of VMM-bypass I/O, which provides InfiniBand service in VM environments (because VMM-bypass I/O provides the same InfiniBand verbs, no changes to MVAPICH2 are needed). Third, we design a communication coordinator, which dynamically creates and tears down IVC connections as the VM migrates.

The communication coordinator keeps track of all the peers to which IVC communication is available. To achieve this, it takes advantage of a data structure called a *Virtual Connection (VC)*. In MVAPICH2, there is a single VC between each pair of computing processes, which encapsulates details about the available communication methods between that specific pair of processes as well as other state information. As shown in Figure 8, the communication coordinator maintains an *IVC-active* list, which contains all VCs for peer processes on the same physical host. During the initialization stage, this *IVC-active* list is generated by the communication coordinator according to the peer list returned by `ivc_get_hosts`. VCs in the list can be removed or added to this list when the VM migrates.

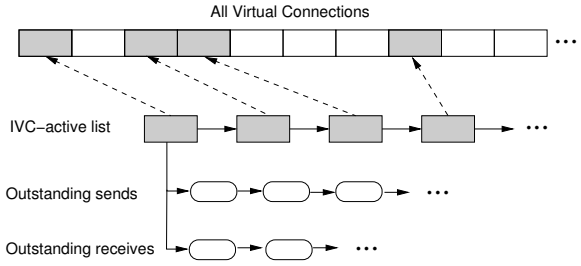


Figure 8: Organization of IVC-active list in MVAPICH2-ivc

Once the application issues a MPI send to a peer process, the data is sent through IVC if the VC to that specific peer is in the *IVC-active* list. As described in Section 4.1, IVC cannot guarantee all data will be sent (or received) by one `ivc_write` (or `ivc_read`) call. Thus, to ensure in order delivery, we must maintain queues of all outstanding send and receive operations for each VC in the *IVC-active* list. Operations on these queues are retried when possible.

5.2 Virtual Machine Migration

As discussed in Section 4.3, IVC issues application callbacks upon migration. Correspondingly, applications are expected to stop communicating through IVC to peers on the original physical host and can start IVC communication to peers on the new physical host. In MVAPICH2-ivc, the communication coordinator is responsible to adapt to such changes. The coordinator associates an *ivc state* to each VC. As shown in Figure 9, there are four states possible: IVC_CLOSED, IVC_ACTIVE, IVC_CONNECTED and IVC_SUSPENDING. At the initialization stage, each VC is either in the IVC_ACTIVE state if the IVC connection is set up, or in the IVC_CLOSED state, which indicates IVC is not available and communication to that peer has to go through the network.

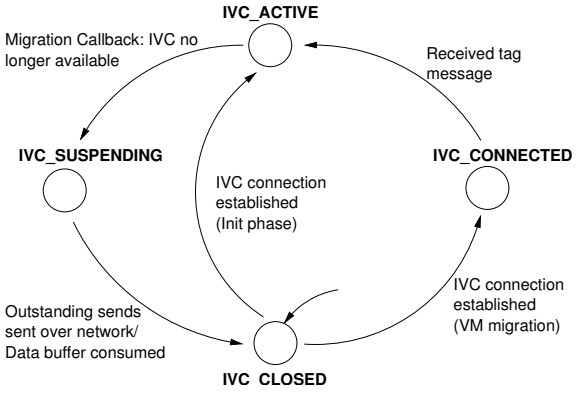


Figure 9: State transition graph of *ivc state*

When a VM migrates, IVC communication will no longer be available to peers on the original host. As we have discussed, the communication coordinator will be notified, along with a data buffer containing the contents of the receive ring when IVC is torn down. The coordinator then changes the state to IVC_SUSPENDING. In this state, all MPI-level send operations are temporarily blocked until the coordinator transmits all outstanding requests in the IVC outstanding send queue through the network channel. Also, all MPI-level receive operations are fulfilled from the data buffer received from IVC until all data in the buffer is consumed. Both of these steps are necessary to guarantee in order delivery of MPI messages. Next, the coordinator changes the *ivc state* to IVC_CLOSED and removes the VC from the IVC active list. Communication between this pair of processes then flows through the network channel.

Once migrated, IVC will be available to peers on the new host. The coordinator will get a callback from IVC and setup IVC connections to eligible peers. IVC cannot be immediately used, however, since there may be pending messages on the network channel. To reach a consistent state, the communication coordinators on both sides of the VC change the *ivc state* to IVC_CONNECTED, and send a flush message through the network channel. Once the coordinator receives a flush message, no more messages will arrive from the network channel and the *ivc state* is changed to IVC_ACTIVE and VC added to the IVC active list. Both sides can now communicate through the IVC channel.

