この記事は、インテル® デベロッパー・ゾーンに掲載されている「Using the Intel® MPI Library on Intel® Xeon Phi™ Coprocessor Systems」の日本語参考訳です。

インテル® Xeon Phi™ コプロセッサー・システムにおけるインテル® MPI ライブラリーの使用

目次

第 1 章 – はじめに

この記事は、インテル® Xeon Phi™ コプロセッサーを搭載する開発プラットフォームで (インテル® MPI ライブラリーを利用して) メッセージ・パッシング・インターフェース (MPI) アプリケーションのコードを記述して実行する開発者向けに作成したものです。

この記事ではインテル® MPI ライブラリー 4.1 Linux* 版を使っています。インテル® MPI ライブラリーは、インテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ コプロセッサーの両方で動作します。

1.1 – 概要

インテル® MPI ライブラリー Linux* 版は、ANL* MPICH2* (http://www.mcs.anl.gov/research/projects/mpich2/) および OSU* MVAPICH2* (http://mvapich.cse.ohio-state.edu/download/mvapich2/) をベースとするマルチファブリック・メッセージ・パッシング・ライブラリーで、MPI-2.2 仕様を実装しています。

現在、インテル® MPI ライブラリー Linux* 版は、インテル® C++ コンパイラー Linux* 版 (バージョン 12.1 以降) およびインテル® Fortran コンパイラー Linux* 版 (バージョン 12.1 以降) をサポートしています。コードは、C、C++、Fortran 77 および Fortran 90 で記述できます。

1.2 – 互換性

インテル® MPI ライブラリー Linux* 版は、以下のディストリビューションを含む、さまざまなオペレーティング・システムをサポートしています。

- Red Hat* Enterprise Linux 64 ビット 6.0 カーネル 2.6.32-71
- Red Hat* Enterprise Linux 64 ビット 6.1 カーネル 2.6.32-131
- Red Hat* Enterprise Linux 64 ビット 6.2 カーネル 2.6.32-220
- Red Hat* Enterprise Linux 64 ビット 6.3 カーネル 2.6.32-279
- SUSE* Linux Enterprise Server 11 SP1 カーネル 2.6.32.12-0.7-default
- SUSE* Linux Enterprise Server 11 SP2 カーネル 3.0.13-0.27-default

上記のプラットフォームで実行するインテル® メニーコア・プラットフォーム・システム・ソフトウェア (MPSS) に応じて、適切なコンパイラー (インテル® Composer XE 2013 Linux* 版) を使用する必要があります。

インテル® MPI ライブラリー 4.1 Linux* 版は複数のインテル® Xeon Phi™ コプロセッサーをサポートしていることに注意してください。

第 2 章では、MPSS 2.1 にインテル® MPI ライブラリー 4.1 をインストールする方法を説明します。第 3 章では、インテル® Xeon Phi™ コプロセッサーで MPI サンプルコードを実行する方法を説明します。

第 2 章 – インテル® MPI ライブラリーのインストール

2.1 – インテル® MPI ライブラリーのインストール

最初に、インテル® C/C++ コンパイラーおよびインテル® Fortran コンパイラーの最新バージョンを適切にインストールする必要があります。この記事では、バージョン 2013 を使用しています。

これらのソフトウェア開発ツールについては、
http://www.xlsoft.com/jp/products/intel/products.html を参照してください。以下の手順は、インテル® MPI ライブラリーの l_mpi_p_4.1.0.024.tgz ファイルを入手済みであることを前提としています。

l_mpi_p_4.1.0.024.tgz ファイルを展開します。

```
# 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
```

install.sh スクリプトを実行して指示に従います。一般ユーザーの場合、インストール・ディレクトリーは $HOME/intel/impi/4.1.0.024 になります。root ユーザーの場合は、/opt/intel/impi/4.1.0.024 ディレクトリーにインストールされます。

```
# 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 – 前処理

コプロセッサー上で MPI アプリケーションを最初に実行する前に、各インテル® Xeon Phi™ コプロセッサーの次のディレクトリーに MPI ライブラリーをコピーします。この例では、2 つのコプロセッサー・カードにコピーしています。最初のコプロセッサーは IP アドレス 172.31.1.1 でアクセスし、2 つ目のコプロセッサーは IP アドレス 172.31.2.1 でアクセスします。すべてのコプロセッサーには一意の IP アドレスが割り当てられるため、個々にアクセスできます。最初のプロセッサーは mic0 として、またはその IP アドレスで参照できます。同様に、2 つ目のプロセッサーは mic1 として、またはその IP アドレスで参照できます。

