HPL with MKL

Nicole Orzan

February 11, 2018

In this exercise we have to run the HPL benchmark compiled with the MKL library. HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers, while MKL is an Intel's library that accelerates math processing routines to increase application performance, which includes highly vectorized and threaded Linear Algebra, Fast Fourier Transforms, Vector Math and Statistics functions. We have to run this benchmark in order to find the best combination of variables which allows us to reach the best performance.
I worked on Ulysses cluster.

The theoretical peak performance is the FLoating point Operations Per Second (FLOPs), which indicates the number of floating point operations that can be executed in one second by the CPU. It can be calculated in this way: FLOPs = n cores * clock rate * n sockets * n floating point operations.
On Ulysses cluster its value is: 10 * 2.8*10^9* 2 * (4*2) = 448 Gflops/s.

First of all I get the HPL package from netlib:
wget http://www.netlib.org/benchmark/hpl/hpl-2.2.tar.gz

Then I created a makefile for HPL. To do this I used the file Make.Linux.ATHLON.CBLAS - because it was the most general one - creating a copy named Make.Nic in the top directory. I had to change some parts of the makefile, for which I used also some tips given by https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor to compile against MKL library.

```
#defining arch
ARCH         = Nic
...
#defining top directory
TOPdir       = /home/norzan/hpl_mkl/hpl-2.2
...
#indicating the MKL library path
LAdir        = $(MKLROOT)
ifndef  LAinc
LAinc        = $(LAdir)/include
endif
ifndef  LAlib
LAlib        = -L$(LAdir)/lib/intel64 \
               -Wl,--start-group \
               $(LAdir)/lib/intel64/libmkl_intel_lp64.a \
               $(LAdir)/lib/intel64/libmkl_gnu_thread.a \
               $(LAdir)/lib/intel64/libmkl_core.a \
               -Wl,--end-group -lgomp -lpthread -lm -ldl
endif
...
#defining the use of mpi compiler
CC           = mpicc
...
#insert some flags
CCFLAGS      = -m64 $(HPL_DEFS) -O3 -Wall
```

Then I compiled with the command make arch=<arch>.
I get an executable file in the directory bin/Nic: xhpl. To ran it I tuned some parameters inside HPL.dat file, such as:

- N, the number of the problems to be runned;
- Ns, the problem sizes;
- # of NBs, the number of block sizes to be runned;
- NBs ;
- # of process grids (P x Q);
- # of Ps and Qs; they specify the number of process rows and columns of each grid you want to run on;

As you can see I had to choose the size of the problems to be ran (which means the size of the matrices), so that it occupied the 75% of the RAM.
On Ulysses' nodes we have 64 GB so I needed to have 48 GB occupied. Since we have double numbers (8 bytes each) we obtain $48/8 = 6$ as N^2, so N=$\sqrt{6} = 77459.66$, which is as ideal number to be chosen as size of the matrix. I decided to try with 65542 which is a power of two near to that number (so I can use blocks multiple of 2 too).
I chosen blocks of sizes 128, 192 (which should to be the number for which HPL gives better results) and 256. Then I tuned the values of P and Q: an Ulysses' node has 20 cores so PxQ=20. I used then the 2 combinations P=4, Q=5 and P=5, Q=4 to see which one was the best.

Results

```
N      :   65536
NB     :     128      192      256
PMAP   : Row-major process mapping
P      :       4
Q      :       5
PFACT  :    Left    Crout    Right
NBMIN  :       2        4
NDIV   :       2
RFACT  :    Left    Crout    Right
BCAST  :   1ring
DEPTH  :       0
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words

================================
T/V           N      NB     P     Q     Time     Gflops
--------------------------------
WR00L2L2   65536    128     4     5     577.78   3.248e+02
```

This case reaches the 72% of the theoretical Peak Performance, which is not so much...
Inverting values of P and Q, instead:

```
================================
T/V           N      NB     P     Q     Time     Gflops
--------------------------------
WR00L2L2    65536   128     5     4     471.91   3.977e+02
```

which is 88% of the Peak Performance. Seems like this combination of P and Q is the best one.
We can see that the block size 128 seems to be the one for which the performance is at its best.