6. EVALUATION

In this section we present performance evaluation of a VM-based HPC environment running MVAPICH2-ivc with each process in a separate VM. We first evaluate the benefits achieved through IVC using a set of micro-benchmark and application-level benchmarks. We show that on multi-core systems MVAPICH2-ivc shows clear improvement compared with unmodified MVAPICH2, which cannot take advantage of shared memory communication when processes are on distinct VMs. We demonstrate that performance of MVAPICH2-ivc is very close to that of MVAPICH2 in a native (non-virtualized) environment. Evaluation on up to 128 processes shows that a VM-based HPC environment can deliver very close application-level performance compared to a native environment.

6.1 Experimental Setup

The experiments are carried out on two testbeds. Testbed A consists of 64 computing nodes. There are 32 nodes with dual Opteron 254 (single core) processors and 4GB of RAM each and 32 nodes with dual Intel 3.6 GHz Xeon processors and 2GB RAM each. Testbed B consists of computing nodes with dual Intel Clovertown (quad-core) processors, for a total of 8 cores and 4GB of RAM. Both testbeds are connected through PCI-Express DDR InfiniBand HCAs (20 Gbps). Xen-3.0.4 with the 2.6.16.38 kernel is used on all computing nodes for VM-based environments. We launch the same number of VMs as the number of processors (cores) on each physical host, and host one computing process per VM.

We evaluate VM-based environments with MVAPICH2-ivc and unmodified MVAPICH2, each using VMM-bypass I/O. We also compare against the performance of MVAPICH2 in native environments. More specifically, our evaluation is conducted with the following three configurations:

- **IVC** - VM-based environment running MVAPICH2-ivc, which communicates through IVC if the processes are hosted in VMs on the same physical host.
- **No-IVC** - VM-based environment running unmodified MVAPICH2, which always communicates through network since each process is in a separate VM.
- **Native** - Native environment running unmodified MVAPICH2, which uses shared memory communication between processes on the same node.

6.2 Micro-benchmark Evaluation

In this section, the performance of MVAPICH2-ivc is evaluated using a set of micro-benchmarks. First a comparison of basic latency and bandwidth achieved by each of the three configurations, when the computing processes are on the same physical host is performed. Next, the performance of collective operations using Intel MPI Benchmarks (IMB 3.0) [16] is measured.

Figure 10 illustrates the MPI-level latency reported by OSU benchmarks [26]. For various message sizes, this test 'ping-pongs' messages between two processes for a number of iterations and reports the average one-way latency observed. IVC communication is able to achieve latency around 1.2μs for 4 byte messages, which, in the worst case, is only about 0.2μs higher than MVAPICH2 communicating through shared memory in a native environment. The IVC latency is slightly higher because MVAPICH2 has recently incorporated several optimizations for shared memory communication [5]. We plan to incorporate those optimizations in the future, as they should be applicable to IVC as well. In both cases, the latency is much lower than the No-IVC case, where communication via network loopback shows 3.16μs latency for 4 byte messages.

Figure 11 presents MPI-level uni-directional bandwidth. In this test, a process sends a window of messages to its peer using non-blocking MPI sends and waits for an acknowledgment. The total message volume sent divided by the total time is reported as bandwidth. Both IVC and native cases

achieve much higher bandwidth for medium-sized messages than in the No-IVC case. An interesting observation is that IVC achieves higher bandwidth than the native case. This can be due to two reasons: first, MVAPICH2's optimized shared memory communication requires more complicated protocols for sending large messages, thus adding some overhead; second, IVC uses only 16 pages as the shared memory region, but MVAPICH2 uses a few million bytes. At a micro-benchmark level, a larger shared memory buffer can slightly hurt performance due to cache effects. A smaller shared memory region, however, holds less data, thus requiring the peer to receive data fast enough to maintain a high throughput. This can lead to less efficient communication progress for applications using large messages very frequently because processes are likely to be skewed. For IVC, we find 16 pages as a good choice for shared memory buffer size [1].

