インテル® Xeon Phi™ プロセッサー上で MPI for Python* (mpi4py) を使用する

この記事は、インテル® デベロッパー・ゾーンに公開されている「Exploring MPI for Python* on Intel® Xeon Phi™ Processor」の日本語参考訳です。

はじめに

メッセージ・パッシング・インターフェイス (MPI) (英語) は、分散メモリー・プログラミング向けに標準化されたメッセージ・パッシング・ライブラリー・インターフェイスです。MPI は、分散メモリー・アーキテクチャーに適しているため、ハイパフォーマンス・コンピューティング (HPC) 分野で広く使用されています。

Python* は、モジュールとパッケージをサポートする、近代的で強力なインタープリターであり、C/C++ 拡張をサポートします。多くの HPC アプリケーションは、高速化のため C や FORTRAN で記述されていますが、Python* は単純でモジュールをサポートするため、プロトタイプの概念実証を素早く作成したり、迅速なアプリケーション開発を実現できます。

MPI for Python (mpi4py) パッケージは、Python* と MPI 標準をバインドします。mpi4py パッケージは、MPI の構文とセマンティクスを変換し、Python* オブジェクトを使用して通信します。そのため、プログラマーは MPI アプリケーションを Python* で素早く実装することができます。mpi4py はオブジェクト指向です。mpi4py では、MPI 標準のすべての関数が使用できるわけではありませんが、一般的な関数は使用できます。mpi4pi の詳細は、こちら (英語) を参照してください。mpi4py の `COMM_WORLD` は、コミュニケーターの基本クラスのインスタンスです。

mpi4py は 2 種類の通信をサポートします。

- 汎用 Python* オブジェクトの通信: コミュニケーター・オブジェクトのメソッドの頭文字は小文字です (例: `send()`、`recv()`、`bcast()`、`scatter()`、`gather()`)。送信するオブジェクトは通信呼び出しの引数として渡されます。
- バッファー形式のオブジェクトの通信: コミュニケーター・オブジェクトのメソッドの頭文字は大文字です (例: `Send()`、`Recv()`、`Bcast()`、`Scatter()`、`Gather()`)。これらの呼び出しのバッファー引数は、タプルで指定します。バッファー形式のオブジェクトの通信のほうが、Python* オブジェクトの通信よりも高速です。

インテル® Distribution for Python* 2017

インテル® Distribution for Python* は、Python* インタープリターのバイナリー・ディストリビューションです。NumPy*、SciPy*、Jupyter、matplotlib、mpi4py などのコア Python* パッケージを高速化します。また、インテル® マス・カーネル・ライブラリー (インテル® MKL)、インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL)、pyDAAL、インテル® MPI ライブラリー、インテル® スレッディング・ビルディング・ブロック (インテル® TBB) を統合します。

インテル® Distribution for Python* 2017 は、Windows® 7 以降、Linux*、OS X* の各オペレーティング・システム向けに、2 つ (Python* 2.7.x および Python* 3.5.x) のパッケージが用意されています。パッケージは、スタンドアロンで、またはインテル® Parallel Studio XE 2017 とともにインストールできます。

インテル® Distribution for Python* の mpi4py は、ネイティブのインテル® MPI ライブラリー実装向けの Python* ラッパーです。この記事では、Python* で MPI プログラムを記述し、OpenMP* (英語) スレッドとインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX-512) 命令 (英語) を使用して、インテルのマルチコア・アーキテクチャーを活用する方法を示します。

インテル® Distribution for Python* は、Python* 2 と Python* 3 をサポートしています。インテル® Distribution for Python* には、2 つ (Python* 2.7 および Python* 3.5) のパッケージがあります。この例では、インテル® Distribution for Python* 2.7 for Linux* (`l_python27_pu_2017.0.035.tgz`) を、インテル® Xeon Phi™ プロセッサー 7250 (68 コア、1.40GHz、コアあたり 4 つのハードウェア・スレッド (合計 272 ハードウェア・スレッド)) にインストールします。以下のように、パッケージのコンテンツを展開した後、インストール・スクリプトを実行してインストールします。

```
$ tar -xvzf l_python27_pu_2017.0.035.tgz
$ cd l_python27_pu_2017.0.035
$ ./install.sh
```

インストールが完了したら、ルートのインテル® Distribution for Python* の Conda 環境をアクティブ化します。

```
$ source /opt/intel/intelpython27/bin/activate root
```

