Sistemas Multinúcleo e Distribuídos
Mestrado em Engenharia Informática
Instituto Superior de Engenharia do Porto

Optimizing Large-Scale Data Processing on Multicore Systems

Introduction

The goal of this project is to implement and analyze different approaches to process large volumes of data efficiently on multicore systems. The project focuses on finding the frequency of words in English texts by processing a large Wikipedia data dump. The performance of each implementation will be measured and analyzed to understand how different concurrency models affect execution time and resource utilization. We accept groups up to 3 students each.

Objectives

The project requires the development and analysis of the following implementations:

1. **Sequential Solution**: A baseline implementation that processes the data sequentially without any concurrency.
2. **Multithreaded Solution (Without Thread Pools)**: An explicit multithreaded implementation where threads are manually managed, focusing on work distribution and synchronization.
3. **Multithreaded Solution (With Thread Pools)**: An implementation utilizing thread pools to manage threads efficiently, reducing the overhead of thread creation and destruction.
4. **Fork/Join Framework Solution**: A solution leveraging the Fork/Join framework to recursively split tasks and join results, suitable for divide-and-conquer algorithms.
5. **CompletableFutures-Based Solution**: An asynchronous implementation using `CompletableFuture` to manage tasks and their dependencies without explicit thread management.
6. **Garbage Collector Tuning**: Configure and tune Java garbage collectors to optimize memory management and improve application performance.

Problem Description

The task involves processing a large XML file containing English Wikipedia pages (`https://dumps.wikimedia.org/enwiki/`) to find and count the frequency of each word. The input file is several megabytes in size, and efficient processing is crucial to handle such large volumes of data.

Parsing

Parsing is provided in the starter code using standard Java XML processing libraries. Although parsing is sequential and considered a fixed overhead, students interested in optimizing this phase can propose improvements for additional credit.

Implementation Approaches

1. Sequential Solution

The sequential solution serves as a baseline for performance comparison. It processes the input file sequentially, counts the occurrences of each word, and identifies the most common words. This implementation helps in understanding the limitations of sequential processing in handling large data volumes on multicore systems.

Sequential processing is straightforward but doesn't utilize the available cores in a multicore system, leading to suboptimal performance. Establishing a baseline allows us to quantify the benefits of concurrent approaches.

2. Multithreaded Solution (Without Thread Pools)

In this approach, multiple threads are explicitly created and managed to process different parts of the data concurrently. A common strategy like the producer-consumer pattern can be employed for work distribution.

Manually managing threads provides fine-grained control over concurrency but introduces complexity in thread coordination and resource management. This approach helps in understanding the challenges of thread synchronization and data sharing.

3. Multithreaded Solution (With Thread Pools)

This implementation utilizes thread pools to manage threads more efficiently. The thread pool handles thread creation and reuse, reducing the overhead associated with thread lifecycle management.

Using thread pools simplifies thread management and can lead to better resource utilization. It allows for scalable solutions that can adjust the number of active threads based on the workload and system capabilities.

4. Fork/Join Framework Solution

The Fork/Join framework is used to implement a solution that recursively splits the task into subtasks, processes them in parallel, and combines the results.

The Fork/Join framework is designed for tasks that can be broken down into smaller, independent subtasks. It efficiently utilizes all available processors and is suitable for divide-and-conquer algorithms, potentially leading to significant performance improvements.

5. CompletableFutures-Based Solution

This approach uses `CompletableFuture` for asynchronous programming. It allows tasks to be executed in non-blocking ways and handles the composition of asynchronous operations.

`CompletableFuture` provides a high-level API for writing asynchronous code without manually managing threads. It simplifies handling of dependent tasks and can lead to more readable and maintainable code.

6. Garbage Collector Tuning

Java provides several garbage collectors that operate concurrently with applications to manage memory by reclaiming unused objects, employing different strategies to optimize performance. You should provide evidence of such configuration work, including documentation or logs demonstrating the configuration and tuning of the garbage collector used in your application, and clearly explain why you selected a particular garbage collector, referencing how it aligns with your application's performance characteristics and resource usage patterns. Additionally, analyze how the chosen garbage collector affects your application's performance and resource utilization, including any improvements observed.

Concurrency and Synchronization

Appropriate synchronization mechanisms must be used to ensure thread safety and data consistency across all concurrent implementations. Choices of data structures (e.g., thread-safe collections) and synchronization techniques (e.g., locks, atomic variables) should be justified and their impact on performance analyzed.

