TP2

Youness Anouar

February 5, 2026

Contents

---

1 Exercise 1

1.1 Questions 1–2: Manual Unrolling Baseline

We sum an array of $N = 10^6$ doubles using a manual unroll factor $U \in [1, 32]$. The execution time $T(U)$ is shown in Fig. 1.

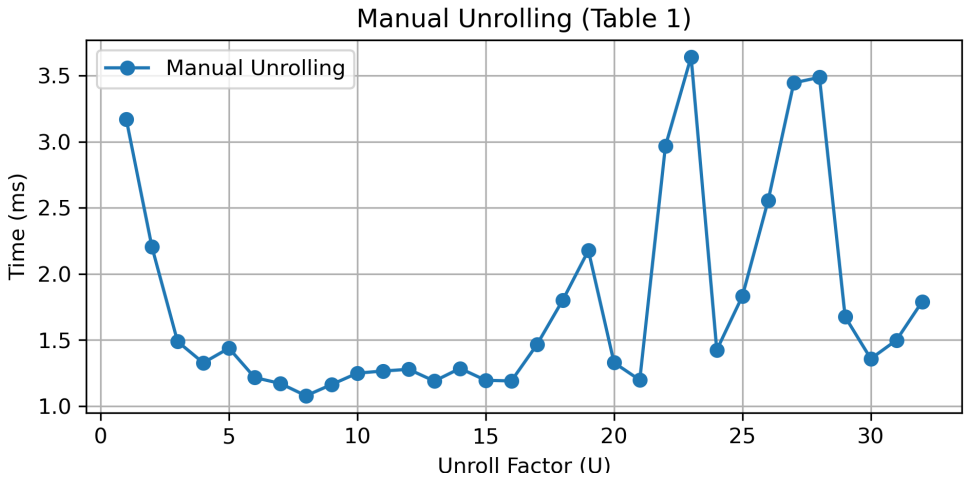

Figure 1: Manual unrolling with doubles

Observation:

- Time drops quickly when going from $U = 1$ to medium U.
- Best run in the shown experiment: around $U = 8$–10 (≈ 1.1 ms vs. 3.2 ms for $U = 1$).
- Very large U makes performance noisy and sometimes worse.

1.2 Question 3: Best Unroll Factor Stability

Across multiple runs the best U is not fixed; typical best values are in $[6, 10]$ (e.g., $U = 8$ in one run). Small timing noise and interactions with caches, branch prediction, and OS scheduling explain why the exact optimum moves, but the "good region" is stable.

1.3 Questions 4–5: Manual Unrolling vs. `-O2`

We now compare `-O0` and `-O2` while still manually unrolling. Results are summarized in Fig. 2.

Key points:

- `-O0`: best time ≈ 1.22 ms at $U = 10$ (about 65–70% faster than $U = 1$).
- `-O2`: best time ≈ 0.65 ms at $U = 6$ (also $\approx 70\%$ faster than $U = 1$ at `-O2`).
- Even with `-O2`, manual unrolling helps: the compiler does not fully remove loop overhead for this kernel.
- The best U with `-O2` is smaller (around 6) than with `-O0`, because the compiler already performs part of the work (strength reduction, instruction scheduling, partial unrolling).

Conclusion for Q5:

- Manual unrolling is clearly beneficial at `-O0`.
- With `-O2`, a moderate unroll factor plus compiler optimizations gives the best performance.
- The compiler alone does not completely replace manual unrolling here.

1.4 Question 6: Effect of Data Type

We repeat the experiment with `float`, `int`, and `short` (Fig. 3). For `short` we use $N = 10^4$ to avoid overflow.