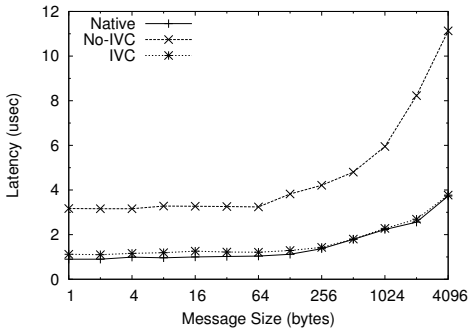


Figure 10: Latency

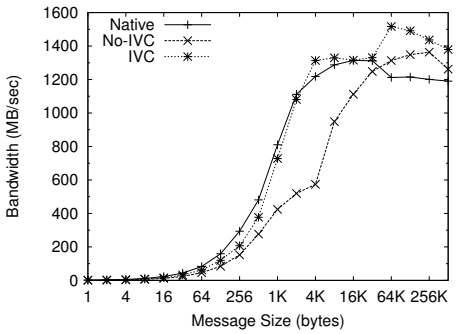


Figure 11: Bandwidth

Figures 12 and 13 illustrate the ability of MVAPICH2-ivc to automatically select the best available communication method after VM migration. In Figure 12, we keep running a latency test with 2KB messages. The test is first carried out between two VMs on distinct physical hosts. At around iteration 400, we migrate one VM so that both VMs are hosted on the same physical node. From the figure it can be observed that the latency drops from 9.2μs to 2.8μs because MVAPICH2-ivc starts to use IVC for communication. At about iteration 1100, the VMs are again migrated to distinct physical hosts, which causes the latency to increase to the original 9.2μs. The drastic increase in latency during migration is because network communication freezes during VM migration, which is explained in more detail in [14]. In Figure 13, the same trend for a bandwidth test is observed, which reports bandwidth achieved for 2KB messages while the VM migrates. When two VMs are located on the same host (iteration 1100 to iteration 2000), communication through IVC increases the bandwidth from around 720MB/s to 1100MB/s.

[1]We do not go for larger number of pages because the current Xen-3.0.4 allows at most 1K pages to be shared per VM. While this restriction can be fixed in the future, 16 page shared buffer will allow us to support up to 60 VMs per physical node with the current implementation.

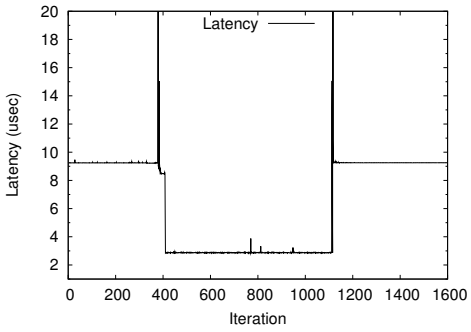


Figure 12: Migration during latency test

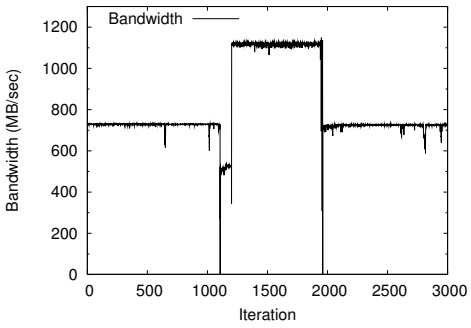
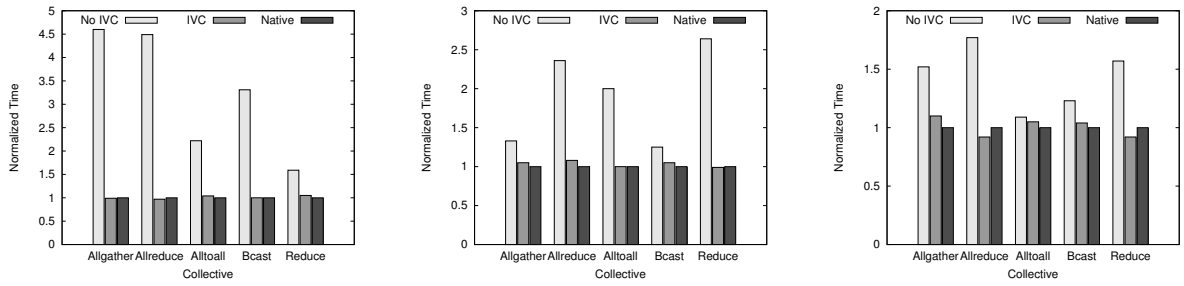


