Profile Util library: A quick and easy way to get MPI, OpenMP and GPU runtime information

Pascal Jahan Elahi[1]

Pawsey Supercomputing Research Centre, Kensington, WA, Australia
pascal.elahi@pawsey.org.au

Abstract. We present `profile_util`, a quick and simple way of profiling codes. This is a MPI, OpenMP, and GPU enabled C++17 library. The GPU interface is compatible with both HIP and CUDA and is compatible with more than a single GPU per MPI process. It provides a means of logging MPI, OpenMP and GPU related information, such as number of threads, core affinity, GPU properties. It can also be used to log memory usage or available memory. The library can also report the time taken on both host and devices, statistics of CPU usage, statistics of GPU usage, memory usage and power. The simple API means minimal changes are required to any C++ library.

1 Introduction

Addressing performance bottlenecks in a code is key to ensuring that it can run efficiently at scale. Profiling tools are essential for identifying performance issues by measuring various metrics such as CPU usage, GPU usage, memory consumption, and execution time. There are a wide variety of tools that provide a comprehensive view of a code. However, these are typically closed-source commercial products, often requiring instrumenting a code, or are tailored to a specific API (see for example Linaro Forge, Intel VTune, NVIDIA Nsight Compute, Omniperf, Paraver[2], Tau[8], or ScoreP[6]). Moreover, they are not designed to provide a view into a code's performance in daily production-scale runs, where minimal impact and a higher level view is desirable.

Here we present `profile_util`, an open-source library which can be simply integrated into codes for production-scale runs (freely available https://github.com/pelahi/profile_util.git. This C++17 code uses the CMAKE build system, and MPI [1], OpenMP [3], and GPU (CUDA and HIP) parallel APIs.

2 Description

The library is designed so that it can be built with any one or combination of OpenMP, MPI, and CUDA/HIP or even for codes that are purely serial. The library only makes use of the C++17 standard library, the relevant libraries required for the desired parallelism profiling, and a few simple Linux utilities, such as `ps` or for GPUs, `nvidia-smi`/`rocm-smi`.

Inclusion into a code base is a matter of using a simple set of APIs and compiling the code with the appropriate library. An example is provided and we will discuss some key API calls.

Listing 1.1. Sample code using the API

```
// simple MPI+OpenMP code
#include <iostream>
#include <vector>
#include <mpi.h>
// include the profile_uti.h header
#include <profile_util.h>
int main() {
    // init MPI
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    // Set the Logging communicator of the profile util library
    MPISetLoggingComm(comm);
    // a call to get parallel information of code
    LogParallelAPI();
    // a call to get thread affinity
    LogBinding();
    // construct timer for openmp loop
    auto timer = NewTimer();
    std::vector<double> vec(100); auto sum = 0.0;
    // parallel loop, add LOGGING() to ensure openmp logging.
    #pragma omp parallel default(none) shared(vec) LOGGING()
        reduction(+:sum)
    {
        // get affinity for every thread spawned.
        #pragma omp critical
        LogThreadAffinity();
        #pragma omp for
        for (auto &x:vec) {sum+=x;}
    }
    //report time taken
    LogTimeTaken(timer);
    MPI_Finalize(); return 0;
}
```

Listing 1.2. Sample build

```
g++ -L/<path/lib> -I/<path>/include -lprofile_util source.cpp -o exe
```

The key API calls fall into several categories: general parallel information, timing and performance, and memory. Every logging call will report what rank called the routine, from what function and what line in the file and at what time. The API uses `std::cout` for `Log*()` calls but there is also an interface where the user can pass a specific `ostream` using `Logger*(ostream)` calls. There are

also MPI variants where only rank 0 in the communicator passed to the library reports the information, `MPIRank0*()`

2.1 General parallel information

These calls can provide information about the number of MPI ranks, number of threads, the core affinity for MPI process, the number of GPUs per MPI process and the information about the GPUs. The key calls are:

•`LogParallelAPI()` reports the parallel API's used. Example output:

```
Parallel API's
========
MPI Comm world size 2
OpenMP version 201811 with total number of threads = 2 with total number
    of allowed levels 1
Using GPUs: Running with HIP and found 4 devices
```

