TP2: Foundations of Parallel Computing

Loop Optimizations, Instruction Scheduling, and Scaling Laws

Khaoula JELLAL

January 2026

1 Introduction

The objective of this practical work is to explore fundamental techniques for improving program performance through compiler-level and architectural optimizations. We analyze loop unrolling and instruction scheduling to understand how Instruction-Level Parallelism (ILP) and memory bandwidth affect execution time. Furthermore, we apply Amdahl's and Gustafson's laws to empirical data to evaluate the theoretical speedup of parallelizable code segments.

2 Exercise 1: Loop Optimizations

We manually unrolled a summation loop for an array of $N = 100,000,000$ elements to evaluate trade-offs between instruction overhead, ILP, and hardware resource limitations.

2.1 Performance Analysis: Unrolling Factors and Compiler Optimization

We tested unrolling factors $U \in \{1, 2, 3, 4, 6, 8, 10, 12, 16, 32\}$ under `-O0` and `-O2` compilation modes.

Unrolling Factor (U)	**Time -O0 (ms)**	**Time -O2 (ms)**
1 (Baseline)	515.19	149.00
3	183.96	107.29
12	141.52	**98.48**
32	**114.34**	156.09

Table 1: Impact of unrolling factors on execution time for double-precision summation.

Key Observations:

- At `-O0`, increasing U from 1 to 4 yields a 67% speedup due to reduced loop overhead. Performance saturates beyond $U = 12$.
- At `-O2`, optimal performance occurs at $U = 12$ (98.48 ms). Beyond this, **register pressure** and **I-cache pollution** cause degradation.
- Even the best manual unrolling at `-O0` (114 ms) cannot match compiler `-O2` performance, demonstrating that modern compilers perform holistic optimizations (register allocation, scheduling, vectorization) unreachable by source-level changes alone.

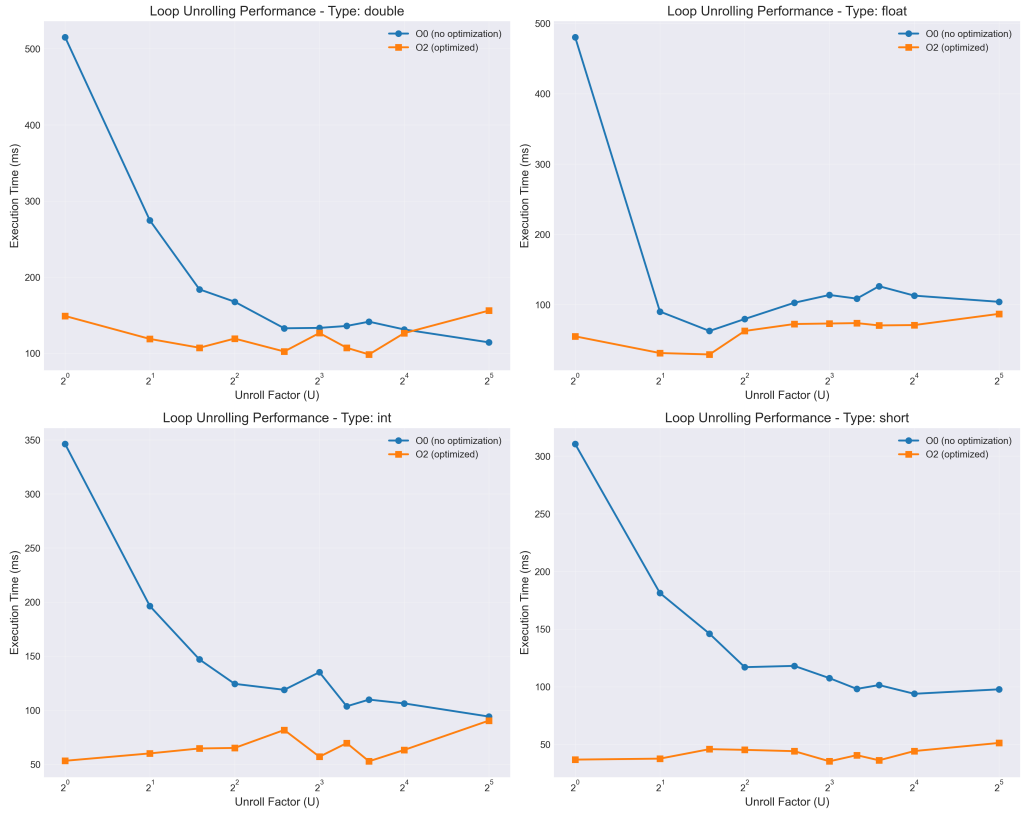

Figure 1: Execution time vs. unrolling factor U for `-O0` and `-O2`.

