Assignment 6: Parallel Python for HPC

PHYS 5v48.001 – HPC S25, Aaron Smith

ABSTRACT

This assignment explores multiple parallel Python approaches for an intensely CPU-bound problem: generating random variates from a Lorentzian distribution through *inverse transform sampling*. You will implement the sampling routine in pure Python, then adapt or integrate it into each concurrency strategy: (i) `threading`, (ii) `multiprocessing`, (iii) `concurrent.futures.ProcessPoolExecutor`, (iv) `asyncio`, (v) `dask`, (vi) `numba`, (vii) `joblib`, and (viii) `mpire`. Your goal is to measure runtime and scaling for 1, 2, 4, 8, 16, ... cores, producing a plot summarizing the performance. You will also verify correctness of at least one of the parallel approaches by comparing the resulting histogram to the Lorentzian PDF. This assignment consolidates your knowledge of Python concurrency, GIL constraints, HPC scaling, and the nuance in selecting an appropriate parallel tool.

1. INVERSE TRANSFORM SAMPLING (LORENTZIAN)

1. **Methodology:** You will generate random numbers from a non-uniform distribution using inverse transform sampling, a fundamental technique in Monte Carlo simulations. The idea is to generate uniform random numbers and transform them through the inverse cumulative distribution function (CDF) of a target distribution. In this case, the Lorentzian distribution with half-width at half-maximum (HWHM) $\Gamma = 1$ has the probability density function (PDF) with $x \in \mathbb{R}$:

$$p(x) = \frac{\Gamma/\pi}{\Gamma^2 + x^2}\,. \tag{1}$$

   What is the normalized CDF $F(x)$ for this distribution? What is the inverse CDF $F^{-1}(u)$ for $u \in (0, 1)$? Derive the result that inverse transform sampling uses: $x = \Gamma/\tan(\pi u)$, where $u \sim \text{Uniform}(0, 1)$.

   Note: This is purely CPU-bound. We want to test how effectively different concurrency strategies can scale up to multiple cores. Pure Python is hindered by the GIL so we will use NumPy as a base implementation.

2. **Base NumPy Implementation:**

   - Generate n uniform random numbers $u_i \in (0, 1)$.
   - Transform each to $x_i = \Gamma/\tan(\pi u_i)$, setting $\Gamma = 1$.
   - Accumulate a histogram, e.g., 100 bins spanning $[-10, 10]$).
   - Compare the normalized histogram to the PDF $p(x)$ for visual verification.

```python
def lorentzian_histogram(n, bins=100, xmin=-10, xmax=10):
    """
    Sample n random points from the Lorentzian distribution
    using inverse transform sampling. Make a histogram with
    the specified bin count and range. Returns counts.
    """
    u = np.random.random(n)  # Uniform(0,1)
    x = 1. / np.tan(np.pi * u)  # x = 1/tan(pi*u)
    counts, _ = np.histogram(x, bins=bins, range=(xmin, xmax))
    return counts  # No need to return bin edges for uniform bins
```

   This provides a baseline that accepts n samples and returns a histogram of counts. You should set up your full pipeline to generate the histogram and plot the results to verify correctness before parallelizing.

2. APPROACHES TO PARALLELIZATION

You will reimplement or wrap `lorentzian_histogram` in each of the following concurrency libraries. Some require rewriting the function for concurrency. Others can run the existing function in parallel, with overhead.

2.1 Threading: `threading` Module

Note: Threading is subject to the GIL for CPU-bound tasks, so you might see minimal speedup. The overhead of locks and global data might also hamper performance.

```
# thread_lorentz.py
import threading

def add_chunk(n, counts, lock, bins=100, xmin=-10, xmax=10):
    """
    Generate n samples and add to global counts.
    """
    local_counts = lorentzian_histogram(n, bins, xmin, xmax)
    # Acquire lock to merge partial counts into global
    with lock:
        counts += local_counts

def run_threaded(n, n_threads=4, bins=100, xmin=-10, xmax=10):
    """
    Run the Lorentzian sampling in parallel using threads.
    """
    # Split n samples among processes
    chunks = (n // n_cores) * np.ones(n_cores, dtype=int)
    chunks[:n % n_cores] += 1  # Distribute remainder
    threads = [None] * n_threads  # Thread list
    counts = np.zeros(bins)  # Global counts
    lock = threading.Lock()  # Lock for global data
    for i in range(n_threads):
        t = threading.Thread(target=add_chunk, args=(chunks[i], counts, lock, bins, xmin, xmax))
        t.start()  # Start thread
        threads[i] = t
    for t in threads:
        t.join()  # Wait for all threads to finish
    return counts
```

