# OSv-MPI: A prototype MPI implementation for the OSv cloud operating system[*]

Joseph Anthony C. Hermocilla
Institute of Computer Science
College of Arts and Sciences
University of the Philippines Los Baños
jchermocilla@up.edu.ph

Eliezer A. Albacea
Institute of Computer Science
College of Arts and Sciences
University of the Philippines Los Baños
eaalbacea@up.edu.ph

## ABSTRACT

In this paper we present OSv-MPI, a prototype MPI implementation to enable HPC applications that use the MPI standard to run on virtual machines with OSv, a new cloud operating system, as guest. OSv-MPI provides a library that can be linked to existing MPI applications and a set of utilities to execute the resulting binaries in an OSv instance. We tested our implementation using simple applications that use the primitives MPI_Send() and MPI_Recv() and show that correct results are obtained. We also present CPU usage statistics while running some of the applications.

## CCS Concepts

**•Networks → Cloud computing; •Computer systems organization → Cloud computing; Client-server architectures; •Software and its engineering → Message passing; Massively parallel systems;**

## 1. INTRODUCTION

Computational scientists usually run their high-performance computing (HPC) applications on dedicated supercomputers or physical clusters. To speed up computations, parallel processing is used. A parallel programming environment is composed of a *job scheduler*, *process manager*, and *parallel library*. The job scheduler describes which resources(compute nodes) a parallel job, which consists of multiple processes, will run. The process manager starts and ends processes associated with the parallel job. The parallel library provides processes a mechanism to communicate[2]. Several parallel programming environments exist but the most popular and widely used is the Message Passing Interface (MPI) standard[4][1]. Open source implementations of MPI exists such as MPICH[2] and OpenMPI[3]. These implementations are available for various hardware architectures and operating systems.

Physical clusters however are expensive to procure, setup, and maintain. They are composed of physical machines (with their own power, CPU, main memory, and secondary storage) that are connected via high-speed networks such as Gigabit Ethernet or Infiniband. Cloud computing, *Infrastructure as a Service (IaaS)* in particular, provides an alternative such that setup and maintenance costs are reduced because of on-demand provisioning of virtual machines(VM)[8][1]. Cloud providers allow users to start and terminate VMs that run general-purpose desktop or server operating systems as guest. VMs are booted with disk images specific to a virtualization product such as KVM[4], Xen[5], or VirtualBox[6]. Figure 1 shows the list of disk images supported by the P2C cloud[6].

A running VM in a cloud is often referred to as *instance*. Virtual clusters are the equivalent of physical clusters in the the cloud. Amazon EC2[7], Microsoft Azure[8], and Google CE[9] are popular commercial or public cloud providers which provide customers access to VMs and clusters for a fee. Private clouds, which are used internally in an organization, are also common in research and academic institutions. These private clouds are deployed using open source cloud frameworks such as OpenStack[10](Figure 2). Instances in the cloud can run HPC applications as long as the guest operating system supports an MPI implementation. Figure 3 shows an example MPI application running on a virtual cluster using MPICH[6].

The adoption of the cloud for running HPC applications however is still limited[9]. Since virtualization adds another layer (the hypervisor) above the physical hardware, some performance degradation is inevitable, particularly in the networking stack. In addition, general-purpose operating systems that run on the virtual machines add more abstraction layers (process management, file system, networking, etc), further degrading performance. To address this, several approaches have been proposed in the literature and

[*]Code:http://srg.ics.uplb.edu.ph/resources/downloads/osv-mpi-source-ncite2016.zip
[1]https://www.mpi-forum.org/docs
[2]https://www.mpich.org
[3]https://www.open-mpi.org
[4]http://www.linux-kvm.org
[5]https://www.xenproject.org
[6]https://www.virtualbox.org
[7]https://aws.amazon.com/ec2
[8]https://azure.microsoft.com
[9]https://cloud.google.com/compute
[10]https://www.openstack.org

**Figure 1: Disk images available on the P2C private cloud.**

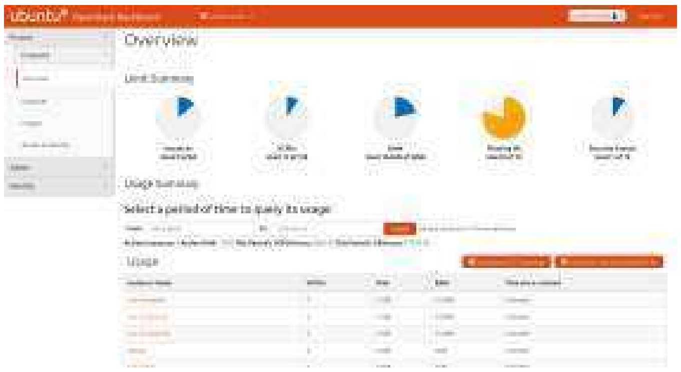


**Figure 2: Dashboard of the P2C private cloud based on OpenStack.**

one of which is the development of new operating systems that is optimized for virtual machines[3]. OSv from Cloudius Systems is one such operating system[7].

