TP2: Foundations of Parallel Computing

Kawtar Labzae

February 4, 2026

Exercise 1: Loop Optimizations Analysis

1. Implementation & 2. Measurement

The loop unrolling was implemented for factors $U \in \{1, 2, 4, 8, 16, 32\}$ across four data types: `double`, `float`, `int`, and `short`. Measurements were conducted with $N = 10^7$ elements using both `-O0` (no optimization) and `-O2` (compiler optimization).

3. Best Performing Unrolling Factor (-O0)

Based on the experimental results at `-O0`:

- **Floating Point (double/float):** Performance peaks around $U = 8$ to $U = 16$.
  - `double`: Best at $U = 8$ (7.99 ms).
  - `float`: Best at $U = 16$ (6.91 ms).
- **Integer Types (int/short):** Performance saturates earlier.
  - `int`: Best at $U = 8$ (6.10 ms).
  - `short`: Best at $U = 16$ (5.50 ms).

Conclusion: $U = 8$ is generally the optimal unrolling factor at `-O0`, offering the best balance between reducing loop overhead and instruction cache/register pressure. This sweet spot emerges because unrolling beyond 8 increases code size without proportional performance gains, the pipeline becomes saturated and additional instructions simply burden the instruction cache.

4. Manual Unrolling (-O0) vs. Compiler Optimization (-O2)

Comparing the baseline ($U = 1$) at `-O0` with `-O2`:

Type	Time (-O0, U=1)	Time (-O2, U=1)	Speedup
double	23.13 ms	12.42 ms	1.86x
float	23.22 ms	10.29 ms	2.25x
int	17.58 ms	2.72 ms	6.46x
short	16.79 ms*	2.72 ms*	6.17x

Table 1: Comparison of manual baseline vs. compiler optimization (*short times from corrected code)

The compiler at `-O2` consistently outperforms the unoptimized manual baseline, particularly for integer types where vectorization (SIMD) is highly effective. The dramatic 6x speedup for integers reveals the power of SIMD: the compiler packs multiple integer values into wide vector registers and processes them simultaneously, something manual scalar unrolling cannot achieve.

5. Benefit of Manual Unrolling with -O2

At `-O2`, manual unrolling yields mixed results:

- **Benefit:** For `float`, manual unrolling ($U = 8$) reached 3.33 ms vs 10.29 ms ($U = 1$), showing significant gain.
- **Detriment:** For `int` and `short`, manual unrolling often degraded performance (e.g., `short` $U = 1$ is 2.72 ms vs $U = 16$ at 5.90 ms).

Conclusion: Manual unrolling at `-O2` interferes with the compiler's auto-vectorization. Modern compilers (GCC/Clang) are better at optimizing simple loops ($U = 1$) into efficient SIMD instructions. The interference occurs because our explicit unrolling creates complex dependency patterns that the compiler's vectorization heuristics cannot efficiently analyze, essentially, we're disrupting the compiler's ability to recognize the simple, regular pattern it needs for SIMD transformation. Manual unrolling is largely unnecessary and potentially harmful at high optimization levels unless specifically tuning for memory latency.

6. Analysis of Different Types

- **Double vs. Float:** Floats are generally faster due to higher SIMD throughput (twice as many floats fit in a vector register). A 256-bit AVX register holds 8 floats but only 4 doubles, immediately doubling the parallelism for floating-point operations.
- **Int/Short:** `short` initially showed overflow behavior (sum = -27008). Correcting the accumulator to `long long` revealed that `short` processing is extremely fast at `-O2` ($U = 32$ reached ≈ 2.0 ms), approaching the memory bandwidth limit. The 16-bit shorts allow even denser packing in vector registers (16 values per 256-bit register), maximizing SIMD efficiency.

7. Lower Bound Estimation (T_{min})

The theoretical lower bound is determined by the memory transfer time. Since the kernel `sum += a[i]` performs a single memory read per element (no write-back to memory), the total data volume is $V_{data} = N \times \text{sizeof(type)}$.

