The HPL Exercise

In this exercise, we chose node at Chameleon to run the HPL test. We had built and installed the HPL benchmark with the Intel MKL and Intel MPI library and used scripts for compiling. Then we ran the HPL benchmark on a single with our own setup and tuning the N, NB, and P x Q setups. Finally, we computed the theoretical peak Flop/s of the CPU and examined the CPU frequency running on the node.

1.HPL benchmark building and installing

1.1 Cluster Description

We chose cluster at Chameleon to run the HPL test. The hardware and software information that we applied for (2 nodes in total but 1 node for HPL test) in Chameleon is shown in Table 1.1.1, 1.1.2 and 1.2.

Item	NUMA	CPU(s)	Architecture	Name
Chameleon CPU Compute Server	NUMA node(s):2 NUMA node0 CPU(s): 0,2,4,6,8,10,12,14,16,18 ,20,22 NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19 ,21,23	24 CPU op-modes:32bit,64bit	x86_64	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

CPU (MHz)	Byte Order	Caches (sum of all)	Memory(MB)
Max:3300.00 Min:1200.00	Little Endian	L1d: 768 KiB (24 instances) L1i: 768 KiB (24 instances) L2: 6 MiB (24 instances) L3: 60 MiB (2 instances)	Available: 126954 Total:128811

TABLE 1.1.1,1.1.2: Chameleon Cluster Hardware Configuration

Classification	Description	Installation Path	Version
OS	GNU/Linux	-	5.15.0-86-generic
Distributor ID	Ubuntu 22.04.3 LTS	-	22.04
Toolkit	Intel® oneAPI HPC Toolkit	/opt/intel	2023.2

Compiler	Intel Parallel StudioXE	/opt/intel/api	2023.2
BLAS	Intel MKL	/opt/intel/api/mkl	2023.2
MPI	Intel MPI	/opt/intel/api/mpi	2023.2
HPL	HPL CPU	-	HPL-2.3

TABLE 1.2: Chameleon Cluster Software Environment Configuration

In the following test, we would run the HPL benchmark to test the performance of the nodes, where one node contained 24 cores and 64GB memory in total.

1.2 Libraries Chosen

1.2.1 MPI Choosing: Intel MPI

In the test we use Intel MPI. The Intel MPI Library is a multi-fabric message passing library that implements the Message Passing Interface.

There are several reasons why we chose Intel MPI.

First of all, it has high performance optimization. Intel MPI is highly optimized for Intel processors, leveraging various features of Intel architecture to achieve superior performance. This includes instruction set optimizations and performance tuning tailored for Intel processors, often resulting in better communication performance on Intel hardware.

Secondly, it contains good Integration with Intel Libraries and Compilers. Intel MPI integrates well with Intel MKL, harnessing optimizations to improve application performance.

Thirdly, when using Intel MPI, we benefited from other Intel tools such as Intel@Vtune Performance Analyzer, which can determine the HPL hotspot function for later program improvement and optimization.

Last but not least, Intel MPI supports large-scale parallel computing, making it suitable for systems with thousands of processors. It incorporates optimizations and techniques to ensure good scalability on large clusters, which will surely benefit our later HPL testing.

By using the Intel MPI library, we could compile and run HPL on multiple cluster interconnects.

1.2.2 BLAS Choosing: Intel MKL

We chose the Intel MKL on the platform as the BLAS we use.

The Intel MKL contains BLAS level1,2,3 and LAPACK and many other Math Libraries, which support linear algebra operations, including linear equation solving, eigenvalue problems, matrix factorization. It will definitely support for computing the HPL test.

Also, according to the testing we did before, the Intel MKL did much benefit to

the performance of system floating-point operations than other BLAS like OpenBLAS. The integration and Multi-thread support let it became our choice.

1.3 Compiling HPL

We first downloaded the latest version of the HPL test package (hpl-2.3.tar.gz) from http://www.netlib.org/benchmark/hpl/, unzipped the file (using the scripts: tar -xvf hpl-2.3.tar.gz) , and we got a directory called hpl-2.3 .