Our contribution presented in this paper is OSv-MPI, a prototype MPI implementation to enable HPC applications that use the MPI standard to run on virtual machines with OSv as the guest operating system. A summary of the features of OSv relevant to the design and implementation of OSv-MPI is presented in the next section. The rest of the paper discusses the design and implementation of OSv-MPI, as well as some test results.

## 2. OSV

OSv's design attempts to eliminate unnecessary OS abstractions that are not needed when running applications or services in the cloud. It follows a *one-application-to-one-VM model* and lets the hypervisor performs the process isolation instead of the guest operating system[7]. By using a single address space for all the threads and the kernel itself, costly context switches are eliminated. Other features of OSv that are relevant to our work are the following:

- OSv does not use spinlocks and implement lock-free algorithms in the kernel instead. All work in the kernel are also done in threads using mutexes. This ensures that performance does not suffer.

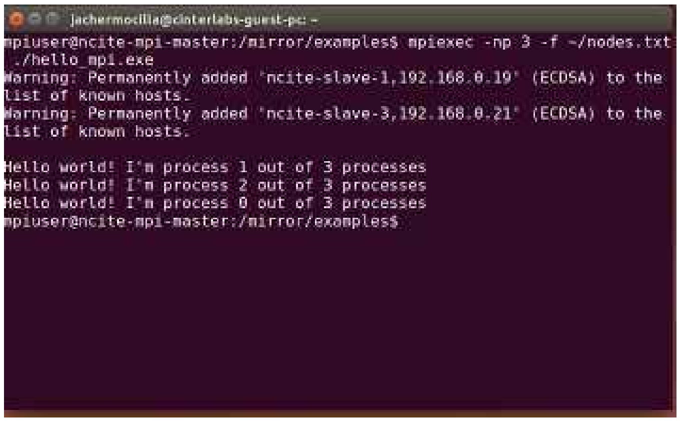
- OSv supports the ELF format and thus can execute Linux binaries linked with glibc. A dynamic linker resolves calls to the Linux ABI to OSv functions implemented in the OSv kernel. In order to provide compatibility, OSv emulates many of the Linux programming interface.

- For the filesystem, OSv has the traditional virtual file system (VFS) design and uses ZFS. Other filesystems are also implemented such as ramfs, devfs, and procfs.

- The networking subsystem of OSv uses network channels. It is a single producer/single consumer queue for transferring packets to the application thread. This reduces lock contention thereby improving network performance. All of these are exposed through the socket API.

- The thread scheduler in OSv is designed to be lock-free, preemptive, tick-less, scalable, and efficient. OSv also supports the pthread library.

- OSv provides a REST API to allow users to interact with a running instance. OSv also has a web based admin interface(Figure 4) to an instance accessible at TCP port 8000 and a command line interface (CLI) (Figure 5) connecting to the same port. The CLI supports a limited set of commands such as *ls* and *cat*.



```
mpiuser@ncite-mpi-master:/mirror/examples$ mpiexec -np 3 -f ~/nodes.txt
 ./hello_mpi.exe
Warning: Permanently added 'ncite-slave-1,192.168.0.19' (ECDSA) to the
list of known hosts.
Warning: Permanently added 'ncite-slave-3,192.168.0.21' (ECDSA) to the
list of known hosts.

Hello world! I'm process 1 out of 3 processes
Hello world! I'm process 2 out of 3 processes
Hello world! I'm process 0 out of 3 processes
mpiuser@ncite-mpi-master:/mirror/examples$
```

