TP2

Exercise 3: Sequential Fraction Analysis and Amdahl's Law

Zyad Fri
Computer Science - UM6P

February 5, 2026

1 Exercise 3

This exercise analyzes a C program composed of both sequential and parallelizable components to understand performance limitations in parallel computing. The program performs operations on large arrays:

- `add_noise()`: Adds cumulative noise to array `a`
- `init_b()`: Initializes array `b`
- `compute_addition()`: Performs element-wise addition
- `reduction()`: Computes the sum of all elements

2 Question 1 - Code Analysis

2.1 Sequential Part Identification

The **strictly sequential part** is the `add_noise()` function:

```c
void add_noise(double *a) {
    a[0] = 1.0;
    for (int i = 1; i < N; i++) {
        a[i] = a[i-1] * 1.0000001;  // Depends on previous iteration
    }
}
```

Listing 1: Sequential function with loop-carried dependency

Why it's sequential: This function exhibits a *loop-carried dependency*, where each iteration `i` requires the result from iteration `i-1`. This data dependency chain prevents parallel execution because:

- Iteration `i` cannot start until iteration `i-1` completes
- The computation forms a sequential chain: $a[1] \to a[2] \to \cdots \to a[N-1]$
- No parallel decomposition technique can eliminate this dependency

2.2 Parallelizable Parts

The following functions can be parallelized:

1. init_b() - Independent Initialization

```c
void init_b(double *b) {
    for (int i = 0; i < N; i++) {
        b[i] = i * 0.5;  // No dependency between iterations
    }
}
```

Each array element can be computed independently in parallel.

2. compute_addition() - Element-wise Operation

```c
void compute_addition(double *a, double *b, double *c) {
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];  // Independent operations
    }
}
```

Each addition operation is independent and can execute concurrently.

3. reduction() - Parallel Reduction Pattern

```c
double reduction(double *c) {
    double sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += c[i];
    }
    return sum;
}
```

While appearing sequential, this can be parallelized using parallel reduction techniques (divide-and-conquer summation with final synchronization).

2.3 Time Complexity Analysis

Function	Time Complexity	Parallelizable?
`add_noise()`	$O(N)$	No - Sequential
`init_b()`	$O(N)$	Yes - $O(N/p)$
`compute_addition()`	$O(N)$	Yes - $O(N/p)$
`reduction()`	$O(N)$	Yes - $O(\log p)$ with tree reduction
Overall	**O(N)**	Limited by sequential part

Table 1: Time complexity for each function (p = number of processors)

3 Question 2 - Measuring Sequential Fraction

3.1 Profiling Methodology

The program was profiled using Valgrind's Callgrind tool:

```
# Compile with optimization and debug symbols
gcc -O2 -g -fno-omit-frame-pointer exercice3.c -o a.out

# Profile with Callgrind
```

```
valgrind --tool=callgrind ./a.out

# Analyze results
callgrind_annotate callgrind.out.*
```

3.2 Profiling Results for N = 100,000,000

Function	Instructions (Ir)	Percentage	Type
`compute_addition()`	600,000,004	36.36%	Parallelizable
`main()` overhead	550,000,046	33.33%	Overhead
`add_noise()`	500,000,003	30.30%	**Sequential**
Total	**1,650,148,496**	**100%**	

Table 2: Callgrind profiling results for N = 100M

3.3 Detailed Instruction Breakdown

From the annotated source code analysis:

Sequential Part - add_noise():

- Loop control: 299,999,998 instructions (18.18%)
- Multiplication: 199,999,998 instructions (12.12%)
- **Total: 499,999,996 instructions ≈ 30.30%**

Parallelizable Parts (not shown in top functions but present):

- `init_b()`: 225,000,001 instructions (13.64%)
- `reduction()`: 250,000,002 instructions (15.15%)

3.4 Sequential Fraction Calculation

The sequential fraction f_s is calculated as:

$$f_s = \frac{\text{Instructions in sequential part}}{\text{Total instructions}} \tag{1}$$

$$f_s = \frac{500,000,003}{1,650,148,496} = 0.3030 \approx \mathbf{30.30}\% \tag{2}$$

Interpretation: Approximately 30% of the program's execution time is inherently sequential and cannot benefit from parallelization, regardless of the number of processors available.