To compile HPL, we first created a Makefile. For this file, the HPL's naming convention was Make.arch, where arch means the name of the computer architecture. Considering that our platform were using Intel processors, we modified it based on the file setup/Make.Linux_Intel64, and copied it to the previous directory. The content which should be modified is showed as follows:

1.3.1 HPL Directory Structure / HPL library

```
1.  TOPdir = $(HOME)/hpltest/hpl-2.3
2.  INCdir = $(TOPdir)/include
3.  BINdir = $(TOPdir)/bin/$(ARCH)
4.  LIBdir = $(TOPdir)/lib/$(ARCH)
```

Here we modified the contents of 'TOPdir' to the directory path of hpl-2.3. And we also rewrote the BINdir and LIBdir according to our actual directory hpl-2.3/bin and hpl-2.3/lib .

1.3.2 Message Passing library (MPI)

```
1.  MPdir = /opt/intel/oneapi/mpi/latest
2.  MPinc = -I$(MPdir)/include
3.  MPlib = $(MPdir)/lib/release/libmpi.a
```

Based on the installation path of Intel MPI on the test platform, we modified the path information corresponding to 'MPdir'. We also modified the corresponding path to 'MPlib'.

1.3.3 Linear Algebra library Intel MKL

```
1.  LAdir = /opt/intel/oneapi/mkl/2023.2.0
2.  LAinc = $(LAdir)/include
3.  LAlib = -L$(LAdir)/lib/intel64 \
4.  -Wl,--start-group \
5.  $(LAdir)/lib/intel64/libmkl_intel_lp64.a \
6.  $(LAdir)/lib/intel64/libmkl_intel_thread.a \
7.  $(LAdir)/lib/intel64/libmkl_core.a \
```

```
8.      -Wl,--end-group -lpthread -
```

we modified the path information corresponding to 'LAdir' since we have installed the version 2023.2 . To use the self-installed math library, we modified the path information and the library file corresponding to 'LAlib' .

1.3.4 Compilers / linkers - Optimization flags

```
1.      CC = /opt/intel/oneapi/mpi/latest/bin/mpiicc
2.      CCNOOPT = $(HPL_DEFS)
3.      CCFLAGS = $(HPL_DEFS) -O3 -w -ansi-alias -i-static -z noexecstack \
4.      -z relro -z now -nocompchk -Wall
5.      -qopenmp
6.      LINKER = $(CC)
7.      LINKFLAGS = $(CCFLAGS) -mt_mpi
```

According to the Intel Compiler installation path on the test platform, we modified the CC compiler path information to the path to mpiicc. We also add -qopenmp to help compile the Compiler.

1.3.5 HPL Compilation

After modifying the Make.Linux_Intel64 file, we compiled HPL. The command is as following:

```
1.      make arch=Linux_Intel64
```

The compilation ended correctly, and the file 'HPL.dat' and 'xhpl' will be generated in the bin/Linux_Intel64 directory. 'HPL.dat' is the configuration file and 'xhpl' is the test program which should be executed. At this point, we finished the compilation of the HPL and related software.

2. Running HPL benchmark on a single node

2.1 HPL.dat Settings and tuning

After the compilation of the HPL benchmark, we now started to set up the running configuration by setting HPL.dat file.

Here we had mainly 3 items to change: problem size N, block size NB, process grid ratio P x Q .

First we determined NB. The matrix is split into the block sized NB to cyclically allocated to each process, where NB refers to the block granularity (size). The choice

of NBs is related to many hardware and software factors. The size of the NBs should be as close as possible to the size of the Cache line, which can not only exert the Catch performance but also reduce the cache conflict. Here by experience, we chose NB = 192.

N is the dimension of the solved linear system Ax = b. Theoretically, the value of N needs to consider the ratio of node memory capacity and the operation access. Larger matrix scales will increase the ratio of computing and communication for better performance. According to the empirical formula,

$$N = \sqrt{memory(GB) \times 1024^3 \times \propto / 8}$$

And $\alpha \leq 80\%$ is suitable. From 1.1 we know that the free memory in the node is about 125GB, take $\alpha = 80\%$,then $N \approx 115000$. Since that if N is divisible by NB, the node can handle the problem with maximally utilization. So, we can take N as 113472.

