Performance of Matrix Multiplication with Python Multiprocessing

1 Introduction

Lecture 1 introduced Amdahl's Law. In this lab, we illustrate it through Python experiments. First, we implement parallelism in Python using `Multiprocessing`, and then we measure performance (execution time, memory usage, and energy consumption). The first part of the lab explains multiprocessing; the second shows how to measure performance.

1.1 Multiprocessing

We compute the cube of each element in an integer array in parallel. We use `Multiprocessing` instead of `Multithreading` because `Multiprocessing` distributes computations across multiple cores, whereas `Multithreading` cannot run parallel parts simultaneously on different cores.

```
import multiprocessing
import importlib
import cube

LEN = 50
NUM_PROCESSES = 4

number_sequence = range(LEN)
result = multiprocessing.Array('i', LEN)
num_per_processes = LEN // NUM_PROCESSES
remaining = LEN % NUM_PROCESSES
processes = []
start = 0

for i in range(NUM_PROCESSES):
    end = start + num_per_processes + (1 if i < remaining else 0)
    p = multiprocessing.Process(target=cube.func, args=(number_sequence, start, end, result))
    processes.append(p)
    p.start()
    start = end

for p in processes:
    p.join()

print(f"No. of Processes: {NUM_PROCESSES}")
print(f"Result: {list(result)}")
```

Listing 1: Multiprocessing.

In Listing 1, we create multiple processes, passing each a target function `func` and its parameters. We store the processes in an array so that we can wait for their completion using `join()`. The function `func` receives a range of numbers, `start` and `end` indices, and a shared `result` variable declared as `multiprocessing.Array()` of integers with length `LEN`. In Jupyter Lab, we define `func` in a separate module (module `cube`, Listing 2) and import it into the main program.

```
def func(number_sequence, start, end, result):
    for i in range(start, end):
        result[i] = pow(number_sequence[i], 3)
```

Listing 2: Module `cube`.

1.2 Performance Measurements

1.2.1 Execution Time

Python offers several ways to measure execution time. In this lab, we use the function `default_timer()` from the package `timeit` as illustrated in Listing 3. The elapsed time is measured in seconds. We typically measure the time of a code snippet at least 10 times and then calculate the average.

```
import timeit

start_time = timeit.default_timer()
# Your code
end_time = timeit.default_timer()
execution time = end_time - start_time
```

Listing 3: Time Measurement using `timeit`.

1.2.2 Memory Usage

Memory usage can be measured using Python's built-in `tracemalloc` module. This module traces memory allocations, which are memory reserved for objects such as lists, integers, and strings. `Tracemalloc` allows us to take snapshots at different points in a program and extracts memory usage statistics that help identify which parts of the code consume the most memory. We can start and stop tracing as needed. Listing 4 demonstrates how to print statistics and report the total allocated memory for a code snippet. The statistics are sorted by line number. Other sorting criteria are also available; see the Python documentation for details.

```
import tracemalloc

tracemalloc.start() # Start tracing memory allocations

# Your code

snapshot = tracemalloc.take_snapshot() # Take a snapshot of the current memory allocations
tracemalloc.stop() # Stop tracing memory allocations

stats = snapshot.statistics('lineno') # Get statistics for memory usage, sorted according to lineno

print("Top 10 memory-consuming lines:")
for stat in stats[:10]:
    print(stat)

# We also find the total memory allocated by summing the memory allocations
total_allocated = sum(stat.size for stat in stats) / 1024 # e.g., in KiB
print("Total memory allocated: ", total_allocated)
```

Listing 4: Memory Allocations using `tracemalloc`.

1.2.3 Energy Consumption

Energy consumption can be measured using the Running Average Power Limit (RAPL) interface provided by Intel processors. This requires Linux and `sudo` rights. In this lab, we describe only the measurement procedure. Linux users on Intel processors can try this on their own systems.

RAPL tracks energy usage using hardware counters that automatically increment. These counters are accessible to software through model-specific registers (MSRs). The supported power domains include:

- package: measures the energy of the entire CPU package, including all processor cores, uncore components (like last-level cache and memory controller), and any integrated GPU. Essentially, everything physically on the CPU chip is included.
- `core`: measures the energy used specifically by the processor cores, excluding the integrated GPU.
- `uncore`: measures energy consumed by the parts of the CPU outside the cores, such as the last-level cache, memory controller, and interconnects.

- `system or platform`: measures the energy usage of the entire computing platform, including the CPU package (cores, uncore, integrated GPU), memory, chipset, and other board-level components. This gives a holistic view of the total system power, beyond just the processor.