Figure 13: Migration during bandwidth test

Next, the performance of collective operations using Intel MPI Benchmarks is evaluated. In Figure 14, several of the most commonly used collective operations are compared using all three configurations. The tests are conducted on Testbed B, using 2 nodes with 8 cores each (in total 16 processes). Collective performance is reported for small (16 bytes), medium (16KB), and large (256KB) size operations, with all results normalized to the Native configuration. Similar to the trends observed in the latency and bandwidth benchmarks, IVC significantly reduces the time needed for each collective operation as compared with the No-IVC case. It is able to deliver comparable performance as a native environment. Another important observation is that even though the No-IVC case achieves almost the same bandwidth for 256KB messages as IVC in Figure 11, it still performs significantly worse for some of the collective operations. This is due to the effect of network contention. On multi-core systems with 8 cores per node, the network performance can be largely degraded when 8 computing processes access the network simultaneously. Though the processes are also competing for memory bandwidth when communicating through IVC, memory bandwidth is typically much higher. This effect shows up in large message collec-



(a) 16 Bytes (b) 16K Bytes (c) 256K Bytes

Figure 14: Comparison of collective operations (2 nodes with 8 cores each)

tives, which further demonstrates the importance of shared memory communication for VM-based environments.

6.3 Application-level Benchmark Evaluation

In this section, several application-level benchmarks are used to evaluate the performance of MVAPICH2-ivc. As in the micro-benchmarks evaluation, we evaluate configurations of IVC (MVAPICH2-ivc in VM environment), No-IVC (unmodified MVAPICH2 in a VM environment) and Native (MVAPICH2 in a native environment).

We use several applications in our evaluations. These applications have various communication patterns, allowing a thorough performance comparison of our three configurations. The applications used in the evaluation are:

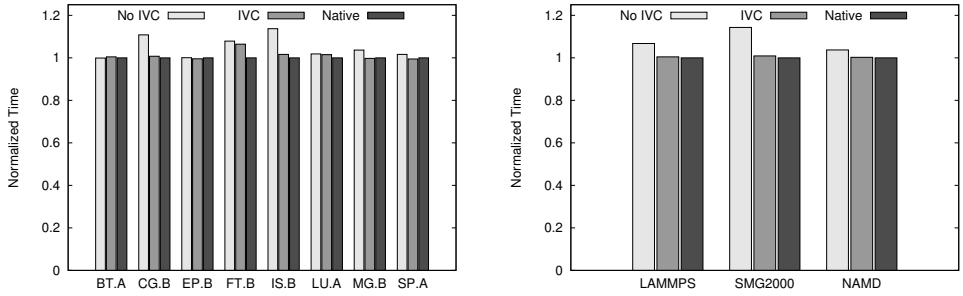
- **NAS Parallel Benchmark Suite** - NAS [28] contains a set of benchmarks which are derived from the computing kernels common on Computational Fluid Dynamics (CFD) applications.
- **LAMMPS** - LAMMPS stands for Large-scale Atomic/ Molecular Massively Parallel Simulator [20]. It is a classical molecular dynamics simulator from Sandia National Laboratory.
- **NAMD** - NAMD is a molecular dynamics program for high performance simulation of large biomolecular systems [32]. It is based on Charm++ parallel objects, which is a machine independent parallel programming system. NAMD can use various data sets as input files. We use one called `apoa1`, which models a bloodstream lipoprotein particle.
- **SMG2000** - SMG2000 [3] is a parallel semicoarsening multigrid solver for the linear systems on distributed memory computers. It is written in C using MPI.
- **HPL** - High Performance Linpack (HPL) is the parallel implementation of Linpack [31] and the performance measure for ranking the computer systems of the Top 500 supercomputer list.

Figure 15 shows the evaluation results on Testbed B, with all results normalized to performance achieved in the native environment. IVC is able to greatly close the performance gap between the No-IVC and native cases. Compared with No-IVC, IVC improves performance by up to 11% for NAS Parallel Benchmarks – IS (11%) and CG (9%). We observe a 5.9%, 11.8%, and 3.4% improvement in LAMMPS, SMG2000 and NAMD, respectively.

