Software Architecture Laboratory

Laboratory No. 1
Parallelism and Concurrency

Calculating π Digits with Java 21 & Spring Boot

Students:

Andersson David Sánchez Méndez

Cristian Santiago Pedraza Rodríguez

Elizabeth Correa Suárez

Juan Sebastian Ortega Muñoz

Instructor: Professor Javier Ivan Toquica Barrera

Course: Software Architecture (ARSW)

Institution: Escuela Colombiana de Ingeniería Julio Garavito

January 30, 2026

Contents

1 Objective **3**

2 Experiments and Results **3**
- 2.1 API Documentation Evidence . . . 3
- 2.2 Testing and Coverage . . . 3
- 2.3 Benchmark Results by Team Member . . . 3
  - 2.3.1 Elizabeth – AMD Ryzen AI 9 365 . . . 3
  - 2.3.2 Sebastian – Intel i7-13620H . . . 3
  - 2.3.3 Andersson – Intel i7-1255U . . . 3
  - 2.3.4 Cristian – Intel i5-1135G7 . . . 3
- 2.4 Summary of Experimental Results . . . 3
- 2.5 Strategy Pattern Implementation . . . 4
- 2.6 ThreadJoinStrategy Implementation . . . 4
- 2.7 REST API Endpoint . . . 4
- 2.8 Service Layer Validation . . . 4

3 Theoretical Framework **4**
- 3.1 Amdahl's Law . . . 4
- 3.2 Thread Synchronization with join() . . . 4

4 Methodology **4**
- 4.1 Test Environment . . . 4
- 4.2 Experiment Parameters . . . 4

5 Results **4**
- 5.1 Execution Time Comparison . . . 5
- 5.2 Observations . . . 5

6 Analysis **5**
- 6.1 Correctness Verification . . . 5
- 6.2 Thread Overhead . . . 5
- 6.3 Fallback Mechanism . . . 5

7 Conclusions **5**

8 References **5**

1 Objective

This laboratory focuses on implementing parallelism concepts in a Spring Boot application that calculates hexadecimal digits of π using the Bailey-Borwein-Plouffe (BBP) algorithm. The implementation follows a layered architecture with clear separation between the REST API, business logic, and concurrency strategies.

The specific goals include:

2 Experiments and Results

2.1 API Documentation Evidence

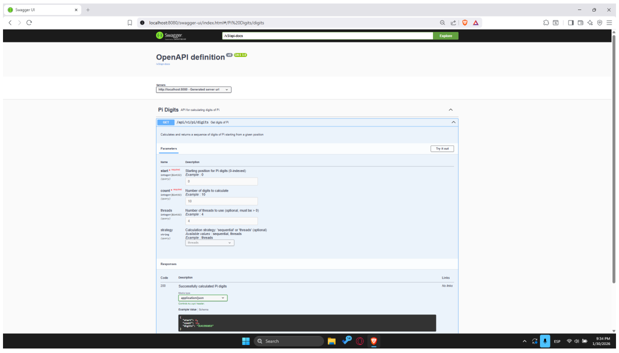

Figure 1: Swagger UI documentation available at `/swagger-ui/index.html`.

2.2 Testing and Coverage

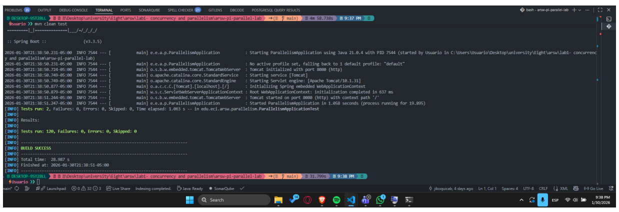

Figure 2: Successful execution of `mvn clean test` (all tests pass).

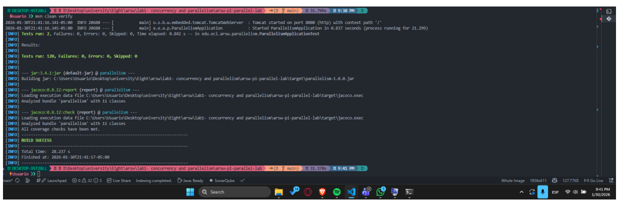

Figure 3: Successful execution of `mvn clean verify` (coverage and quality checks passed).