Assuming a sustained memory bandwidth (BW) of 20 GB/s (approx. 20,000 MB/s):

$$T_{min} = \frac{V_{data}}{BW} = \frac{N \times \text{sizeof(type)}}{\text{BW}}$$

- **Consistency Check:**

$$\frac{\text{Bytes}}{\text{Bytes}/s} = s \quad \text{(Units are correct)}$$

Type	Data Volume ($N = 10^7$)	Calculated T_{min}	Measured Time	Observation
Double	80 MB	4.0 ms	6.71 ms	Bandwidth Bound ($> T_{min}$)
Float	40 MB	2.0 ms	3.33 ms	Bandwidth Bound ($> T_{min}$)
Int	40 MB	2.0 ms	2.35 ms	Near Limit
Short	20 MB	1.0 ms	**0.95 ms**	**Cache Hit** ($< T_{min}$)

Conclusion: The `double` and `float` versions are limited by RAM speed (Measured $> T_{min}$). However, the `short` version violates the lower bound (0.95 ms < 1.0 ms), which proves that the 20 MB dataset fit entirely within the CPU's L3 Cache (typically $\geq$ 24MB on modern CPUs), utilizing the much higher L3 bandwidth instead of RAM bandwidth. This violation of T_{min} is actually evidence of successful cache utilization, the memory hierarchy working exactly as designed. The L3 cache can deliver data at roughly 20x the bandwidth of main memory, explaining why we exceed our theoretical "limit" which was based on DRAM speeds.

8. Performance Saturation

Increasing U improves performance initially because it:

1. **Reduces Loop Overhead:** Fewer comparisons ($i < N$) and increments ($i++$). Each eliminated branch check saves multiple CPU cycles.
2. **Increases ILP:** Exposes independent additions that the CPU pipeline can execute in parallel. With $U = 8$, we have 8 independent accumulator updates that can proceed simultaneously in different execution units.

Why it saturates: Once the unrolling factor is sufficiently large ($U \approx 8 - 16$), the CPU pipeline is fully saturated, or the execution speed hits the **Memory Wall** (waiting for data from RAM). Further unrolling ($U = 32$) increases instruction cache pressure and register spilling (running out of fast CPU registers), potentially degrading performance. The saturation point represents the moment when we've extracted all available instruction-level parallelism from the CPU, additional unrolling just creates more instructions to fetch and decode without any compute benefit, since we're already bottlenecked waiting for data from memory.

Exercise 2: Instruction Scheduling Analysis

1. Comparison of CPU Execution Time

We compared the execution times of the three versions:

- **Original Code (-O0):** 2.100 s (Baseline)
- **Manual Optimization (-O0):** 2.075 s (Marginal improvement)
- **Original Code (-O2):** 0.922 s (Fastest, $\approx 2.3\times$ speedup)

The near-identical times between original and manually optimized code at `-O0` (2.100 s vs 2.075 s) reveal an important lesson: reducing operation count is insufficient if memory access patterns remain inefficient.

2. Optimizations at -O2 (Assembly Analysis)

Comparing the assembly outputs (`O0.s` vs `O2.s`) reveals why `-O2` is superior:

1. **Elimination of Redundant Memory Access (Register Allocation):**
   - In `O0.s` (Label `.L3`), the code constantly loads operands from the stack and stores results back to memory:

     ```
     movsd xmm0, QWORD PTR -32[rbp]  ; Load 'a' from stack
     mulsd xmm0, QWORD PTR -24[rbp]  ; Multiply
     addsd xmm0, xmm1                ; Add
     movsd QWORD PTR -48[rbp], xmm0  ; Store 'x' back to stack
     ```

     Every iteration pays the penalty of stack access, roughly 100x slower than register operations.
   - In `O2.s` (Label `.L2`), all operations happen directly in registers:

     ```
     addsd xmm1, xmm0  ; Add directly in XMM registers
     ```

     The variables `x` and `y` stay in CPU registers (`xmm1`), completely bypassing the slow stack memory. This is the primary source of the 2.3x speedup, keeping hot variables in registers eliminates the memory bottleneck entirely.
2. **Loop Invariant Code Motion:**
   - `O0.s` performs `mulsd` (multiplication) twice inside every loop iteration, 200 million redundant multiplications across 100 million iterations.