We further analyze the communication patterns of these benchmarks. Figure 16(a) introduces a communication rate metric, which is the total volume of messages sent by each process divided by execution time. This represents an approximation of how frequent the application communicates. As we can see, for several applications which have a high communication rate, such as NAS-IS, NAS-CG, SMG2000 and LAMMPS, IVC achieves performance improvement as compared with No-IVC. When running on a larger number of computing nodes, some applications could benefit less from IVC because a larger percentage of peers are not located on the same physical host. However, Figure 16(b) suggests that IVC is still very important in many cases. We analyze the percentage of data volume that will be sent through IVC on clusters with 8 core computing nodes. We find that IVC communication is well above average, especially for CG, MG, SMG2000 and LAMMPS, the percentage of IVC communication is higher than 50% even on a 64 core cluster[2]. This is because some applications tend to communicate frequently between neighbors, which have high probability to be on the same host for multi-core systems. For those applications, the benefits of IVC are expected to be observable.

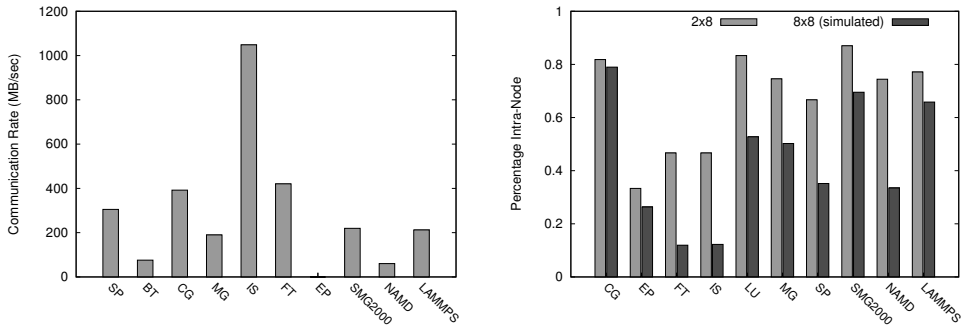
We also observe that IVC achieves comparable performance with the native configuration. The overhead is marginal (within 1.5%) in most cases. The only exception is NAS-FT, where a performance degradation of 6% is observed. This is due to the main communication pattern of NAS-FT, an all-to-all personalized operation with very large message sizes. In MVAPICH2 (and also MVAPICH2-ivc), all-to-all personalized operations are implemented on top of non-blocking send operations. As noted earlier, IVC only uses a 16 page shared memory space, which takes multiple iterations of the buffer space to send out a large message, hurting communication progress. Meanwhile, MVAPICH2 incorporated an optimized shared memory communication proposed by Chai et al. [5], which leads to better performance than IVC. Fortunately, FT is currently the only application we have noticed which has such communication pattern. Thus, in most cases

[2]Data is collected through simulation on Testbed A, which has a larger number of processors.



(a) NAS Parallel Benchmarks (b) LAMMPS, SMG2000 and NAMD

Figure 15: Application-level evaluation on Testbed B (2 nodes with 8 cores each)



(a) Communication Rate (2 x 8cores) (b) Percentage volume of IVC communication

Figure 16: Communication patterns of the evaluated applications

we will not notice this performance gap. Optimizing IVC for better performance under such communication patterns is also planned, since optimizations used in MVAPICH2 are also possible for IVC.

In order to examine the performance of MVAPICH2-ivc on a larger scale cluster, a 64-node VM-based environment was setup on Testbed A. Figures 17 and 18 show the performance comparison of NAS Parallel Benchmarks and HPL on Testbed A. Because systems of Testbed A are 2 processors per node with single-core only, the percentage of IVC communication is small compared to inter-node communication through the network. Thus, IVC and No-IVC configurations achieve almost the same performance here and the No-IVC configuration is omitted for conciseness. Compared with the Native configuration, we observe that the VM-based environment performs comparably. In most cases the performance difference is around 1%, except for NAS-FT, which degrades around 5% because of its large message all-to-all personalized communication pattern.

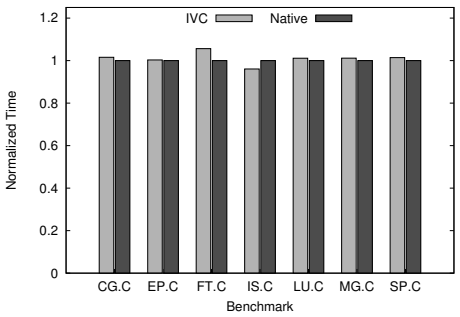


Figure 17: Normalized Execution time of NAS (64 nodes with 2 processors each)