**Figure 3: A hello world application running on a virtual MPI cluster deployed on P2C.**

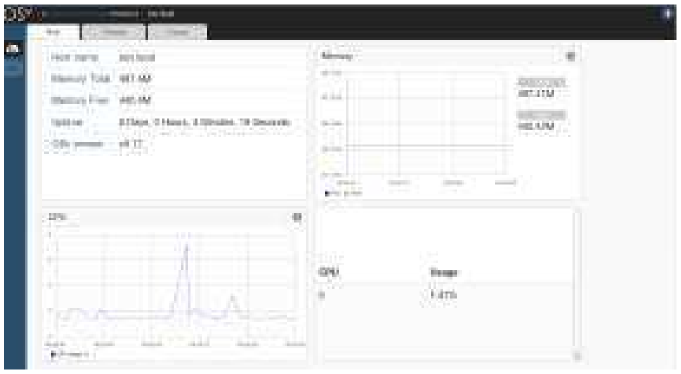
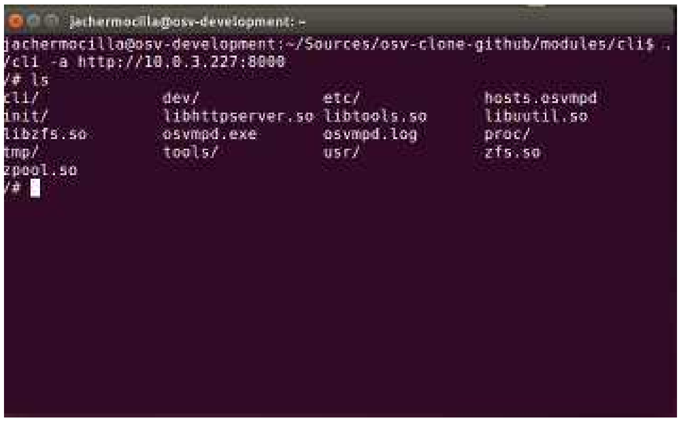


**Figure 4: OSv web admin interface.**

## 3. OSV-MPI DESIGN

OSv-MPI's design allows existing MPI applications to run on OSv with minimal, if no modifications unless there are



```
jachermocilla@osv-development: ~
jachermocilla@osv-development:~/Sources/osv-clone-github/modules/cli$ .
/cli -a http://10.0.3.227:8000
/# ls
cli/            dev/              etc/            hosts.osvmpd
init/           libhttpserver.so  libtools.so     libuutil.so
libzfs.so       osvmpd.exe        osvmpd.log      proc/
tmp/            tools/            usr/            zfs.so
zpool.so
/# 
```

**Figure 5: OSv command line interface via REST API.**

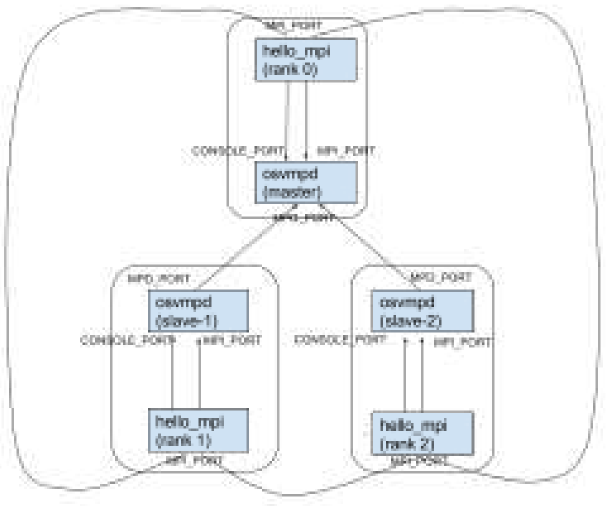


**Figure 6: Architecture of OSv-MPI.**

dependencies that are not fully supported in OSv. Thus, a familiar interface is provided to the users. There are three main components in OSv-MPI, namely *osvmpd*, *osvmpi*, and *bootstrap utilities*. Figure 6 shows the architecture of OSv-MPI. Rounded rectangles represent an OSv instance. Rectangles represent MPI applications and osvmpd processes. We constrained, for now, that only one MPI application connects to an osvmpd process. All communications between the components use TCP sockets. The components are described in the subsections that follow.

## 3.1 osvmpd

osvmpd provides the runtime environment on which MPI applications connect. osvmpd must be started first before MPI applications are run. One osvmpd process is designated as the master while the rest are slaves. At startup, the master reads the hosts file and uploads it, together with the osvmpd binary, to the other nodes, whose addresses are specified in the hosts file. The master then waits for incoming connections from MPI applications or peer osvmpd processes. The slaves on the other hand, once started, load the hosts file to determine the address of the master. Slaves then send messages to the master to get their ranks and the total number of nodes in the cluster. They then wait for connections from MPI applications (Listing 1).