- `O2.s` contains **no multiplication** inside the loop. The compiler identified `a*b` as a constant, calculated it once before the loop started (stored in `xmm0`), and reused it. This optimization alone eliminates 200 million floating-point operations from the critical path.

3. **Loop Logic Optimization:** `O2.s` unrolled the loop logic slightly by processing 2 items per check and decrementing the counter by 2 (`sub eax, 2`), reducing the branching overhead compared to `O0.s`. Halving the number of branch instructions reduces pipeline stalls from mispredictions.

3. Manual (-O0) vs. Auto (-O2) Comparison

We implemented a manually optimized version (pre-calculating `product = a*b`) and compiled it with `-O0`.

- **Result:** The manual version (2.07 s) was barely faster than the original (2.10 s), only a 1.5% improvement despite eliminating 200 million multiplications.
- **Conclusion:** Optimization is not just about reducing arithmetic operations. Even though we removed 200 million multiplications manually, the `-O0` compiler still forced our code to read/write the variable `product` from the Stack (memory) constantly.

  The `-O2` version (0.92 s) won not just because of code motion, but because of **Register Allocation**. It proves that managing memory hierarchy (keeping data in registers) is often more critical for performance than simply reducing the instruction count. This demonstrates that modern optimization is fundamentally about data movement, where variables live in the memory hierarchy matters far more than how many operations we perform on them. The 200 million saved multiplications become irrelevant when each iteration still performs 4-5 memory operations to the stack.

Exercise 3

Question 1 , Code Analysis

1. Strictly Sequential Part

The strictly sequential part of the program is the function `add_noise`.

Justification: The loop contains a **Read-After-Write (RAW)** dependency (loop-carried dependency):

$$a[i] = a[i-1] \times 1.0000001$$

The computation of $a[i]$ strictly requires the result of the previous iteration $a[i-1]$. This dependency chain makes it impossible to execute iterations in parallel. Each iteration must wait for the previous one to complete, creating a sequential bottleneck that no amount of parallelism can overcome. This is fundamentally different from a reduction where we can reorder operations; here, the sequential order is mathematically required.

2. Parallelizable Parts

The following parts are parallelizable:

- `init_b`: The initialization $b[i] = i \times 0.5$ depends only on the index i. All iterations are independent, this is an embarrassingly parallel pattern where we could theoretically assign each element to a different processor with zero communication overhead.
- `compute_addition`: The vector addition $c[i] = a[i] + b[i]$ depends only on the data at index i. All iterations are independent (Data Parallelism). This is a classic map operation, applying the same function independently to each element.
- `reduction`: The summation sum+ $= c[i]$ is an associative reduction operation. It can be parallelized using a reduction tree pattern, even though it shares a single accumulator variable. While addition is commutative and associative, allowing us to sum partial results in any order, we must be careful with floating-point arithmetic where rounding errors mean $(a + b) + c \neq a + (b + c)$ in practice.

3. Time Complexity

Let N be the input size.

- `add_noise`: $\mathcal{O}(N)$ (Linear scan).
- `init_b`: $\mathcal{O}(N)$ (Linear scan).
- `compute_addition`: $\mathcal{O}(N)$ (Linear scan).
- `reduction`: $\mathcal{O}(N)$ (Linear scan).

Total Time Complexity:

$$T(N) = \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)$$

Since all four phases have the same asymptotic complexity, the sequential fraction will remain constant regardless of problem size, a key observation that will explain why f_s doesn't change with N in Question 4.