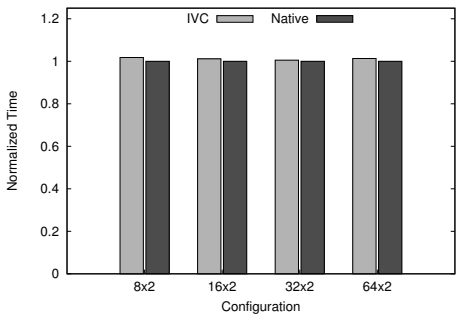


Figure 18: Normalized Execution time of HPL

7. RELATED WORK

In this paper, we have focused on VM-based environments for high performance computing. VMs have been a popular research topic in recent years and are being deployed in industry production environments, through such products as VMware VirtualCenter [37] and Xen Enterprise [39]. VM technologies have the potential to greatly benefit HPC applications. Mergen et al. [23] had a thorough discussion on the values of hardware virtualization in HPC environments.

There have been several research works focused on addressing manageability issues of HPC environment through VM technologies, such as VM-based proactive fault tolerance proposed by Mueller et al. [27].

Several researchers have deployed experimental VM-based computing environments. Figueiredo et al. [9] proposed grid computing based on virtualization and middleware mechanisms to manage VMs in a distributed environment. In our earlier work [13], we have further explored using VMs in cluster-based HPC environments. In this paper, we focused on the cluster environment and proposed inter-VM communication (IVC), which is designed to alleviate another performance overhead of VM-based computing. Inter-VM communication is very important to achieve fine-grained control without sacrificing performance on multi-core systems.

The para-virtualized Xen network device driver also uses shared memory pages to communicate among VMs on the same physical host. However, it has more overhead including network stack processing and interrupt handling compared with IVC. IVC communication is performed through polling at user space, which is much more desired for HPC applications. IVC is based on grant table mechanisms available to Xen-compatible kernels. We notice that there is a recent effort from the Xen community to expose the grant table for user space usage [39]. Such a method could simplify the design of IVC, however, other issues that we have discussed, such as the design of easy-to-use communication APIs and handling VM migration, are still required. There are several other researchers studying inter-VM communication in different problem domains. For example, Muir et al. [25] have proposed *Proper*, a service running on PlanetLab to allow multiple services to cooperate and interact with each other. Kourai et al. [18] proposed *HyperSpector*, which achieves secure intrusion detection in distributed computer systems through inter-VM cooperation. XenFS [39] aims to improve file system performance through inter-VM cache sharing.

We have proposed and implemented a VM-aware MPI in this paper, which is a multi-method MPI that is able to communicate through both the network and IVC. There are several published studies on multi-method MPIs, including [4, 11, 12, 17, 30, 36]. Most of these assume static configurations of available communication methods. Some of them support switching communication methods at runtime, but the main purpose is network fail-over [11, 12, 36]. MVAPICH2-ivc is designed for an environment where available communication methods may change due to migration. The MPI level can adapt to a new communication method more smoothly by taking advantage of library callbacks.

Our detailed performance evaluation shows very little overhead of MVAPICH2-ivc for virtualization in HPC environments. VMM-bypass I/O [21] and its migration support [14] plays an important role by drastically reducing the I/O overhead in VM environments by taking advantage of user level communication. Willmann [38] also proposed CDNA (Concurrent Direct Network Access), which provides direct network access on a programmable and reconfigurable FPGA-based Gigabit Ethernet network interface in VM environments.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the performance concerns of deploying a VM-based environment for high performance computing. One concern is the trade-off between fine-grain management of parallel applications at the level of each individual process and the ability to communicate via shared memory. Correspondingly, we propose VM-aware communication libraries, including an Inter-VM Communication (IVC) library and MVAPICH2-ivc, an MPI library based on IVC. MVAPICH2-ivc is able to communicate via shared memory for processes on the same physical host, even if they are on separate VMs. This provides the ability to host one computing process per VM, thus achieving the flexibility of fine-grain management without sacrificing the efficiency of shared memory communication. We evaluate MVAPICH2-ivc on clusters featuring multi-core systems and high performance InfiniBand interconnects. Our evaluation demonstrates that MVAPICH2-ivc can improve NAS Parallel Benchmark performance by up to 11% in a VM-based environment on an eight-core Intel Clovertown system. Application-level performance achieved by MVAPICH2-ivc on 128 processes shows only marginal overhead as compared to MVAPICH2 running in a native environment. Our work demonstrates that performance overhead should no longer be a major concern for VM-based HPC environments. With additional manageability, VM-based environments provide a solution to achieve both high performance and high productivity computing.