並列コンピューティング: OpenMP* と SIMD

マルチスレッドの Python* ワークロードは、インテル® TBB により最適化されたスレッド・スケジュールを利用できます。別のアプローチとして、OpenMP* によりインテルのマルチコア・アーキテクチャーの利点を活用することもできます。このセクションでは、OpenMP* スレッドと C 数学ライブラリーを Cython* で実装する方法を示します。

Cython は、ネイティブ言語にビルド可能なインタープリター言語です。Python* に似ていますが、C 関数呼び出し、C 形式の変数およびクラス属性宣言をサポートします。Cython は、Python* プログラムの実行を高速化する外部 C ライブラリーのラップに使用します。Cython は、`import` 文によりメイン Python* プログラムで使用される C 拡張モジュールを生成します。

例えば、拡張モジュールを生成するには、Cython コード (拡張子 `.pyx`) を記述します。次に、`.pyx` ファイルを Cython コンパイラーでコンパイルして、C コード (`.c` ファイル) に変換します。そして、`.c` ファイルを C コンパイラーでコンパイルして、共有オブジェクト・ライブラリー (`.so` ファイル) を生成します。

Cython コードをビルドする 1 つの方法は、disutils `setup.py` ファイルを記述することです (disutils は Python* モジュールを分散するために使用されます)。次の `multithreads.pyx` ファイルで、`vector_log_multiplication` 関数は A 配列と B 配列の各エントリーの `log(a)*log(b)` を計算して、C 配列に結果を格納します。並列ループ (`prange`) を使用して複数のスレッドを並列に実行できるようにします。log 関数は C 数学ライブラリーからインポートされます。`getnumthreads()` 関数はスレッド数を返します。

```
$ cat multithreads.pyx
cimport cython
import numpy as np
cimport openmp
from libc.math cimport log
from cython.parallel cimport prange
from cython.parallel cimport parallel

@cython.boundscheck(False)
@cython.wraparound(False)
def vector_log_multiplication(double[:] A, double[:] B, double[:] C):
```

```
    cdef int N = A.shape[0]
    cdef int i

    with nogil:
        for i in prange(N, schedule=&amp;apos;static&amp;apos;):
            C[i] = log(A[i]) * log(B[i])

def getnumthreads():
    cdef int num_threads

    with nogil, parallel():
        num_threads = openmp.omp_get_num_threads()
        with gil:
            return num_threads
```

setup.py ファイルは、拡張モジュールを生成する setuptools ビルドプロセスを起動します。デフォルトでは、この setup.py は GNU* GCC を使用して Python* 拡張の C コードをコンパイルします。インテル® Xeon Phi™ プロセッサーでインテル® AVX-512 と OpenMP* スレッドを利用するには、インテル® C++ コンパイラー (icc) で -xMIC-avx512 オプションと -qopenmp オプションを指定してコンパイルします。setup.py ファイルの作成方法については、Python* ドキュメントの「Writing the Setup Script (セットアップ・スクリプトを記述する)」(英語) を参照してください。

```
$ cat setup.py
from distutils.core import setup
from Cython.Build import cythonize
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
  name = "multithreads",
  cmdclass = {"build_ext": build_ext},
  ext_modules = [
    Extension("multithreads",
              ["multithreads.pyx"],
              libraries=["m"],
              extra_compile_args = ["-O3", "-xMIC-avx512", "-qopenmp" ],
              extra_link_args=[&amp;apos;-qopenmp&amp;apos;, &amp;apos;-xMIC-avx512&amp;apos;]
              )
  ]

)
```

この例では、インテル® Parallel Studio XE 2017 Update 1 を使用しています。最初に、インテル® C コンパイラーの環境変数を設定します。

```
$ source /opt/intel/parallel_studio_xe_2017.1.043/psxevars.sh intel64
Intel(R) Parallel Studio XE 2017 Update 1 for Linux*
Copyright (C) 2009-2016 Intel Corporation.無断での引用、転載を禁じます。
```