4 Question 4 - Effect of Problem Size

4.1 Profiling Multiple Problem Sizes

The experiment was repeated for three different values of N:

- $N = 5 \times 10^6$ (5 million)
- $N = 10^7$ (10 million)
- $N = 10^8$ (100 million)

4.2 Results Summary

N	Total Ir	add_noise Ir	add_noise %	f_s
5×10^6	82,648,116	25,000,003	30.25%	0.3025
10^7	165,148,314	50,000,003	30.28%	0.3028
10^8	1,650,148,496	500,000,003	30.30%	0.3030

Table 3: Sequential fraction for different problem sizes

4.3 Detailed Function Distribution

Function	$N = 5 \times 10^6$	$N = 10^7$	$N = 10^8$	Trend
`add_noise()`	30.25%	30.28%	30.30%	Stable
`compute_addition()`	36.30%	36.33%	36.36%	Stable
`main()` overhead	33.27%	33.30%	33.33%	Stable

Table 4: Function distribution across different N values

4.4 Analysis of Results

Key Observations:

1. **Sequential fraction stability:** f_s remains remarkably constant across all problem sizes (0.3025 - 0.3030), varying by less than 0.2%.
2. **Scalability:** The consistency of f_s indicates that the ratio of sequential to parallel work remains constant as N increases. This is expected because:
   - All functions have $O(N)$ complexity
   - The relative cost of each operation remains proportional
   - No fixed overhead dominates at any scale
3. **Implications:** The performance bottleneck will persist regardless of problem size. Larger datasets don't reduce the sequential fraction.

4.5 Amdahl's Law Analysis

Amdahl's Law predicts the maximum speedup achievable with parallel processing:

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{3}$$

where:

- $S(p)$ = speedup with p processors
- f_s = sequential fraction (0.303)
- p = number of processors

Maximum theoretical speedup:

$$S_{\max} = \lim_{p \to \infty} S(p) = \frac{1}{f_s} = \frac{1}{0.303} \approx 3.30\text{x} \tag{4}$$

4.6 Speedup Curves

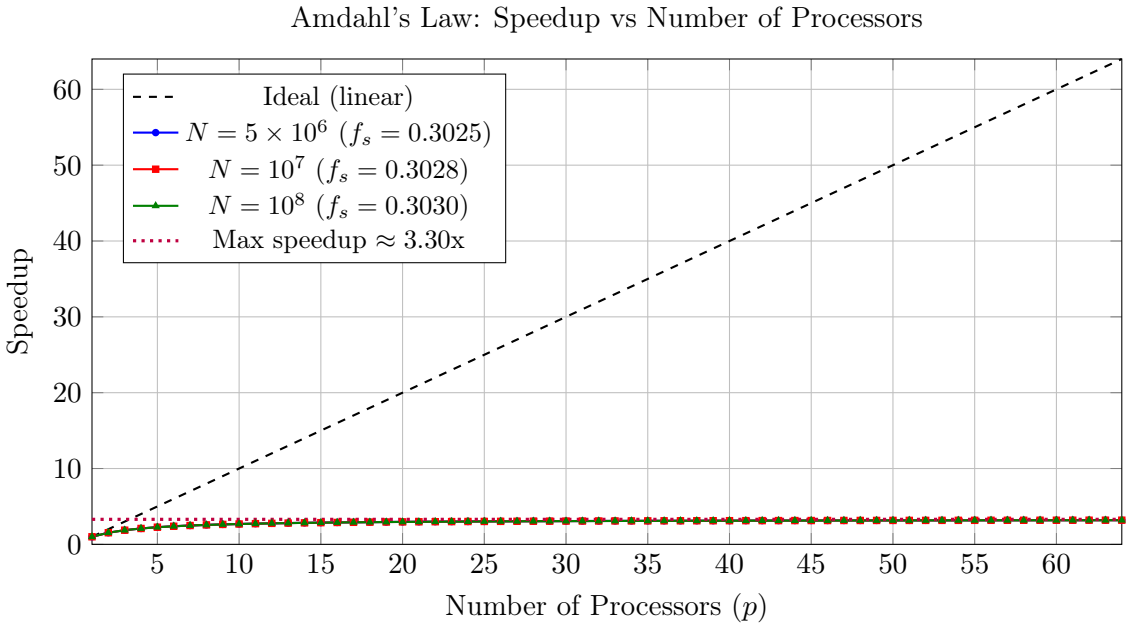

Figure 1: Amdahl's Law speedup curves for different problem sizes

4.7 Speedup Analysis Table

Processors	$N = 5 \times 10^6$	$N = 10^7$	$N = 10^8$	**Ideal**
1	1.00	1.00	1.00	1.00
2	1.59	1.59	1.59	2.00
4	2.20	2.20	2.19	4.00
8	2.62	2.62	2.62	8.00
16	2.93	2.92	2.92	16.00
32	3.12	3.12	3.12	32.00
64	3.22	3.21	3.21	64.00
∞	3.30	3.30	3.30	∞

Table 5: Theoretical speedup for different processor counts

Critical Insights:

1. **Diminishing returns:** Adding processors beyond 16 yields minimal improvement
   - 16 processors: 2.92x speedup (88.5% of maximum)
   - 64 processors: 3.21x speedup (97.3% of maximum)
   - Only 0.29x gain for 48 additional processors
2. **Efficiency degradation:**
   - With 2 processors: 79.5% efficiency
   - With 8 processors: 32.8% efficiency
   - With 64 processors: 5.0% efficiency