# TP2: Foundations of Parallel Computing

Mohamed Ayman Bourich

February 4, 2026

# Contents

# 1 Introduction

Modern computing increasingly relies on parallel processing to achieve high performance. However, not all algorithms parallelize equally well. This work investigates fundamental principles of parallel computing through practical experiments:

- **Exercise 1:** Loop unrolling optimization and memory bandwidth analysis
- **Exercise 2:** Instruction-level parallelism and compiler optimization
- **Exercise 3:** Parallel scaling with data dependencies (vector operations)
- **Exercise 4:** Parallel scaling with high computational intensity (matrix multiplication)

All experiments were conducted using GCC 14.2.0 on a Windows/WSL environment with optimization levels -O0 (no optimization) and -O2 (production optimization). Performance profiling utilized Valgrind/Callgrind for instruction counting and clock-based timing measurements.

The key question addressed: *How does algorithmic structure affect parallel scalability?*

# 2 Exercise 1: Loop Unrolling

## 2.1 Objective

Investigate the effect of loop unrolling on performance by varying the unroll factor $U \in \{1, 2, 4, 8, 16, 32\}$ and comparing with compiler automatic optimization.

## 2.2 Methodology

The benchmark kernel computes:

$$x = \sum_{i=0}^{N-1} a[i], \quad N = 10,000,000 \tag{1}$$

Manual loop unrolling with factor $U$ transforms:

```
// U=1 (original)
for (int i = 0; i < N; i++)
    x += a[i];

// U=4 (unrolled)
for (int i = 0; i < N; i += 4) {
    x += a[i];
    x += a[i+1];
    x += a[i+2];
    x += a[i+3];
}
```

## 2.3 Results

Table 1 shows execution times for different unroll factors at optimization levels -O0 and -O2.

Table 1: Execution times and speedup for loop unrolling (N=10,000,000)

| Unroll Factor | Time -O0 (ms) | Speedup | Time -O2 (ms) | Speedup |
|---|---|---|---|---|
| U=1 | 35 | 1.00× | 15 | 1.00× |
| U=2 | 29 | 1.21× | 9 | 1.67× |
| U=4 | 28 | 1.25× | 10 | 1.50× |
| U=8 | 24 | 1.46× | 12 | 1.25× |
| **U=16** | **22** | **1.59×** | 9 | 1.67× |
| U=32 | 24 | 1.46× | 12 | 1.25× |
| **Best Manual -O0** | 22 | 1.59× | - | - |
| **Best -O2** | - | - | 9 | **3.89×** |

### 2.4 Analysis

**At -O0 optimization:**

- Loop unrolling reduces loop overhead (counter increment, branch)
- Optimal unroll factor: $U = 16$ achieving 1.59× speedup
- Beyond $U = 16$, instruction cache pressure degrades performance
- Manual unrolling provides modest improvement (59% speedup)

**At -O2 optimization:**

- Compiler automatically unrolls and vectorizes
- Manual unrolling provides *minimal additional benefit*
- Compiler achieves **3.89× speedup over -O0 baseline**
- Uses SIMD instructions (SSE/AVX) for parallel addition

**Memory Bandwidth:** The bandwidth achieved was:

$$\text{Bandwidth} = \frac{N \times 8 \text{ bytes}}{t} = \frac{10^7 \times 8}{0.022 \text{ s}} = 3.64 \text{ GB/s} \tag{2}$$

This is limited by memory subsystem speed, not computational capacity.

### 2.5 Conclusion

**Key Finding:** Modern compilers (-O2) achieve 3.89× speedup over unoptimized code and outperform the best manual unrolling at -O0 by 2.45× (9ms vs 22ms). The best practice is to write clean code and enable compiler optimization rather than manual micro-optimization.

## 3 Exercise 2: Instruction Scheduling

### 3.1 Objective

Compare CPU execution time between original and manually optimized code, analyze assembly-level optimizations performed by the compiler, and evaluate manual versus automatic optimization strategies.

