MPI Application Profiling

Gain insights into MPI application (at Scale)

1. Why profiling?
2. Where to start?
3. How to see it for real?
    - BSC Performance Tools quick tour
    - Performance analysis methodology
4. Any second opinion?
5. Sharing some profiles taken from the APAC HPC-AI 2021 and 2022 competition
6. Things we wish we knew before

Why profiling?

In typical practice (with no profiling), to make the workload fast, we normally go with:

1. Elapsed time
2. Scaling plots (speedup, efficiency)
3. Next step (profiles)
   But these approaches are rough estimations:
4. Lacks structural insight (initialization, long runs/ specific timing)
5. Too coarse aggregation (risk of speculating about causes of observed behavior with little capacity of verifying hypotheses)
6. Time is not the only thing to blame

Why profiling?

1. **Realistic understanding** of the structure of the application – actual execution flows to gain "structural insight" of the application at scale.
2. Quantifying the **real opportunities** for enhancing applications performance (know where to blame, e.g., load imbalance, memory issues, communications, floating-point precision, etc.)
3. Know that the **program model** is important (application – algorithm – program model – runtime – architecture – hardware)

Profiling approach is same as build approach.
Three ways to build application on HPC:

1. Module load
    - Loading the binaries provided by the HPC Center's Modules (e.g., QE 7.0).
2. Native built
    - Building the application from source code directly to the user space customized libraries by the available Modules
    - To use existing software libraries available on the HPC system and install some necessary dependencies on Team's project folder
3. Containerized
    - More flexibility
    - Creating the customized environments in a Singularity Container using Inside-Out or Outside-In execution

Where to start?

1. Looking for **existing tools available** at the HPC Center, study the user manual, previous training materials.
2. Start with profiling all versions of the application available via module load.
    - Example of Module Load approach using Arm HPC Tools.
    - Give a quick overview of app profile: collecting execution profile (.map) and generating a report.

> Example: collecting profiles using arm map
>
> **Tool:** ARM MAP on ASPIRE 1 NSCC-Singapore
> **Target workload:** GROMACS 2018.2
> **Input file:** STMV, Lignocellulose
> **Aim:** Single-node-profile – to collect a profile data file (.map file) of a single node

```
module load gromacs/2018.2/gcc493/impi_cude
module load arm/forge/19.1.4
map --profile mpirun -np <int>
mdrun_mpi -s <inputfile_path> -v -noconfout -resethway -nsteps
10000 \ -pin on -nb cpu -ntomp <int>
```

- `--profile` can keep the profiling running.
- Arm MAP is a parallel profiler that shows the running lines of code, and explains.
- The tool does not require any complicated configuration, and design for dummies.
- Use Window ARM Force version to view the .map file. It will show the time breakdown for the application. E.g., most of the time is used in OpenMP. It will also show execution breakdown line-by-line to know which part spends the most CPU time, overhead, etc.

3. Another way is to run the profile module on the HPC Center:

> Example: generate a performance report
> Output file:

```
module load arm/report/19.1.4
perf-report mdrun mpi_12p_1n_2t_2021-10-26_14-54.map
```

- Beautiful HTML report will be generated. It can show what type of application it is, e.g., compute-bound, MPI-bound, or I/O-bound.
- For example, it will show you the OpenMP breakdown.

GROMACS Profile

After the profiling, we found the parameters that can be improved:

1. STMV - `<nstep>` - <nnode x MPI per node x OMP per node> (sensitive to pp x pme ratios)
2. LIGNO - `<nstep>` - <nnode x MPI per node x OMP per node> (has no pme)

Refer to the video to know more about the findings.

What information could we get?

1. The workload is computation or communication intensive and to how much extent?
    - For example, GROMACS is computation intensive.

    - The profiling results confirm that around 15% of the time was used in MPI communication.
2. Which factors affect workload performance, and in which direction?
    - For example, in GROMACS, the domain-decomposition grid size is the dominant factor.
    - The performance of the GROMACS STMV model and GROMACS LIGNO increases when increasing the number of MPI tasks, and reducing the OMP tasks.
    - Setting the values of OMP_NUM_THREADS in the model affects performance.
    - Therefore, we have to test with multiple settings to find the best performance.
3. Which function/ areas of source code are hot spots?

However, the challenge of ARM map is using ARM map to profile the Containerized workload.

How to see the execution behavior for real?

Quick tour of the BSC Performance Tools

Suggest to download Paraver

Success stories and performance reports of profiling applications from Europe

The success stories and technical reports can tell us that what should we do to profile the application and make the performance faster.

The steps we used:

1. Build or download the tools
2. **Extrae** (extract) collecting execution trace: Make sure other jobs are not running else it will extract and exceed the disk quota
3. **Paraver** (on HPC) merge the traces - **Clustering** viewing the burst of execution - **Tracking** see how exec. scales
4. Zip and download .prv file to the local machine - Paraver (on HPC) can filter the useful event to minimize the zip and download time!
5. **Paraver** (PC) visualizing the trace

For example, we work with 4 different versions of GROMACS.
For example, we work with 5 different versions of Quantum Espresso software.

Example of native built approach: Building Quantum espresso 7.1

1. Configure QE
2. Make install with `cmake`
3. Build Quantum Espresso 7.1 on Gadi frontend (configure in /build and install in /bin)
   Refer to the video to see the source codes.