•`LogBinding()` reports the overall binding of cores, GPU information (such as PCI address) for every MPI process. Example output:

```
Core Binding
========
On node nid003012 : MPI Rank 0 : OMP Thread 0 : at nested level 1 : Core
    affinity = 0-7
On node nid003012 : MPI Rank 0 : OMP Thread 1 : at nested level 1 : Core
    affinity = 0-7
Current runtime environment gpu list is :0,1
On node nid003012 : MPI Rank 0 : GPU device 0 Device_Name=
    Bus_ID=0000:c1:00.0 Compute_Units=110 Max_Work_Group_Size=64
    Local_Mem_Size=65536 Global_Mem_Size=68702699520
On node nid003012 : MPI Rank 0 : GPU device 1 Device_Name=
    Bus_ID=0000:c6:00.0 Compute_Units=110 Max_Work_Group_Size=64
    Local_Mem_Size=65536 Global_Mem_Size=68702699520
```

•`LogThreadAffinity()` reports core affinity of threads of a given MPI rank and is designed to be called in a OpenMP environment to see if there are issues with oversubscription.

2.2 Timing & Performance

The library provides a timer class, which allows code to be profiled with minimal additions. This timer can wrap specific calls and is ideal for providing general timing information of key functions, allowing users to check if performance has been greatly impacted when moving architecture or updating the code in a quick fashion. It requires creating a `Timer` object with `auto timer = NewTimer();`. The relevant calls are:

•`LogTimeTaken(timer)` reports the time taken from creation of Timer to point at which this function is called. Example output:

```
@allocate_mem_host L132 (Wed Jul 24 13:41:03 2024) : Time taken between
    : @allocate_mem_host L132 - @allocate_mem_host L106 : 1.296 [s]
```

•`LogTimeTakenOnDevice(timer)` reports the time taken on the GPU device from the creation of the Timer and when this function is called. This makes use of the creation of device events. If the current device is not the one upon creation, the code will move to the device upon creation to get the elapsed time and then move back to the current device. Example output:

```
@allocate_mem_gpu L177 (Wed Jul 24 13:41:03 2024) : Time taken on device
    between : @allocate_mem_gpu L177 - @allocate_mem_gpu L158 : 33 [us]
```

These timers are local to each MPI process. Work is in progress to add a simple API to synchronize, generate and collect a timer across all MPI process in a given MPI communicator and report summary of all times or a statistics of the timing. Currently, some codes taking this approach make use of calls within the `profiling_util` namespace and the timer class functions.

The library also provides a `Sampler` class which uses C++ threads to spawn monitoring processes to get quantities like CPU usage, GPU usage and energy, reporting back the average, standard deviation, minimum and maximum during the sampled period. Like the `Timer` class, it does require creating a sampler with `auto sampler = NewSampler(sample_time_in_seconds);` and, like the timers, samplers are local to each MPI process. The sampler stores information from external process in a hidden file[1], that are processed for statistics over some time interval. The relevant calls are:

•`LogCPUUsage(sampler)` reports the CPU usage over the time sampled. Example output:

```
@main L386 (Wed Jul 24 13:41:19 2024) : CPU Usage (%) statistics taken
    between : @main L386 - @main L353 over 15.532 [s] :
    [ave,std,min,max] = [ 4458.294, 78.093, 95.200, 5536.000 ]
```

•`LogGPUUsage(sampler)` reports the GPU usage of all visible GPUs. Relies on using `nvidia-smi` or `rocm-smi`. Example output:

```
@main L387 (Wed Jul 24 13:41:19 2024) : GPU0 Usage (%) statistics taken
    between : @main L387 - @main L353 over 15.559 [s] :
    [ave,std,min,max] = [ 75.558, 3.619, 0.000, 100.000 ]
```

•`LogGPUEnergy(sampler)` like `LogGPUUsage(sampler)` but reports power consumption along with total energy consumed.

•`LogGPUMem(sampler)` like `LogGPUUsage(sampler)` but reports memory.

•`LogGPUMemUsage(sampler)` like `LogGPUUsage(sampler)` but reports memory in percent used.

[1] Files are located in the runtime directory. Name is `.sampler.<properties>.<unique_id>.txt`

•`LogGPUStatistics(sampler)` like `LogGPUUsage(sampler)` but reports all aspects of GPU state (usage, memory, power).