**Listing 1: Function executed by slave osvmpd processes.**

```
void do_slave_mpd(){
  char response[1024];
  char temp_str[256];
  hl=load_hosts("hosts.osvmpd");
  strcpy(master_ip,hl->ip_address[0]);
  send_to(master_ip,MPD_PORT,"GET_MPD_RANK",
        response);
  sprintf(temp_str,"mpd_rank:␣%s",response);
  my_rank=atoi(response);
  send_to(master_ip,MPD_PORT,"GET_NUM_NODES"
        ,response);
  sprintf(temp_str,"num_nodes:␣%s",response);
  num_nodes=atoi(response);
  send_to(master_ip,MPD_PORT,"GET_RANK_TABLE"
        ,response);
  sprintf(temp_str,"rank_table:␣%s",response);
  comm_listen();
}
```

## 3.2 osvmpi

osvmpi is the library to which MPI applications are linked. In Figure 6, *hello_mpi* is the MPI application that is linked to the osvmpi library. In the current version of the prototype, the following functions from the MPI standard are implemented as part of osvmpi (Listing 2).

**Listing 2: MPI functions implemented in OSv-MPI.**

```
int MPI_Init(int *argc, char ***argv);
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Send(const void *buf, int count,
        MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count,
        MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm,
        MPI_Status *status);
int MPI_Finalize( void );
```

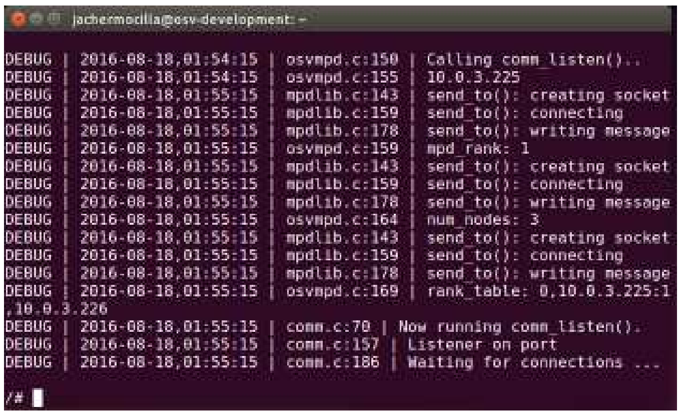
## 3.3 bootstrap utilities

bootstrap utilities, *osvmpd-start.sh* and *mpirun-osv.sh*, enable the MPI applications to be started and executed in OSv. These are equivalent to the traditional *mpdboot* and *mpiexec* commands found in most popular MPI implementations. osvmpd-start.sh starts osvmpd while mpirun-osv.sh executes the MPI application specified as parameter. These scripts make use of *curl*[11] to interact with the REST API of OSv.

## 4. OSV-MPI IMPLEMENTATION

Implementing OSv-MPI presents some challenges. The development machine used is a four-core Intel Core i3-2100

[11]https://curl.haxx.se



```
DEBUG | 2016-08-18,01:54:15 | osvmpd.c:150 | Calling comm_listen()..
DEBUG | 2016-08-18,01:54:15 | osvmpd.c:155 | 10.0.3.225
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:143 | send_to(): creating socket
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:159 | send_to(): connecting
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:178 | send_to(): writing message
DEBUG | 2016-08-18,01:55:15 | osvmpd.c:159 | mpd_rank: 1
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:143 | send_to(): creating socket
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:159 | send_to(): connecting
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:178 | send_to(): writing message
DEBUG | 2016-08-18,01:55:15 | osvmpd.c:164 | num_nodes: 3
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:143 | send_to(): creating socket
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:159 | send_to(): connecting
DEBUG | 2016-08-18,01:55:15 | mpdlib.c:178 | send_to(): writing message
DEBUG | 2016-08-18,01:55:15 | osvmpd.c:169 | rank_table: 0,10.0.3.225:1
,10.0.3.226
DEBUG | 2016-08-18,01:55:15 | comm.c:70 | Now running comm_listen().
DEBUG | 2016-08-18,01:55:15 | comm.c:157 | Listener on port
DEBUG | 2016-08-18,01:55:15 | comm.c:186 | Waiting for connections ...

/# 
```