The Linux kernel provides the `powercap` subsystem to access MSRs through the `sys/class/powercap` directory hierarchy. Each domain appears as a separate subdirectory containing a file named `energy_uj`. To measure energy consumption, we read the `energy_uj` file before and after executing a code snippet, then calculate the difference and check for overflow, as shown in Listing 5. The values are expressed in joules (J).

```
 # Filepaths for RAPL power doman registers
CORE_DOMAIN="/sys/class/powercap/intel-rapl:0:0/energy_uj"
PSYS_DOMAIN="/sys/class/powercap/intel-rapl:1/energy_uj"

def read_energy_uj(file_path):
    try:
        with open(file_path, 'r') as file:
            energy_uj = file.read().strip()
        return int(energy_uj)

    except FileNotFoundError:
        print(f"File not found: {file_path}")
    except PermissionError:
        print(f"Permission denied: {file_path}")
    except ValueError:
        print(f"Could not convert data to an integer.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Read the counters for each domain
initial_core_energy = read_energy_uj(CORE_DOMAIN)
initial_platform_energy = read_energy_uj(PSYS_DOMAIN)

# Your Code

final_core_energy = read_energy_uj(CORE_DOMAIN)
final_platform_energy = read_energy_uj(PSYS_DOMAIN)

core_energy_used = final_core_energy - initial_core_energy
if (core_energy_used < 0):
    print(f"Counter overflow core energy: {core_energy_used}")
else:
    # save the core energy used

platform_energy_used = final_platform_energy - initial_platform_energy
if (platform_energy_used < 0):
    print(f"Counter overflow psys energy: {platform_energy_used}")
else:
     # save the platform energy used
```

Listing 5: Measuring Energy Consumption using `RAPL` and `Powercap`.

Note: The script has to be run with `sudo`. Since the energy counters report total system energy, you should avoid running other major processes during measurements to get a close estimate of the code's energy usage.

2 Exercise - Amdahl's Law

In this exercise, we demonstrate Amdahl's Law in practice. We measure performance during matrix multiplication while varying the number of processes to observe its effects.

2.1 Matrix Multiplication

Matrix multiplication is computationally intensive. We improve performance by running multiple processes on separate cores. As a reminder, the product of a matrix is computed as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \ddots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{m1} & x_{m2} & b_{m3} & \dots & b_{mn} \end{bmatrix} = \begin{bmatrix} c_{11} & b_{12} & c_{13} & \dots & c_{1n} \\ c_{21} & b_{22} & c_{23} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & c_{m3} & \dots & c_{mn} \end{bmatrix}$$

where the element-wise summation for matrix multiplication is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

For simplicity, we use only square matrices ($n = m$) for our performance measurements.

2.2 Example Multiply Function

One simple way to exploit parallelism is to compute the rows of the product in parallel. Depending on the matrix size and the number of available cores, each core computes at least one row of the product. We use Python's `multiprocessing` to assign a range of rows (`start` to `end`) to each process running on a core. The multiplication function is shown in Listing 6. The result matrix must use *flattened indexing* because it is shared among all processes, as required by the `multiprocessing` package.

```
# Multiply elements of m1 and m2 to compute rows (start, end) of product
def mult_rows(m1, m2, start, end, result):
    for i in range(start, end):
        for j in range(DIM):
            for index in range(DIM):
                flat_index = i * DIM + j
                result[flat_index] += m1[i][index] * m2[index][j]
```

Listing 6: Function for row-wise matrix multiplication.

