Using the Intel® MPI Library on Intel® Xeon Phi™ Coprocessor Systems

Version 1.3

Table of Contents

Chapter 1 – Introduction

This document is designed to help users get started writing code and running Message Passing Interface (MPI) applications (using the Intel® MPI library) on a development platform that includes the Intel® Xeon Phi™ Coprocessor.

More specifically, the Intel® MPI library used in this whitepaper is Intel MPI Library 4.1 for Linux* OS. This Intel MPI Library is used for both the Intel® Xeon and Intel® Xeon Phi™ Coprocessor.

1.1 – Overview

The Intel MPI Library for Linux OS is a multi-fabric message passing library based on ANL* MPICH2* (http://www.mcs.anl.gov/research/projects/mpich2/) and OSU* MVAPICH2* (http://mvapich.cse.ohio-state.edu/download/mvapich2/).

The Intel MPI Library for Linux OS implements the Message Passing Interface, version 2.2 (MPI-2.2) specifications.

It currently supports the Intel® C++ Compiler for Linux OS version 12.1 and higher and the Intel® Fortran Compiler for Linux OS version 12.1 and higher. Users can write their code in C, C++, Fortran 77 and Fortran 90.

1.2 – Compatibility

The Intel MPI Library for Linux OS supports a good variety of Operating Systems, including the following distributions:

- Red Hat* Enterprise Linux 64-bit 6.0 kernel 2.6.32-71
- Red Hat Enterprise Linux 64-bit 6.1 kernel 2.6.32-131
- Red Hat Enterprise Linux 64-bit 6.2 kernel 2.6.32-220
- Red Hat Enterprise Linux 64-bit 6.3 kernel 2.6.32-279
- SUSE* Linux Enterprise Server 11 SP1 kernel 2.6.32.12-0.7-default
- SUSE Linux Enterprise Server 11 SP2 kernel 3.0.13-0.27-default

Depending on the Intel® Many-core Platform System Software (MPSS) running on the above platforms, you need to use the correct compiler (Intel® Composer XE 2013 for Linux OS).

Note that the Intel MPI Library 4.1 for Linux OS supports multiple Intel® Xeon Phi™ coprocessors.

The first part of this whitepaper shows how to install the Intel MPI Library 4.1 on MPSS 2.1. The second part shows how to run some MPI sample code on the Intel® Xeon Phi™ Coprocessor.

Chapter 2 – Installing the Intel® MPI Library

2.1 – Installing the Intel MPI Library

To start, you must follow appropriate directions to install the latest versions of the Intel C/C++ Compiler and the Intel Fortran Compiler. In this paper, the version 2013 is used.

You can purchase these Software Development Tools from http://software.intel.com/en-us/linux-tool-suites. These instructions assume that you have the Intel MPI Library tar file `l_mpi_p_4.1.1.036.tgz`. This is the latest stable release of the library at the time of writing this article. To check if a newer version exists, log into the Intel® Registration Center. The below instructions are valid for all current and subsequent releases.

Untar the tar file `l_mpi_p_4.1.1.036.tgz`:

```
# tar -xzvf l_mpi_p_4.1.1.036.tgz
# cd l_mpi_p_4.1.1.036
# ls
cd_eject.sh
INSTALL.html
install.sh
license.txt
pset
Release_Notes.txt
rpm
SilentInstallConfigFile.ini
sshconnectivity.exp
```