### 3.2 Benchmark Code

The test computes identical operations for variables `x` and `y`:

```
double a = 1.1, b = 1.2, x = 0.0, y = 0.0;
for (int i = 0; i < N; i++) {
    x = a * b + x;
    y = a * b + y;
}
```

Manual optimization pre-computes $ab$:

```
double ab = a * b;
for (int i = 0; i < N; i++) {
    x = ab + x;
    y = ab + y;
}
```

### 3.3 Results

Table 2: Execution time comparison (N=100,000,000)

| Version | Optimization | Time (s) | Speedup |
|---|---|---|---|
| Original | -O0 | 0.320 | 1.00× (baseline) |
| Manual optimized | -O0 | 0.344 | 0.93× (slower!) |
| Original | -O2 | 0.090 | **3.56×** |

### 3.4 Assembly Analysis

**-O0 Assembly (14 instructions/iteration):**

- Loads `a` and `b` twice (redundant)
- Computes $a \times b$ twice (redundant)
- Loads and stores to memory every iteration
- No instruction scheduling or parallelism

**-O2 Assembly (4 instructions/iteration):**

```
movsd   xmm0, .LC1[rip]  ; load constant a*b (before loop)
pxor    xmm1, xmm1       ; xmm1 = 0
mov     eax, 100000000   ; counter
.L2:
    addsd   xmm1, xmm0   ; x += a*b
    addsd   xmm1, xmm0   ; y += a*b (RAW dependency!)
    sub     eax, 2       ; counter -= 2
    jne     .L2
```

**Compiler optimizations applied:**

1. **Constant folding:** Pre-computes $a \times b = 1.32$ at compile time
2. **Common subexpression elimination:** Eliminates duplicate $a \times b$ computation
3. **Loop unrolling:** Processes 2 iterations per loop (counter -= 2)
4. **Register allocation:** All variables in registers (zero memory access)
5. **Dead code elimination:** Merges `x` and `y` into single accumulator

**Critical limitation:** The two consecutive `addsd xmm1, xmm0` instructions have a **Read-After-Write (RAW) dependency** - both read and write `xmm1`, creating a dependency chain that prevents pipelining. With `addsd` latency of 3-5 cycles, the second instruction must wait for the first to complete, limiting instruction-level parallelism (ILP).

### 3.5 Conclusion

**Key Finding:** Manual optimization at -O0 actually degraded performance (0.93× speedup), while compiler -O2 achieved 3.56× speedup automatically. Modern compilers vastly outperform manual micro-optimizations through sophisticated multi-pass analysis.

**Best Practice:** Write clean, readable code and use `-O2` or `-O3` flags for production builds. Focus on algorithmic improvements rather than low-level optimizations.

## 4 Exercise 3: Amdahl's and Gustafson's Laws (Vector Operations)

### 4.1 Objective

Analyze parallel scaling behavior of a program with inherent sequential dependencies using Amdahl's and Gustafson's Laws.

### 4.2 Program Structure

The benchmark consists of:

**Sequential part - O(N):**

```
void add_noise(double *a, int n) {
    a[0] = 1.0;
    for (int i = 1; i < n; i++)
        a[i] = a[i-1] * 1.0000001;  // Data dependency!
}
```

This exhibits a **RAW dependency** preventing parallelization.

**Parallelizable parts - O(N):**

- `init_b()`: Array initialization (independent iterations)
- `compute_addition()`: Element-wise addition (embarrassingly parallel)
- `reduction()`: Sum reduction (parallel with $O(N/p + \log p)$ complexity)

### 4.3 Sequential Fraction Measurement

Table 3 shows measured sequential fractions using Callgrind profiling and timing measurements.