インテル® コンパイラー (icc) でこのアプリケーションをコンパイルするには、次のコマンドで setup.py ファイルを実行します。

```
$ LDSHARED="icc -shared" CC=icc python setup.py build_ext –inplace

running build_ext
cythoning multithreads.pyx to multithreads.c
building &apos;multithreads&apos; extension
creating build
creating build/temp.linux-x86_64-2.7
icc -fno-strict-aliasing -Wformat -Wformat-security -D_FORTIFY_SOURCE=2 -fstack-
protector -O3 -fpic -fPIC -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -fPIC
-I/opt/intel/intelpython27/include/python2.7 -c multithreads.c -o
build/temp.linux-x86_64-2.7/multithreads.o -O3 -xMIC-avx512 -march=native -
qopenmp
icc -shared build/temp.linux-x86_64-2.7/multithreads.o -
L/opt/intel/intelpython27/lib -lm -lpython2.7 -o
/home/plse/test/v7/multithreads.so -qopenmp -xMIC-avx512
```

前述のとおり、このプロセスは、まず拡張コード `multithreads.c` を生成します。そして、この拡張コードをインテル® コンパイラーがコンパイルして、ダイナミック共有オブジェクト・ライブラリー `multithreads.so` を作成します。

MPI/OpenMP* のハイブリッド実装を使用して Python* アプリケーションを記述する

このセクションでは、MPI アプリケーションを Python* で記述します。このプログラムは、`mpi4py` モジュールと `multithreads` モジュールをインポートします。MPI アプリケーションは、コミュニケーター・オブジェクト `MPI.COMM_WORLD` を使用して、プロセスセット内の通信可能なプロセスを特定します。MPI 関数 `MPI.COMM_WORLD.Get_size()`、`MPI.COMM_WORLD.Get_rank()`、`MPI.COMM_WORLD.send()`、`MPI.COMM_WORLD.recv()` は、このコミュニケーター・オブジェクトのメソッドです。mpi4py では、MPI 標準とは異なり、`MPI_Init()` と `MPI_Finalize()` を明示的に呼び出す必要がありません。これらの関数は、モジュールのインポート時と Python* プロセスの終了時にそれぞれ呼び出されます。

サンプル Python* アプリケーションは、最初に 1 ～ 2 の範囲の乱数を含む 2 つの大きな入力配列を初期化します。各 MPI ランクは、OpenMP* スレッドを使用して並列に計算を実行します。そして、各 OpenMP* スレッドは、2 つの自然対数 `c = log(a)*log(b)` の積を計算します。ここで、a と b は、1 ～ 2 の範囲の乱数です (`1 <= a, b <= 2`)。各 MPI ランクは、`multithreads.pyx` で定義されている `vector_log_multiplication` 関数を呼び出します。この関数の実行時間は短く、約 1.5 秒です。ここでは、使用されている OpenMP* スレッドの数が分かるように、`timeit` ユーティリティーを使用して関数を 10 回呼び出しています。