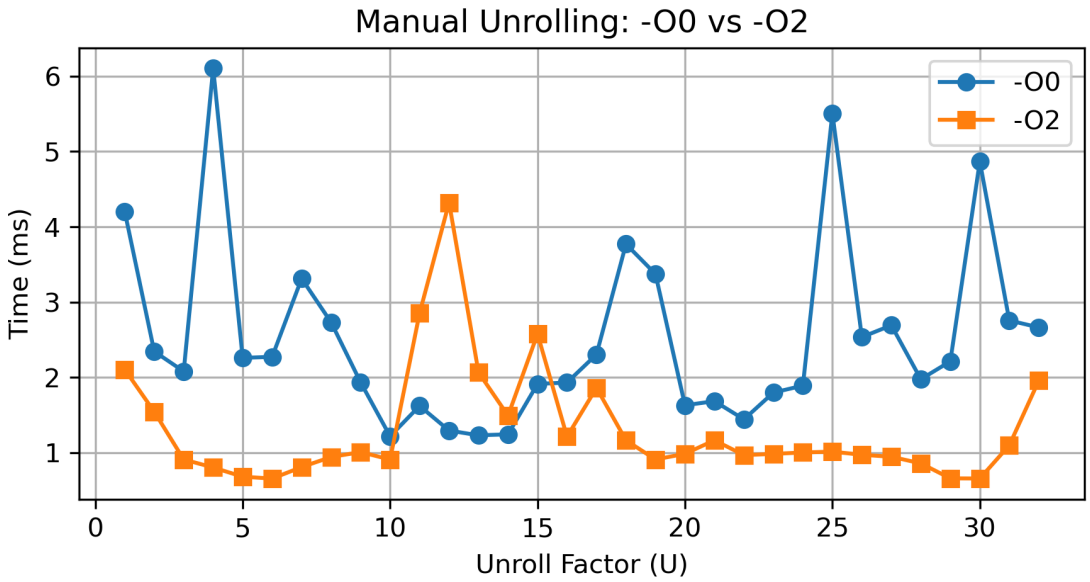

Figure 2: Manual unrolling at `-O0` vs. `-O2`.

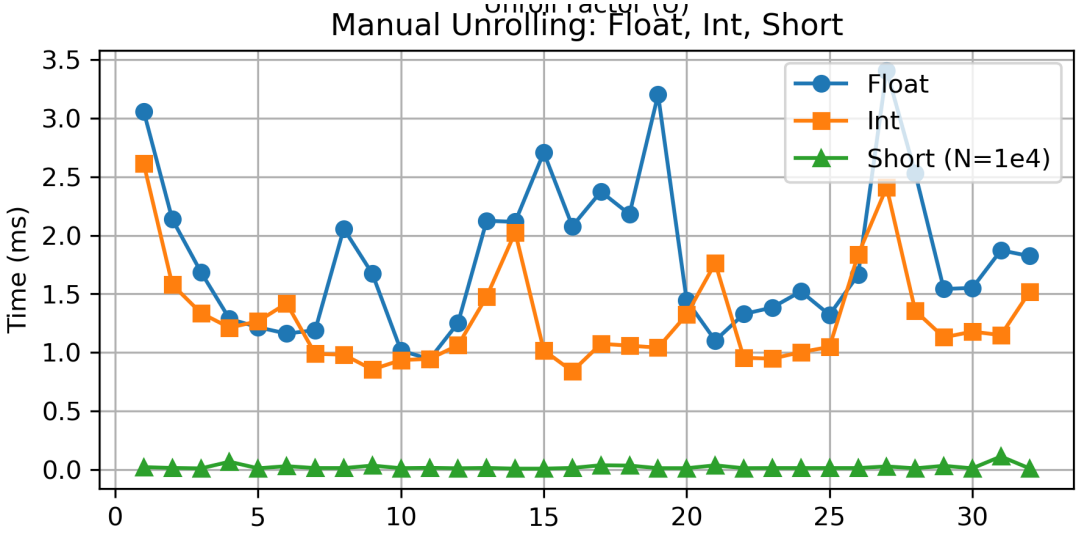

Figure 3: Manual unrolling for `float`, `int`, and `short`.

Observations:

- **Float**: best around $U = 11$ (≈ 0.94 ms).
- **Int**: best around $U = 16$ (≈ 0.84 ms).
- **Short** ($N = 10^4$): best around $U = 14$–15 (≈ 0.008 ms), but timings are very small and noisy.
- Optimal U depends on type; heavier types (double/float) hit compute and bandwidth limits earlier than `int`/`short`.