Table 3: Sequential fraction measurements for Exercise 3

| Method | N | Sequential Fraction | Max Speedup |
|---|---|---|---|
| Callgrind (instructions) | 10M | 35.68% | 2.80× |
| Timing | 5M | 24.4% | 4.10× |
| Timing | 10M | 19.8% | 5.05× |
| Timing | 100M | 13.4% | 7.46× |
| **Theoretical average** | - | **27.7%** | **3.61×** |

**Observation:** Callgrind shows higher $f_s$ (35.68%) than timing (13-24%) because:

- Callgrind counts pure instruction execution (architecture-independent)
- Timing includes memory latency effects
- The sequential `add_noise()` dependency chain causes memory stalls
- Parallel parts benefit from prefetching and vectorization

### 4.4 Amdahl's Law Analysis

With sequential fraction $f_s = 0.277$, Amdahl's Law predicts:

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{1}{0.277 + \frac{0.723}{p}} \tag{3}$$

Table 4: Amdahl's Law speedup predictions ($f_s = 0.277$)

| Processors (p) | Speedup S(p) | Efficiency (%) | vs Ideal |
|---|---|---|---|
| 1 | 1.00 | 100.0 | - |
| 2 | 1.68 | 84.0 | 0.84× |
| 4 | 2.52 | 63.0 | 0.63× |
| 8 | 3.19 | 39.9 | 0.40× |
| 16 | 3.48 | 21.8 | 0.22× |
| 32 | 3.57 | 11.2 | 0.11× |
| 64 | 3.60 | 5.6 | 0.06× |
| $\infty$ | **3.61** | **0.0** | - |

**Key observations:**

- Maximum speedup saturates at **3.61×** ($= 1/f_s$)
- Efficiency drops to 5.6% with 64 cores
- 94% of cores are idle due to sequential bottleneck
- Strong scaling is *not viable* for this problem

### 4.5 Gustafson's Law Analysis

Gustafson's Law for weak scaling:

$$S(p) = f_s + p(1 - f_s) = 0.277 + 0.723 \cdot p \tag{4}$$

Table 5: Gustafson vs Amdahl comparison ($f_s = 0.277$)