Generation of Results

Each implementation should measure and record:

- **Execution Time:** Total time taken to process the dataset.
- **Resource Utilization:** CPU and memory usage during execution.
- **Scalability Analysis:** Performance metrics as the size of the dataset or the number of threads changes.

Results should be presented using automatically generated tables and charts. The analysis should compare the performance of each implementation, discussing:

- **Efficiency Gains:** Improvements over the sequential baseline.
- **Scalability:** How performance scales with increased data size or additional cores.
- **Overhead Analysis:** Impact of thread management and synchronization on performance.
- **Bottlenecks:** Identification of any limitations or areas where performance does not improve as expected.

Starter Code

The starter code includes a parser and a sequential implementation. Configuration parameters such as the number of pages to process and the input file name can be adjusted in the `WordCount.java` file:

```
static final int maxPages = 100000;
static final String fileName = "enwiki.xml";
```

Example output for a partial file of Wikipedia [1]:

```
Max pages: 100000
No more pages!
Processed pages: 27371
Elapsed time: 55554ms
Word: 'the' with total 5078052 occurrences!
Word: 'of' with total 3806097 occurrences!
Word: 'and' with total 2644983 occurrences!
```

Report

The report should include the following content:

- **Cover**: Title, class identification, and authors (student numbers and full names)
- **Introduction**
- **Objectives**
- **Implementation Approaches**
  - Sequential Solution
  - Multithreaded Solution (Without Thread Pools)
  - Multithreaded Solution (With Thread Pools)
  - Fork/Join Framework Solution
  - CompletableFuture-Based Solution
  - Garbage Collector Tuning
- **Concurrency and Synchronization**
- **Performance Analysis**
- **Conclusions**

Each section should include a description of the implementation, justification of the chosen approach, and essential code snippets (avoid including all code).

[1] `https://dumps.wikimedia.org/enwiki/20241020/enwiki-20241020-pages-articles-multistream1.xml-p1p41242.bz2`

Grading

Grading is awarded following the criteria described next:

1. **Correctness and Functionality (60 points)**
    - Multithreaded Implementation without Thread Pools (15 points): Correctly implements multithreading. Manages threads and synchronization properly.
    - Thread Pool Implementation (15 points): Effectively uses `ExecutorService` and thread pools. Optimizes thread pool size for performance.
    - Fork/Join Framework Implementation (15 points): Successfully implements the Fork/Join approach with proper task splitting and result aggregation.
    - CompletableFuture Implementation (10 points): Correctly uses `CompletableFuture` for asynchronous tasks. Handles task completion and combines results appropriately.
    - Garbage Collector Tuning (5 points): Demonstrates effective configuration and tuning of garbage collectors. Provides evidence and justification for choices.
2. **Performance Analysis (30 points)**
    - Benchmarking (15 points): Measures execution time, CPU, memory usage, and garbage collection metrics. Tests with various data sizes and thread counts.
    - Analysis Report (15 points): Presents data using graphs or tables. Provides insights into why certain implementations perform better. Discusses challenges, including garbage collection effects, and solutions.
3. **Code Quality and Documentation (10 points)**
    - Code Organization (5 points): Follows best practices for code structure and naming conventions. Uses appropriate data structures and algorithms.
    - Documentation and Comments (5 points): Includes comments explaining complex sections. Provides a README file with setup and execution instructions.
4. **Understanding of Concepts during Presentation (Student Weighting)**
    - Has a very good understanding of the topic and can explain the developed code well: 100%
    - Has some shortcomings in understanding the topic or (only) in the developed code: 80%
    - Has some shortcomings in understanding both the topic and the developed code: 70%
    - Encounters significant difficulties in mastering the topic or (only) in the developed code: 60%
    - Encounters significant difficulties in mastering both the topic and the developed code: 50%
    - Demonstrates major difficulties in mastering the topic or (only) in the developed code: 40%

- Faces major difficulties in fully understanding both the topic and the developed code: 25%
- No-show or demonstrates a lack of knowledge in both the topic and the developed code: 0%

Student's grade is = Work grading * Student Weighting

Scheduling

The final deadline for the submission of this project is 12th of May of 2025 and the presentations will take place on the following days during TP and PL classes.

Code of Honor

As per the "Código de boas Práticas de Conduta" dated October 27, 2020, students must submit a declaration as described in Article 8 for each project submission. This declaration is a signed commitment to the originality of the submitted work. Failure to submit this declaration will result in the work not being evaluated. Submitting work that violates the declaration will have legal consequences.