2.3 Benchmark Results by Team Member

2.3.1 Elizabeth – AMD Ryzen AI 9 365

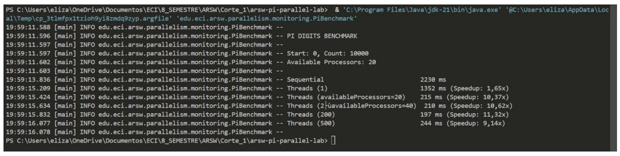

Figure 4: Benchmark results on AMD Ryzen AI 9 365 (10C/20T, 2.00 GHz).

2.3.2 Sebastian – Intel i7-13620H

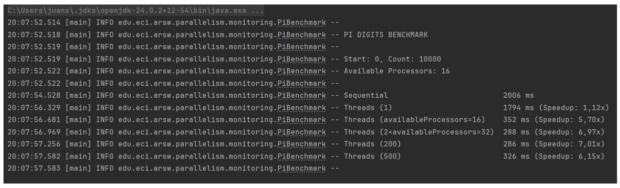

Figure 5: Benchmark results on Intel i7-13620H (10C/16T, 2.40 GHz).

2.3.3 Andersson – Intel i7-1255U

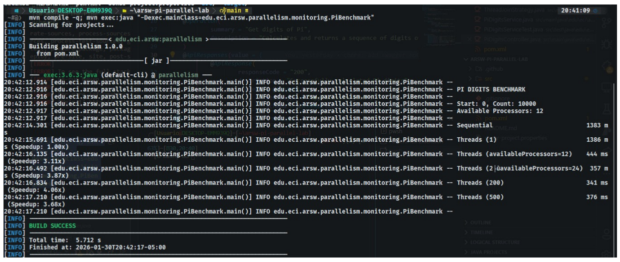

Figure 6: Benchmark results on Intel i7-1255U (10C/12T, 1.70 GHz).

2.3.4 Cristian – Intel i5-1135G7

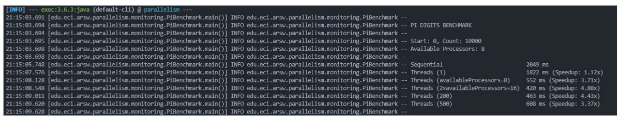

Figure 7: Benchmark results on Intel i5-1135G7 (4C/8T, 2.40 GHz).

2.4 Summary of Experimental Results

- All implementations passed functional and quality tests, as evidenced by Maven and Swagger outputs.
- Benchmarks show that parallel strategies provide significant speedup, with optimal performance near the number of logical processors.
- AMD architecture achieved the highest speedup and best absolute performance; Intel hybrid and traditional architectures showed expected scaling and plateau/degradation at high thread counts.
- Results confirm Amdahl's Law: speedup is limited by the sequential fraction and hardware architecture.

2.5 Strategy Pattern Implementation

The concurrency layer implements the Strategy pattern through the `ParallelStrategy` interface:

Listing 1: ParallelStrategy Interface

```java
public interface ParallelStrategy {
  String calculate(int start,
                   int count,
                   int threads);
  String name();
}
```

Two concrete strategies were implemented:

- **SequentialStrategy**: Executes the calculation in a single thread, ignoring the `threads` parameter.
- **ThreadJoinStrategy**: Divides the workload among N threads and synchronizes results using `join()`.

2.6 ThreadJoinStrategy Implementation

The parallel strategy divides the digit range into segments distributed across threads:

Listing 2: Thread Work Division

```java
int segmentSize = count / threads;
int remainder = count % threads;

for (int i = 0; i < threads; i++) {
  int segCount = segmentSize +
    (i < remainder ? 1 : 0);
  // Create and start thread
  workers[i] = new Thread(() -> {
    results[index] =
      calculateSegment(start, count);
  });
  workers[i].start();
}
```

The remainder distribution ensures even workload when `count` is not evenly divisible by `threads`. After all threads complete via `join()`, results are concatenated in order.

