Programming Machine Learning Algorithms for HPC

- Profiling

Dr. Pierrick Pochelu and Dr. Oscar J. Castro-Lopez

Profiling

Introduction

Why Profiling?

- **Identify Bottlenecks**: Profiling helps you identify which parts of your code are consuming the most time or other resources. This allows you to focus your optimization efforts where they will have the greatest impact.
- **Data-Driven Optimization**: Profiling provides concrete data about your code's performance. This data helps you make informed decisions about where to optimize based on actual measurements rather than guessing or intuition.
- **Prevent Premature Optimization**: Profiling helps you avoid the common pitfall of premature optimization, which can lead to code complexity and reduced maintainability. By profiling first, you can ensure that you're optimizing areas of the code that genuinely need it.
- **Prioritize Efforts**: When dealing with limited resources (time, budget, etc.), profiling helps you prioritize which parts of your code to optimize. You can focus on the critical sections that have the most significant impact on overall performance.

Why Profiling?

- **Avoid Over-Engineering**: Profiling helps you strike a balance between performance and readability/maintainability. Without profiling, you might over-engineer your code for performance, making it more complex than necessary.
- **Benchmark Improvements**: After making changes to your code, profiling allows you to measure the actual impact of those changes. This helps you verify that your optimizations are effective and didn't introduce new issues.
- **Continuous Improvement**: Profiling should be an ongoing process. As your codebase evolves, new bottlenecks may emerge, or the performance characteristics may change. Regular profiling ensures that your code continues to perform well over time.
- **Debugging**: Profilers often provide insights into unexpected behavior or errors in your code. You may discover unintended inefficiencies or even bugs that are only apparent when looking at performance data.

Measure time in python

- There are several ways to measure the execution time in Python. For example:

- time.time() function: measure the total time elapsed to execute the script in seconds.
  - This value is often referred to as "wall-clock time" or "real time."
  - It includes the time spent in sleeping or waiting for I/O operations, making it suitable for measuring the total elapsed time for a program or a specific task.

- time.process_time(): returns the current CPU time used by the current process in seconds, as a floating-point number.
  - This value represents the amount of CPU time consumed by your program and excludes time spent in sleep or waiting for I/O.

Measure time in python

```python
import time
start_time = time.time()
time.sleep(2.4)
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time} seconds")
```

```python
import time
start_cpu_time = time.process_time()
time.sleep(2.4)
end_cpu_time = time.process_time()
elapsed_cpu_time = end_cpu_time - start_cpu_time
print(f"CPU time used: {elapsed_cpu_time} seconds")
```

Measure time in python

- timeit module: module in Python is a built-in library that provides a simple way to measure the execution time of small code snippets.
  - It has both a Command-Line Interface as well as a callable one.
  - Measures "wall-clock" time.
  - For more details: https://docs.python.org/3/library/timeit.html

```
import timeit

code_to_measure = """
result = sum(range(1000))
"""

time_taken = timeit.timeit(code_to_measure, number=10000)

print(f"Time taken: {time_taken} seconds")
```

Profiling with cProfile

- Python contains a built-in code profiler
- cProfile will run the function and collect data on how much time is spent in each function or method called.
- Usage:

python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)

Profiling with prun

The output consist on the following columns:

- ncalls: number of times a function or method is called.
- tottime: total time spent in the function excluding time spent in subfunctions it calls (in seconds).
- percall: average time per call to the tottime of the function (tottime / ncalls).
- cumtime: cumulative time that indicates the total time spent in function including subfunctions.
- percall: average time percall to the cumtime of the function (cumtime / ncalls).
- filename:lineno(function): provides information about the function or method, indicates the file, line number, and name.

Line profiling

- For a more detailed and granular analysis of code performance, as well as a clear and comprehensible report, you can employ the line profiler. (lprun).
- This can be particularly valuable when you need to pinpoint specific bottlenecks or areas for optimization within your codebase.
- By generating a line-by-line report, lprun allows you to identify exactly where the most time-consuming operations occur, aiding in the optimization process and enabling you to achieve optimal code performance.

Line profiling

- First, we need to install the line profiler library:

```
pip install line_profiler
```

- Add a decorator to the functions we want to profile
- Then we profile the program with the following command:

```
kernprof -l my_script.py
```

- We can check the output with the following command:

```
python -m line_profiler my_script.py.lprof
```

Line profiling with lprun

The output of the line profiler is the following:

- Line #: line number of the code being profiled.
- Hits: indicates how many times each line was executed.
- Time: shows the total time spent on the execution of the line.
- Per Hit: shows the average time, in milliseconds spent on each execution of the line.
- % time: shows the percentage of total execution time spent on each line.
- Line Contents: contains the actual code corresponding to the line being profiled.