以下は、`mpi_sample.py` にあるアプリケーションのソースコードです。

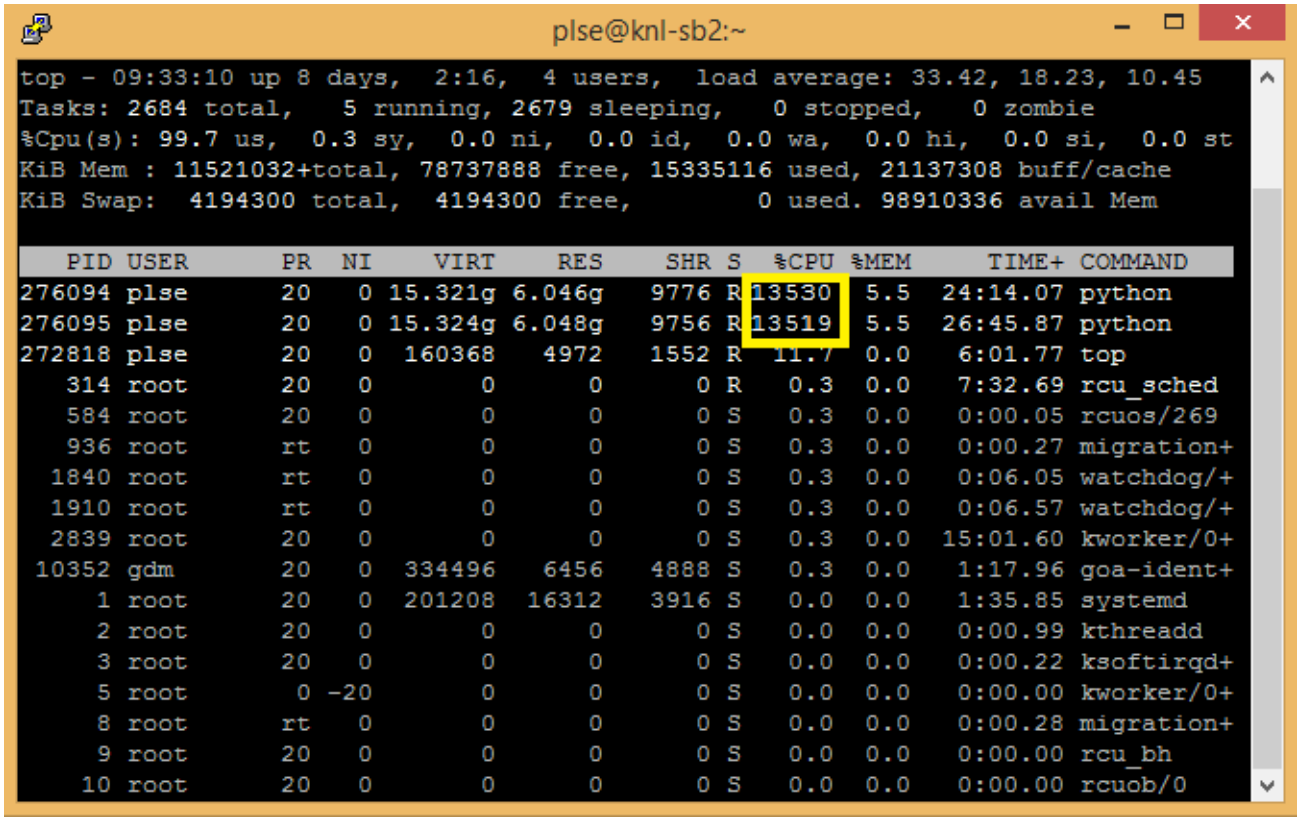
```
from mpi4py import MPI
from multithreads import *
import numpy as np
import timeit

def time_vector_log_multiplication():
    vector_log_multiplication(A, B, C)

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

THOUSAND = 1024
FACTOR = 512
NUM_TOTAL_ELEMENTS = FACTOR * THOUSAND * THOUSAND
NUM_ELEMENTS_RANK = NUM_TOTAL_ELEMENTS / size
repeat = 10
```

```
numthread = getnumthreads()

if rank == 0:
   print "Initialize arrays for %d million of elements" % FACTOR

A = 1 + np.random.rand(NUM_ELEMENTS_RANK)
B = 1 + np.random.rand(NUM_ELEMENTS_RANK)
C = np.zeros(A.shape)

if rank == 0:
    print "Start timing ..."
    print "Call vector_log_multiplication with iter = %d" % repeat
    t1 =  timeit.timeit("time_vector_log_multiplication()", setup="from __main__
import time_vector_log_multiplication",number=repeat)
    print "Rank %d of %d running on %s with %d threads in %d seconds" % (rank,
size, name, numthread, t1)

    for i in xrange(1, size):
        rank, size, name, numthread, t1 = MPI.COMM_WORLD.recv(source=i, tag=1)
        print "Rank %d of %d running on %s with %d threads in %d seconds" % (rank,
size, name, numthread, t1)
    print "End  timing ..."

else:
    t1 =  timeit.timeit("time_vector_log_multiplication()", setup="from __main__
import time_vector_log_multiplication",number=repeat)
    MPI.COMM_WORLD.send((rank, size, name, numthread, t1), dest=0, tag=1)
```

次のコマンドを実行して、2 つの MPI ランクで上記の Python* アプリケーションを起動します。

```
$ mpirun -host localhost -n 2 python mpi_sample.py

Initialize arrays for 512 million of elements
Start timing ...
Call vector_log_multiplication with iter = 10
Rank 0 of 2 running on knl-sb2.jf.intel.com with 136 threads in 14 seconds
Rank 1 of 2 running on knl-sb2.jf.intel.com with 136 threads in 15 seconds
End  timing ...
```