```
# 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
...
```

MPI ライブラリーを手動でコピーする代わりに、次のスクリプトを実行することもできます。実際にインストールするバージョンに合わせて、ディレクトリー名を変更してください。

```
#!/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
```

複数のカードを使用している場合は、MPSS をピア・ツー・ピア型に設定します。

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

第 3 章 – サンプル MPI プログラムのコンパイルと実行

このセクションでは、C で記述されたサンプル MPI プログラムを、ホストおよびインテル® Xeon Phi™ コプロセッサー向けにコンパイルして実行する方法を説明します。

インテル® MPI ライブラリーは 3 つのプログラミング・モデルをサポートします。

- **コプロセッサーのみのモデル**: このネイティブモードでは、MPI ランクはコプロセッサーにのみ存在します。アプリケーションはホストまたはコプロセッサーから起動できます。
- **シンメトリック・モデル**: このモードでは、MPI ランクはホストとコプロセッサーに存在します。
- **MPI オフロードモデル**: このモードでは、MPI ランクはホストにのみ存在します。MPI ランクは、インテル® C/C++ コンパイラーまたはインテル® Fortran コンパイラーのオフロード機能によってワークロードをコプロセッサーへオフロードします。

次に、シンメトリック・モデルで MPI アプリケーションをビルドして実行する方法を示します。

このサンプルプログラムはモンテカルロ法を用いて Pi(π) の計算を推定します。原点を中心とする立方体で囲まれた球体を考えます。球体の半径は r で、立方体のエッジ長は 2r とします。球体と立方体の容積は次の式で表現できます。

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

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

座標系の最初の八分円には球体と立方体両方の容積の 1/8 が含まれます。八分円の容積は次の式で表現できます。

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

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

この八分円内の立方体に N_c 個のポイントを一様にランダムで生成すると、次の比率に従って、約 N_s 個のポイントが球体の容積の内部に含まれることが想定されます。

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

Pi (π) の推定値は次の式で計算できます。

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

ここで、N_c は最初の八分円に存在する立方体の部分に生成されるポイントの数、N_s は最初の八分円に存在する球体の部分に含まれるポイントの総数です。

この実装では、ランク 0 (プロセス) はほかの n ランクにワークを分割します。各ランクにはワークの一部が割り当てられ、合計 (sum) は数 Pi を求めるのに使用されます。ランク 0 は、x 軸を等しい n 個のセグメントに分割します。各ランクは割り当てられたセグメントに (N_C / n) 個のポイントを生成した後、球体の最初の八分円に含まれるポイントの数を計算します。

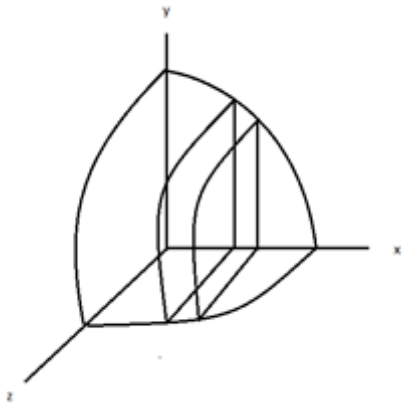

図 1 – 各ランクで最初の八分円の個々の部分を処理する

擬似コードを以下に示します。

```
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
```

プログラム montecarlo.c をコンパイルする前に、インテル® Xeon Phi™ コプロセッサー向けにコンパイラーおよびインテル® MPI ライブラリーの環境を設定する必要があります。実際にインストールされているバージョンに合わせて、ディレクトリー名を変更してください。

