AUGUST 2019

Performance monitoring using intel performance counters for HEP applications

AUTHOR:
Khadidja Hadj Henni

SUPERVISOR:
Pablo Llopis Sanmillan

SUMMARY

The HPC service at CERN provides linux batch infrastructure to run high performance computing applications that require MPI clusters.The HPC cluster is therefore dedicated to run MPI programs. with numerous applications running on the cluster and big number of users demanding resources to run their jobs,a monitoring tool is needed to measure performances and so this project aims to integrate a framework that includes intel performance tools in the hpc cluster in order to gather data for both users and service managers about job execution efficiency and give feedback to users about how to improve the performance .

TABLE OF CONTENTS

1. HPC RESOURCES:

The HPC cluster is structured into four partitions:

- batch short
- batch long
- BE short
- BE long

the terms “short” and “long” refer to the job execution time limit, jobs running on short partitions have time limit of two days whereas jobs that run on long partitions can take several days to months.

Batch nodes:

- CPU: 2x Intel(R) Xeon(R) CPU E5-2650 v2 (16 physical cores, 32 hyperthreaded)
- Memory: 128GB DDR3 1600Mhz (8x16GiB M393B2G70QH0-YK0 DIMMs)

BE nodes:

- CPU: 2x Intel(R) Xeon(R) CPU E5-2630 v3 (20 physical cores, 40 hyperthreaded)
- Memory: 128GB DDR4 2400Mhz (8x 16GiB 18ASF2G72PDZ-2G3B1 DIMMs)

2. HPC RESOURCES UTILIZATION EFFICIENCY

Measuring the execution efficiency allow users to know if the code is optimized enough to take advantage of the machine’s computing potentials and capabilities, the more an application is optimized the higher its performance goes and less time it takes to finish the execution.

Resource utilization metrics are the first indicators of job execution efficiency because applications performance greatly depends on how well resources were used and so we shall first explore the most important metrics to measure resource utilisation efficiency.

Collecting data and metrics about HPC applications was made simple with SLURM which is a cluster workload manager that was implemented in the HPC cluster. The accounting data for jobs invoked with slurm are stored in the slurm database from which is it possible to collect data and resource utilisation metrics using **sacct** command, this command displays information about jobs (job steps, status, default exit codes ...) as well as the accounting information, there are numerous job accounting fields however in this project we only focus on resources (CPU and memory) fields. The most relevant metrics are the following:

CPUTime: time used and it equals the job elapsed time multiplied by number of allocated CPUs. This metric does not indicate if the CPU was used efficiently or not because it only depends on one facture and that is the elapsed time which is the amount of time the job took to finish its execution.

TotalCPU time: equals the sum of system CPU and user CPU time used by the job or job step.

System CPU time is the amount of time the CPU was busy executing code in the kernel space whereas user CPU time is the amount of time CPU was busy executing code in the user space. The total CPU time is more precise because it only measures the amount of time in which the CPU was actually busy executing code instructions.

To measure the CPU utilization efficiency, we calculate the ratio of total CPU time to the elapsed time as defined as follows:

$$\mathbf{r} = \boldsymbol{CPU\ utlization\ efficiency} = \frac{\boldsymbol{total\ CPU\ time}}{\boldsymbol{elapsed\ time}} = \frac{\textbf{total CPU time} * \textbf{number of allocated CPUs}}{\boldsymbol{CPU\ time}}$$

if the ratio **r** equals or is close to one (r=1 or r~1) that means that CPU was used very efficiently but if the value of the ratio **r** was far less than one (r <1) then in that case the CPU was not being used in most of the execution time.

to print the CPU time and the total CPU time of a certain job the command **sacct** is used with the option **--format**, this option allows users to tailor job accounting fields and print the desirable fields in the desirable order as well. For example:

sacct -j 106580 --format=JobName,Partition,AllocCPUS,AllocNodes,NTasks,Elapsed,State,CPUTime,TotalCPU

in this example the **-j** option is used to specify the job by its ID and the format of the output is defined by the option **–format**

