Integrating performance analysis in JupyterLab for OpenMP and MPI

Klaus Noelp, Lena Oden

University of Hagen

Overview

- Motivation
- OpenMP and MPI in Jupyter Notebooks
- Goals of our performance tool
- Case study: Developing compression algorithms
- Summary

Why integrate performance analysis?

- We teach parallel programming and code performance
- Performance tools are difficult to use for beginners
- Manual analysis is time-consuming

> **Integrating performance analysis saves time and effort**

Related work

Bridging between Data Science and Performance Analysis: Tracing of Jupyter Notebooks

Elias Werner
Lalith Manjunath
Jan Frenzel
Sunna Torge
{elias.werner,jan.frenzel,sunna.torge}@tu-dresden.de
lalith.manjunath@mailbox.tu-dresden.de
Technische Universität Dresden
Center for Information Services and High Performance Computing (ZIH)
Dresden, Germany

ABSTRACT

In the last years, an increasing amount of available data has led to new application approaches and an application field that is now called data science (DS). Such applications often require low runtimes while having to deal with restricted compute resources. Up to now, we perceive that the DS community lacks tool support for runtime and resource usage investigations. Thus, we present an approach that combines DS and performance analysis from the High Performance Computing domain. Our concept integrates the measurement framework Score-P in Jupyter, a popular editor for the development of DS applications. We designed and implemented a custom Jupyter kernel that collects runtime data and applied it

1 INTRODUCTION

The availability of large amounts of data in different application domains like Internet of Things (IoT), mobile data, health data and many others, enabled a broad variety of data analytics approaches and data-driven AI, leading to new applications and systems. In the area of machine learning (ML), deep learning (DL) approaches were applied extremely successful e. g. in computer vision or natural language processing and enabled systems for every day use, but also in specific domains as e. g. the medical domain [24], material science [22], transportation [16] and many others. Comparably, other approaches based on explorative statistics and unsupervised learning, making use of large amounts of data, led to data science (DS)

https://github.com/score-p/scorep_jupyter_kernel_python

Quickstart (from README)

1. Download archive from https://fernuni-hagen.sciebo.de/s/mwRMDXht2TshHx8
2. Extract archive and `cd`
3. Run prebuilt image (or build from Dockerfile)

```
1 docker run --rm --name xeus -p 8888:8888 -v "$(pwd):/home/jovyan/materials:Z" \
2     -it docker.io/fuhagen/xeus:2024-09-06
```

Goals of our performance tool

- Support 3 analysis types:
  1. **Timing**: Test scalability
  2. **Profiling**: Find expensive functions and load imbalances
  3. **Tracing**: See synchronization behavior
- Use compiled code only
- Use comments to mark code regions
- The results are visualized inside JupyterLab

How to run a timing analysis?

- Given: A working C++ notebook

1. Create a main method using only comments

```
1  // start_main
2  ...
3  // end_main
```

2. Add placeholder for the problem size, and add timing sections

```
1  // start_main
2  ...
3  int n = 0;
4  // start_mysection
5  ...
6  // end_mysection
7  // end_main
```

   During analysis the `int n` line becomes `int n = atoi(argv[1]);`

3. Start GUI from the menu bar

Source code replacements

Marker	Replacement	Analysis
`// start_main`	`int main(...){`	all
`// end_main`	`return 0;}`	all
`// start_mysection`	it depends on exec. model*	Timing
`// end_mysection`	it depends on exec. model*	Timing
`// pause_recording`	`SCOREP_RECORDING_OFF();`	Prof./Trac. (opt.)
`// continue_recording`	`SCOREP_RECORDING_ON();`	Prof./Trac. (opt.)

*Possible replacements for `// start_mysection`:

```
1  t_start_{marker}_{count} = MPI_Wtime(); // for "mpi"
2  t_start_{marker}_{count} = omp_get_wtime(); // for "openmp"
3  clock_gettime(CLOCK_MONOTONIC, &start); // for "serial"
```

How to run a timing analysis?