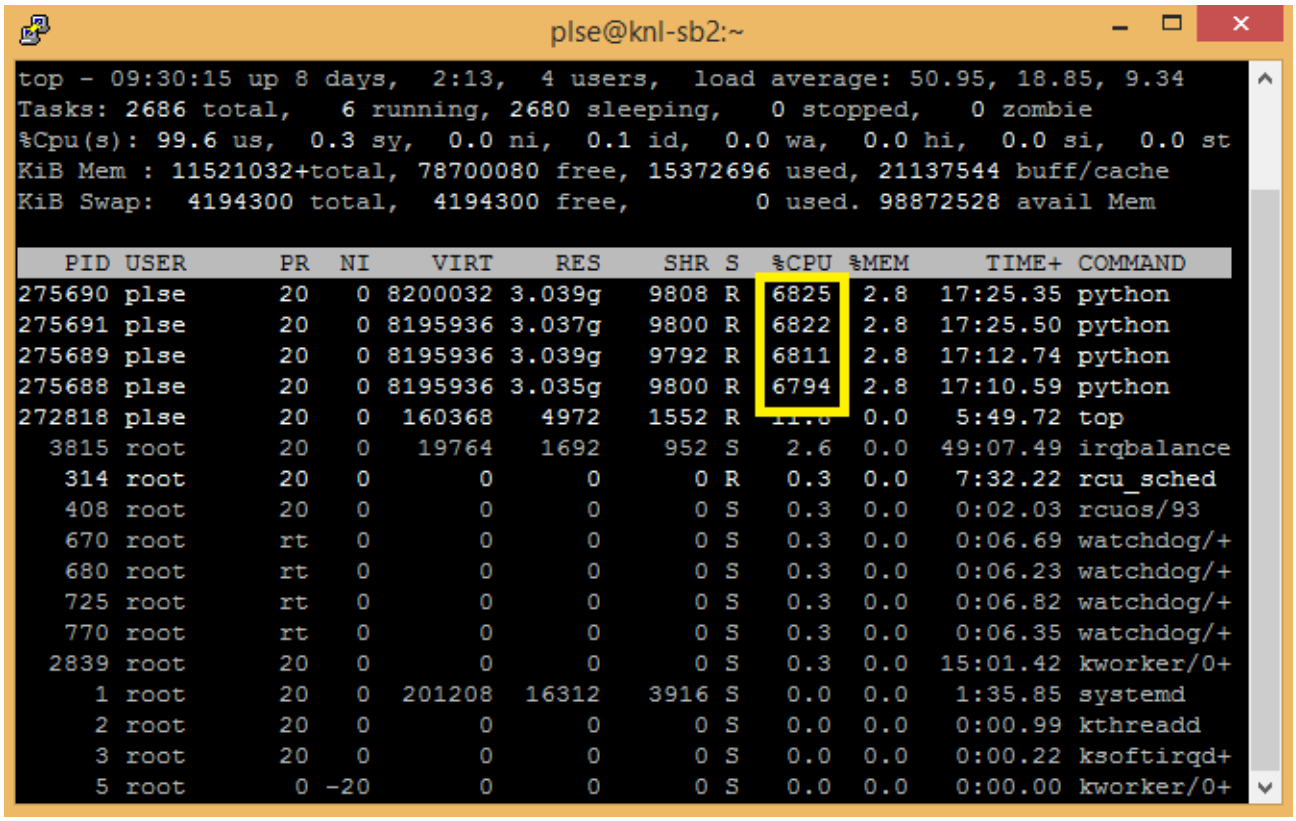
Python* プログラムの実行中、新しいターミナルの `top` コマンドには 2 つの MPI ランク (2 つの Python* プロセス) が表示されます。メインモジュールがループに入る (“`Start timing ...`” メッセージの出力) 時点で、`top` コマンドは 136 スレッドが実行中 (~13600%CPU) であるとレポートしています。デフォルトでは、このシステムで利用可能な 272 のハードウェア・スレッドがすべて 2 つの MPI ランクによって使用されるため、各 MPI ランクのスレッド数は 272/2 = 136 になります。



実行時に MPI に関する詳細情報を取得するには、I_MPI_DEBUG 環境変数の値を 0 ～ 1000 の範囲に設定します。次のコマンドは、4 つの MPI ランクを実行し、I_MPI_DEBUG を 4 に設定します。top コマンドの出力が示すように、各ランクの OpenMP* スレッド数は 272/4 = 68 になります。