2.3 Experiment

Now we write a program to explore Amdahl's Law using matrix multiplication. The program consists of a *sequential* part and a *parallel* part. The *sequential* part consists of reading sensor values from two different sources. The sensors are arranged in a grid, whose dimension is equal to the size of our square matrices. The values are randomly generated integers between 0 and 9, and are stored in the CSV files `sensor1.csv` and `sensor2.csv`. They are then loaded into matrices. You can also generate new sensor values using the provided code snippet. The sensor readings are repeated a fixed number of times, and the average is computed. The *parallel* part distributes matrix rows among multiple processes, starts the processes, and waits for their completion. Both the sequential and parallel components are shown in Listing 7.

```
import numpy as np
import csv
import multiprocessing

DIM = 12 #dimension of square matrix
MAX_NUM_PROCESSES = 10 # number of processes (< num of cores on PC)
NUM_READ_SENSOR = 1 # number of times sensor values are read
DELAY = 0.05 # delay between each sensor readings
NUM_REPEAT = 10 # number of times the timing measurements are taken

# Ensure MAX_NUM_PROCESSES does not exceed DIM
if MAX_NUM_PROCESSES > DIM:
    print("The maximum number of processes must be less than or equal to the number of rows in the
        matrix!")
    exit()

# Sequential part: read sensor1 and sensor 2 values as 2 matrices every t seconds
# and calculate average
# Parallel part: perform matrix multiplication on 2 matrices
# using a variable number of processes
num_processes = 1
```

```
m1 = np.zeros((DIM, DIM))
m2 = np.zeros((DIM, DIM))

while num_processes <= MAX_NUM_PROCESSES:
    print("No. of Processes: ", num_processes)
        # Sequential part
        start_seq = timeit.default_timer()
        for _ in range(NUM_READ_SENSOR):
            # Read the sensor1 data
            with open('sensor1.csv', 'r') as file:
                reader = csv.reader(file)
                read_dat_1 = np.array([row for row in reader], dtype=int)

            # Read the sensor2 data
            with open('sensor2.csv', 'r') as file:
                reader = csv.reader(file)
                read_dat_2 = np.array([row for row in reader], dtype=int)

            # Accumulate read sensor values
            m1 += read_dat_1
            m2 += read_dat_2

            time.sleep(DELAY)

        # Calculate average of read sensor values
        m1 = (m1/NUM_READ_SENSOR).astype(int)
        m2 = (m2/NUM_READ_SENSOR).astype(int)

        # Parallel part
        # Matrix multiplication using variable number of processes
        result = multiprocessing.Array('i', DIM*DIM)
        num_rows_per_process = DIM // num_processes
        remaining_rows = DIM % num_processes
        processes = []
        start = 0
        start_par = timeit.default_timer()
        for i in range(num_processes):
            end = start + num_rows_per_process + (1 if i < remaining_rows else 0)
            # fill in this part using multiprocessing to create process to execute mult_rows()
            processes.append(p)
            p.start()
            start = end

        for p in processes:
            p.join()

        result = (np.array(result)).reshape(DIM, DIM)

    # Verify result
    if not (np.array_equal(np.dot(m1,m2), result)):
        print("Verification failed!")

plt.show()
```

Listing 7: Sequential and parallel parts of the test program.

According to Amdahl's Law, the ratio between sequential and parallel execution time determines the overall speedup. This is controlled in the experiment by two constants: `NUM_READ_SENSOR` and `DELAY`, which increase the time spent in the sequential section. In contrast, the execution time of the parallel section depends on the matrix dimension `DIM` of the matrix and the number of processes `MAX_NUM_PROCESSES`.

2.3.1 Task 1 - Implement Multiprocessing

Set `DIM = 4` and `MAX_NUM_PROCESSES = 2`. Create the processes and pass the function `mult_rows()` to each process. Verify if the product is correct.

2.3.2 Task 2 - Measure Execution Time

Using the procedure described in Section 1.2.1, measure the execution time of both the sequential and parallel parts. Repeat the experiment at least ten times and compute the average execution time.

2.3.3 Task 3 - Calculate Speedup

We cannot directly use the formula from the lecture because the exact fractions of sequential and parallel execution are unknown. Instead, calculate the speedup using Equation 1.

$$S = \frac{T_s}{T_p} \tag{1}$$

where

- S is the speedup,
- T_s is the execution time of the program when the parallel section runs with 1 process (baseline),
- T_p is the execution time of the program when the parallel section runs with p processes.

Here T_s equals the sequential part plus the time the parallel part needs with one process. T_p equals the sequential part plus the time the parallel part needs with p processes. As p increases, the time of the parallel part is expected to decrease.

Vary the parameters and calculate the speedup for $p = 1$ to 8, limited by the number of cores on your computer. Create a table of results and plot a graph of speedup versus the number of processes. Explain and discuss your results.

2.3.4 Task 3 – Memory Usage

Using the procedure described in Section 1.2.2, measure the memory usage of both the sequential and parallel parts of the program, and, in particular, the memory used by each process. To measure memory usage in one process, include the `tracemalloc` instructions *inside* the `mult_rows` function and return the collected statistics to the main program. Present the results in a table and report a short analysis of your observations.

2.3.5 Task 4 – Energy Consumption

Using the procedure described in Section 1.2.3, record the core and platform energy consumption for both the sequential and parallel parts of the program. Summarize the results in a table and plot a graph of energy consumption. Discuss your observations. Energy measurements can only be performed on a Linux system with `sudo` privileges. If this is not possible, describe the expected results based on your understanding of the workload and Amdahl's Law.

Enjoy this experiment! We are pleased to answer any questions you may have.

Note: All code snippets are available as text files on Moodle.