Example of containerized-built approach: QE 7.1 (without OpenMP)

1. Build a singularity image with all dependencies
2. Build QE7.1 on GADI
   Refer to the video to know more about the findings.

Example of containerized-built approach: GROMACS 2021.3 and 2022.beta
Building BSC Tools: Extrae, Paraver, Clustering, Dimemas

1. Build a singularity image with all dependencies
2. Build GROMACS on NSCC
   Some libraries might not be available in the host. More than 20 libraries can be configured.
   Refer to the video to know more about the findings.

Collecting trace

After the tool is built, we must collect trace. It's like submit a job to run the application, but need to call the tool before running the application.

1. `job.pbs` : Tracing job
2. `Extrae.xml` : Defines things to be collected
3. `trace.sh` : Sets preload libraries
   Refer to the video to know more about the findings.

We can ***download the MPI profile examples***. recording | 00:58:00

Have a trace file already, then what?

1. Analyze reference "original" versions of the code

    - Analyze the pure MPI first and see how the application looks like recording | 1:00:18 - we are expecting the CPUs to have parallel at 1 (100% parallel), which GOMAC has
    - Even "non-optimized" versions
    - Try to avoid implementing "solutions" before a deep understanding of behavior and potential gains in your specific case
2. Refer the recommended journey recording | 1:05:00
    - More demonstration recording | 1:08:00
3. Identify the structure
    - Clear iterative pattern
    - With an initialization phase
    - All iterations are similar - we can select a few to analyze
    - Usually use "MPI calls" view or "Useful Duration" - the color indicates the CPU time used - homogeneous (green) means the CPU is working and not wasting
4. Paraver shows how GROMACS behave
    - Can use histogram and some numbers, even using Python code to summarize and visualize it (`simplemetrics.py`) recording | 1:34:00
    - More advanced tool: Clustering (`clustering-jobs.pbs`) - the clustering results can be shown in many files, such as .csv, .prv, .gnuplot recording | 1:39:00
    - The plot can show that when we increase the number of nodes (parameters), how the computation performance (high is good).
5. Dimemas
    - Simulate ideal connection - no delay between intra-nodes.
    - Comes with the preset configuration of ideal network `ideal.cfg` .
    - It can show that if we have ideal connection, what is the duration of computational time.

And still...
the trees can hide the forest...
Sometimes it can be not easy for finalization (e.g., there are 6 iterations and 1 finalization phase, iterations are not regular along the time - different patterns of load balance/ durations)
Thus...
Need to abstract top down models
To steer the identification of causes of performance issues and how they can be counteracted.
Has hybridizing (MPI+OpenMP) fundamental mechanisms to achieve better

performance?
Should we improve the communication scheduling, the element numbering, etc.
* What are expected gains if we address an issue?

Good performance metrics are easy to measure and communicate.
We need to find another way to confirm our observation.

1. Floating-point operations per second (flop/s; not FLOPS)
2. Other popular metrics: response time, throughput, bandwidth, utilization, mean time between failures (MTBF), supportable load, speedup, scalability (weak/ strong)

Evaluation methodology and their tradeoffs

1. Analytical modeling
    - Limited resources requirements; fast (in theory)
    - Rarely accurate; simplified assumptions
2. Computer simulation
    - Possible without system, no interference, explore variations
    - Infeasible for large systems, very slow, semi-accurate
3. Real measurement
    - Most convincing; fast; can be very accurate
    - High costs; not very flexible, outside interference

10-steps scientific benchmarking in theory
Old school metrics

1. State benchmark goal (latency, throughput recording | 1:50:00, energy efficiency) - Define boundaries (internal/ external influences)
2. List services and outcomes (workload input data set, DB queries)
3. Select metrics (speedup, Gflop/s, accuracy, availability)
4. Select factors to study (parameters to be varied during evaluation)
5. Select evaluation technique (analytic modeling, simulation, measurement)
6. Select workload (appropriately sized for the experiment)
7. Design experiment for max. information w/ min effort (scripts, timings, allocation, version control)
8. Analyze and interpret data (consider the variability, use statistics)
9. Document process and present results (communication, easy understanding)

10. Improve your methodology (learn from your and others mistakes to do better next time)

Evaluation of CPU performance

- CPU performance is evaluated by its **response time**, i.e. how long it takes to execute a program and produce results
- The response time of a program A can be split into 3 parts:
  - User CPU time: the time that the CPU spends for executing A recording | 1:49:54
  - System CPU time: the time that the CPU spends for OS to facilitate the execution of A
  - Waiting time of A: the time that the CPU does not execute A, caused by waiting the completion of I/O operations and by the execution of other programs because of time sharing

Performance metrics for parallel programs

1. Parallel runtime
2. Speedup factor or parallel speedup recording | 1:51:05
3. Efficiency
4. **Amdahl's law**
5. **Gustafson-Basis's law**
6. **The Karp-Flatt metric** - good recording | 1:53:00

Refer to the recording for more demonstrations recording | 1:54:40.

Things we wish we knew before

Important map about Gadi steps recording | 2:05:52.
Understand the file structure.
Profiling with kernel of CUDA.