**Figure 7: Log output contained in osvmpd.log.**

at 3.10GHz with 4GB RAM running Ubuntu 14.04 LTS. OSv was written in C++ and some Lua scripts. The source code for OSv was cloned from Github[12] and version 0.17 was checked out. This version was used since it is the latest version that worked with the P2C used for testing. After cloning, a script was executed which downloaded the dependencies needed to build the OSv VM image. The image was built with the web and the CLI interface modules. After the build, the image (about 26.7MB in size) was uploaded to P2C as *osv-cli-v0.17.jach* (Figure 1). The uploaded image can now be used to create OSv instances for testing. As shown in Figure 2, three instances were created for testing namely, *osv-cli-master*, *osv-cli-slave-01*, and *osv-cli-slave-02*. Each instance is allocated 1 virtual cpu, 512MB of memory, and 21GB disk space.

Binaries for osvmpd and osvmpi were built on the development machine using the *make* utility since there are no development tools inside an OSv instance. The generated binaries work in OSv because of the effort of its developers to provide application binary compatibility with Linux as discussed previously.

### 4.1 Logging

There is no remote terminal access (such SSH) to an OSv instance thus it is difficult to debug OSv-MPI. A logging library, implemented in *log.c*, was created to record events during execution. Sample log output from osvmpd.log is shown in Figure 7.

### 4.2 Hosts file

In order to specify the hosts that belong to a cluster, a *hosts.osvmpd* file must be created which contains the IP addresses of the OSv instances. The first entry in the hosts file is designated as the master node.

### 4.3 REST utilities

The primary interface to an OSv instance is via a REST API, thus utility functions were created to perform HTTP related operations. For instance, to perform file upload, a function using the HTTP POST request method was implemented in *http_utils.c*. This function is used by the master osvmpd process to upload a copy of its binary to slave nodes.

### 4.4 Interprocess Communication

In osvmpd, the communication routines are implemented in the *comm_listen()* function in *comm.c*. Low-level communication is done through TCP sockets using the *select()* system call. This allows multiple clients to connect without using threads. osvmpd processes listen on three ports(Figure 6). MPI_PORT waits for data coming from an MPI process. MPD_PORT listens for data from peer osvmpd processes. CONSOLE_PORT accepts the standard output (stdout) of MPI applications. The output are then consolidated on the master osvmpd process in a file named *stdout.txt*. This serves as the final output of a run.

In osvmpi, the MPI process has a listener thread that waits for data from peer MPI applications. This listener is initialized in the *MPI_Init()* function implementation after the *process_table* structure has been received from the master osvmpd process. process_table maps the rank of an MPI application to the IP address of the node it is running. This enables an MPI application to directly send data to peer MPI applications through the MPI_PORT(Figure 6). Thus, data is no longer passed through the osvmpd process where the MPI applications are attached, reducing transfer time.

## 5. EVALUATION

To test our implementation, we compiled some example MPI applications and linked them to the OSv-MPI library (*osvmpi.a* in Listing 3). We then used the bootstrap utilities to execute them on the three-node OSv cluster in P2C with the IP addresses in hosts.osvmpd. The output shown in the figures were captured by connecting to the CLI of the master node where stdout.txt is generated by the master osvmpd process. Listing 3 shows the commands to build and execute MPI applications using OSv-MPI.

**Listing 3: Compiling, linking, and executing MPI applications on the development machine.**

```
$gcc -std=gnu99 -pie -fpie -rdynamic \
        -Wall -c -o hello_mpi.o hello_mpi.c
$gcc -o hello_mpi.exe \
        -std=gnu99 \
        -pie -fpie \
        -rdynamic \
        -Wall hello_mpi.o -pthread \
        -ljson-c osvmpi.a
$./osvmpd-start.sh
$./mpirun-osv.sh -np 3 -f hosts.osvmpd \
        -x hello_mpi.exe
```

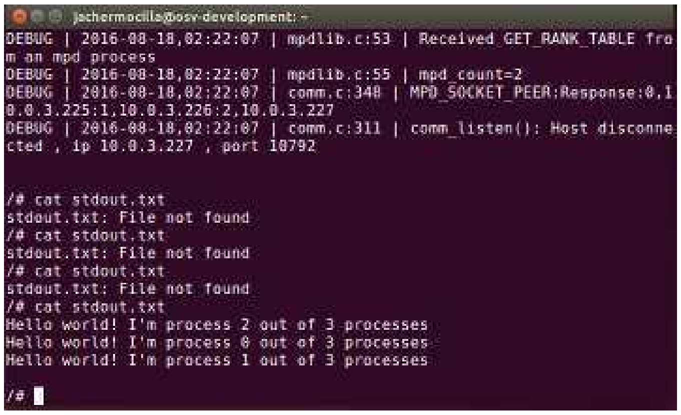
### 5.1 Hello World

The *hello world* application simply outputs the rank of a process. As shown in Figure 8, initially the stdout.txt file is not found. This is because the execution of the MPI applications have not yet completed. On the fourth try, the final output is finally shown.