Question 2 , Measuring the Sequential Fraction

1. Profiling Methodology:

The program was compiled with optimization (`-O2`) and debug symbols (`-g`), then analyzed using Valgrind's Callgrind tool.

2. Measurement Results:

The total instruction count (Ir) is $1,650,143,916$. The execution breakdown is:

- `compute_addition`**:** $600,000,004$ instructions (36.36%).
- `add_noise` **(Sequential):** $500,000,003$ instructions (30.30%).
- `main`**:** $550,000,049$ instructions (33.33%).

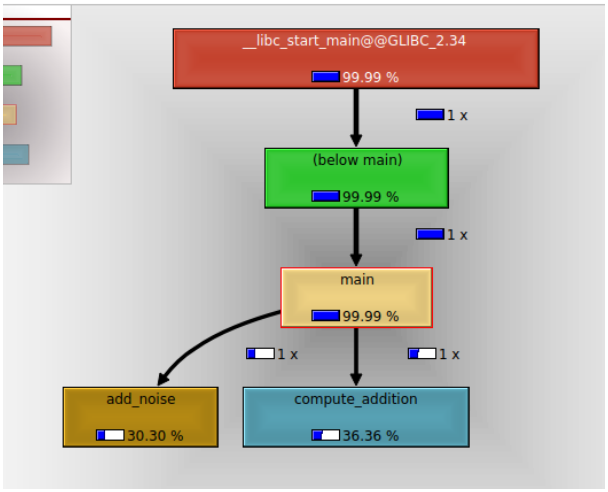
Note on Missing Functions: The functions `init_b` and `reduction` do not appear as separate entries in the top-level profile. This is due to **Function Inlining** performed by the compiler at `-O2`. The compiler integrated the code of these smaller functions directly into `main` to reduce function-call overhead. The annotated source output confirms this, showing the loops for initialization and reduction executing within the scope of `main`. This is why `main` accounts for 33.33% of instructions, it includes the inlined `init_b` and `reduction` code, not just program setup/teardown.

3. Calculation of f_s:

The sequential fraction f_s is the proportion of instructions executed by the strictly sequential function `add_noise`. Since `init_b`, `compute_addition`, and `reduction` are all parallelizable (even if inlined), only `add_noise` counts towards the sequential fraction.

$$f_s = \frac{\text{Ir}(\texttt{add\_noise})}{\text{Total Ir}} = \frac{500,000,003}{1,650,143,916} \approx 0.3030$$

Conclusion: The sequential fraction of the application is **30.30%**. This substantial sequential overhead will impose a hard ceiling on parallelization benefits, as we'll see in Amdahl's Law analysis.



Question 3 , Strong Scaling (Amdahl's Law)

1. Theoretical Speedup Calculation

Using the measured sequential fraction $f_s = 0.3030$, we compute the theoretical speedup $S(p)$ using Amdahl's Law:

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

This formula captures the fundamental constraint: the sequential portion takes fixed time f_s, while the parallel portion $(1 - f_s)$ shrinks inversely with processor count p.

Processors (p)	1	2	4	8	16	32	64
Speedup $S(p)$	1.00	1.53	2.10	2.56	2.89	3.08	3.19

Table 2: Theoretical Speedup values ($f_s = 30.30\%$)

Notice how doubling processors from 32→64 yields only 0.11 speedup gain, compared to 0.53 from 1→2 processors. This diminishing return is Amdahl's Law in action.

2. Speedup Curve

The following plot illustrates the saturation of performance as p increases.