1.5 Question 7: Theoretical Lower Bound (Bandwidth)

We approximate a lower bound assuming the kernel is purely memory-bandwidth limited:

$$T_{\min} \approx \frac{N \times \text{sizeof(type)}}{\text{BW}},$$

with $N = 10^6$ (except `short`: $N = 10^4$) and BW ≈ 30 GB/s.

- **Double** (8 B): 8 MB $\Rightarrow T_{\min} \approx 0.27$ ms.
  Best measured ≈ 1.08 ms ($\sim 4\times$ slower).
- **Float** (4 B): 4 MB $\Rightarrow T_{\min} \approx 0.13$ ms.
  Best measured ≈ 0.94 ms ($\sim 7\times$ slower).
- **Int** (4 B): same bound, best ≈ 0.84 ms ($\sim 6\times$ slower).
- **Short** ($N = 10^4$, 2 B): 20 KB $\Rightarrow T_{\min} \approx 0.0007$ ms.
  Best measured ≈ 0.008 ms ($\sim 11\times$ slower) and dominated by timer overhead.

Interpretation:

- All measured times are well above the bandwidth bound: the kernel is not purely memory-bandwidth limited.
- Main overheads: loop control, instruction dependencies, pipeline effects, memory latency, and timer resolution for tiny runtimes.
- Loop unrolling reduces *computational* overhead but does not change the bandwidth limit.

1.6 Question 8: Why Performance Improves Then Saturates

Trend of $T(U)$:

- For small U (e.g., $1 \leq U \leq 8$–16):
  - Fewer loop iterations (N/U instead of N) $\Rightarrow$ less branch and counter overhead.
  - More independent operations per iteration $\Rightarrow$ better instruction-level parallelism and pipeline usage.
  - Result: strong speedup when increasing U from 1.
- For large U:
  - Loop body becomes large $\Rightarrow$ more I-cache pressure and register pressure (spills).
  - Execution units and memory system approach saturation; further ILP does not help.
  - Small perturbations (cache effects, branch prediction, OS noise) make timings noisy; performance can degrade.

Overall, the kernel transitions from compute-bound at low U to close to a memory/latency bound at high U. The best performance is obtained with a moderate unroll factor (roughly $6 \leq U \leq 16$ depending on type and optimization level).

2 Exercise 2

2.1 Question 1: Performance at `-O0` vs. `-O2`

Measured results:

- `-O2`: $x = 131999999.662353$, $y = 131999999.662353$, time $= 0.107$ s
- `-O0`: $x = 131999999.662353$, $y = 131999999.662353$, time $= 0.271$ s

2.2 Question 2: Analysis of Compiler Optimizations

At `-O0`, each iteration performs two floating-point multiplications and frequent loads/stores from the stack for a, b, x, and y, with less efficient loop control.

At `-O2`, the compiler:

- Hoists the product $a \times b$ out of the loop (loop-invariant code motion), storing it in a register.
- Uses a single accumulator in an XMM register, minimizing memory traffic.
- Holds the loop counter in a register and uses efficient branch instructions, improving instruction scheduling and instruction-level parallelism.

2.3 Question 3: Manual Optimization vs. Compiler Optimization

Manually optimized (`-O0`, with $a \times b$ hoisted):

- $x = 131999999.662353$, $y = 131999999.662353$, time $= 0.265\,\text{s}$

Comparison:

- Original `-O0`: 0.271 s (slowest)
- Manual optimization: 0.265 s (improved)
- `-O2`: 0.107 s (fastest)

Conclusion: Manual optimization at `-O0` improves performance over the naive version, but is still much slower than the compiler-optimized `-O2` version. Modern compilers apply more aggressive and effective low-level optimizations than are practical to do by hand; thus, writing simple code and enabling `-O2` usually yields the best performance.