Type	Size (B)	Best U	Time (ms)	Speedup
`double`	8	12	98.48	1.00×
`float`	4	3	29.23	3.37×
`int`	4	12	52.72	1.87×
`short`	2	8	35.32	2.79×

Table 2: Performance comparison across data types at `-O2`.

2.2 Impact of Data Type on Performance

Table 2 compares performance across different data types.

Analysis: `float` achieves the highest speedup (3.37×) due to reduced memory footprint and efficient SIMD vectorization. Surprisingly, `int` is slower than `float` despite simpler arithmetic—modern compilers vectorize floating-point operations more efficiently. `short` underperforms due to alignment penalties and less efficient 16-bit SIMD packing.

2.3 Memory Bandwidth Analysis

We estimate the theoretical minimum time as $T_{\text{min}} = \frac{N \times \text{sizeof(type)}}{\text{BW}}$, assuming 20 GB/s streaming bandwidth:

All measured times significantly exceed theoretical minimums. The bottleneck is **memory latency** and **cache effects**, not bandwidth. `float` achieves the highest efficiency (68.4%) due to better cache utilization and vectorization.

Type	Size (MB)	$T_{\min}$ (ms)	Measured (ms)	Efficiency
`double`	800	40.0	98.48	40.6%
`float`	400	20.0	29.23	68.4%
`int`	400	20.0	52.72	37.9%
`short`	200	10.0	35.32	28.3%

Table 3: Theoretical vs. measured performance at `-O2`.

2.4 Why Unrolling Saturates

Performance evolves through three regimes as U increases:

1. **Loop Overhead Reduction ($U = 1$ to $U = 4$):** Fewer branch/increment instructions per element processed.
2. **ILP Exploitation ($U = 4$ to $U = 12$):** Out-of-order execution processes independent additions in parallel.
3. **Resource Saturation ($U > 12$):** Performance degrades due to:
   - **Register spilling:** Limited registers force stack access.
   - **I-cache pollution:** Large loop bodies evict critical instructions.
   - **Bandwidth limitation:** Memory subsystem fully saturated; additional ILP provides no benefit.

Conclusion: Manual unrolling provides gains at `-O0`, but `-O2` automatically selects near-optimal factors. Ultimate performance is bounded by memory subsystem limits, not computational throughput.

3 Exercise 2: Instruction Scheduling

Instruction scheduling reorders operations to prevent pipeline stalls caused by high-latency instructions like multiplications.

3.1 Assembly Comparison and Optimization Analysis

The transition from `-O0` to `-O2` illustrates a fundamental shift in how the processor handles the pipeline, moving from naive stack-heavy operations to aggressive hardware utilization.

3.1.1 Observation at -O0: Stack-Based Bottlenecks

The assembly generated at `-O0` reveals a "naive" compilation strategy where the state is constantly synchronized with memory. We observed:

- **Redundant Load-After-Store:** The values of `a` and `b` are reloaded from memory (`[rbp-32]` and `[rbp-24]`) for both streams. This introduces unnecessary **load-use latencies**.
- **High Register Pressure on xmm0:** The compiler reuses `xmm0` for every intermediate calculation, creating a **Read-After-Write (RAW)** dependency chain that prevents the CPU from executing Stream 1 and Stream 2 in parallel.

3.1.2 Observation at -O2: Architectural Exploitation

At `-O2`, the compiler eliminates the overhead of the C-language abstraction to exploit the underlying hardware:

1. **Loop-Invariant Code Motion (LICM):** The product $a \times b$ is hoisted out of the loop. In the assembly, we see this value pre-loaded into a register, turning a high-latency `mulsd` into a simple `addsd` inside the loop.
2. **Loop Unrolling & Branch Reduction:** The loop counter decrements by 2 (`sub eax, 2`), effectively halving the number of `jne` jumps. This reduces the pressure on the **Branch Predictor** and minimizes pipeline flushes.
3. **Exploiting Superscalar Execution:** By allocating independent `xmm` registers for x and y, the compiler enables **Instruction-Level Parallelism (ILP)**. The CPU can dispatch multiple `addsd` instructions to different functional units simultaneously.

3.2 Manual Optimization and Final Conclusion

By manually hoisting the multiplication (`double ab = a * b`), we achieved a 9% performance gain at `-O0` (0.528s $\rightarrow$ 0.480s).