- **JobName**: name of the job
- **Partition**: identifies the partition on which the job ran
- **AllocCPUS**: count of allocated CPUs
- **NTasks**: total number of tasks in a job
- **Elapsed**: The job elapsed time, the format of this field output is as follows: [DD-[HH:]]:MM: SS where DD: days, HH: hours, MM: months, SS: seconds
- **State**: job status

CPUTime and TotalCPU have the same format as Elapsed

Even if it is useful to detect resource usage inefficiencies this metric remains very simple and limited and does not tell users how they should optimize their code. In addition to that it can be sometimes misinterpreted, for instance, often we think of 100% CPU as a sign of high performance while the truth may not be that, but instead the CPU could be stalled (waiting for memory I/O) and not really executing code instructions

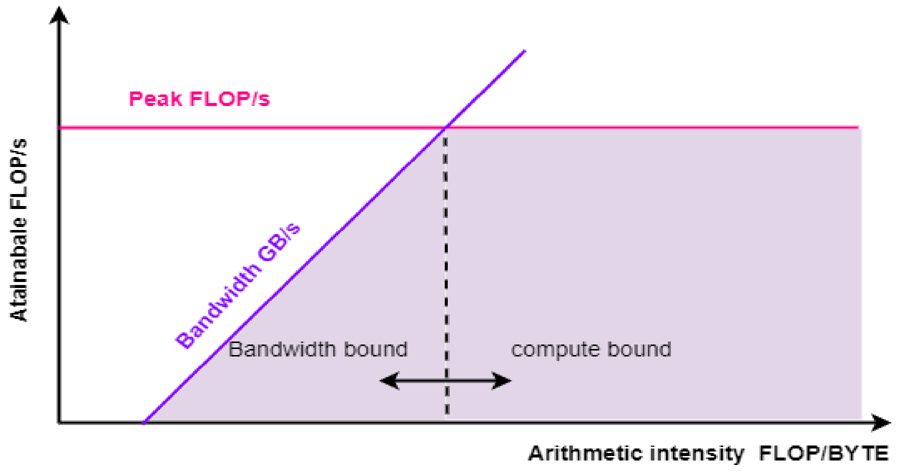
3. ROOFLINE MODEL

The Roofline model provides users with a way to understand the trade-off between data-movement and computation to find out which part of the hardware is limiting the code (CPU or memory), and how close the application is from reaching the machine bounds. It also helps visualize application performance, actual and potential and provide guidance for optimizing the program.

There are two basic concepts that we need to build the Roofline model: **FLOP/s** and **arithmetic intensity**. FLOP/s is the number of floating-point operations that the processor can execute per second and the arithmetic intensity is the ratio of total floating-point operations (Flops) performed by a given code or a code section, to the total data movement (Bytes) required to support those flops

NB: Note that there’s a difference between **FLOP/s** and **Flops** in this report we distinguish between the two terms where the term FLOP/s refers to the computation speed and Flops is the count (number) of floating-point operations.

Roofline model can be visualized by plotting the performance bound (GFLOP/s) as a function of Arithmetic Intensity as shown below:



The horizontal line indicates the machine's maximum FLOP/s while the diagonal line represents the machine peak bandwidth. The intersection of these two lines will determine the machine's achievable performances (the curve shown in gey). Since any given program has a specific arithmetic intensity, when we run it we record how many FLOP/s is achieved and plot it on the graph ,shown in figure above , the coordinates of the application can tell us tell how far we are from reaching the bounds and what type of bound it is (memory or CPU).

To build the roofline model we first measure the machine's peak performance and peak bandwidth that will constitute the roofs of our model, this could be done using the vendor specifications that give an insight of the machine maximum performances and capabilities however in practice those peaks may not be achievable by any code running on the machine because of various physical factors related to the job's execution environment such as power and energy. However, it is possible to build more realistic roofs that are close enough to the theoretical machine bounds using the Empirical Roofline toolkit.

a. EMPIRICAL ROOFLINE TOOLKIT

The ERT empirically determines the machine's required characterizations for the roofline model, these characterizations include the maximum bandwidth of each level of the memory hierarchy and the maximum GFLOP/s. The roofs are obtained by running various micro kernel codes.