2.3 Memory

These APIs report the memory usage by the process and state of memory on the node that the process is running on. Key APIs are:

•`LogMemUsage()` reports current and peak memory usage by the process. Example output:

```
[00000] @main L947 (Wed Jul 24 11:00:56 2024) : Node memory report @
    main L947 :
   Node : nid002950 : VM current/peak/change : 33.097 [GiB] / 91.230
       [MiB] / 0 [B]; RSS current/peak/change : 183.777 [MiB] / 0 [B] /
       0 [B]
```

•`LogSystemMem()` reports the memory state of the node on which the process is running. Example output:

```
[00000] @main L948 (Wed Jul 24 11:00:56 2024) : Node system memory
    report @ main L948 :
   Node : nid002950 : Total : 251.193 [GiB]; Used : 24.997 [GiB]; Free
       : 229.620 [GiB]; Shared: 1.429 [GiB]; Cache : 7.451 [GiB]; Avail
       : 226.196 [GiB];
```

3 Use-cases

3.1 MPI performance

A number of users of the Setonix HPE Cray EX system located at the Pawsey Supercomputing centre were having issues running MPI jobs [We refer readers to 5, for more details]. Multi-node jobs with many MPI processes with large message sizes and internode communication were crashing with numerous different reported errors. The error messages were varied and a single code could encounter all of them if run multiple times, so initially, it was not obvious what was causing the wide variety of errors. Due to the severity of the crashes, core dumps were not particularly useful. Further investigations using open-source codes with useful logging information (such as VELOCIraptor, [4] and SWIFTSIM, [7]) indicated the errors occurred during MPI communication and were more likely to occur with increasing per node memory usage and/or increasing message sizes.

By incorporating the library into a simple MPI unit, we were able to gain invaluable information. The logs produced by the code showed that the available node memory did not match the memory used by the MPI processes plus that required for running the OS. Significantly more memory was being used *if and only if* after some communication had occurred. It became quite clear based on

the logs that the MPI associated libraries had at least one memory leak and these leaks only occurred when multi-node communication was involved.

For example, running a 8 node, 50 MPI process per node job would show during an `MPI_Allreduce` that the code along with the OS expected to use ~ 77 GB of memory yet library would report ~ 105 GB being consumed on the node. As the number of processes increased, this unaccounted for excess memory would increase as well, along with the likelihood of encountering uninformative errors like "bus error". This information combined with extra information of node state was vital to HPE Cray for diagnosing the issue.

3.2 GPU performance

While running on an Nvidia Grace Hopper system, users encountered unusual performance degradation in the Hopper GPU. There were indications that this degradation was occurring when the load on the Grace CPU was high. Again, by incorporating the library into a simple code designed to run compute on both CPUs and GPUs at the same time we were able to confirm and quantify the degradation and when it occurred.

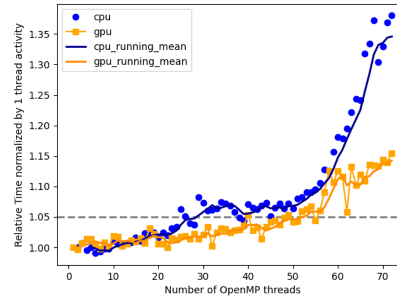

Fig. 1. GH200 Performance: Time taken to run a kernel on GPU and CPU as the number of threads running on the Grace CPU is increased normalized by the time taken when a single thread is running on the Grace CPU. Worse performance is values > 1.

In Fig. 1, we show the time taken to complete a computationally intensive, CPU bound kernel on the CPU and GPU as the number of CPU threads is increased. There is a small steady increase in the time taken on the GPU to complete the kernel till ~ 54 cores are running on the Grace CPU. At this point, there is a marked increase in the time taken on the GPU and the CPU as well, with GPU slowing down by $\sim 15\%$ and CPUs slowing down by $\sim 20 - 30\%$.