Conclusion: Even with manual source-level improvements, the `-O0` version remains an order of magnitude slower than `-O2` (0.078s). This gap proves that the primary bottleneck is not just the arithmetic complexity, but the **Memory Wall**. Manual optimization cannot force the compiler to perform efficient **Register Allocation**; as long as the code remains at `-O0`, the CPU is throttled by the latency of the stack (L1 cache/RAM) rather than the speed of the execution units.

4 Exercise 3: Amdahl's and Gustafson's Laws

In this exercise, we analyze the theoretical limits of parallelization for a program containing both sequential and parallelizable components.

4.1 Question 1: Code Analysis

To determine the parallel potential of the program, we examine the data dependencies within each function:

1. **Strictly Sequential Part:** The function `add_noise` is strictly sequential. The operation `a[i] = a[i-1] * 1.0000001` creates a **loop-carried dependency**. Because iteration i requires the result of $i-1$, these iterations cannot be executed concurrently.
2. **Parallelizable Part:**
   - `init_b`: Each element of the array is assigned independently (`b[i] = i * 0.5`).
   - `compute_addition`: The addition is element-wise; `c[i]` depends only on `a[i]` and `b[i]`.
   - `reduction`: While the global sum is a single value, partial sums can be computed independently and combined (associative reduction).
3. **Complexity Analysis:** All four functions consist of a single loop iterating from 0 to N. Therefore, the time complexity for each individual part is $O(N)$.

4.2 Question 2: Measuring the Sequential Fraction (f_s)

We measured the sequential fraction f_s using two methods:

1. **Instruction Counts** from `callgrind` for $N = 10^8$ (Table 4).
2. **Execution Time** for varying N (Table 5).

4.2.1 Instruction-Based f_s

Function	Instruction Count (Ir)	Status
`add_noise`	1,799,999,997	Sequential
`init_b`	1,300,000,012	Parallelizable
`compute_addition`	2,200,000,014	Parallelizable
`reduction`	1,200,000,013	Parallelizable
Total Program	**6,500,157,241**	100%

Table 4: Callgrind profiling results for $N = 10^8$.

$$f_s^{(\text{instruction})} = \frac{Ir_{\text{sequential}}}{Ir_{\text{total}}} = \frac{1,799,999,997}{6,500,157,241} \approx \mathbf{0.2769}$$

4.2.2 Time-Based f_s for Varying N

N	Sequential Time (s)	Total Time (s)	f_s (from time)
5×10^6	0.011	0.029	0.3793
10^7	0.022	0.053	0.4151
10^8	0.159	0.501	0.3174

Table 5: Measured execution times and derived f_s values.

We observe that f_s varies with N, showing that the sequential overhead is not strictly proportional to problem size.

4.3 Question 3: Strong Scaling (Amdahl's Law)

Amdahl's Law defines the theoretical speedup $S(p)$ for a fixed problem size given p processors:

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

Using our time-based f_s values from Table 5, we compute the speedup for each N (Table 6).

Analysis: The speedup saturates quickly—typically by $p = 16$—because the parallel execution time becomes negligible compared to the fixed sequential time (`add_noise`). The maximum speedup is capped at $1/f_s$, which ranges from 2.41× to 3.15× for our tested N. This confirms that even with infinite processors, the sequential fraction imposes a hard performance limit under strong scaling.

4.4 Question 4: Effect of Problem Size

From Table 5, we see that f_s is not constant; it fluctuates between 0.38 and 0.32. This variation suggests that the overhead of the sequential part does not scale perfectly linearly with N, possibly due to cache effects or measurement noise. Nevertheless, the overall trend shows that larger N slightly reduces f_s, allowing marginally better strong scaling.

p	$N = 5 \times 10^6$ ($f_s = 0.3793$)	$N = 10^7$ ($f_s = 0.4151$)	$N = 10^8$ ($f_s = 0.3174$)
1	1.00	1.00	1.00
2	1.45	1.41	1.52
4	1.87	1.78	2.05
8	2.19	2.05	2.48
16	2.39	2.21	2.78
32	2.51	2.31	2.95
64	2.57	2.36	3.05
Max ($p \to \infty$)	2.64	2.41	3.15

Table 6: Amdahl speedup predictions for varying N and f_s.