Run the install.sh script and follow the instructions. The installation will be placed, for a specific user, into the installation directory `$HOME/intel/impi/4.1.1.036`. For the root user, it will be installed into the `/opt/intel/impi/4.1.1.036` directory assuming you are installing the library with root permission.

```
# sudo ./install.sh
# ls -l /opt/intel/impi/4.1.1.036
total 208
-rw-r--r-- 1 root root 28556 Aug 31 07:48 Doc_Index.html
-rw-r--r-- 1 root root  9398 Aug 31 07:48 README.txt
lrwxrwxrwx 1 root root     8 Sep 22 17:07 bin -> ia32/bin
lrwxrwxrwx 1 root root    11 Sep 22 17:07 bin64 -> intel64/bin
drwxr-xr-x 2 root root  4096 Sep 22 17:07 binding
drwxr-xr-x 3 root root  4096 Sep 22 17:07 data
drwxr-xr-x 4 root root  4096 Sep 22 17:07 doc
lrwxrwxrwx 1 root root     8 Sep 22 17:07 etc -> ia32/etc
lrwxrwxrwx 1 root root    11 Sep 22 17:07 etc64 -> intel64/etc
drwxr-xr-x 6 root root  4096 Sep 22 17:07 ia32
-rw-r--r-- 1 root root   309 Sep 22 17:07 impi.uninstall.config
lrwxrwxrwx 1 root root    12 Sep 22 17:07 include -> ia32/include
lrwxrwxrwx 1 root root    15 Sep 22 17:07 include64 -> intel64/include
drwxr-xr-x 6 root root  4096 Sep 22 17:07 intel64
lrwxrwxrwx 1 root root     8 Sep 22 17:07 lib -> ia32/lib
lrwxrwxrwx 1 root root    11 Sep 22 17:07 lib64 -> intel64/lib
drwxr-xr-x 3 root root  4096 Sep 22 17:07 man
drwxr-xr-x 6 root root  4096 Sep 22 17:07 mic
-rw-r--r-- 1 root root 28728 Aug 31 07:48 mpi-rtEULA.txt
-rw-r--r-- 1 root root   491 Sep  7 04:12 mpi-rtsupport.txt
-rw-r--r-- 1 root root 28728 Aug 31 07:48 mpiEULA.txt
-rw-r--r-- 1 root root   283 Sep  7 04:12 mpisupport.txt
-rw-r--r-- 1 root root  2770 Aug 31 07:48 redist-rt.txt
-rw-r--r-- 1 root root  1524 Aug 31 07:48 redist.txt
drwxr-xr-x 2 root root  4096 Sep 22 17:07 test
-rw-r--r-- 1 root root  7762 Sep 22 15:28 uninstall.log
-rwxr-xr-x 1 root root 41314 Sep  7 04:12 uninstall.sh
```

2.2 – Preparation

Before the first run of an MPI application on the coprocessors, copy the MPI libraries to the following directories on each Intel® Xeon Phi™ coprocessor equipped on this system. In this example, we issue the copy to two coprocessors: the first coprocessor is accessible via the IP address 172.31.1.1 and the second coprocessor has 172.31.2.1 as its IP address. Note that all coprocessors have unique IP addresses since they are treated as just other uniquely addressable machines. You can refer to the first coprocessor as mic0 or its IP address; similarly, you can refer to the second coprocessor as mic1 or its IP address).

```
# sudo scp /opt/intel/impi/4.1.1.036/mic/bin/* mic0:/bin/
mpiexec                                                     100% 1061KB   1.0MB/s   00:00
pmi_proxy                                                   100%  871KB 871.4KB/s   00:00
...
# sudo scp /opt/intel/impi/4.1.1.036/mic/lib/* mic0:/lib64/
libmpi.so.4.1                                               100% 4391KB   4.3MB/s   00:00
libmpigf.so.4.1                                             100%  321KB 320.8KB/s   00:00
libmpigc4.so.4.1                                            100%  175KB 175.2KB/s   00:00
...
# sudo scp /opt/intel/composer_xe_2013.4.183/compiler/lib/mic/* mic0:/lib64/
libimf.so                                                   100% 2516KB   2.5MB/s   00:01
libsvml.so                                                  100% 4985KB   4.9MB/s   00:01
libintlc.so.5                                               100%  128KB 128.1KB/s   00:00
...
# sudo scp /opt/intel/impi/4.1.1.036/mic/bin/* mic1:/bin/
mpiexec                                                     100% 1061KB   1.0MB/s   00:00
pmi_proxy                                                   100%  871KB 871.4KB/s   00:00
# sudo scp /opt/intel/impi/4.1.1.036/mic/lib/* mic1:/lib64/
libmpi.so.4.1                                               100% 4391KB   4.3MB/s   00:00
libmpigf.so.4.1                                             100%  321KB 320.8KB/s   00:00
libmpigc4.so.4.1                                            100%  175KB 175.2KB/s   00:00
...
# sudo scp /opt/intel/composer_xe_2013.4.183/compiler/lib/mic/* mic1:/lib64/
libimf.so                                                   100% 2516KB   2.5MB/s   00:01
libsvml.so                                                  100% 4985KB   4.9MB/s   00:01
libintlc.so.5                                               100%  128KB 128.1KB/s   00:00
...
```