Looking at the CPU and GPU usage and parallel efficiency in Fig. 2, it is not a reduction in usage. The usage and maximum usage does not drop significantly at the same point at which there is a noticeable performance drop. In fact, for the Grace CPU, there is an increase in the amount of usage and efficiency. There is a

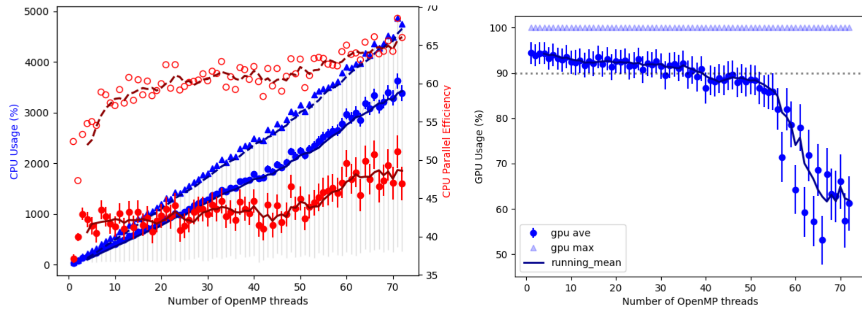

Fig. 2. GH200 Usage: Grace usage (blue points) and efficiency (red points) as number of threads increased (top) and Hopper Usage. For both, we plot the average ± standard deviation along with the maximum (blue triangles and open red points).

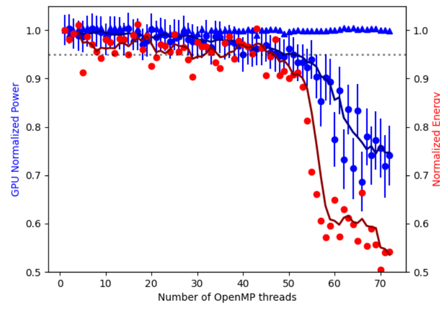

Fig. 3. GH200 Power: Power (blue) and energy (red) consumed by Hopper. We plot the average ± standard deviation along with the maximum for the power. We plot the running mean as well.

drop in the average GPU usage and increased variability, though the maximum remains the same.

The Hopper's drop in average usage is reflected in the drop in average power consumed as seen in Fig. 3. This analysis is operationally informative since it allows us to configure a GH200 based cluster and inform HPC Centres how they may want to setup shared GH200 access.

4 Summary

We have presented `profile_util`, a C++17 MPI, OpenMP, and GPU enabled library that can be incorporated into an existing code base with minimal changes. We highlight two cases where this approach to profiling was very useful in identifying performance and stability issues. Future work will add interfaces for Intel GPUs using `sycl` and oneAPI.

Acknowledgements

We would like to acknowledge the Whadjuk people of the Noongar nation as the traditional custodians of this country, where the Pawsey Supercomputing Research Centre is located and where we live and work. We pay our respects to Noongar elders past, present, and emerging. This work was supported by resources provided by the Pawsey Supercomputing Research Centre with funding from the Australian Government and the Government of Western Australia.

Bibliography

[1] Message passing interface forum, mpi: A message-passing interface standard. https://www.mpi-forum.org/, https://hpc.nmsu.edu/discovery/mpi/introduction/ (March 1994), accessed: 2022-12-12

[2] Computadors, D., Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. WoTUG-18 **44** (03 1995)

[3] Dagum, L., Menon, R.: `OpenMP`: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE **5**(1), 46–55 (1998)

[4] Elahi, P.J., *et al.*: Hunting for galaxies and halos in simulations with VELOCIraptor. Publications of the Astronomical Society of Australia **36**, e021 (May 2019). https://doi.org/10.1017/pasa.2019.12

[5] Elahi, P.J., Meyer, C.: Stress-less MPI Stress Tests. In: CUG23. CUG, vol. 2023 (May 2023)

[6] Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In: Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2011. pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[7] Schaller, M., Borrow, J., Draper, P.W., Ivkovic, M., McAlpine, S., Vandenbroucke, B., Bahé, Y., Chaikin, E., Chalk, A.B.G., Chan, T.K., Correa, C., van Daalen, M., Elbers, W., Gonnet, P., Hausammann, L., Helly, J., Huško, F., Kegerreis, J.A., Nobels, F.S.J., Ploeckinger, S., Revaz, Y., Roper, W.J., Ruiz-Bonilla, S., Sandnes, T.D., Uyttenhove, Y., Willis, J.S., Xiang, Z.: SWIFT: A modern highly-parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications. Monthly Notices of the Royal Astronomical Society **530**(2), 2378–2419 (May 2024). https://doi.org/10.1093/mnras/stae922

[8] Shende, S.S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482