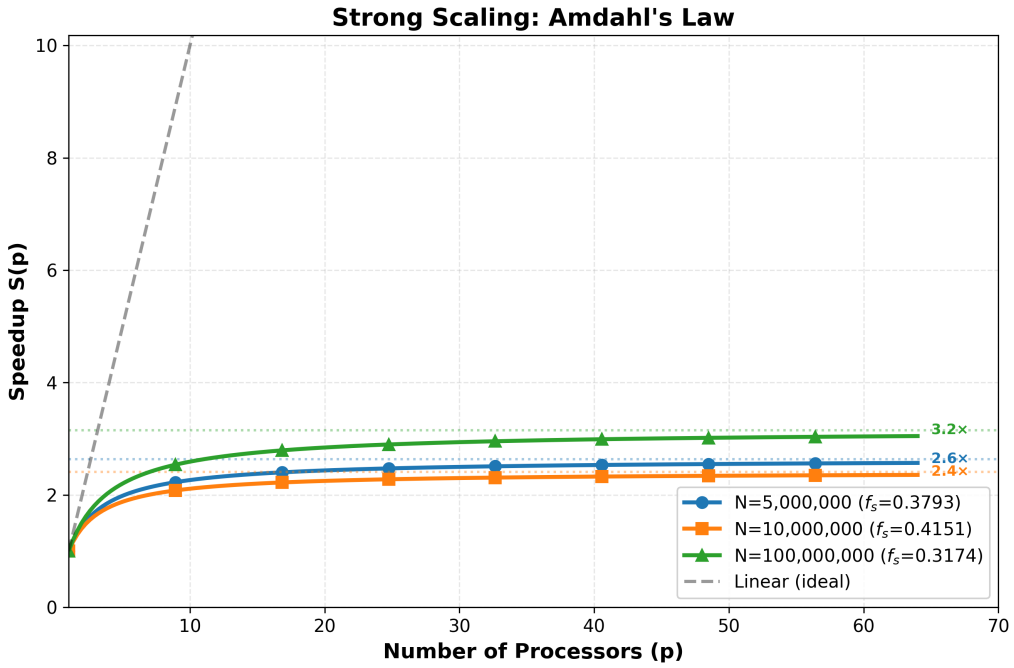

Figure 2: Strong scaling curves for different problem sizes N, showing saturation due to the sequential bottleneck.

4.5 Question 5: Weak Scaling (Gustafson's Law)

Gustafson's Law models the case where the problem size grows with p, keeping the work per processor constant:

$$S(p) = p - f_s(p - 1)$$

p	$N = 5 \times 10^6$ ($f_s = 0.3793$)	$N = 10^7$ ($f_s = 0.4151$)	$N = 10^8$ ($f_s = 0.3174$)
1	1.00	1.00	1.00
2	1.62	1.58	1.68
4	2.86	2.75	3.05
8	5.34	5.09	5.78
16	10.31	9.77	11.24
32	20.24	19.13	22.16
64	40.10	37.85	44.01

Table 7: Gustafson speedup predictions for varying N.

Comparison of Laws:

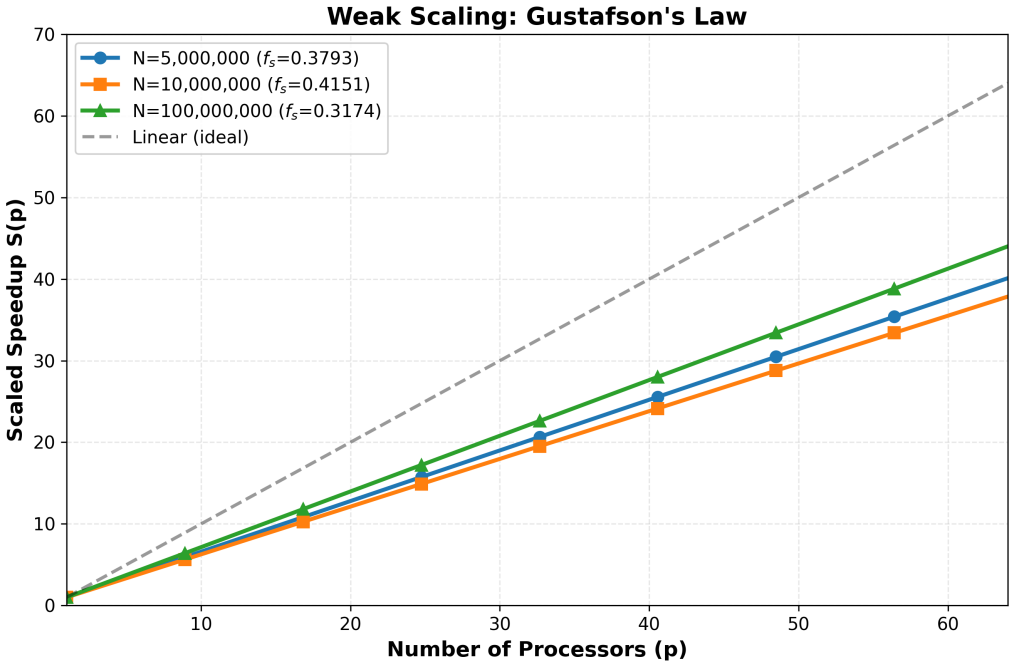

Figure 3: Weak scaling curves showing near-linear speedup as the problem grows with p.

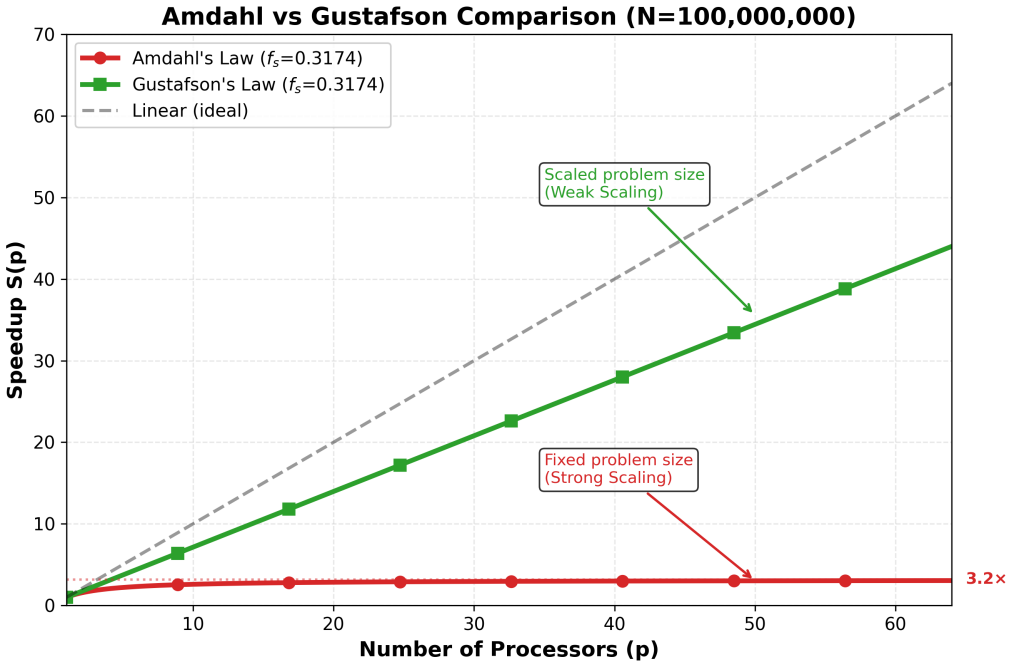

Figure 4: Comparison of Amdahl's (saturating) and Gustafson's (scaling) laws for $N = 10^8$, $f_s = 0.3174$.

- **Amdahl's Law** predicts a rigid ceiling (max $\approx 3.15\times$) because the problem size is fixed.
- **Gustafson's Law** shows near-linear speedup (up to $44\times$ at $p = 64$) because the sequential fraction becomes negligible as the total work increases.

This demonstrates that parallel computing is most effective for large-scale problems where the workload can grow with available resources.

5 Exercise 4: Matrix-Matrix Multiplication with Sequential Noise

In this exercise, we analyze a program combining a sequential noise generator with a parallelizable matrix-matrix multiplication (MatMul) kernel. Unlike Exercise 3 where both parts were $O(N)$, here the parallel MatMul is $O(N^3)$ while the sequential noise generation remains $O(N)$. This allows us to study how computational complexity affects parallel scaling.

The code structure:

- `generate_noise(noise)`: Sequential $O(N)$ initialization
- `matmul(A, B, C, noise)`: Parallelizable $O(N^3)$ multiplication

5.1 Code Analysis and Complexity

The program consists of three components with distinct complexities:

Function	Complexity	Parallelizable?
`generate_noise`	$O(N)$	No (loop-carried dependency)
`init_matrix`	$O(N^2)$	Yes (embarrassingly parallel)
`matmul`	$O(N^3)$	Yes (independent computations)

Table 8: Complexity analysis of Exercise 4 components.