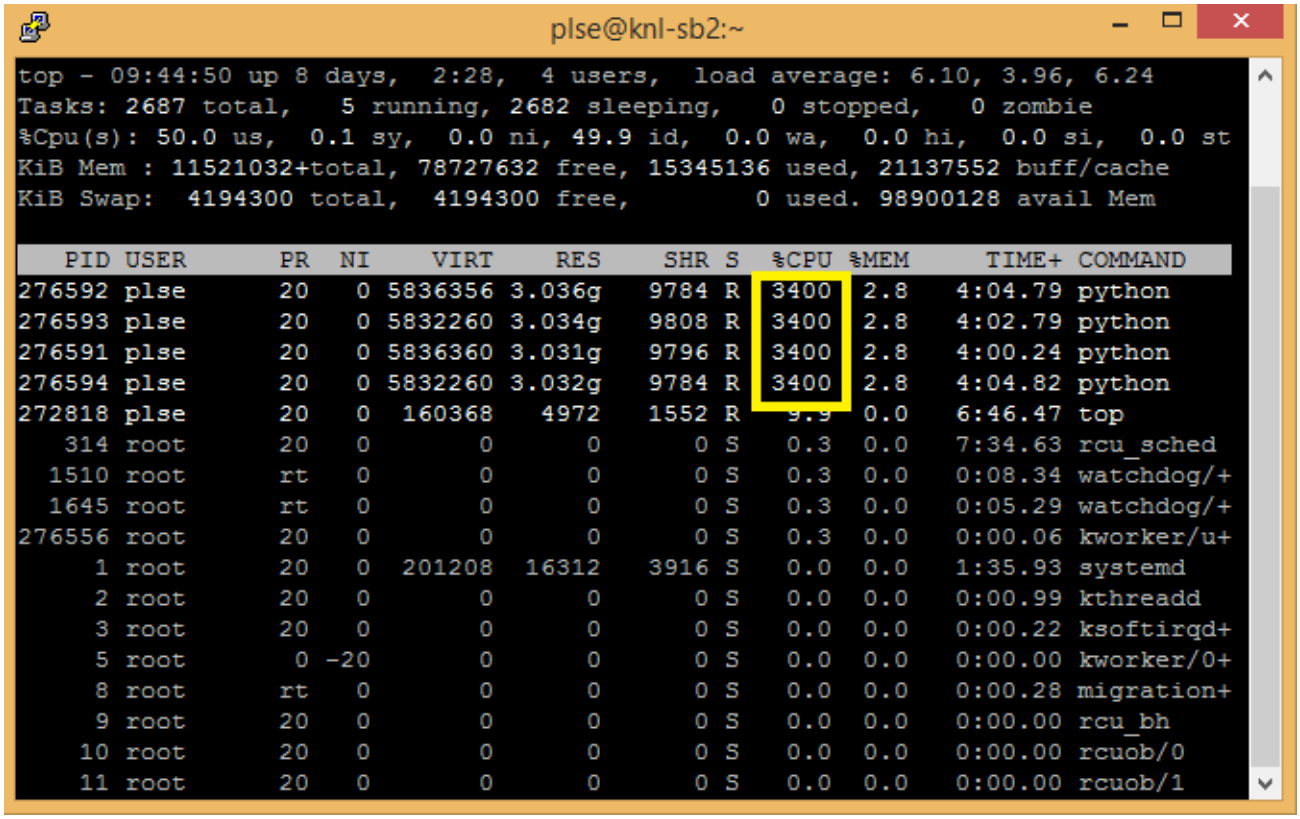
```
$ mpirun -n 4 -genv I_MPI_DEBUG 4 python mpi_sample.py

[0] MPI startup(): Multi-threaded optimized library
[0] MPI startup(): shm data transfer mode
[1] MPI startup(): shm data transfer mode
[2] MPI startup(): shm data transfer mode
[3] MPI startup(): shm data transfer mode
[0] MPI startup(): Rank    Pid      Node name             Pin cpu
[0] MPI startup(): 0       84484    knl-sb2.jf.intel.com
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,
204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220}
[0] MPI startup(): 1       84485    knl-sb2.jf.intel.com
{17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,85,86,87,88,89,90,91,92,93,94
                                                ,95,96,97,98,99,100,101,153,154,155,1
56,157,158,159,160,161,162,163,164,165,166,
167,168,169,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237}
[0] MPI startup(): 2       84486    knl-sb2.jf.intel.com
{34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,102,103,104,105,106,107,108,1
09,110,111,112,113,114,115,116,117,118,170,171,172,173,174,175,176,177,178,179,18
0,181,182,183,184,185,186,238,239,240,241,242,243,244,245,246,247,248,249,250,251
,252,253,254}
[0] MPI startup(): 3       84487    knl-sb2.jf.intel.com
{51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,119,120,121,122,123,124,125,1
26,127,128,129,130,131,132,133,134,135,187,188,189,190,191,192,193,194,195,196,19
7,198,199,200,201,202,203,255,256,257,258,259,260,261,262,263,264,265,266,267,268
,269,270,271}
Initialize arrays for 512 million of elements
```

```
Start timing ...
Call vector_log_multiplication with iter = 10
Rank 0 of 4 running on knl-sb2.jf.intel.com with 68 threads in 16 seconds
Rank 1 of 4 running on knl-sb2.jf.intel.com with 68 threads in 15 seconds
Rank 2 of 4 running on knl-sb2.jf.intel.com with 68 threads in 15 seconds
Rank 3 of 4 running on knl-sb2.jf.intel.com with 68 threads in 15 seconds
End  timing ...
```