Instead of copying the MPI libraries manually, you can also run the script below

```
#!/bin/sh

export COPROCESSORS="mic0 mic1"
export BINDIR="/opt/intel/impi/4.1.1.036/mic/bin"
export LIBDIR="/opt/intel/impi/4.1.1.036/mic/lib"
export COMPILERLIB="/opt/intel/composer_xe_2013.4.183/compiler/lib/mic"

for coprocessor in `echo $COPROCESSORS`
```

```
do
   for prog in mpiexec mpiexec.hydra pmi_proxy mpirun
   do
      sudo scp $BINDIR/$prog $coprocessor:/bin/$prog
   done

   for lib in libmpi.so.4.1 libmpigf.so.4.1 libmpigc4.so.4.1
   do
      sudo scp $LIBDIR/$lib $coprocessor:/lib64/$lib
   done

   for lib in libimf.so libsvml.so libintlc.so.5
   do
      sudo scp $COMPILERLIB/$lib $coprocessor:/lib64/$lib
   done
done
```

For multi-card usage, configure MPSS peer-to-peer:

```
# sudo /sbin/sysctl -w net.ipv4.ip_forward=1
```

Chapter 3 – Compiling and Running the Sample MPI Program

This section includes a sample MPI program written in C. We will show how to compile and run the program for the host and also for the Intel® Xeon Phi™ Coprocessor.

Intel® MPI Library supports three programming models:

- **Co-processor only model**: in this native mode, the MPI ranks reside solely inside the coprocessor. The application can be launched from the host or the coprocessor.
- **Symmetric model**: in this mode, the MPI ranks reside on the host and the coprocessors.
- **MPI Offload model**: in this mode, the MPI ranks reside solely on the host. The MPI ranks use offload capabilities of the Intel® C/C++ Compiler or Intel® Fortran Compiler to offload some workloads to the coprocessors.

For illustration purposes, the following example shows how to build and run an MPI application in symmetric model.

The sample program estimates the calculation of Pi (π) using a Monte Carlo method. Consider a sphere centered at the origin and circumscribed by a cube: the sphere's radius is r and the cube edge length is 2r. The volumes of a sphere and a cube are given by

$$V_{sphere} = \frac{4\pi r^3}{3}$$

$$V_{cube} = (2r)^3 = 8r^3$$

The first octant of the coordinate system contains one eighth of the volumes of both the sphere and the cube; the volumes in that octant are given by:

$$v_{sphere} = \frac{\pi r^3}{6}$$

$$v_{cube} = r^3$$

If we generate N_c points uniformly and randomly in the cube within this octant, we expect that about N_s points will be inside the volume of sphere according to the following ratio:

$$\frac{Nc}{Ns} = \frac{6r^3}{\pi r^3} = \frac{6}{\pi}$$

Therefore, the estimated Pi (π) is calculated by

$$\pi = \frac{6\,Ns}{Nc}$$

where N_c is the number of points generated in the portion of the cube residing in the first octant, and N_s is the total number of points found inside the portion of the sphere residing in the first octant.

In the implementation, rank 0 (process) is responsible for dividing the work among the other *n* ranks. Each rank is assigned a chunk of work, and the summation is used to estimate the number Pi. Rank 0 divides the *x-axis* into *n* equal segments. Each rank generates (*N_C/n)* points in the assigned segment, and then computes the number of points in the first octant of the sphere.

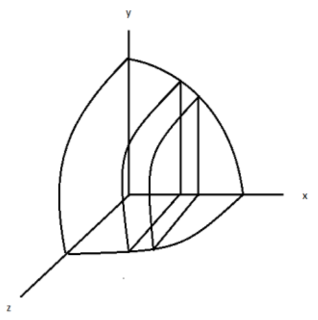

Figure 1 – Each rank handles a separate slide in the first octant.

The pseudo code is shown below

```
        Rank 0 generate n random seed
```

```
Rank 0 broadcast all random seeds to n rank
For each rank i [0, n-1]
       receive the corresponding seed
       set num_inside = 0
       For j=0 to $Nc$ / n
              generate a point with coordinates
                     x between [i/n, (i+1)/n]
                     y between [0, 1]
                     z between [0, 1]
              compute the distance d = x^2 + y^2 + z^2
              if distance d <= 1, increment num_inside
       Send num_inside back to rank 0
Rank 0 set $Ns$ to the sum of all num_inside
Rank 0 compute Pi = 6 * $Ns$ / $Nc$
```

Before compiling the program, called `montecarlo.c`, you need to establish the proper environment settings for the compiler and for the Intel MPI Library for Intel® Xeon Phi™ Coprocessor

```
# source /opt/intel/composer_xe_2013.4.183/bin/compilervars.sh intel64
# source /opt/intel/impi/4.1.1.036/ bin64/mpivars.sh
```