The `matmul` function dominates execution time for large N, while the sequential `generate_noise` grows only linearly. This creates a fundamentally different scaling profile than Exercise 3.

5.2 Question 1: Sequential Fraction Measurement

Using Callgrind instruction counts (Ir), we calculated the sequential fraction f_s for three matrix sizes:

N	Sequential (Ir)	Total (Ir)	f_s
256	1,282	119,619,043	1.07×10^{-5}
512	2,563	947,506,309	2.70×10^{-6}
1024	5,123	7,547,383,380	6.79×10^{-7}

Table 9: Sequential fraction decreases as N increases.

Verification of $O(N^3)$ Complexity: Doubling N from 256 to 512 increases total instructions by $\approx 7.92\times$ (expected: $8\times$). From 512 to 1024, the increase is $\approx 7.97\times$. This confirms the $O(N^3)$ scaling of the dominant `matmul` function.

5.3 Question 2: Strong Scaling (Amdahl's Law)

For $N = 1024$ with $f_s = 6.79 \times 10^{-7}$:

Analysis: Unlike Exercise 3 (which saturated at $3.15\times$), the MatMul kernel achieves near-perfect scaling. With $f_s < 10^{-6}$, the sequential portion is effectively invisible—adding more processors continues to yield proportional speedup up to hundreds of cores.

5.4 Question 3: Weak Scaling (Gustafson's Law)

For all tested values of N, Gustafson's Law predicts:

$$S(p) = p - f_s(p-1) \approx p$$

Processors (p)	Speedup	Efficiency (%)
1	1.00	100.00
16	16.00	100.00
64	64.00	100.00
128	127.99	99.99
256	255.96	99.98
∞ (max)	1,473,235×	—

Table 10: Near-perfect Amdahl speedup for $N = 1024$.

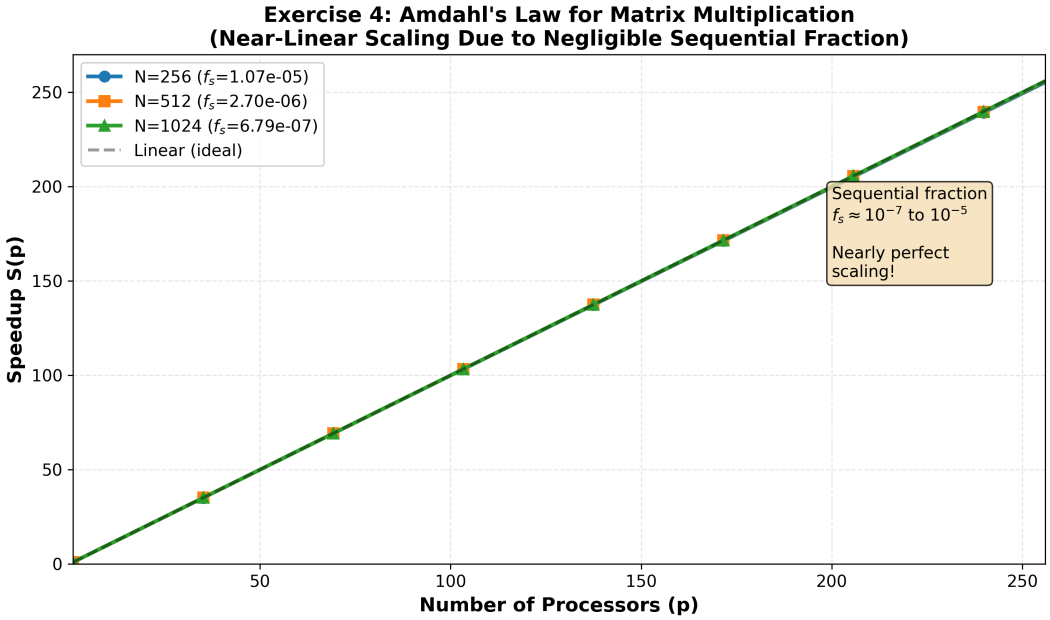

Figure 5: Amdahl's Law for matrix multiplication. The sequential bottleneck is negligible, resulting in near-linear scaling even for large p.

Because $f_s \ll 1$, the Gustafson speedup is nearly identical to the number of processors:

This demonstrates that when both the problem size and the number of processors scale together, the MatMul workload achieves ideal parallel efficiency.