2.2 Multiprocessing: `multiprocessing` Module

Note: Each child process is GIL-free. We minimize the overhead of locks and global data by only merging the counts at the end.

```
# mp_lorentz.py
import multiprocessing

def run_multiproc(n, n_cores=4, bins=100, xmin=-10, xmax=10):
    """
    Run the Lorentzian sampling in parallel using processes.
    """
    # Split n samples among processes
    chunks = (n // n_cores) * np.ones(n_cores, dtype=int)
```

```
    chunks[:n % n_cores] += 1  # Distribute remainder
    # Use partial function to reset default arguments (bins, xmin, xmax)
    from functools import partial
    lorentzian_hist_func = partial(lorentzian_histogram, bins=bins, xmin=xmin, xmax=xmax)
    # Use Pool to distribute chunks to processes
    with multiprocessing.Pool(n_cores) as pool:
        results = pool.map(lorentzian_hist_func, chunks)
    return np.sum(results, axis=0)  # Aggregate results
```

2.3 ProcessPoolExecutor: concurrent.futures

```
# ppe_lorentz.py
from concurrent.futures import ProcessPoolExecutor

def run_ppe(n, max_workers=4, bins=100, xmin=-10, xmax=10):
    """
    Run the Lorentzian sampling in parallel using ProcessPoolExecutor.
    """
    chunks = (n // max_workers) * np.ones(max_workers, dtype=int)  # Split n samples among workers
    chunks[:n % max_workers] += 1  # Distribute remainder
    with ProcessPoolExecutor(max_workers=max_workers) as executor:
        futures = [executor.submit(lorentzian_histogram, chunk, bins, xmin, xmax) for chunk in chunks]
        results = [f.result() for f in futures]  # Collect results
    return np.sum(results, axis=0)  # Aggregate results
```

2.4 AsyncIO: asyncio

Note: This is typically not beneficial for CPU-bound tasks under the GIL, but we demonstrate for completeness.

```
# async_lorentz.py
import asyncio

async def async_lorentzian_histogram(n, bins=100, xmin=-10, xmax=10):
    """
    Async wrapper for lorentzian_histogram. Since lorentzian_histogram
    is CPU-bound and synchronous, we call it directly.
    """
    return lorentzian_histogram(n, bins, xmin, xmax)

async def add_chunk(n, counts, bins=100, xmin=-10, xmax=10, n_subchunks=10):
    """
    Generate n samples in subchunks and add to global counts.
    """
    # Split n samples among sub-chunks
    sub_chunks = (n // n_subchunks) * np.ones(n_subchunks, dtype=int)
    sub_chunks[:n % n_subchunks] += 1  # Distribute remainder
    # Gather results from subchunks
    local_counts = await asyncio.gather(*[
        async_lorentzian_histogram(chunk, bins, xmin, xmax)
        for chunk in sub_chunks
    ])
    counts += np.sum(local_counts, axis=0)  # Merge partial counts

async def get_counts(n, n_tasks=4, bins=100, xmin=-10, xmax=10, n_subchunks=10):
```

```
    """
    Async function to run the Lorentzian sampling in parallel using asyncio.
    """
    # Split n samples among tasks
    chunks = (n // n_tasks) * np.ones(n_tasks, dtype=int)
    chunks[:n % n_tasks] += 1  # Distribute remainder
    counts = np.zeros(bins)  # Global counts
    tasks = [
        asyncio.create_task(add_chunk(chunk, counts, bins, xmin, xmax, n_subchunks))
        for chunk in chunks
    ]
    await asyncio.gather(*tasks)  # Wait for all tasks to finish
    return counts

def run_async(n, n_tasks=4, bins=100, xmin=-10, xmax=10, n_subchunks=10):
    """
    Run the Lorentzian sampling in parallel using asyncio.
    """
    return asyncio.run(get_counts(n, n_tasks, bins, xmin, xmax, n_subchunks))
```

2.5 Dask Example: `dask.delayed`

Note: By default, Dask uses threads, so it might be GIL-limited for CPU tasks. You can configure Dask to use processes with `dask.config.set(scheduler='processes')` or similar.

```
# dask_lorentz.py
import dask
from dask import delayed

@delayed
def delayed_lorentzian_histogram(n, bins=100, xmin=-10, xmax=10):
    """
    Delayed function for lorentzian_histogram.
    """
    return lorentzian_histogram(n, bins, xmin, xmax)

def run_dask(n, n_tasks=4):
    """
    Run the Lorentzian sampling in parallel using Dask.
    """
    # Split n samples among tasks
    chunks = (n // n_tasks) * np.ones(n_tasks, dtype=int)
    chunks[:n % n_tasks] += 1  # Distribute remainder
    tasks = [delayed_lorentzian_histogram(chunk) for chunk in chunks]
    results = dask.compute(*tasks)  # Compute all tasks
    return np.sum(results, axis=0)  # Aggregate results
```