```
# source /opt/intel/composer_xe_2013.0.079/bin/compilervars.sh
intel64
# source /opt/intel/impi/4.1.0.024/mic/bin/mpivars.sh
```

コプロセッサー用アプリケーション montecarlo.mic をビルドします。

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

ホストで実行するアプリケーションをビルドするには、インテル® 64 対応のインテル® MPI ライブラリーの環境を設定する必要があります。実際にインストールされているバージョンに合わせて、ディレクトリー名を変更してください。

```
# source /opt/intel/impi/4.1.0.024/intel64/bin/mpivars.sh
```

ホスト用アプリケーションをビルドします。

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

scp コマンドでバイナリー (montecarlo.mic) をコプロセッサーの /tmp ディレクトリーにアップロードします。この例では、2 つのコプロセッサーにコピーします。

```
# 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
```

ホストとコプロセッサー間の MPI 通信を有効にします。

```
# export I_MPI_MIC=enable
```

mpirun コマンドでアプリケーションを開始します。また、–n オプションで MPI プロセスの数を、–host オプションでマシン名を指定します。

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

複数のホスト (“:” で区切って指定) でアプリケーションを実行することができます。最初の MPI ランク (ランク 0) は常にコマンドの最初に指定します。

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

上記の例は、hostname1 でランク 0 を開始します。

次に、ホストでアプリケーションを実行します。次の mpirun コマンドは、ホストで 2 つのランク、コプロセッサー MIC0 で 3 つのランク、コプロセッサー MIC1 で 5 つのランクのアプリケーションを開始します。

```
# 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
```

シンメトリック・モデルでこの処理を簡単に行うには、mpirun コマンドの –machinefile オプションと I_MPI_MIC_POSTFIX 環境変数を併せて使用します。この場合、すべての実行ファイルが、ホスト、MIC0 カードおよび MIC1 カードの同じ位置に存在することを確認してください。

I_MPI_MIC_POSTFIX 環境変数は、カードで実行するときに .mic 接尾辞を追加するようにライブラリーに指示します (実行ファイルの名前は montecarlo.mic になるため)。

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

(<*host*>:<*#_ranks*> 形式で) ホストファイルにランクのマッピングを設定します。

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

実行ファイルを実行します。

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

この構文の便利な点は、ランクの数を変更する場合、あるいはカードを追加する場合、hosts_file を編集するだけで済むことです。

ホストから、コプロセッサー mic0 および mic1 でのみ動作するアプリケーションを交互に起動できます。

```
# 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
```

別の方法として、コプロセッサー mic0 に ssh を使用してアプリケーションを起動することもできます。

```
# 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
```

このセクションでは、シンメトリック・モデルで簡単な MPI アプリケーションをコンパイルして実行する方法を説明しました。ヘテロジニアス・コンピューティング・システムでは、各計算ユニットのパフォーマンスが異なるため、このシステム動作がロード・インバランスの原因となります。ヘテロジニアス・システムで実行する複雑な MPI プログラムの動作を解析して理解するには、インテル® Trace Analyzer & Collector (ITAC、http://software.intel.com/en-us/intel-trace-analyzer) を利用すると良いでしょう。このツールにより、ボトルネックをすぐに識別し、ロードバランスを評価し、パフォーマンスを解析し、通信 hotspot を特定することができます。ITAC は、複数の計算ユニットを備えたクラスターで、MPI プログラムをデバッグしてパフォーマンスを向上させるために不可欠なツールです。ITAC の詳細は、「インテル® Trace Analyzer & Collector を使用して MPI のロード・インバランスを理解する」(英語) を参照してください。

付録

サンプル MPI プログラムのコードを次に示します。

```
/*
// 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 MPIn");
      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 %sn", 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 %sn", 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 spheren", 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=%16.12fn",
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;
}
```

著者紹介

Loc Q Nguyen。ダラス大学で MBA を、マギル大学で電気工学の修士号を、モントリオール理工科大学で電気工学の学士号を取得しています。現在は、インテル コーポレーションのソフトウェア & サービスグループのソフトウェア・エンジニアで、コンピューター・ネットワーク、コンピューター・グラフィックス、並列処理を研究しています。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。