In the future, we plan to further optimize the performance of IVC, especially for large message communication patterns. We will explore efficient runtime management of VM-based computing. For example, how to efficiently launch a virtual cluster with customized configurations, how to efficiently checkpoint parallel applications in a VM environment, etc. Solutions can be from multiple system-level perspectives such as OS, file system, or network support. We also plan to address other potential issues for VM-based computing environments, such as reducing the memory overhead of the OS running on each separate VM.

Acknowledgments

We would like to thank Dr. Jiuxing Liu and Dr. Bulent Abali from IBM T. J. Research Center for the valuable discussion and suggestions. We thank Carrie Casto for proof reading this paper. We would also like to thank the anonymous reviewers for their insightful comments, which help to improve the final version of this paper.

This research is supported in part by the following grants and equipment donations to the Ohio State University: Department of Energy's Grant #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0702675; grants from Intel, Mellanox, Sun, Cisco, and Linux Networx; and equipment donations from Apple, AMD, IBM, Intel, Microway, Pathscale, Silverstorm and Sun.

9. REFERENCES

[1] Argonne National Laboratory. http://www-unix.mcs.anl.gov/mpi/mpich/.

[2] Argonne National Laboratory. Zeptoos: The small linux for big computers. http://www-unix.mcs.anl.gov/zeptoos/.

[3] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21:1823–1834, 2000.

[4] D. Buntinas, G. Mercier, and W. Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 521–530, Washington, DC, USA, 2006.

[5] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, September 2006.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, Botson, MA, 2005.

[7] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003.

[8] FastOS: Forum to Address Scalable Technology for runtime and Operating Systems. http://www.cs.unm.edu/ fastos/.

[9] R. Figueiredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS), May 2003.*, 2003.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, UK, 2004.

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[12] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A Network-failure-tolerant Message-passing System for Terascale Clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.

[13] W. Huang, J. Liu, B. Abali, and D. K. Panda. A Case for High Performance Computing with Virtual Machines. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS'06)*, Cairns, Australia, June 2006.

[14] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: Migrating OS-bypass Networks in Virtual Machines. In *Proceedings of the third ACM/USENIX Conference on Virtual Execution Environments (VEE'07)*, San Diego, California, June 2007.

[15] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.

[16] Intel Corporation. Intel Cluster Toolkit 3.0 for Linux.

[17] N. T. Karonis, B. R. Toonen, and I. T. Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Int. J. Parallel Program.*, 63(5):551–563, 2003.

[18] K. Kourai and S. Chiba. HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection. In *Proceedings of the first ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*, 2005.

[19] O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proceedings of EuroSys 2006*, Leuven, Belgium, April 2006.

[20] LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov/.

[21] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of USENIX '06*, Boston, MA, May 2006.

[22] Mellanox Technologies. http://www.mellanox.com.

[23] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for High-Performance Computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, 2006.

[24] MOLAR: Modular Linux and Adaptive Runtime Support for High-end Computing Operating and Runtime Systems. http://forge-fre.ornl.gov/molar/.

[25] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman. Proper: Privileged Operations in a Virtualised System Environment. In *USENIX 05*, Anaheim, CA, April 2005.

[26] MVAPICH/MVAPICH2 Project Website. http://mvapich.cse.ohio-state.edu.

[27] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, Seattle, WA, June 2007.

[28] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[29] Open Fabrics Alliance. http://www.openfabrics.org.

[30] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *Workshop on Novel Uses of System Area Networks (SAN-1), in Conjuction with HPCA-8*, Cambridge, MA, Feb 2002.

[31] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[32] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[33] Portable Batch System (OpenPBS). http://www.openpbs.org/.

[34] I. Pratt. Xen Virtualization. Linux World 2005 Virtualization BOF Presentation.

[35] SLURM: A Highly Scalable Resource Manager. http://www.llnl.gov/linux/slurm/.

[36] A. Vishnu, P. Gupta, A. Mamidala, and D. Panda. A Software Based Approach for Providing Network Fault Tolerance in Clusters with uDAPL interface: MPI Level Design and Performance Evaluation. In *Proceedings of SuperComputing (SC'06)*, Tampa, FL, Nov 2006.

[37] VMware – Virtual Infrastructure Software. http://www.vmware.com.

[38] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, Phoenix, AZ, Feb. 2007.

[39] XenSource. http://www.xensource.com/.