The generated Kernel code is adapted to the machine's architecture through the compilation options in the ERT configuration file and Thus each type of node in the HPC cluster has a specific ERT configuration file.

i. Configuration

The configuration file contains lines that define the parameters needed by ERT to compile the specified derivers and kernels, run them and gather the results the final roofline characterizations of machine.

ERT utilize MPI, OpenMP and Cuda but since the goal is to measure the roofs of machines that only run MPI applications (HPC cluster), we will be using only MPI and OpenMP with ERT

The configuration options used to obtain the roofs of HPC nodes are given in the table down below:

Option	Explanation
ERT_RESULTS	This option specifies the directory where all results of running the ERT will be kept.
ERT DRIVER	the driver code to use with the selected kernel code There is currently only one driver, “driver1”
ERT KERNEL	the kernel code to use with the selected driver code There is currently only one kernel, “kernel1”.
ERT_MPI	set to True or false. this option specifies whether to compile with MPI or not
ERT_MPI_CFLAGS	compilation flags specific to MPI that are needed or wanted.
ERT_MPI_LDFLAGS	linking/loading flags specific to MPI that are needed or wanted.
ERT_OPENMP	set to true or false. specifies whether to compile with Openmp or not
ERT_OPENMP_CFLAGS	compilation flags specific to OpenMP that are needed or wanted.
ERT_OPENMP_LDFLAGS	linking/loading flags specific to OpenMP that are needed or wanted.
ERT_FLOPS	A set of integers specifying the FLOPS per computational element
ERT_ALIGN	An integer specifies the alignment (in bits) of the data being manipulated.
ERT_CC	The compiler to use to build the ERT test(s)
ERT_CFLAGS	the flags to use when compiling the code
ERT_LD	the linker/loader to use when compiling the code
ERT_RUN	The command used to run the driver/kernel code that is built
ERT_PROC_THREADS	A set of integers that constrain valid products of the number of MPI processes and the number of OpenMP threads. For example, if this is ‘8’ then only combinations of MPI processes and OpenMP threads have to multiply to give 8, e.g., 2 and 4, 8 and 1, are run.
ERT_MPI_PROCS	A set of integers specifying possible number of MPI processes to run
ERT_OPENMP_THREADS	A set of integers specifying possible number of OpenMP threads to run

ERT_NUM_EXPERIMENTS	The number of times to rerun the same code with all the parameters set the same
ERT_MEMORY_MAX	The maximum number of bytes to allocate/use for the entire run.
ERT_WORKING_SET_MIN	The minimum size (in 8-byte, double precision floating point numbers) for an individual process/thread working set.
ERT_TRIALS_MIN	The minimum number of times to repeatedly run the loop over the working set.
ERT_GNUPLOT	How GNUplot is invoked.

The value of the previous configuration parameters used to measure the roofs for batch partition nodes are given in the following table:

Option	Value in the ERT config file for Batch nodes
ERT_RESULTS	Results.batch.nodes
ERT DRIVER	driver1
ERT KERNEL	kernel1
ERT_MPI	True
ERT_MPI_CFLAGS	-I/usr/local/mpi/mvapich2/2.2/include/
ERT_MPI_LDFLAGS	-lmpi -L /usr/local/mpi/mvapich2/2.2/lib64/
ERT_OPENMP	True
ERT_OPENMP_CFLAGS	-qopenmp
ERT_OPENMP_LDFLAGS	-qopenmp
ERT_FLOPS	1,2,4,8,16
ERT_ALIGN	64
ERT_CC	icpc -std=c++11

ERT_CFLAGS	-O3 -xHost -fno-alias -fno-fnalias -DERT_INTEL
ERT_LD	icpc
ERT_RUN	**export** OMP_NUM_THREADS=ERT_OPENMP_THREADS; mpirun -np ERT_MPI_PROCS ERT_CODE
ERT_PROC_THREADS	1-16
ERT_MPI_PROCS	1,2,4,8,16
ERT_OPENMP_THREADS	1,2
ERT_NUM_EXPERIMENTS	3
ERT_MEMORY_MAX	1073741824
ERT_WORKING_SET_MIN	1
ERT_TRIALS_MIN	1
ERT_GNUPLOT	gnuplot

ii. Overview on how ERT works

Running ERT is done by using the command "ert" with a single argument that specifies the path to the ERT configuration file.