### 5.2 Send-Receive

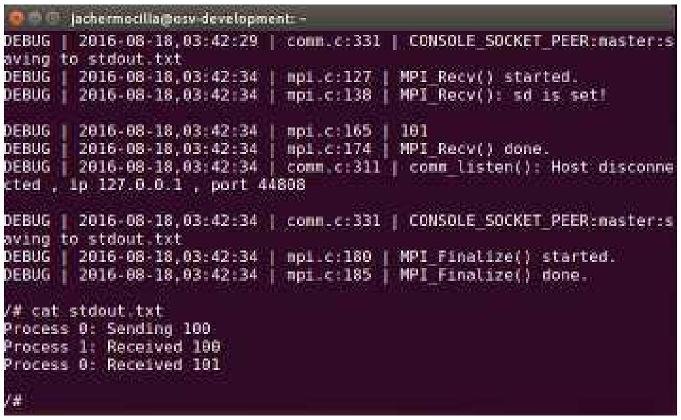
The *send-receive* application shows an example exchange of numbers between two MPI application processes. Process 0 sends 100 to Process 1. Process 1 increments the data it received from Process 0, then sends back the result to Process 0 (Figure 9).

[12]https://github.com/cloudius-systems/osv



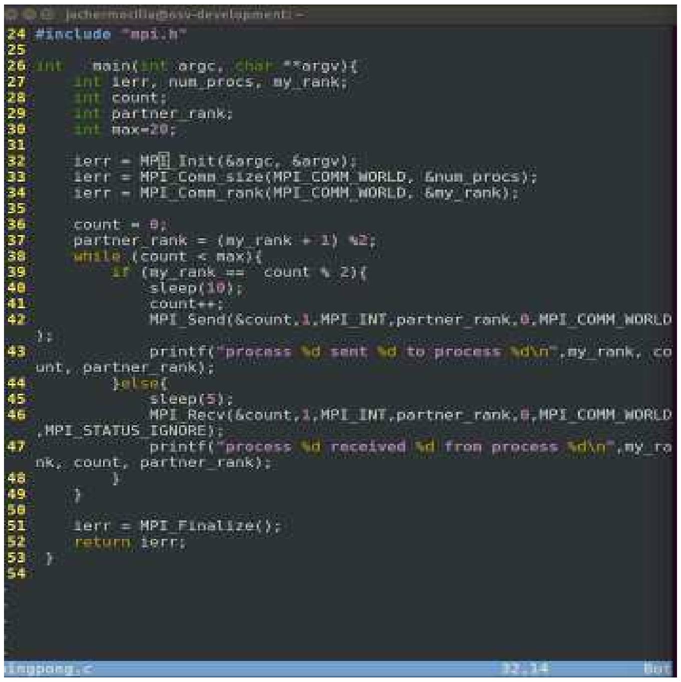
```
jachermocilla@osv-development: ~
DEBUG | 2016-08-18,02:22:07 | mpdlib.c:53 | Received GET_RANK_TABLE fro
m an mpd process
DEBUG | 2016-08-18,02:22:07 | mpdlib.c:55 | mpd_count=2
DEBUG | 2016-08-18,02:22:07 | comm.c:348 | MPD_SOCKET_PEER:Response:0,1
0.0.3.225:1,10.0.3.226:2,10.0.3.227
DEBUG | 2016-08-18,02:22:07 | comm.c:311 | comm_listen(): Host disconne
cted , ip 10.0.3.227 , port 10792


/# cat stdout.txt
stdout.txt: File not found
/# cat stdout.txt
stdout.txt: File not found
/# cat stdout.txt
stdout.txt: File not found
/# cat stdout.txt
Hello world! I'm process 2 out of 3 processes
Hello world! I'm process 0 out of 3 processes
Hello world! I'm process 1 out of 3 processes

/# 
```