OMP_NUM_THREADS 環境変数を設定することで、並列領域で使用される各ランクの OpenMP* スレッド数を指定できます。次のコマンドは、4 つの MPI ランク、MPI ランクごとに 34 スレッド (コアあたり 2 スレッド) を開始します。

```
$  mpirun -host localhost -n 4 -genv OMP_NUM_THREADS 34 python mpi_sample.py

Initialize arrays for 512 million of elements
Start timing ...
Call vector_log_multiplication with iter = 10
Rank 0 of 4 running on knl-sb2.jf.intel.com with 34 threads in 18 seconds
Rank 1 of 4 running on knl-sb2.jf.intel.com with 34 threads in 17 seconds
Rank 2 of 4 running on knl-sb2.jf.intel.com with 34 threads in 17 seconds
Rank 3 of 4 running on knl-sb2.jf.intel.com with 34 threads in 17 seconds
End  timing ...
```



最後に、プログラムが MCDRAM (インテル® Xeon Phi™ プロセッサーの高帯域幅メモリー) にメモリーを割り当てるように強制することができます。例えば、プログラムの実行前に "numactl –hardware" コマンドを実行すると、システムに 2 つの NUMA ノードがあり、ノード 0 は CPU と 96GB DDR4 メモリー、ノード 1 はオンボードの 16GB MCDRAM メモリーであることが分かります。

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108
109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168
169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188
189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208
209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228
229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248
249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268
269 270 271
node 0 size: 98200 MB
node 0 free: 73585 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15925 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

次のコマンドを実行すると、可能な場合は MCDRAM にメモリーが割り当てられます。

```
$ mpirun -n 4 numactl --preferred 1 python mpi_sample.py
```

プログラムの実行中、MCDRAM (NUMA ノード 1) にメモリーが割り当てられたことを確認できます。

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104
105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164
165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244
245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264
265 266 267 268 269 270 271
node 0 size: 98200 MB
node 0 free: 73590 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 3428 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

上記のコードは、適切な設定を使用して、インテル® Xeon® プロセッサー・ベースのシステムでも試すことができます。例えば、インテル® Xeon® プロセッサー E5-2690 v4 ベースのシステムでは、`-xMIC-AVX512` の代わりに `-xCORE-AVX2` を使用し、利用可能なスレッド数を 272 から 28 に変更します。インテル® Xeon® プロセッサー E5-2690 v4 には、高帯域幅メモリー (MCDRAM) は搭載されていません。

まとめ

この記事では、MPI for Python (mpi4py) とインテル® Distribution for Python* を利用する方法を説明しました。さらに、OpenMP* とインテル® AVX-512 命令を利用して、インテル® Xeon Phi™ プロセッサー・アーキテクチャーを最大限に活用する方法も示しました。簡単な例を用いて、OpenMP* を使用して並列 Cython 関数を記述し、インテル® コンパイラーでインテル® AVX-512 を有効にするオプションを指定してコンパイルし、MPI Python* プログラムに統合して、インテル® Xeon Phi™ プロセッサー・アーキテクチャーの利点が得られることを示しました。

参考資料

- MPI Forum (英語)
- MPI for Python (英語)
- インテル® Distribution for Python*
- インテル® Parallel Studio XE 2017
- インテル® MPI ライブラリー
- インテル® AVX-512 命令 (英語)
- OpenMP* (英語)

- Cython C-Extensions for Python* (英語)
- Writing the Setup Script (英語)

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、並列コンピューティング、コンピューター・グラフィックスを研究しています。

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。