5.5 Question 4: Comparison with Exercise 3

The contrast between Exercise 3 and Exercise 4 illustrates the profound impact of computational complexity on parallel scalability:

Metric	Exercise 3 ($O(N)$)	Exercise 4 ($O(N^3)$)
Sequential fraction f_s	0.3174	6.79×10^{-7}
Max Amdahl speedup	3.15×	1,473,235×
Speedup at $p = 64$	3.05×	64.00×
Parallel efficiency at $p = 64$	4.8%	100.0%

Table 11: Quantitative comparison of parallel performance.

Key Observations:

1. **Complexity Growth:** In Exercise 3, both sequential and parallel parts scale as $O(N)$,

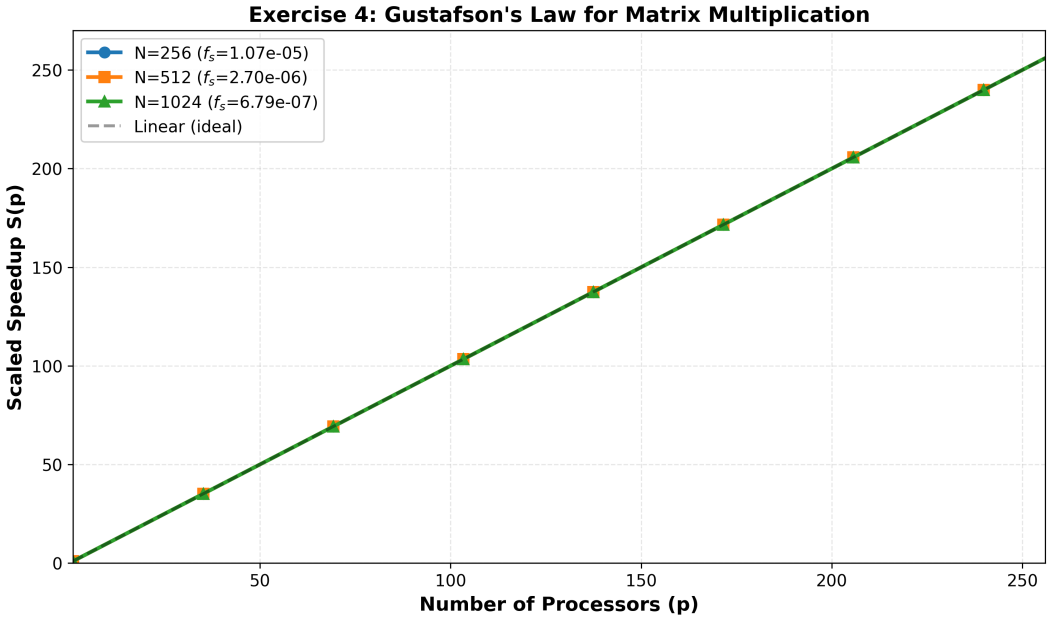

Figure 6: Gustafson's Law shows perfect linear scaling. All three curves are indistinguishable from the ideal line.