3 Exercise 3

3.1 Question 1: Code Analysis

- **Strictly Sequential Parts**: The loop in `add_noise` is strictly sequential because each iteration depends on the previous one ($a[i]$ uses $a[i-1]$). The loop in `reduction` accumulates a sum; while the provided implementation is sequential, it can be parallelized using a tree-based reduction, so it is not inherently strictly sequential in the same way as `add_noise`.
- **Parallelizable Parts**:
  - `init_b`: each $b[i]$ is independent.
  - `compute_addition`: each $c[i]$ depends only on corresponding a and b indices.
  - `reduction`: amenable to parallel reduction strategies.
- **Complexity**: All main functions (`add_noise`, `init_b`, `compute_addition`, `reduction`) have $\Theta(N)$ time complexity.

3.2 Question 2: Profiling with Callgrind

We profiled the application ($N = 10^8$) using Valgrind/Callgrind. The instruction counts (Ir) summary is:

- **Total Program**: $\approx 2.2 \times 10^9$ instructions.
- `main`: $\approx 1.1 \times 10^9$ (50%).
- `compute_addition`: $\approx 6.0 \times 10^8$ (27.27%).
- `add_noise`: $\approx 5.0 \times 10^8$ (22.73%).

The strictly sequential fraction (f_s) corresponds to `add_noise`:

$$f_s = \frac{\text{Ir}_{\text{seq}}}{\text{Ir}_{\text{total}}} = \frac{500,000,002}{2,200,140,720} \approx 0.227 \quad (22.7\%).$$

3.3 Question 3: Amdahl's Law Prediction

Using Amdahl's Law $S(p) = \frac{1}{f_s + \frac{1-f_s}{p}}$ with $f_s \approx 0.23$:

Cores (p)	Theoretical Speedup $S(p)$
1	1.00
2	1.63
4	2.38
8	3.09
16	3.63
32	3.98
64	4.18

As $p \to \infty$, the maximum speedup is bounded by $1/f_s \approx 4.35$.

3.4 Question 4: Consistency Across Problem Sizes

We repeated the profiling for $N = 5 \times 10^6$ and $N = 10^7$. In both cases, the calculated sequential fraction remained $f_s \approx 0.227$.

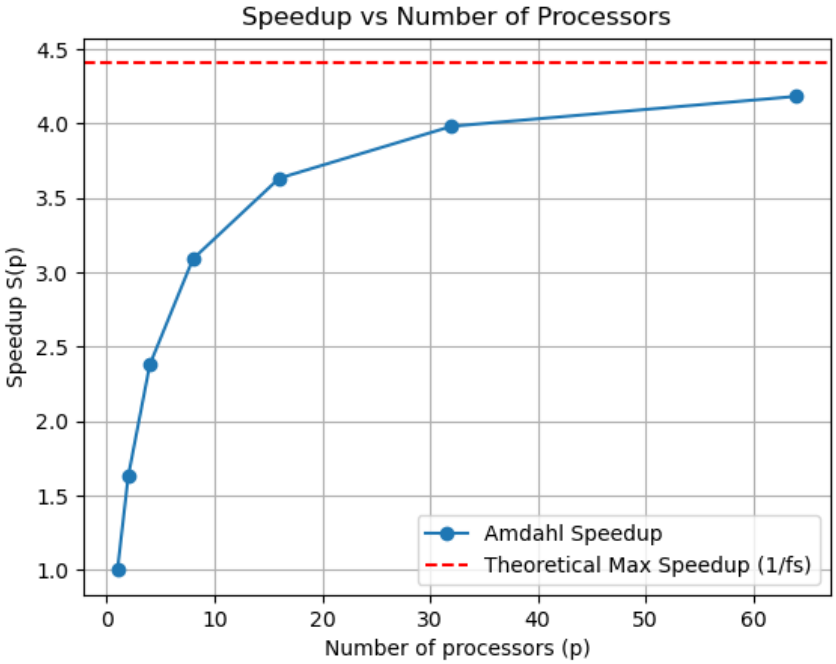

Figure 4: Theoretical Speedup vs Number of Processors (Amdahl's Law).

The speedup saturates quickly due to the fixed sequential overhead.