P×Q represents the number of parallel processes. For the cluster nodes we used for testing, we would use 2 threads for each of the NUMA nodes in the CPU, and each node we will have 12 processes. That means that P×Q = 2. In theory, $P \leqslant Q$ will have a better floating-point performance, because column-oriented traffic is much costing than horizontal communication. Above all, we take P = 1, Q = 2.

Next we still have many configuration in HPL.dat. Here we referenced https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/ and use the configuration it generated. And we finished PFACTs, RFACTs and the rest of the settings. Figure2.1.1 are the whole HPL.dat settings:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
113472 30 34 35  Ns
1            # of NBs
192 2 3 4      NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
1 1 4        Ps
2 4 1        Qs
16.0         threshold
1            # of panel fact
2 1 0        PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4 2          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1 0 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
~
~
```

Figure2.1.1 HPL.dat settings

2.2 Using the binary 'xhpl' from the vendor

Now we try to run HPL benchmark on our node. Since we get a 'xhpl' file generated in the directory, we can use it to run the HPL benchmark using the following command:

```
NP=4
mpirun -np $NP ./xhpl
```

The NP means the number of process we assign. In actual running, we write a simple bash command ' job_lsf.bash 'to run HPL benchmark. The following are the commands:

```
#!/bin/bash
export OMP_NUM_THREADS=12
mpirun -n 2 ~/hpltest/hpl-2.3/bin/Linux_Intel64/xhpl > result.out
```

The combination of these commands means that the xhpl executable file is launched simultaneously on two MPI processes, and OpenMP parallel programming techniques are employed to create 12 threads within each process to perform computational tasks. The result would be seen in result.out if succeeded.

Then We execute it with this command:

```
bash job_lsf.bash
```

After that the node would run HPL test.

2.3 The Final Output

Here we simply listed the parameter values that the HPL used and the last 31 rows of the final result in result.out. List2.3.1 is parameter values used in HPL testing and List2.3.2 is the final result:

```
N      :  113472
NB     :     192
PMAP   : Row-major process mapping
P      :       1
Q      :       2
PFACT  :   Right
NBMIN  :       4
NDIV   :       2
RFACT  :   Crout
BCAST  :   1ring
DEPTH  :       0
SWAP   : Mix (threshold = 64)
```

```
L1      : transposed form
U       : transposed form
EQUIL   : yes
ALIGN   : 8 double precision words
```

List2.3.1 Parameter Values used in HPL testing

```
Column=000112128 Fraction=98.8% Gflops=6.036e+02
Column=000112320 Fraction=99.0% Gflops=6.036e+02
Column=000112512 Fraction=99.2% Gflops=6.036e+02
Column=000112704 Fraction=99.3% Gflops=6.036e+02
Column=000112896 Fraction=99.5% Gflops=6.036e+02
Column=000113088 Fraction=99.7% Gflops=6.036e+02
Column=000113280 Fraction=99.8% Gflops=6.036e+02
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00C2R4      113472   192     1     2            1614.98             6.0314e+02
HPL_pdgesv() start time Wed Oct 11 21:51:18 2023

HPL_pdgesv() end time   Wed Oct 11 22:18:13 2023

--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV-
Max aggregated wall time rfact . . . :               7.70
+ Max aggregated wall time pfact . . :               4.42
+ Max aggregated wall time mxswp . . :               0.84
Max aggregated wall time update  . . :            1553.67
+ Max aggregated wall time laswp . . :             147.94
Max aggregated wall time up tr sv  . :               1.20
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=   2.06534730e-03 ...... PASSED
================================================================================

Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------
```

List2.3.2 Final Result

So, the briefly result is that the node ran the test in 1614.98s, and its Gflop/s is 6.0314e+02 = 603.14 Gflop/s.

3. Theoretical peak Flop/s Computation

3.1 Theoretical peak FLOP/s Computation

Now we compute the theoretical peak Flop/s for our single using node. According to the Computation Formula:

$$Rpeak = Ncores \times CPU\ frequency \times \frac{flops}{cycle} \text{ ①}$$

$$\frac{flops}{cycle} = \frac{256(vector\ length)}{64} \times 2(FMA) \times 2(FP\ unit) = 16 \text{ ②}$$

From 1.1 we know the Ncores = 24, CPU frequency = 2.5GHz. So one node's Rpeak is:

$$Rpeak = 24 \times 2.5 \times 16\ G\ flop/s = 960Gflop/s$$

That is higher than our result 603.14Gflop/s. If we take our testing result as the Rmax, our node's LINPACK efficiency is:

$$\eta = \frac{Rmax}{Rpeak} \times 100\% = 62.83\%$$

3.2 CPU's actual frequency while running HPL

To know the CPU's actual frequency while running, we use the following command to catch the CPU information to CPUrunning.txt .

```
1.    cat /proc/cpuinfo > CPUrunning.txt
```

We open the file and check the processor id and its cpu MHz. The following graph3.2.1 and graph3.2.2 are the processor and its cpu MHz. They are grouped according to the NUMA node.

processor: 0 cpu MHz: 2083.859	processor: 2 cpu MHz: 2083.894	processor: 4 cpu MHz: 2083.955	processor: 6 cpu MHz: 2083.989	processor: 8 cpu MHz: 2084.028	processor: 10 cpu MHz: 2084.057
processor: 12 cpu MHz: 2084.097	processor: 14 cpu MHz: 2084.131	processor: 16 cpu MHz: 2084.162	processor: 18 cpu MHz: 2084.196	processor: 20 cpu MHz: 2084.233	processor: 22 cpu MHz: 2084.264

Graph3.2.1 processors in NUMA node0

processor: 1 cpu MHz: 2316.720	processor: 3 cpu MHz: 2316.560	processor: 5 cpu MHz: 2316.453	processor: 7 cpu MHz: 2316.355	processor: 9 cpu MHz: 2316.259	processor: 11 cpu MHz: 2316.171
processor: 13 cpu MHz: 2316.075	processor: 15 cpu MHz: 2315.982	processor: 17 cpu MHz: 2315.886	processor: 19 cpu MHz: 2315.785	processor: 21 cpu MHz: 2315.696	processor: 23 cpu MHz: 2315.612

Graph3.2.2 processors in NUMA node1

As we can see, the processors in NUMA node0 use 2084 MHz CPU frequency in average, and the processors in NUMA node1 use 2316 MHz CPU frequency in average. And all of the processors' CPU frequency are lower than nominal frequency. We take the average CPU MHz $\frac{2084+2316}{2}\ MHz\ =\ 2200MHz = 2.2GHz$, and put it into the Rpeak computation formula② and get:

$$Rpeak' = 24\ \times\ 2.2 \times\ 16\ G\ flop/s\ = 844.8Gflop/s$$

Which shows that if we can booster CPU frequency, we can harvest higher Gflop/s.

Summary

In this exercise, the focus was on conducting an High-Performance Linpack test on a specific Chameleon node. The process began with the construction and installation of the HPL benchmark, utilizing the Intel MKL and Intel MPI library. To streamline this procedure, predefined scripts were employed for compilation.

The subsequent step involved executing the HPL benchmark on a single node while customizing and fine-tuning crucial parameters such as N (problem size), NB (block size), and the P x Q configuration. The aim was to optimize the performance and efficiency of the computation.

Finally, we encompassed an assessment of the theoretical peak Flop/s by the CPU. Additionally, a close examination of the CPU's operating frequency on the node was conducted.

Overall, we learn how to install and have HPL test on Chameleon. And we learn the basic parameters of HPL test, which help us know the main direction to optimize the test. What's more, we found that the CPU frequency is lower than its nominal frequency, which make us think the possibility to optimize it.