Project Report: ETL Optimization Using Parallelism Techniques in C#

1. Project Overview

This project demonstrates and evaluates the application of various parallelism techniques to optimize a sample ETL (Extract, Transform, Load) pipeline implemented in C#. The primary goal is to compare the runtime performance improvements achieved by leveraging modern C# parallel programming features and CPU-level parallelism against a baseline sequential implementation.

The ETL process involves: 1. **Extract:** Reading a large dataset of random integers (simulated as an in-memory array `rawData`). 2. **Transform:** Applying a simple transformation (multiplying each integer by 2). 3. **Load:** Filtering the transformed data (keeping only numbers divisible by 3) and storing the result.

2. Implemented Parallelism Techniques

The following techniques were implemented and benchmarked within the `Program.cs` code:

- **Sequential:** A baseline implementation where Extract, Transform, and Load stages are executed one after another in a single thread.
- **Task Parallel (Refactored):** Uses `Parallel.For` to parallelize the work within each ETL stage (Extract, Transform, Load). The stages themselves still run sequentially, but the processing of data within each stage is parallelized across available CPU cores. `ConcurrentBag<int>` is used for thread-safe collection during the parallel Load stage.
- **Data Parallel (Refactored):** Functionally similar to the Task Parallel approach in this implementation, also using `Parallel.For` within each stage and `ConcurrentBag<int>` for the Load stage. It emphasizes the concept of applying the same operation to multiple data elements concurrently.
- **Pipeline Parallel:** Implements a pipeline using `BlockingCollection<int>` where the Extract, Transform, and Load stages run as concurrent tasks. Data flows from one stage to the next through bounded buffers, allowing stages to operate simultaneously on different chunks of data.
- **SIMD + ILP (Refactored):** Leverages Single Instruction, Multiple Data (SIMD) using `System.Numerics.Vector<int>` to perform the Transform operation (multiplication) on multiple data elements simultaneously within a single CPU instruction. Instruction-Level Parallelism (ILP) is implicitly utilized by the CPU during these operations. The Load stage is parallelized using PLINQ (`AsParallel().Where()`).

- **Combined (SIMD + Data Parallel):** This hybrid approach aims for maximum performance by combining Data Parallelism with SIMD. It uses `Parallel.For` to distribute chunks of the data across cores, and within each parallel task, it uses SIMD instructions for the Transform operation. The Load stage is parallelized using PLINQ.

3. Performance Measurement Methodology

Each technique was executed on the same dataset (`DataSize = 10,000,000` integers). The execution time was measured using `System.Diagnostics.Stopwatch`. Garbage collection (`GC.Collect()`, `GC.WaitForPendingFinalizers()`) was explicitly invoked before each measurement to minimize its impact on the results.

4. Performance Results

The measured execution times (in milliseconds) for each technique were as follows (sorted fastest to slowest based on the last test run):

- Combined (SIMD + Data Parallel): 97.68 ms
- SIMD + ILP (Refactored): 120.92 ms
- Data Parallel (Refactored): 285.00 ms
- Sequential: 324.06 ms
- Task Parallel (Refactored): 372.76 ms
- Pipeline Parallel: 14088.64 ms

(Note: Performance can vary based on the specific hardware and runtime environment.)

5. Performance Comparison Chart

The following chart visually compares the execution times:

`/home/ubuntu/chart.png`

6. Analysis and Discussion

- **SIMD Effectiveness:** The results clearly demonstrate the significant performance advantage provided by SIMD instructions for the numeric transformation task. Both the `SIMD + ILP` and `Combined` approaches, which utilize SIMD, were considerably faster (roughly 2.7x to 3.3x) than the sequential baseline.
- **Combined Approach:** The `Combined (SIMD + Data Parallel)` method achieved the best performance, indicating that leveraging both task-level parallelism (`Parallel.For`) and instruction-level parallelism (SIMD) is highly effective for this workload.
- **Basic Parallelism Overhead:** The `Task Parallel` and `Data Parallel` implementations, while conceptually parallel, showed performance slightly worse than or similar to the sequential version. This suggests

that for the relatively simple operations in this ETL pipeline and the given data size, the overhead associated with managing parallel tasks (`Parallel.For` and `ConcurrentBag`) might negate or outweigh the benefits of parallel execution within stages.
- **Pipeline Performance:** The `Pipeline Parallel` implementation using `BlockingCollection` performed poorly in this test. The overhead associated with managing the buffers, context switching between tasks, and potential locking within `BlockingCollection` likely dominated the execution time. While pipelining can be beneficial for tasks with significant I/O or varying stage complexities, it was not well-suited for this CPU-bound, relatively balanced workload without further optimization (e.g., batching).

7. Conclusion

This project successfully implemented and benchmarked various parallelism techniques for optimizing a sample C# ETL pipeline. The findings highlight the substantial performance gains achievable through SIMD for suitable numeric operations. Combining SIMD with higher-level data parallelism (`Parallel.For` and PLINQ) yielded the most efficient results for this specific problem. The experiment also illustrated that simpler parallelization strategies might introduce overhead that negates benefits for certain workloads, and pipeline parallelism requires careful implementation and tuning to be effective.