3.5 Question 5: Gustafson's Law

Gustafson's Law (weak scaling) assumes the problem size grows with p:

$$S_G(p) = p - f_s \times (p - 1)$$

Cores (p)	Gustafson Speedup $S_G(p)$
1	1.000
2	1.773
4	3.319
8	6.411
16	12.595
32	24.963
64	49.699

Table 1: Comparison of Scaling Laws

Law	Scaling	Growth Behavior
Amdahl	Strong	Saturates quickly (limited by f_s)
Gustafson	Weak	Grows almost linearly

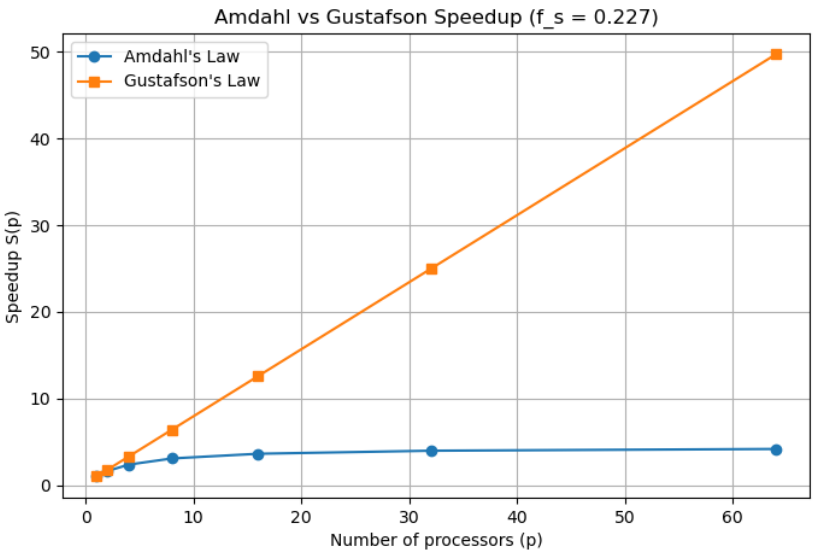

Figure 5: Amdahl vs Gustafson Speedup.

Conclusion: Amdahl's Law shows the limits of parallelization for fixed workloads, while Gustafson's Law highlights the potential for near-linear scaling when the workload increases proportional to the processor count.

4 Exercise 4

4.1 Question 1: Profiling Analysis (N=512)

We profiled a matrix multiplication application ($N = 512$) using Valgrind/Callgrind. The instruction counts summary is:

- **Total Program**: 948,847,389 instructions.
- `matmul`: 941,888,011 (99.27%).
- `init_matrix` (2×): 6,815,754 (0.72%).
- `generate_noise`: 2,562 (≈ 0.00%).

Within `matmul`, the innermost `k` loop accounts for approximately 99.05% of the total instructions, indicating that almost all computation time is spent in the matrix multiplication kernel.

Parallelizability Analysis:

- The outer loops over `i` and `j` in `matmul` are fully parallelizable, as each (i, j) computation is independent.
- The inner `k` loop is a reduction (sum), which is sequential per (i, j) element but can be computed in parallel for different elements.
- `init_matrix` is fully parallelizable (each element independent).
- `generate_noise` is strictly sequential due to loop-carried dependency: `noise[i] = noise[i-1] * 1.0000001`.

The sequential fraction corresponds only to `generate_noise`:

$$f_s = \frac{2,562}{948,847,389} \approx 0.00027\% \quad \Rightarrow \quad f_s \approx 0.003$$

This means the code is highly amenable to parallelization, with nearly all computation being parallel.

4.2 Question 2: Amdahl's Law – Strong Scaling (N=512)

Using Amdahl's Law for strong scaling with $f_s \approx 0.003$:

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

The theoretical maximum speedup as $p \to \infty$ is $1/f_s \approx 333\times$.