ERT first reads the configuration file then it creates the output directory specified by "ERT RESULTS" and copies the configuration file there, then for each value of "ERT_FLOP" it builds the roofline characterization code with that number of FLOPs per element.

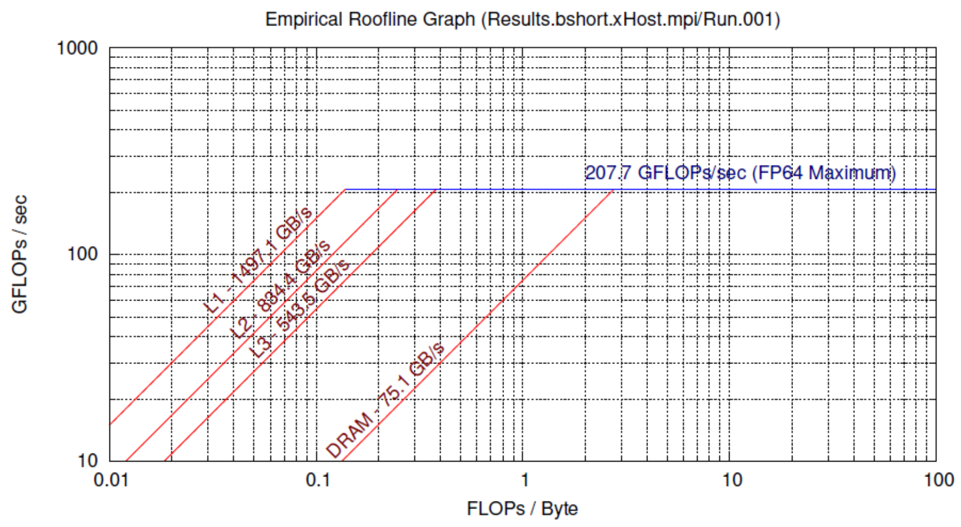
The built code is executed using different combination of number of processes and number of threads that fulfil the constraint specified by "ERT PROCS". In a more programmatic way : For each number of MPI processes specified by "ERT MPI PROCS" and each number of OpenMP threads specified by "ERT OPENMP THREADS" if the product of the number of MPI processes and OpenMP threads is included in "ERT PROCS", the code that was built is executed according to the MPI and OpenMP specification (number of processes /threads).

For each combination of number of processes/threads the code is executed as many times as specified by "ERT NUM EXPERIMENTS".

AT the end ERT gathers all the obtained results to produce the final roofline results as a graph generated by GNUplot in PostScript format and a JSON output file

iii. ROOFLINE MODEL OF BATCH NODES

The following figure shows the obtained roofs for Batch partition nodes



- Peak performance: 207.7 GFLOP/s
- Maximum bandwidth:
    - DRAM: **75.14 GB/s**
    - L1: **1497.12 GB/s**
    - L2: **834.44 GB/s**
    - L3: **543.53 GB/s**

The next step will be to use these roofs to analyse HPC applications performances and to do so we need to measure the FLOPS/s and the arithmetic intensity of the applications and then plot it on the obtained graphs. once this is done users will be able to determine which bound is affecting the application's performance (memory or CPU) and optimize the code accordingly

b. Calculating arithmetic intensity and FLOPs/s of HPC applications

Application performance in flops/s is the number of floating-point operations performed by the application per second

$$Application\ \ Flops/s = \frac{flops\ count}{run\ time}$$

The application run time can be obtained using the slurm accounting command "**sacc**t" and since the HPC nodes have intel architecture, flops count can be measured using intel software development emulator SDE.