**Figure 8: Sample run of the *hello world* application.**



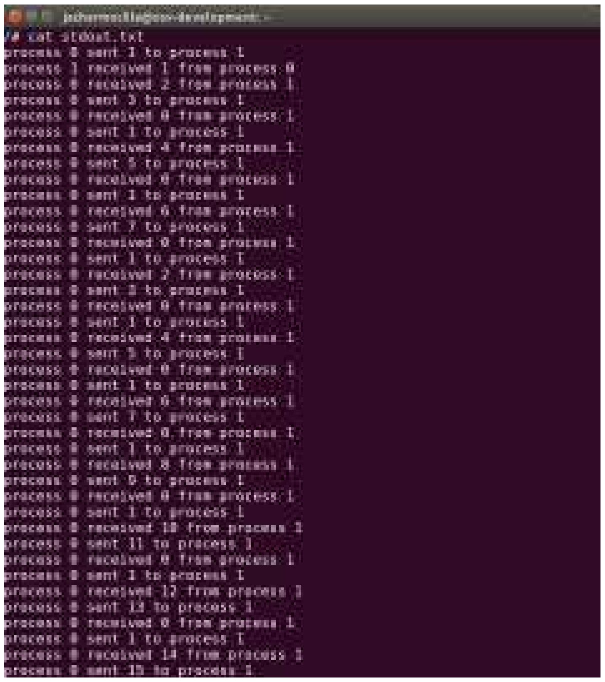
```
jachermocilla@osv-development: ~
DEBUG | 2016-08-18,03:42:29 | comm.c:331 | CONSOLE_SOCKET_PEER:master:s
aving to stdout.txt
DEBUG | 2016-08-18,03:42:34 | mpi.c:127 | MPI_Recv() started.
DEBUG | 2016-08-18,03:42:34 | mpi.c:138 | MPI_Recv(): sd is set!

DEBUG | 2016-08-18,03:42:34 | mpi.c:165 | 101
DEBUG | 2016-08-18,03:42:34 | mpi.c:174 | MPI_Recv() done.
DEBUG | 2016-08-18,03:42:34 | comm.c:311 | comm_listen(): Host disconne
cted , ip 127.0.0.1 , port 44808

DEBUG | 2016-08-18,03:42:34 | comm.c:331 | CONSOLE_SOCKET_PEER:master:s
aving to stdout.txt
DEBUG | 2016-08-18,03:42:34 | mpi.c:180 | MPI_Finalize() started.
DEBUG | 2016-08-18,03:42:34 | mpi.c:185 | MPI_Finalize() done.

/# cat stdout.txt
Process 0: Sending 100
Process 1: Received 100
Process 0: Received 101

/#
```

**Figure 9: Sample run of *send-receive* application.**



```c
jachermocilla@osv-development: ~
#include "mpi.h"

int   main(int argc, char **argv){
    int ierr, num_procs, my_rank;
    int count;
    int partner_rank;
    int max=20;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    count = 0;
    partner_rank = (my_rank + 1) %2;
    while (count < max){
        if (my_rank ==  count % 2){
            sleep(10);
            count++;
            MPI_Send(&count,1,MPI_INT,partner_rank,0,MPI_COMM_WORLD
);
            printf("process %d sent %d to process %d\n",my_rank, co
unt, partner_rank);
        }else{
            sleep(5);
            MPI_Recv(&count,1,MPI_INT,partner_rank,0,MPI_COMM_WORLD
,MPI_STATUS_IGNORE);
            printf("process %d received %d from process %d\n",my_ra
nk, count, partner_rank);
        }
    }

    ierr = MPI_Finalize();
    return ierr;
 }

pingpong.c                                          32,14          Bot
```

**Figure 10: Source for the *pingpong* application.**



```
jachermocilla@osv-development: ~
/# cat stdout.txt
process 0 sent 1 to process 1
process 1 received 1 from process 0
process 0 received 2 from process 1
process 0 sent 3 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 4 from process 1
process 0 sent 5 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 6 from process 1
process 0 sent 7 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 2 from process 1
process 0 sent 3 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 4 from process 1
process 0 sent 5 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 6 from process 1
process 0 sent 7 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 8 from process 1
process 0 sent 9 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 10 from process 1
process 0 sent 11 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 12 from process 1
process 0 sent 13 to process 1
process 0 received 0 from process 1
process 0 sent 1 to process 1
process 0 received 14 from process 1
process 0 sent 15 to process 1
```

**Figure 11: Sample run of the *pingpong* application.**

## 5.3 Pingpong

A more complex example, *pingpong*, performs exchange of data between two processes for 20 iterations. The source and sample run are shown in Figure 10 and Figure 11 respectively.