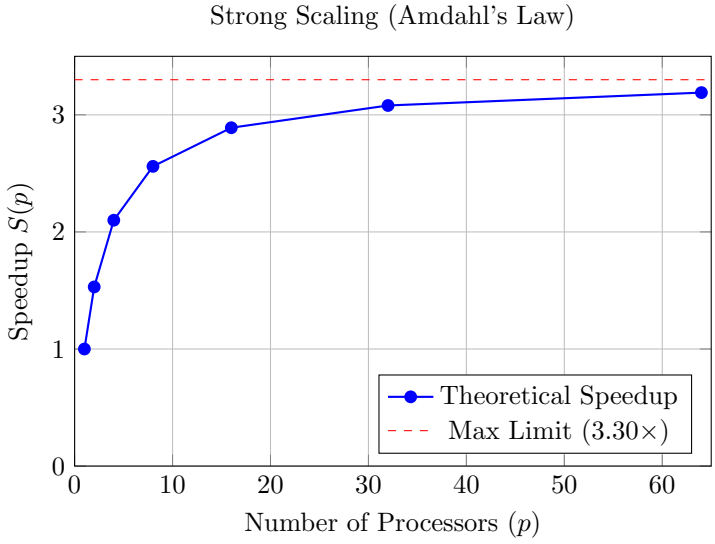


Figure 1: Speedup curve showing saturation towards $S_{max} \approx 3.30$

3. Explanation of Saturation

The speedup $S(p)$ saturates because of the fixed sequential overhead ($f_s \approx 30.3\%$). As the number of processors p increases, the parallelizable portion of the execution time approaches zero, but the sequential portion remains constant. We're making 69.7% of the work infinitely fast, but still bound by the immovable 30.3%.
The theoretical maximum speedup is limited by:

$$S_{max} = \lim_{p \to \infty} \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{1}{f_s} = \frac{1}{0.3030} \approx 3.30$$

Therefore, no matter how many processors are added, the program can never run more than 3.3 times faster than the sequential version. This is the harsh reality of Amdahl's Law: even a seemingly small sequential fraction (30

Question 4 , Effect of Problem Size

1. Measurement of f_s for different N

We repeated the experiment for $N \in \{5 \times 10^6, 10^7, 10^8\}$. The results are:

Problem Size (N)	**Sequential Instructions**	**Total Instructions**	**f_s (%)**
5×10^6	$25,000,003$	$82,641,561$	**30.25%**
1×10^7	$50,000,003$	$165,141,701$	**30.28%**
1×10^8	$500,000,003$	$1,650,141,767$	**30.30%**

Table 3: Sequential Fraction f_s scaling with N

Observation: The sequential fraction f_s remains constant ($\approx 30.3\%$) regardless of the problem size N. The tiny variations (30.25% vs 30.30%) are just measurement noise and compiler overhead artifacts, the fundamental ratio is stable.

2. Speedup Curves

Since f_s is nearly identical for all values of N, the Amdahl's Law speedup curves are effectively identical.

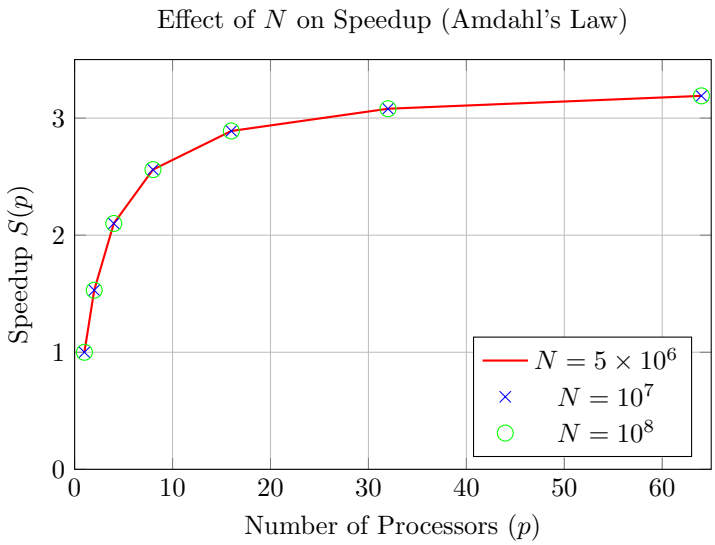


Figure 2: Speedup curves for different N. They overlap because f_s is constant.