SDE traces dynamic instructions and includes a histogram tool that captures different information about the code instructions such as the instruction length and type.

an example of SDE command line to measure the flops of an application running on the HPC cluster:

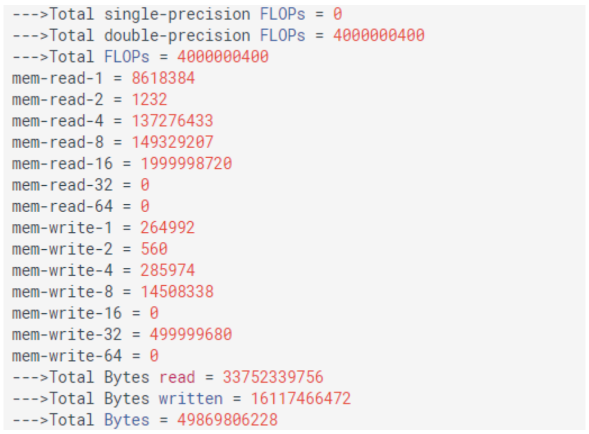
srun -p batch-short -t 1:00:00 -n 64 sde64 -ivb -d -omix file.out -i -global_region --user-application

- **Ivb**: is used to specify Intel Ivy Bridge CPU
- **-d**: specifies to only collect dynamic profile information
- **-o** : specifies the output file
- **-global_region**: to include any threads spawned by a process (for OpenMP)
- **-i**: specifies that each process will have a unique file name based on process ID

SDE will create a file for each process generated by the application, and if the application contains threads (OpennMP) then information related to threads are included in the same files (the option global-region enables this)

The output file has too many information and it is not easy to read, therefor we use the script **parse-sde.sh** in https://bitbucket.org/dwdoerf/stream-ai-example/src/master/parse-sde.sh to parse the results

The below figure shows an example of parsing SDE output files:



```
--->Total single-precision FLOPs = 0
--->Total double-precision FLOPs = 4000000400
--->Total FLOPs = 4000000400
mem-read-1 = 8618384
mem-read-2 = 1232
mem-read-4 = 137276433
mem-read-8 = 149329207
mem-read-16 = 1999998720
mem-read-32 = 0
mem-read-64 = 0
mem-write-1 = 264992
mem-write-2 = 560
mem-write-4 = 285974
mem-write-8 = 14508338
mem-write-16 = 0
mem-write-32 = 499999680
mem-write-64 = 0
--->Total Bytes read = 33752339756
--->Total Bytes written = 16117466472
--->Total Bytes = 49869806228
```

Calculating L1 level arithmetic intensity:

The arithmetic intensity is the ratio of total flops to the total bytes count, the bytes count can be calculated for different memory levels, and since the SDE reports the total bytes on L1 level we can calculate the L1 level arithmetic intensity as follows:

$$\boldsymbol{AI(L1)} \quad = \frac{\boldsymbol{total\ flops}}{\boldsymbol{total\ bytes\ on\ L1\ level}}$$

L1 level AI of the above example is: $AI(L1) = \frac{4000000400}{49869806228} = 0.0802$

It is important to know that running SDE can slow down the execution and degrade application's performances for example we ran intel MKL benchmark that consists on matrix multiplication, the number of multiplication and the size of the matrix were big enough to stress the CPU and so the run time without SDE was about 25 s but when running it with SDE to measure the flops the run time was far more greater 67 s which is about 2 times the original elapsed time.

4. Conclusion and future work

The project took an experimental approach in the sense where many tools and methodologies were tested on the HPC cluster: SLURM accounting, Roofline model, ERT and SDE. The present report documents these tools and explains different ways to understand and measure HPC application performances however it did not cover the implementation phase due to time limitation. Hopefully this is a milestone toward future work that includes combining intel performance tools in one framework that can provide users with enough data and metrics measured by a diversity of tools to enable them to analyze and improve the performance of HPC applications.

5. REFERENCES

- https://docs.nersc.gov/programming/performance-debugging-tools/roofline/

- CERN Batch Service User Guide: http://batchdocs.web.cern.ch/batchdocs/index.html
- https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
- https://insidehpc.com/2019/02/improving-hpc-performance-with-the-roofline-model/
- https://bitbucket.org/berkeleylab/cs-roofline-toolkit/
- https://software.intel.com/en-us/articles/intel-software-development-emulator
- https://slurm.schedmd.com/quickstart.html