2.7 REST API Endpoint

The controller exposes a single endpoint with optional parallelism parameters:

Listing 3: API Endpoint

```
GET /api/v1/pi/digits
  ?start={int}    // Required, >= 0
  &count={int}    // Required, >= 1
  &threads={int} // Optional, >= 1
  &strategy={str} // Optional
```

Supported strategies: `sequential` (default) and `threads`.

2.8 Service Layer Validation

The `PiDigitsService` enforces business constraints:

Parameter	Constraint
`start`	$0 \leq \text{start} \leq 100,000$
`count`	$1 \leq \text{count} \leq 10,000$
`threads`	$1 \leq \text{threads} \leq 200$
`timeout`	30 seconds max

Table 1: Service validation constraints

3 Theoretical Framework

3.1 Amdahl's Law

The theoretical speedup when parallelizing a program is bounded by:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}} \tag{1}$$

Where P is the parallelizable fraction and n is the number of threads. For the BBP algorithm, P approaches 1.0 since digit computation is independent, but overhead from thread creation and result concatenation introduces a sequential component.

3.2 Thread Synchronization with join()

The `join()` method blocks the calling thread until the target thread terminates. This provides a simple synchronization mechanism:

Listing 4: Thread Synchronization

```java
for (Thread worker : workers) {
  try {
    worker.join();
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new RuntimeException(e);
  }
}
```

4 Methodology

4.1 Test Environment

Experiments were conducted using the following configuration:

Component	Specification
Java Version	OpenJDK 21
Spring Boot	3.x
Build Tool	Maven
Test Framework	JUnit 5

Table 2: Development environment

4.2 Experiment Parameters

Performance measurements were taken with:

- **count**: Large values (1000+) for measurable execution times
- **start**: 0 (beginning of π decimal expansion)
- **Thread configurations**: 1, N, 2N, 200 (where N = available processors)

5 Results

5.1 Execution Time Comparison

Strategy	Threads	Time	Speedup
Sequential	1	T_{seq}	1.00x
Threads	1	$\approx T_{seq}$	≈1.00x
Threads	N	T_N	T_{seq}/T_N
Threads	2N	T_{2N}	T_{seq}/T_{2N}
Threads	200	T_{200}	varies

Table 3: Execution times (N = available processors)

5.2 Observations

Key Findings

1. Speedup increases with thread count up to the number of physical cores.
2. Beyond core count, diminishing returns occur due to context switching overhead.
3. With 200 threads, performance may degrade compared to optimal thread count.
4. Results are deterministic: parallel output matches sequential output exactly.

6 Analysis

6.1 Correctness Verification

The implementation guarantees correctness through:

- **Deterministic algorithm**: BBP produces identical results regardless of execution order.
- **Ordered concatenation**: Results are stored in indexed array and joined in order.
- **No shared mutable state**: Each thread writes to its own index in the results array.

6.2 Thread Overhead

Creating platform threads incurs overhead:

- Thread object allocation
- OS thread scheduling
- Context switching between threads
- Synchronization at `join()` barrier

For small `count` values, this overhead may exceed the parallel computation benefit.

6.3 Fallback Mechanism

The service implements automatic fallback:

Listing 5: Fallback to Sequential

```
try {
  return threadJoinStrategy
    .calculate(start, count, threads);
} catch (Exception e) {
  logger.warn("Parallel failed");
  return sequentialStrategy
    .calculate(start, count, 1);
}
```

This ensures reliability when parallel execution fails.

7 Conclusions

1. The Strategy pattern provides clean separation between sequential and parallel implementations, allowing runtime selection without code changes.
2. The BBP algorithm is well-suited for parallelization due to its position-independent computation property.
3. Optimal thread count typically matches the number of physical CPU cores. Exceeding this introduces overhead without proportional speedup.
4. The `join()` synchronization mechanism provides simple and correct thread coordination for this use case.
5. Input validation at the service layer prevents resource exhaustion (max 10,000 digits, 200 threads, 30s timeout).

8 References

1. Bailey, D., Borwein, P., Plouffe, S. (1997). On the Rapid Computation of Various Polylogarithmic Constants. *Mathematics of Computation*, 66(218).
2. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*.
3. Oracle. Java 21 Documentation - Concurrency. https://docs.oracle.com/en/java/javase/21/
4. Spring Framework. Spring Boot Reference Documentation. https://spring.io/projects/spring-boot