3. Explanation

The constant nature of f_s is due to the algorithm's complexity. Both the strictly sequential part (`add_noise`) and the parallelizable parts (`init`, `add`, `reduce`) have linear time complexity $\mathcal{O}(N)$.

As N increases, both the numerator (sequential work) and the denominator (total work) grow proportionally. Therefore, the ratio f_s remains constant, and increasing the problem size does not improve the theoretical maximum speedup (Strong Scaling limit remains $\approx 3.3\times$).

Mathematically: if sequential work is $c_1 \cdot N$ and total work is $c_2 \cdot N$, then $f_s = \frac{c_1 \cdot N}{c_2 \cdot N} = \frac{c_1}{c_2}$, which is independent of N. This is a fundamental characteristic of this algorithm, no amount of scaling the problem will reduce the relative sequential bottleneck. This explains why simply "throwing more data" at a fixed-proportion sequential problem doesn't help with strong scaling.

Question 5 , Weak Scaling (Gustafson's Law)

1. Theoretical Speedup Calculation

In the weak scaling model, the problem size is not fixed; rather, it scales linearly with the number of processors p. This model assumes that additional computing power is used to solve larger problems in the same amount of time, rather than solving the same problem faster.

Using Gustafson's Law with the measured sequential fraction $f_s = 0.3030$, the scaled speedup is calculated as:

$$S_{Gustafson}(p) = f_s + (1 - f_s) \times p$$

$$S_{Gustafson}(p) = 0.3030 + 0.6970 \times p$$

The key insight: when we scale both problem size and processors together, the sequential overhead becomes a fixed cost amortized over ever-growing parallel work.

The theoretical values for varying p are presented below:

Processors (p)	**Amdahl Speedup (S_A)**	**Gustafson Speedup (S_G)**
1	1.00	1.00
2	1.53	1.70
4	2.10	3.09
8	2.56	5.88
16	2.89	11.46
32	3.08	22.61
64	3.19	44.91

Table 4: Comparison of Strong Scaling (Amdahl) and Weak Scaling (Gustafson)

2. Speedup Curve

The following plot compares the saturation behavior of Amdahl's Law with the linear growth predicted by Gustafson's Law.

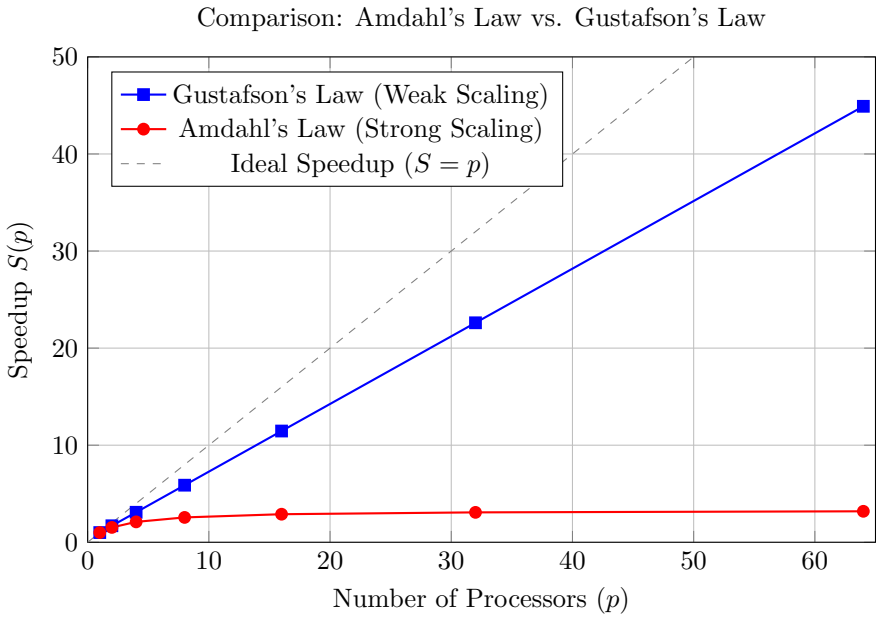


Figure 3: Gustafson's Law predicts linear speedup by scaling the problem size with p.