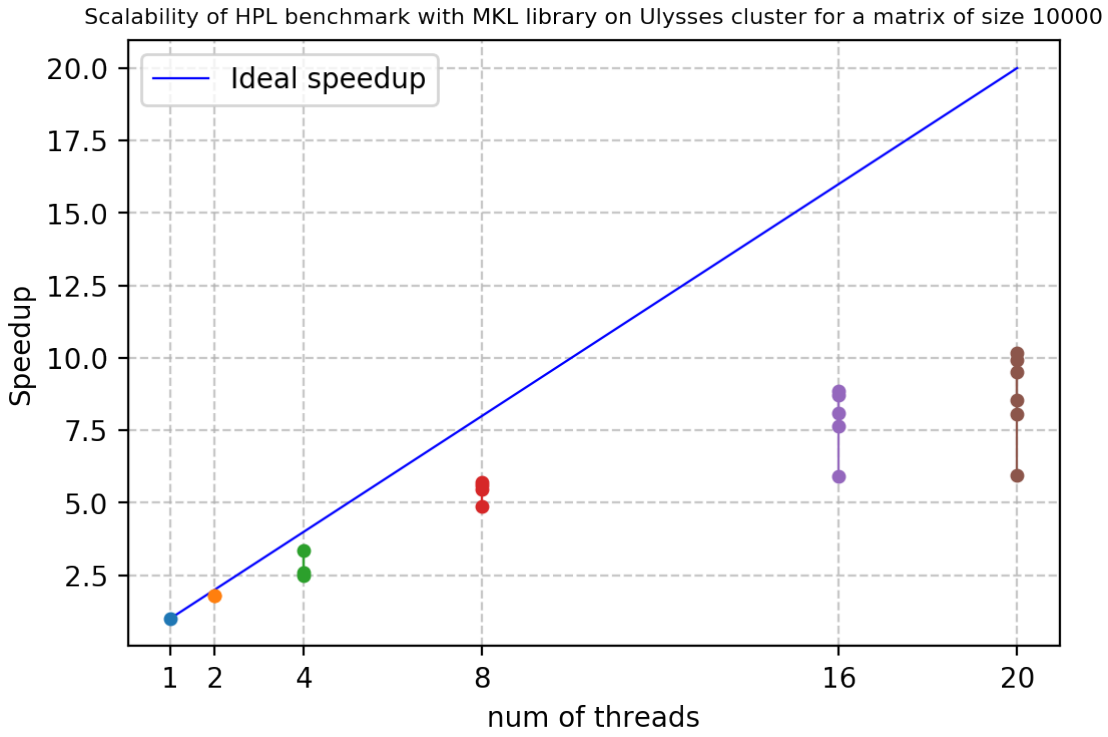

Figure 1: Strong scalability of HPL benchmark compiled against MKL library, for a matrix of size 1000 and blocks of size 128.

Scalability

Then I checked the Speedup of HPL. To compute it I fixed the matrix size and I changed the number of threads for 1 up to 20. In figure 1 you can see an image of the strong scalability I obtained from HPL benchmark with MKL library on a matrix of size 1000 and blocks of size 128. The different points you can see for every fixed number of threads were obtained changing the combination of PxQ (for example, when we have PxQ=16 we can have 5 possible combinations: 1*16, 2*8, 4*4, 8*2 and 16*1; I notice that, as supposed, the combinations that had an higher speedup value were the "central" ones; in the example 5*4 and 4*5).
In figure 2 you can see an image of the efficiency computed on the same dataset.

Precompiled HPL

I get the precompiled binary files provided by intel from
https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite
I unzipped the file l_mklb_p_2018.1.009.tgz and then I ran the precompiled binary file ./runme_xeon64.

```
Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
1000   1000   4       101.3180 133.2907
2000   2000   4       212.1919 214.0705
5000   5008   4       219.2405 222.5120
10000  10000  4       321.4467 321.6213
15000  15000  4       379.7967 380.0693
18000  18008  4       407.0823 407.1996
20000  20016  4       410.5489 410.5952
22000  22008  4       418.1688 418.2686
25000  25000  4       419.1335 419.3067
```

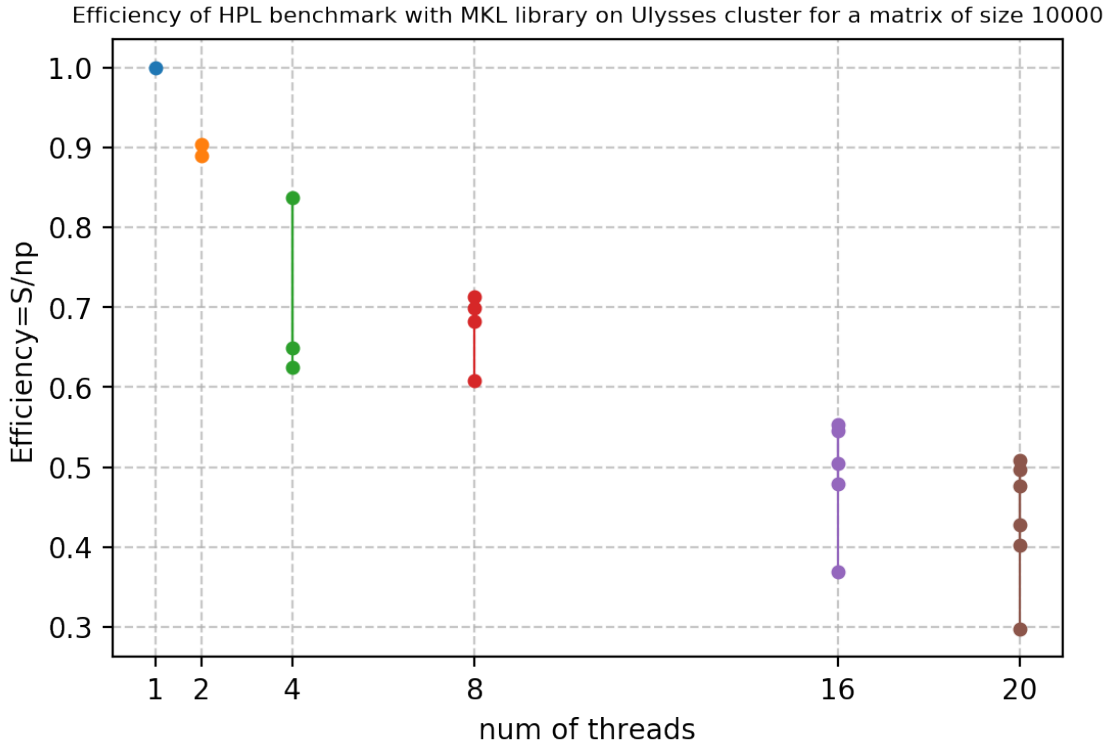

Figure 2: Efficiency of HPL benchmark compiled against MKL library, for a matrix of size 1000 and blocks of size 128. Computed as speedup/n of threads

```
26000   26000   4          422.3966  422.6081
27000   27000   4          422.5188  422.5188
30000   30000   1          423.9991  423.9991
35000   35000   1          425.3995  425.3995
40000   40000   1          435.2235  435.2235
45000   45000   1          433.5398  433.5398
```

So it reaches the 97% of the theoretical Peak Performance.