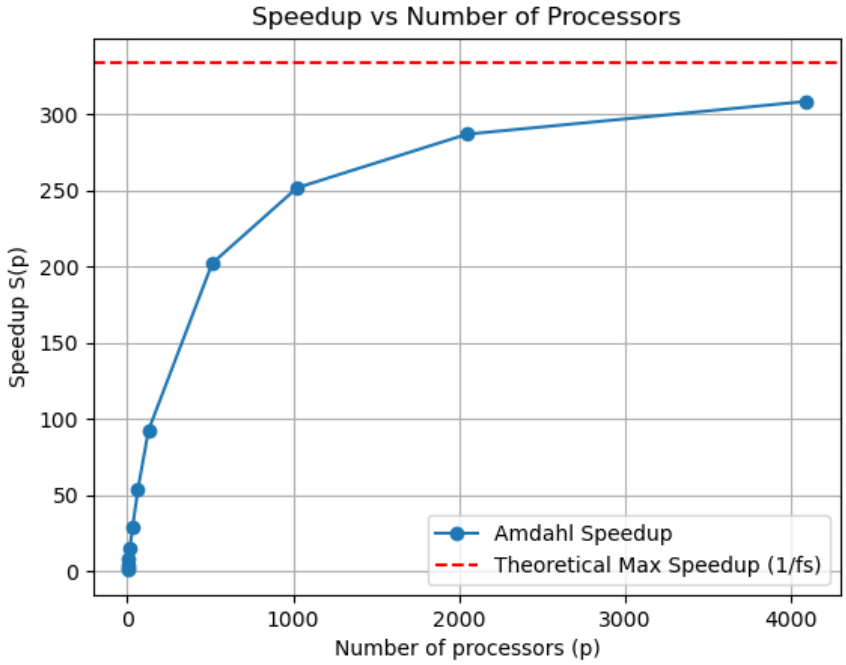

Figure 6: Amdahl's Law speedup vs. number of processors (N=512).

The code achieves near-linear speedup for a large number of processors, limited only by the small sequential fraction.

4.3 Question 3: Gustafson's Law – Weak Scaling (N=512)

Using Gustafson's Law for weak scaling:

$$S_G(p) = p - f_s \times (p - 1)$$

With $f_s \approx 0.003$, Gustafson's Law predicts nearly perfect linear scaling.

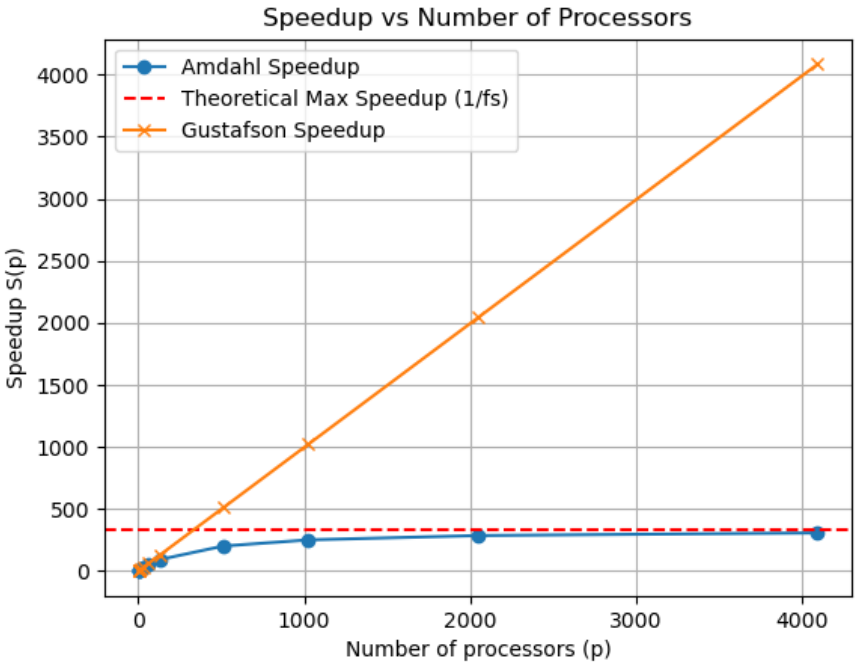

Figure 7: Amdahl vs. Gustafson speedup comparison (N=512).

4.4 Question 4: Consistency Analysis (N=128)

We repeated the profiling for $N = 128$. The instruction counts summary is:

- **Total Program**: 15,396,011 instructions.
- `matmul`: 14,828,683 (96.32%).
- `init_matrix` (2×): 425,994 (2.77%).
- `generate_noise`: 642 (≈ 0.004%).