Build the application `montecarlo.mic` for the coprocessor:

```
# mpiicc –mmic montecarlo.c -o montecarlo.mic
```

Build the application for the host:

```
# mpiicc montecarlo.c -o montecarlo.host
```

Upload the application `montecarlo.mic` to the `/tmp` directory on the coprocessors using the `scp` command. In this example, we issue the copy to two coprocessors.

```
# sudo scp ./montecarlo.mic mic0:/tmp/montecarlo.mic
montecarlo.mic                                   100%   15KB  15.5KB/s   00:00
# sudo scp ./montecarlo.mic mic1:/tmp/montecarlo.mic
montecarlo.mic                                   100%   15KB  15.5KB/s   00:00
```

Enable the MPI communication between host and coprocessors:

```
# export I_MPI_MIC=enable
```

The command `mpirun` starts the application. Also, the flag `–n` specifies the number of MPI processes and the flag `–host` specifies the machine name:

```
# mpirun –n <# of processes>  -host <hostname> <application>
```

We can run the application on multiple hosts by separating them by ":". The first MPI rank (rank 0) always starts on the first part of the command:

```
# mpirun –n <# of processes>  -host <hostname1> <application> : –n <# of
processes>  -host <hostname2> <application>
```

This should start the rank 0 on `hostname1`.

Now run the application on the host. The `mpirun` command shown below starts the application with 2 ranks on the host, 3 ranks on the coprocessor MIC0 and 5 ranks on coprocessor MIC1:

```
# mpirun -n 2 -host knightscorner1 ./montecarlo.host \
: -n 3 -host mic0 /tmp/montecarlo.mic \
: -n 5 -host mic1 /tmp/montecarlo.mic
Hello world: rank 0 of 10 running on knightscorner1
Hello world: rank 1 of 10 running on knightscorner1
Hello world: rank 2 of 10 running on knightscorner1-mic0
Hello world: rank 3 of 10 running on knightscorner1-mic0
Hello world: rank 4 of 10 running on knightscorner1-mic0
Hello world: rank 5 of 10 running on knightscorner1-mic1
Hello world: rank 6 of 10 running on knightscorner1-mic1
Hello world: rank 7 of 10 running on knightscorner1-mic1
Hello world: rank 8 of 10 running on knightscorner1-mic1
Hello world: rank 9 of 10 running on knightscorner1-mic1
Elapsed time from rank 0:      14.79 (sec)
Elapsed time from rank 1:      14.90 (sec)
Elapsed time from rank 2:     219.87 (sec)
Elapsed time from rank 3:     218.24 (sec)
Elapsed time from rank 4:     218.29 (sec)
Elapsed time from rank 5:     218.94 (sec)
Elapsed time from rank 6:     218.74 (sec)
Elapsed time from rank 7:     218.42 (sec)
Elapsed time from rank 8:     217.93 (sec)
Elapsed time from rank 9:     217.35 (sec)
Out of 4294967295 points, there are 2248861895 points inside the sphere =>
pi=  3.141623973846
```

A short-hand way of doing this in symmetric mode will be to use the `–machinefile` option for the `mpirun` command in coordination with the `I_MPI_MIC_POSTFIX` environment variable. In this case, make sure all executables are in the same location on the host and MIC0 and MIC1 cards.

The `I_MPI_MIC_POSTFIX` environment variable simply tells the library to add the .mic postfix when running on the cards (since the executables there are called `montecarlo.mic`).

```
# export I_MPI_MIC_POSTFIX=.mic
```

Now set the rank mapping in your hosts file (by using the *<host>:<#_ranks>* format):

```
# cat hosts_file
knightscorner1:2
mic0:3
mic1:5
```

And run your executable:

```
# cp ./montecarlo.host /tmp/montecarlo
# mpirun -machinefile hosts_file /tmp/montecarlo
```

The nice thing about this syntax is that you only have to edit the `hosts_file` when deciding to change your number of ranks, or need to add more cards.