2.6 Numba Approach for Parallelism

Warning: `np.random.random()` inside a parallel loop might be suboptimal. Possibly each iteration might conflict. A better approach might be seeding or chunk-summing. But let's keep it simple for demonstration.

```
# numba_lorentz.py
import numpy as np
```

```python
from numba import njit, prange, atomic

@njit(parallel=True, nogil=True)
def lorentzian_histogram_numba(n, bins=100, xmin=-10, xmax=10):
    """
    Sample n random points from the Lorentzian distribution
    using inverse transform sampling. Make a histogram with
    the specified bin count and range. Returns counts.
    """
    xfac = bins / (xmax - xmin)  # Factor to map x to bin index
    counts = np.zeros(bins)  # Initialize counts
    for i in prange(n):
        u = np.random.random()  # Uniform(0,1)
        x = 1. / np.tan(np.pi * u)  # x = 1/tan(pi*u)
        ix = int((x - xmin) * xfac)  # Map x to bin index
        if 0 <= ix < bins:
            atomic.add(counts, ix, 1)  # Atomic increment
    return counts
```

2.7 Joblib

Tip: `batch_size` can finetune the performance. Also, for large data, consider further partial histograms to reduce memory usage.

```python
# joblib_lorentz.py
from joblib import Parallel, delayed

def run_joblib(n, n_jobs=4, bins=100, xmin=-10, xmax=10):
    """
    Run the Lorentzian sampling in parallel using joblib.
    """
    # Split n samples among jobs
    chunks = (n // n_jobs) * np.ones(n_jobs, dtype=int)
    chunks[:n % n_jobs] += 1  # Distribute remainder
    results = Parallel(n_jobs=n_jobs)(
        delayed(lorentzian_histogram)(chunk, bins, xmin, xmax)
        for chunk in chunks
    )
    return np.sum(results, axis=0)  # Aggregate results
```

2.8 mpire

Note: mpire automates chunking, dynamic scheduling. Good for HPC if tasks differ in complexity.

```python
# mpire_lorentz.py
from mpire import WorkerPool

def run_mpire(n, n_jobs=4, bins=100, xmin=-10, xmax=10):
    """
    Run the Lorentzian sampling in parallel using mpire.
    """
    # Split n samples among jobs
    chunks = (n // n_jobs) * np.ones(n_jobs, dtype=int)
    chunks[:n % n_jobs] += 1  # Distribute remainder
```

```
    with WorkerPool(n_jobs=n_jobs) as pool:
        results = pool.map(lorentzian_histogram, chunks, bins, xmin, xmax)
    return np.sum(results, axis=0)  # Aggregate results
```

3. BENCHMARKING AND HPC SCALING

1. **Test Cases:** For the CPU-bound Lorentz sampling, choose a large n (like 10^7 or 10^8) that runs within your patience threshold on HPC. To improve the statistics, you can adjust the number samples along with the number of cores if you think weak scaling is appropriate. Also, adjust the number of bins in the histogram to see how it affects performance (e.g. 10, 100, 1000). Run each approach with concurrency levels in {1,2,4,8,16,...} cores (or threads, processes, workers, tasks, jobs).

2. **Plot Results:** For each approach and core count, record the total runtime (and possibly memory usage if you are curious). Plot the runtime per core as a function of the number of cores, including each approach in the same figure. Use a log-log plot to capture the scaling behavior. Also plot the speedup relative to the serial runtime to more intuitively understand the performance gains. Summarize the results in a table or list giving the method and main takeaways.

3. **Final HPC Run:** Attempt a larger run, e.g. $n = 10^9$ or even 10^{10} for the approach you found to be the fastest. Keep the runtime under 10 minutes. Were your scaling predictions accurate? Did you encounter any anomalies or unexpected behavior?

4. REPORTING GUIDELINES

1. **Submission Format**:

   - Provide a concise PDF that includes:
     (a) Code snippets of each concurrency approach.
     (b) HPC logs or short job scripts.
     (c) Table or chart of concurrency vs. runtime or speedup for each approach.
     (d) Observations on overhead, GIL, parallel speedups, memory usage, etc.

2. **Evaluation Criteria**:

   - *Correctness & Completeness*: Each approach tested on HPC at multiple concurrency levels, data collected, final HPC run.
   - *Clarity*: Code well-organized, commentary or docstrings describing concurrency usage.
   - *Analysis & Discussion*: Reflection on GIL-limited vs. process-based approaches, overhead or synergy with HPC, any run-time anomalies, final hist vs. PDF check.

This assignment explored multiple Python concurrency tools for a CPU-bound sampling problem. You should now understand the different approaches, including their pros and cons. Good luck and happy coding!