Sequential fraction (only `generate_noise`):

$$f_s = \frac{642}{15,255,319} \approx 0.000042 \quad (0.0042\%)$$

Maximum Amdahl speedup: $1/f_s \approx 23,809\times$.

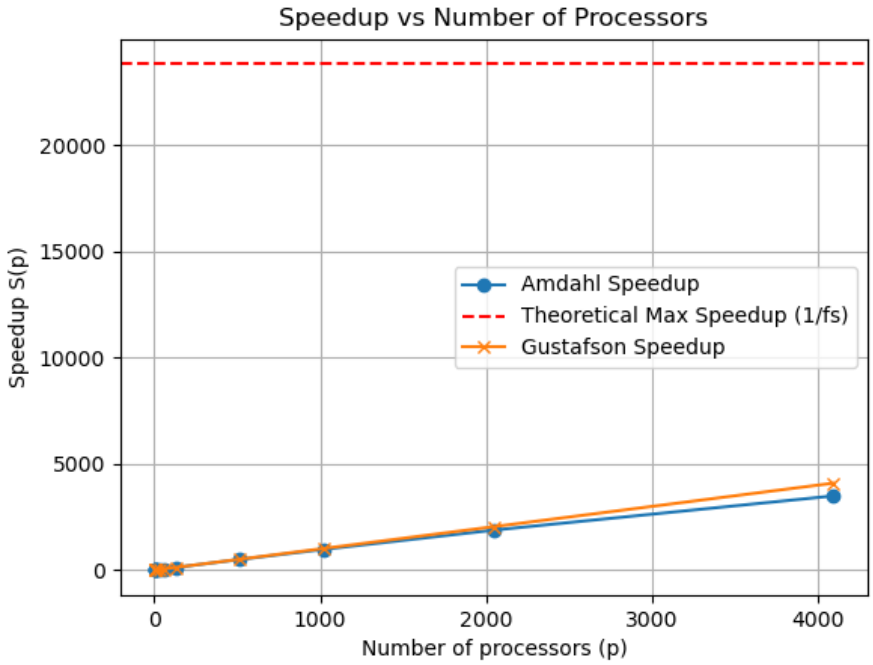

Figure 8: Amdahl vs. Gustafson speedup comparison (N=128).

4.5 Comparison: Exercise 3 vs. Exercise 4

Metric	Exercise 3	Exercise 4 (N=512)
Sequential function	`add_noise`	`generate_noise`
Sequential fraction f_s	22.7%	0.0003%
Parallel fraction	77.3%	99.9997%
Max Amdahl speedup $(1/f_s)$	$\approx 4.35\times$	$\approx 333,333\times$
Scaling at 64 cores (Amdahl)	$4.18\times$	$\approx 64\times$
Scaling at 64 cores (Gustafson)	$49.7\times$	$\approx 64\times$

Key Observations:

- **Sequential fraction impact**: Exercise 3 has a significant sequential portion (22.7%), severely limiting speedup under Amdahl's Law. Exercise 4's sequential fraction is negligible ($< 0.001\%$), allowing near-linear scaling.
- **Complexity differences**: In Exercise 3, all functions scale as $O(N)$, so the sequential fraction remains constant regardless of problem size. In Exercise 4, `matmul` scales as $O(N^3)$ while `generate_noise` scales as $O(N)$, causing the sequential fraction to *decrease* as N grows.
- **Amdahl vs. Gustafson**: For Exercise 3, Gustafson's Law provides much better scaling predictions than Amdahl's Law due to the large f_s. For Exercise 4, both laws predict nearly identical (linear) scaling because $f_s \approx 0$.
- **Practical implications**: Exercise 3 represents a poorly parallelizable workload where weak scaling is essential for good performance. Exercise 4 represents an ideal parallelizable workload where strong scaling is achievable.

Conclusion: The sequential fraction f_s is the critical factor determining parallel scalability. Algorithms with $O(N^k)$ parallel work and $O(N)$ sequential overhead (like matrix multiplication with $k = 3$) become increasingly parallelizable as problem size grows, while algorithms where all components scale equally (like Exercise 3) maintain a fixed scalability limit.