From the host, you can alternately launch the application running only on the coprocessors `mic0` and `mic1`:

```
# mpirun -n 3 -host mic0 /tmp/montecarlo.mic : -n 5 -host mic1 \
/tmp/montecarlo.mic
Hello world: rank 0 of 8 running on knightscorner1-mic0
Hello world: rank 1 of 8 running on knightscorner1-mic0
Hello world: rank 2 of 8 running on knightscorner1-mic0
Hello world: rank 3 of 8 running on knightscorner1-mic1
Hello world: rank 4 of 8 running on knightscorner1-mic1
Hello world: rank 5 of 8 running on knightscorner1-mic1
Hello world: rank 6 of 8 running on knightscorner1-mic1
Hello world: rank 7 of 8 running on knightscorner1-mic1
Elapsed time from rank 0:     273.71 (sec)
Elapsed time from rank 1:     273.20 (sec)
Elapsed time from rank 2:     273.66 (sec)
Elapsed time from rank 3:     273.84 (sec)
Elapsed time from rank 4:     273.53 (sec)
Elapsed time from rank 5:     273.24 (sec)
Elapsed time from rank 6:     272.59 (sec)
Elapsed time from rank 7:     271.64 (sec)
Out of 4294967295 points, there are 2248861679 points inside the sphere =>
pi= 3.141623497009
```

As an alternative, you can `ssh` to the coprocessor `mic0` and launch the application from there:

```
# ssh mic0
# mpirun -n 3 /tmp/montecarlo.mic
Hello world: rank 0 of 3 running on knightscorner1-mic0
Hello world: rank 1 of 3 running on knightscorner1-mic0
Hello world: rank 2 of 3 running on knightscorner1-mic0
Elapsed time from rank 0:     732.09 (sec)
Elapsed time from rank 1:     727.86 (sec)
Elapsed time from rank 2:     724.82 (sec)
Out of 4294967295 points, there are 2248845386 points inside the sphere =>
pi=  3.141600608826
```

This section showed how to compile and run a simple MPI application in symmetric model. In a heterogeneous computing system, the performance in each computational unit is different and this system behavior leads to the load imbalance problem. The Intel® Trace Analyzer and Collector (or ITAC, in http://software.intel.com/en-us/intel-trace-analyzer) can be used to analyze and understand the behavior of a complex MPI program running on a heterogeneous system. Using ITAC, users can quickly identify bottlenecks, evaluate load balancing, analyze performance, and identify communication hotspots. This powerful tool is essential to debug and improve the performance of a MPI program running on a cluster with multiple computational units. For more details on using ITAC, users are encouraged to read the whitepaper "Understanding MPI Load Imbalance with Intel® Trace Analyzer and Collector" available on http://software.intel.com/mic-developer .