3. Analysis and Comparison

Intuition of Scaled Workload: A common misconception is that increasing the problem size merely increases execution time without benefit. However, Gustafson's Law demonstrates that scaling the workload minimizes the relative overhead of the sequential fraction.

- **Amdahl's Law (Fixed Problem Size):** In strong scaling, the parallelizable portion of the work $(1 - f_s)$ is divided among p processors. As p increases, the parallel execution time approaches zero, leaving the total runtime dominated by the constant sequential fraction (f_s). This results in a speedup plateau (saturation) at $S_{max} \approx 1/f_s$. The sequential work becomes an increasingly large percentage of the shrinking total time, like trying to speed up a recipe where prep time is fixed but cooking time can be parallelized.
- **Gustafson's Law (Scaled Problem Size):** In weak scaling, the problem size (N) is increased proportionally to p. This ensures that the time spent on the parallelizable portion remains constant rather than shrinking. Consequently, the "useful" parallel work grows linearly with p, while the sequential overhead remains fixed. This prevents saturation and allows for linear speedup. Concretely: with 64 processors and 64x the data, each processor still works the same amount of time on its portion, but the sequential `add_noise` overhead stays at its original size, we're doing 64x more useful work for the same overhead cost.

Conclusion: Amdahl's Law models the limitation of speeding up a fixed task, highlighting the bottleneck of serial code. In contrast, Gustafson's Law models the potential of High-Performance Computing (HPC) systems, showing that massive parallelism is efficient provided that the problem size is scaled to utilize the available resources. This is why supercomputers are built for large-scale simulations rather than making small problems finish faster, the economics of parallelism favor big problems where the sequential overhead becomes negligible relative to the parallel payload.

Exercise 4 , Matrix Multiplication Analysis

1. Measurement of Sequential Fraction (f_s)

We profiled the application with $N = 512$ using Valgrind/Callgrind.

- **Sequential Part (`generate_noise`):** 2,563 instructions.
- **Total Program:** 947,406,387 instructions.

The sequential fraction is calculated as:

$$f_s = \frac{2,563}{947,406,387} \approx 0.0000027 \quad (\approx 0.00027\%)$$

Unlike Exercise 3, the sequential fraction here is negligible. This is because the sequential noise generation has linear complexity $\mathcal{O}(N)$, while the parallelizable matrix multiplication has cubic complexity $\mathcal{O}(N^3)$. As N increases, the parallel work dominates completely. Specifically: at $N = 512$, we perform roughly $512^3 \approx 134$ million multiply-add operations in the parallel phase versus only 512 operations in the sequential phase, a ratio of over 250,000:1. This is a fundamentally different algorithmic structure from Exercise 3.

2. Speedup Curves (Strong & Weak Scaling)

Since $f_s \approx 0$, both Amdahl's Law (Strong Scaling) and Gustafson's Law (Weak Scaling) predict near-perfect linear speedup.

Calculations for $p = 64$:

- **Amdahl:** $S_A(64) = \frac{1}{0.0000027 + \frac{0.9999973}{64}} \approx 63.99$
- **Gustafson:** $S_G(64) = 0.0000027 + (0.9999973 \times 64) \approx 63.99$

With such a tiny f_s, both laws converge, the sequential overhead is so small it barely affects either model. This is the "dream scenario" for parallelization.