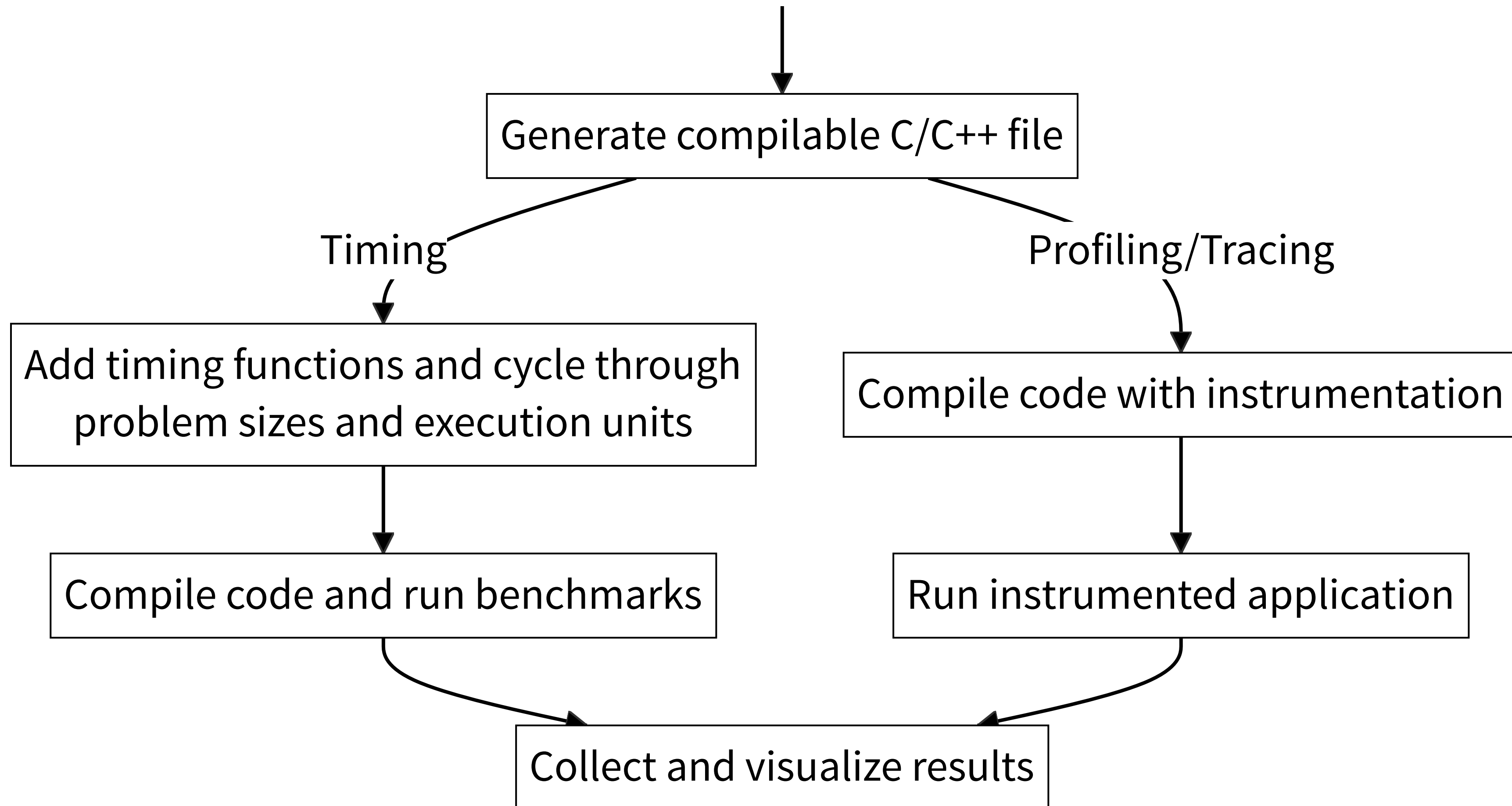
C++ API

```cpp
#include <performance.hpp>
performance p{"/home/jovyan/materials/notebooks/edupar/Pi_EduPar.ipynb"};
display::lazy_image im;
im = p.display_timing({1, 2, 4, 8}, {32, 64}, 10, "-O3", true);
im
```

Python API

```python
import performance.wf_timing.main
from pathlib import Path
plot_path, plot, plots_list, df = performance.wf_timing.main.main( \
    Path("/home/jovyan/materials/notebooks/edupar/Pi_EduPar.ipynb"), "", \
    [1,2,4,8], [32,64], 10, "-O3", True)
plot
```

The automated process



Case study: Developing compression algorithms

- Idea: Change memory layout of floats before compression (Google's snappy library)
- Next week: Results @ *PARS-Workshop 2024* in Ingolstadt
- Now: Demo

Summary

- C++ development and performance analysis is possible in JupyterLab
- It’s easy to add new tools or analysis types

GUI screenshot

File Edit View Run Kernel C++ Performance Tabs Settings Help

Create Performance Diagram for the current notebook

ui_materials | notebooks | Pi

Code

Open in... Python 3 (ipykernel)

The cell below should run automatically.

```
import os •••
```

[1]:

Performance Tool

Timing Profiling Tracing Compiler Optimization

Input File
/home/jovyan/materials/notebooks/Pi_EduPar.ipynb

Number of Execution Units (Threads or Processes)
1,2,4,8

Problem Sizes
1024, 2048, 4096

Number of Iterations
100

Compiler Flags
-O3

☑ Compare Timing Regions

C++ API Python API

```
1 #include <performance.hpp>
2 performance p("/home/jovyan/materials/notebo
3 display::lazy_image im;
4 im = p.display_timing({1,2,4,8}, {1024, 2048,
5
```

Generate Diagram

```
### Python file command runner.py started:
OMP NUM THREADS=8 /home/jovyan/materials/notebooks/Pi EduPar results/Pi EduPar.ex
e 4096 8 /home/jovyan/materials/notebooks/Pi EduPar results/timing 8 4096.csv 15
12 of 12 parameter sets (16 of 100 iterations): problem size = 4096, execution un
its = 8
### Python file command runner.py started:
OMP NUM THREADS=8 /home/jovyan/materials/notebooks/Pi EduPar results/Pi EduPar.ex
e 4096 8 /home/jovyan/materials/notebooks/Pi EduPar results/timing 8 4096.csv 16
12 of 12 parameter sets (17 of 100 iterations): problem size = 4096, execution un
its = 8
### Python file command runner.py started:
OMP NUM THREADS=8 /home/jovyan/materials/notebooks/Pi EduPar results/Pi EduPar.ex
e 4096 8 /home/jovyan/materials/notebooks/Pi EduPar results/timing 8 4096.csv 17
```