Appendix

The code of the sample MPI program is shown below

```
/*
// Copyright 2003-2012 Intel Corporation. All Rights Reserved.
//
// The source code contained or described herein and all documents related
// to the source code ("Material") are owned by Intel Corporation or its
// suppliers or licensors.  Title to the Material remains with Intel Corporation
// or its suppliers and licensors.  The Material is protected by worldwide
// copyright and trade secret laws and treaty provisions.  No part of the
// Material may be used, copied, reproduced, modified, published, uploaded,
// posted, transmitted, distributed, or disclosed in any way without Intel's
// prior express written permission.
//
// No license under any patent, copyright, trade secret or other intellectual
// property right is granted to or conferred upon you by disclosure or delivery
// of the Materials, either expressly, by implication, inducement, estoppel
// or otherwise.  Any license under such intellectual property rights must
// be express and approved by Intel in writing.


#******************************************************************************
# Content: (version 0.5)
#      Based on a Monto Carlo method, this MPI sample code uses volumes to
#      estimate the number PI.
#
#*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <math.h>

#include "mpi.h"

#define MASTER 0
#define TAG_HELLO 4
#define TAG_TEST 5
#define TAG_TIME 6

int main(int argc, char *argv[])
{
  int i, id, remote_id, num_procs;

  MPI_Status stat;
  int namelen;
  char name[MPI_MAX_PROCESSOR_NAME];

  // Start MPI.
  if (MPI_Init (&argc, &argv) != MPI_SUCCESS)
    {
      printf ("Failed to initialize MPI\n");
      return (-1);
    }
  // Create the communicator, and retrieve the number of processes.
  MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

  // Determine the rank of the process.
  MPI_Comm_rank (MPI_COMM_WORLD, &id);

  // Get machine name
  MPI_Get_processor_name (name, &namelen);

  if (id == MASTER)
    {
      printf ("Hello world: rank %d of %d running on %s\n", id, num_procs, name);

      for (i = 1; i<num_procs; i++)
       {
         MPI_Recv (&remote_id, 1, MPI_INT, i, TAG_HELLO, MPI_COMM_WORLD, &stat);
         MPI_Recv (&num_procs, 1, MPI_INT, i, TAG_HELLO, MPI_COMM_WORLD, &stat);

         MPI_Recv (&namelen, 1, MPI_INT, i, TAG_HELLO, MPI_COMM_WORLD, &stat);

         MPI_Recv (name, namelen+1, MPI_CHAR, i, TAG_HELLO, MPI_COMM_WORLD,
&stat);

         printf ("Hello world: rank %d of %d running on %s\n", remote_id,
num_procs, name);
       }
    }
  else
    {
      MPI_Send (&id, 1, MPI_INT, MASTER, TAG_HELLO, MPI_COMM_WORLD);
      MPI_Send (&num_procs, 1, MPI_INT, MASTER, TAG_HELLO, MPI_COMM_WORLD);
      MPI_Send (&namelen, 1, MPI_INT, MASTER, TAG_HELLO, MPI_COMM_WORLD);
      MPI_Send (name, namelen+1, MPI_CHAR, MASTER, TAG_HELLO, MPI_COMM_WORLD);
    }

   // Rank 0 distributes seek randomly to all processes.
  double startprocess, endprocess;

  int distributed_seed = 0;
  int *buff;

  buff = (int *)malloc(num_procs * sizeof(int));

  unsigned int MAX_NUM_POINTS = pow (2,32) - 1;
  unsigned int num_local_points = MAX_NUM_POINTS / num_procs;

  if (id == MASTER)
    {
      srand (time(NULL));

      for (i=0; i<num_procs; i++)
       {
         distributed_seed = rand();
         buff[i] = distributed_seed;
       }
    }
  // Broadcast the seed to all processes
  MPI_Bcast(buff, num_procs, MPI_INT, MASTER, MPI_COMM_WORLD);

  // At this point, every process (including rank 0) has a different seed. Using
their seed,
  // each process generates N points randomly in the interval [1/n, 1, 1]
  startprocess = MPI_Wtime();

  srand (buff[id]);

  unsigned int point = 0;
  unsigned int rand_MAX = 128000;
  float p_x, p_y, p_z;
  float temp, temp2, pi;
  double result;
  unsigned int inside = 0, total_inside = 0;

  for (point=0; point<num_local_points; point++)
    {
      temp = (rand() % (rand_MAX+1));
      p_x = temp / rand_MAX;
      p_x = p_x / num_procs;

      temp2 = (float)id / num_procs;    // id belongs to 0, num_procs-1
      p_x += temp2;

      temp = (rand() % (rand_MAX+1));
      p_y = temp / rand_MAX;

      temp = (rand() % (rand_MAX+1));
      p_z = temp / rand_MAX;

      // Compute the number of points residing inside of the 1/8 of the sphere
      result = p_x * p_x + p_y * p_y + p_z * p_z;

      if (result <= 1)
         {
              inside++;
         }
    }

  double elapsed = MPI_Wtime() - startprocess;

  MPI_Reduce (&inside, &total_inside, 1, MPI_UNSIGNED, MPI_SUM, MASTER,
MPI_COMM_WORLD);


#if DEBUG
  printf ("rank %d counts %u points inside the sphere\n", id, inside);
#endif
  if (id == MASTER)
    {
      double timeprocess[num_procs];

      timeprocess[MASTER] = elapsed;
      printf("Elapsed time from rank %d: %10.2f (sec) \n", MASTER,
timeprocess[MASTER]);
```

```
      for (i=1; i<num_procs; i++)
       {
         // Rank 0 waits for elapsed time value
         MPI_Recv (&timeprocess[i], 1, MPI_DOUBLE, i, TAG_TIME, MPI_COMM_WORLD,
&stat);
         printf("Elapsed time from rank %d: %10.2f (sec) \n", i, timeprocess[i]);
       }

      temp = 6 * (float)total_inside;
      pi = temp / MAX_NUM_POINTS;
      printf ( "Out of %u points, there are %u points inside the sphere =>
pi=%16.12f\n", MAX_NUM_POINTS, total_inside, pi);
    }
  else
    {
      // Send back the processing time (in second)
      MPI_Send (&elapsed, 1, MPI_DOUBLE, MASTER, TAG_TIME, MPI_COMM_WORLD);
    }

  free(buff);

  // Terminate MPI.
  MPI_Finalize();

  return 0;
}
```

About the Author

Loc Q Nguyen received an MBA from University of Dallas, a master's degree in Electrical Engineering from McGill University, and a bachelor's degree in Electrical Engineering from École Polytechnique de Montréal. He is currently a software engineer with Intel Corporation's Software and Services Group. His areas of interest include computer networking, computer graphics, and parallel processing.

Notices