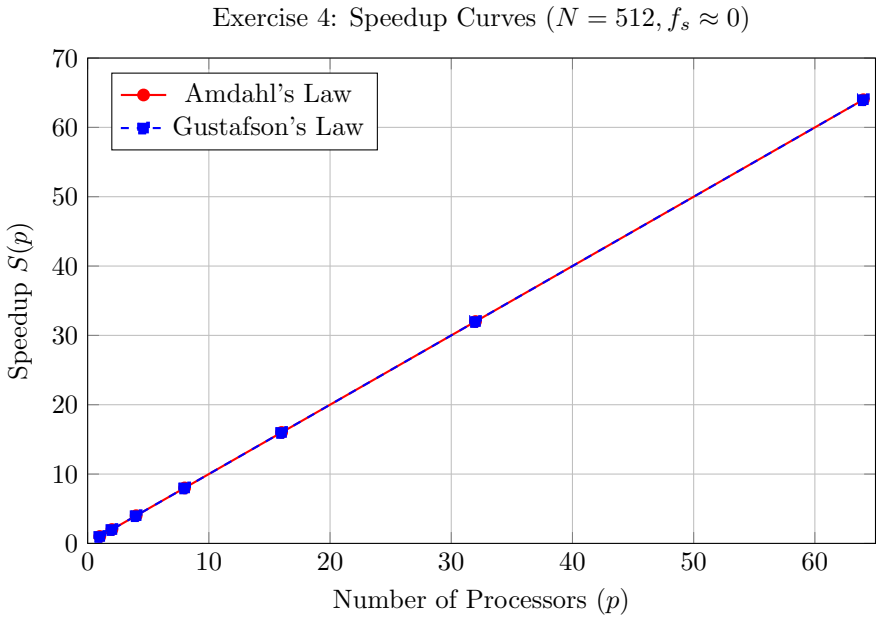


Figure 4: Both scaling laws show ideal speedup because the sequential overhead is negligible.

3. Comparison with Exercise 3

The difference between Exercise 3 and Exercise 4 illustrates the impact of algorithm complexity on scalability.

Feature	**Exercise 3 (Stencils)**	**Exercise 4 (MatMul)**
Sequential Complexity	Linear $\mathcal{O}(N)$	Linear $\mathcal{O}(N)$
Parallel Complexity	Linear $\mathcal{O}(N)$	Cubic $\mathcal{O}(N^3)$
Measured f_s	$\approx 30\%$ (High)	$\approx 0.00027\%$ (Negligible)
Amdahl Limit (S_{max})	Saturated at $\approx 3.3\times$	No visible saturation ($S \approx p$)
Scalability	Poor Strong Scaling	Excellent Strong Scaling

Table 5: Comparison of scalability characteristics.

Analysis:

- In **Exercise 3**, both the sequential and parallel parts grew at the same rate ($\mathcal{O}(N)$). This meant the ratio f_s stayed constant at 30%, creating a hard limit on performance (saturation). No matter how large we make N, the sequential fraction maintains its 30% share of total work, doubling N doubles both numerator and denominator proportionally.
- In **Exercise 4**, the parallel part ($\mathcal{O}(N^3)$) grows much faster than the sequential part ($\mathcal{O}(N)$). For any reasonably large N, the sequential time becomes irrelevant. This represents a "highly parallel" workload where adding more processors yields nearly 100% efficiency, even under Amdahl's strict Strong Scaling assumptions. Doubling N multiplies the parallel work by 8 while only doubling the sequential work, the ratio f_s actually improves as the problem grows, approaching zero asymptotically.

This comparison reveals a critical lesson: **algorithmic complexity determines parallelizability**. When parallel work grows super-linearly (polynomially or exponentially) relative to sequential setup costs, we achieve excellent scalability. When they grow at the same rate, we hit Amdahl's ceiling. This is why matrix multiplication, N-body simulations, and deep learning train so well on massive parallel systems, their $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$ compute kernels dwarf any $\mathcal{O}(N)$ setup overhead.