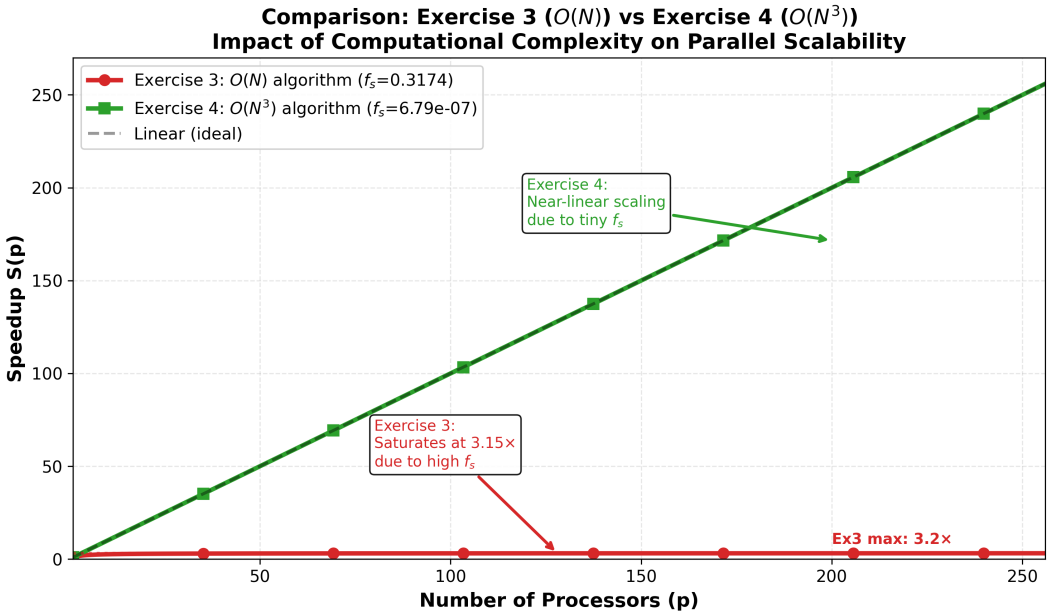

Figure 7: Direct comparison: Exercise 3 ($O(N)$, $f_s = 0.32$) vs. Exercise 4 ($O(N^3)$, $f_s = 6.8 \times 10^{-7}$). The $O(N^3)$ algorithm achieves near-linear scaling while the $O(N)$ algorithm saturates at 3.15×.

keeping f_s relatively constant (≈ 0.32). In Exercise 4, the parallel part scales as $O(N^3)$ while the sequential part remains $O(N)$, causing f_s to vanish as N increases.

2. **Asymptotic Behavior:** Figure 8 shows that f_s decreases approximately as $1/N^2$ (since sequential is $O(N)$ and total is $O(N^3)$). This means that for sufficiently large matrices, the sequential overhead becomes arbitrarily small.

3. **Practical Implications:** Compute-intensive algorithms like matrix multiplication, FFTs, and neural network training are ideal candidates for massive parallelization. In contrast,

I/O-bound or memory-bound workloads (like those in Exercise 3) remain fundamentally limited by their sequential components.

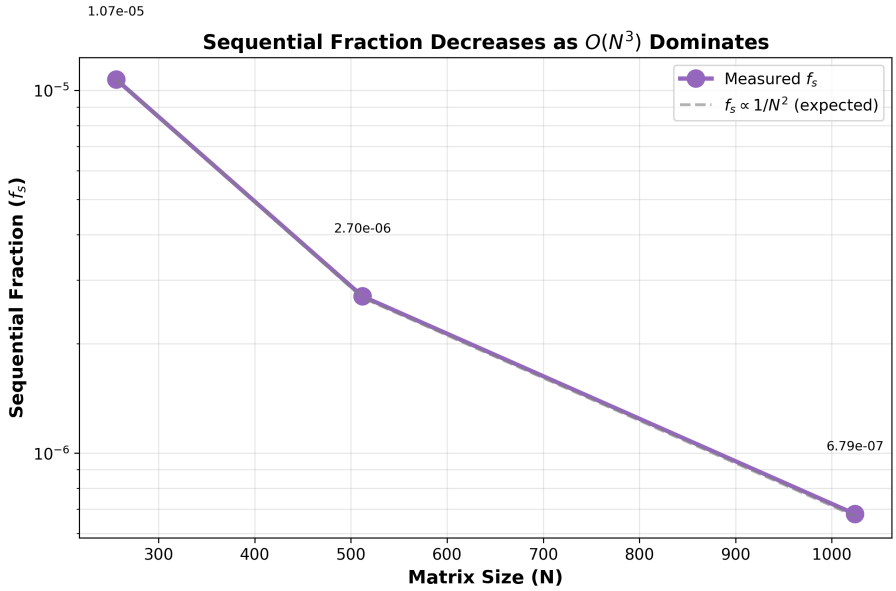

Figure 8: Sequential fraction f_s decreases as $O(1/N^2)$ as the matrix size grows, confirming that the $O(N^3)$ parallel work dominates.

5.6 Conclusion

Exercise 4 demonstrates that **computational complexity is destiny** in parallel computing. While Exercise 3's $O(N)$ operations faced a hard speedup ceiling of 3.15× due to a 32% sequential fraction, Exercise 4's $O(N^3)$ MatMul achieves near-perfect scaling because its sequential overhead vanishes asymptotically. This explains why scientific computing applications (which often involve dense linear algebra, PDEs, and iterative solvers with high complexity) can efficiently utilize thousands of cores, while simple data processing pipelines may struggle to scale beyond a few dozen.

The lesson: when designing parallel algorithms, prioritize problems where the parallelizable work grows faster than the sequential overhead.

6 Conclusion

This TP demonstrated that low-level optimizations like loop unrolling and scheduling can provide significant gains, but are often better handled by modern compilers (`-O2`). However, the ultimate limit to parallel performance is dictated by the code's inherent sequential fraction, as evidenced by our Amdahl and Gustafson scaling models.