**Challenge 5: Workload Profiling, Parallelism, and Architecture Analysis**

**Learning Goals**

* Analyze and profile AI/ML and algorithmic workloads in Python.
* Identify bottlenecks and opportunities for parallelism.
* Think critically about suitable execution architecture.
* Experience "vibe coding" and the problems it exposes.

**Selected Workloads**

1. **Differential Equation Solver** – Euler’s method for solving ODEs.
2. **Quicksort** – Classic recursive sorting algorithm.
3. **Matrix Multiplication** – Manual implementation of matrix product.

**Step 1 & 2: Code Generation**

Generated both serial and parallel versions of all three workloads using Python.

* **diff\_eq\_solver.py** – Euler's method
* **quicksort.py** – Recursive sort with pivot
* **matrix\_mult.py** – Nested loop matrix multiplication
* Parallelized versions using multiprocessing for:
  + parallel\_diff\_eq\_solver.py
  + parallel\_quicksort.py
  + parallel\_matrix\_mult.py

**Step 3: Compiling to Bytecode**

Compiled all six .py files into .pyc bytecode using:

bash

CopyEdit

python -m py\_compile filename.py

Example:

bash

CopyEdit

python -m py\_compile diff\_eq\_solver.py

**Step 4: Disassembling Bytecode**

Disassembled .pyc files using the dis module.

**Virtual Machine:** Python uses **CPython**, a stack-based virtual machine.

**Sample instructions observed:**

* LOAD\_FAST, STORE\_FAST, BINARY\_MULTIPLY, CALL, BUILD\_LIST, etc.

**Instruction frequency compared across workloads using a script:**

| **Instruction** | **diff\_eq\_solver** | **quicksort** | **matrix\_mult** |
| --- | --- | --- | --- |
| CALL | 2 | 4 | 2 |
| BUILD\_LIST | 0 | 0 | 7 |
| FORMAT\_VAL | 2 | 0 | 0 |
| COMPARE\_OP | 1 | 1 | 1 |

**Step 5: Profiling Workloads**

Used cProfile for text-based profiling. Example:

python

CopyEdit

import cProfile

cProfile.run('run\_matrix\_mult()')

**Differential Equation Solver**

* 54 total function calls
* 51 from list .append()
* Lightweight and fast

**Quicksort**

* 1303 function calls (mostly recursion)
* Major calls from list comprehensions and pivot slicing

**Matrix Multiplication**

* 15 function calls
* Dominated by 3 nested loops and list creation

**Step 6: Bottleneck Identification**

|  |  |  |
| --- | --- | --- |
| **Workload** | **Bottleneck** | **Suggestion** |
| Differential Equation | None (linear) | Only optimize for large systems |
| Quicksort | Recursive calls + list slicing | In-place sort or tail-recursion |
| Matrix Multiplication | Triple loops (O(n³)) | Use NumPy or SIMD/GPU |

**Used:**

* multiprocessing.Pool().map() for matrix rows.
* Parallel Quicksort splits at pivot and dispatches in parallel.
* Euler’s method was not parallelized internally but shown as a **batchable workload**.

**Step 7: Parallelism and Architecture Recommendations**

|  |  |  |
| --- | --- | --- |
| **Workload** | **Structure** | **Parallelism Potential** |
| Differential Equation | Sequential ODE solver (Euler’s method) | Within-solver parallelism is limited.  Parallelize **across multiple equations**. |
| Quicksort | Divide-and-conquer recursion | Parallelize quicksort(left) and quicksort(right) using multiprocessing. |
| Matrix Multiplication | Triple-nested loop | High potential. Rows can be computed in parallel. Suitable for SIMD. |

**Differential Equation Solver**

* No branching, predictable flow.
* Suitable for scalar CPU or pipelined RISC core.

**Quicksort**

* Needs recursion stack and branching logic.
* Best fit: **General-purpose CPU** with RISC pipeline.

**Matrix Multiplication**

* Heavy data parallelism, repeatable computation.
* Ideal for: **SIMD**, **GPU cores**, or **TPU-like MAC arrays**.

**Final Takeaways**

* **Python’s stack-based virtual machine** is easy to analyze with tools like dis.
* **Instruction profiling** reveals how different workloads stress the CPU differently.
* **Quicksort** highlights memory and call-stack pressure.
* **Matrix multiplication** benefits greatly from vectorized execution and hardware acceleration.

**Reflection on Vibe Coding**

Vibe coding — quickly prototyping with LLMs — was useful for fast development, but deeper analysis revealed inefficiencies. For example, the matrix multiplication was initially written using raw Python loops, which are not scalable. Profiling and bytecode inspection highlighted performance gaps. This exercise shows that fast, intuitive coding must be supplemented with architectural awareness and analysis for building efficient, scalable systems.

**Conclusion**

This challenge provided practical experience in:

* Writing Python workloads.
* Bytecode inspection and disassembly.
* Instruction analysis.
* Runtime profiling.
* Identifying parallelism.
* Suggesting hardware-friendly architecture.