To get an idea of how much system resources are used, we observed the CPU usage on the master node during the run of the pingpong example through the web admin interface. The result is shown in Figure 12. Other processes were hidden to show only the *osvmpd.exe* process and the *pingpong.exe* process. As can be seen in the figure, >*pingpong.exe* consumes as much as 96.9% of the CPU. >*pingpong.exe* is the thread that listens for connections and keeps the CPU busy.

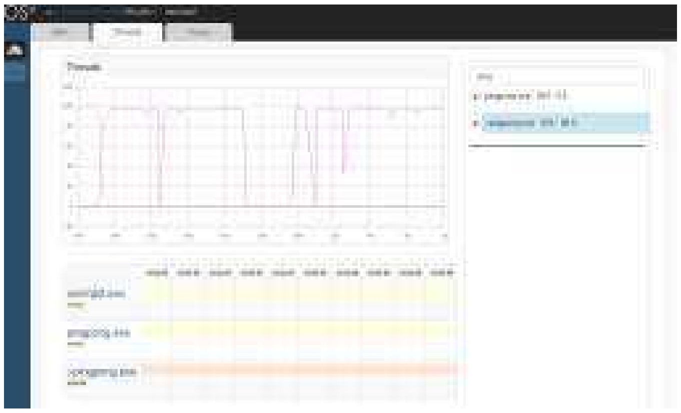


**Figure 12: CPU usage during the execution of the pingpong application.**

## 6. RELATED WORK

There are several existing MPI implementations that are more advanced than OSv-MPI[13]. Some hardware vendors, such as Intel[14], also provide implementations optimized for their machines. In this review of related work, we focus on OpenMPI. OpenMPI addresses issues that are arising from modern machine architectures and networking technologies. Unlike OSv-MPI which supports TCP/IP only, OpenMPI supports other interconnect such as Infiniband, MyriNet, and shared memory. It is also able to gracefully handle network failures. OpenMPI follows a component architecture composed of the MCA backbone, component frameworks, and modules[5].

Porting existing MPI implementations such as OpenMPI to OSv was considered. However, due to the complexity (many abstraction layers) of these implementations, the authors decided to implement a simplified version, OSv-MPI, that adheres to OSv's original design considerations.

## 7. CONCLUSION AND FUTURE WORK

We have shown in this work the possibility of running HPC applications that use the MPI standard can be achieved in OSv through OSv-MPI. This will provide an alternative guest operating system to use when running HPC applications in the cloud.

In the future, additional functions from the MPI standard will be implemented such as MPI_BCast() and MPI_Gather() for collective operations. Also, MPI benchmarks tools need to be executed against OSv-MPI to evaluate its performance and scalability. Finally, security will be enhanced particularly during message transfers.

## 8. ACKOWLEDGMENTS

This work is supported by the Philippine Department of Science and Technology Accelerated Science and Technology Human Resource Development Program. The authors would like to thank Dr. Vladimir Y. Mariano, Dr. Zita VJ. Albacea, Dr. Marlon N. Manalo and the reviewers for their comments.

## 9. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. A view of cloud computing. *Communications of the ACM*, 53(4):50âĂŞ58, 2010.

[2] R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27(11):1417–1429, 2001.

[3] F. Diakhate, M. Perache, R. Namyst, and H. Jourdren. Efficient shared memory message passing for inter-VM communications. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 53–62. Springer, 2008.

[4] C. T. M. Forum. MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, Portland, Oregon, USA, 1993. ACM.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, and others. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 97–104. Springer, 2004.

[6] J. A. C. Hermocilla. P2c: Towards scientific computing on private clouds. In *Proceedings of the 12th National Conference on IT Education (NCITE 2014)*, pages 163–168. Philippine Society of Information Technology Educators, Oct. 2014.

[7] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv-optimizing the operating system for virtual machines. In *2014 usenix annual technical conference (usenix atc 14)*, pages 61–72, 2014.

[8] P. Mell and T. Grance. The NIST definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011.

[9] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of EC2 cloud computing services for scientific computing. In *Cloud Computing*, pages 115–131. Springer, 2010.

[13]http://www.mcs.anl.gov/research/projects/mpi/implementations.html
[14]https://software.intel.com/en-us/intel-mpi-library