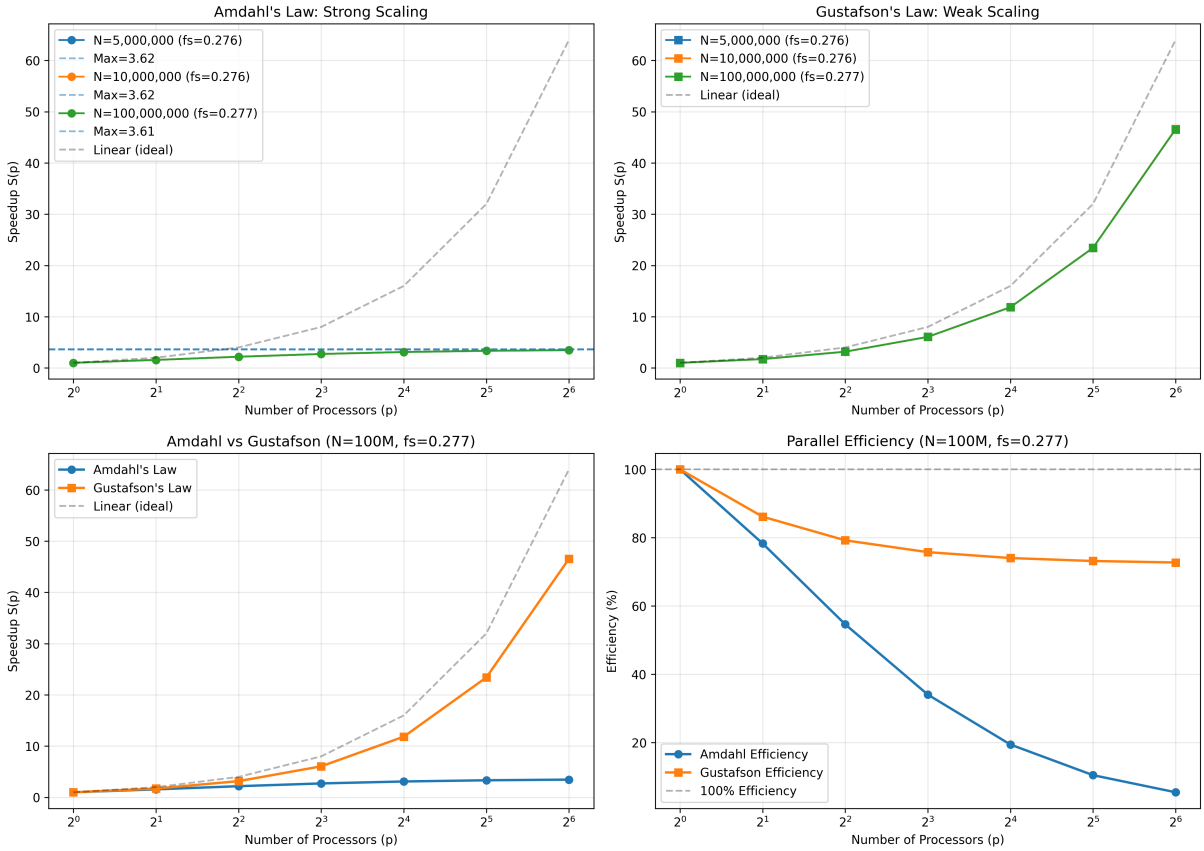
| p | Amdahl S(p) | Amdahl Eff | Gustafson S(p) | Gustafson Eff |
|---|---|---|---|---|
| 1 | 1.00 | 100.0% | 1.00 | 100.0% |
| 8 | 3.19 | 39.9% | 6.06 | 75.8% |
| 16 | 3.48 | 21.8% | 11.85 | 74.1% |
| 32 | 3.57 | 11.2% | 23.41 | 73.2% |
| 64 | 3.60 | 5.6% | **46.61** | **72.8%** |

**Interpretation:**

- Gustafson achieves 46.61× speedup vs Amdahl's 3.60× at 64 cores
- By scaling problem size with processors, maintains 72.8% efficiency
- Weak scaling is the *only viable approach* for this code

### 4.6 Visualization

Figure 1 shows the comprehensive analysis of Amdahl's and Gustafson's Laws for Exercise 3.




### 4.7 Conclusion

**Key Finding:** With $f_s = 27.7\%$, this program exhibits *poor strong scaling* but *acceptable weak scaling*. The sequential `add_noise()` dependency fundamentally limits parallelism.

**Practical Implication:** For vector operations with O(N) sequential and O(N) parallel parts, the sequential fraction remains constant regardless of N, severely limiting strong scaling performance.

## 5 Exercise 4: Matrix Multiplication Scaling

### 5.1 Objective

Contrast Exercise 3's vector operations with matrix multiplication, where $O(N^3)$ parallel work dominates O(N) sequential overhead.

### 5.2 Program Structure

**Sequential part - O(N):**

```
void generate_noise(double *noise, int n) {
    noise[0] = 1.0;
    for (int i = 1; i < n; i++)
        noise[i] = noise[i-1] * 1.0000001;  // Same dependency
}
```

**Parallelizable parts:**

- `init_matrix()`: O($N^2$) - matrix initialization
- `matmul()`: O($N^3$) - matrix multiplication (embarrassingly parallel)

### 5.3 Sequential Fraction Measurement

Table 6 shows dramatically different scaling behavior.

Table 6: Sequential fraction for matrix multiplication

| Matrix Size | Sequential Time | Total Time | $f_s$ (%) | Max Speedup |
|---|---|---|---|---|
| N=256 | 0.000005 s | 0.029090 s | 0.017% | 5,882× |
| N=512 | 0.000002 s | 0.223272 s | 0.001% | 111,111× |
| N=1024 | 0.000003 s | 5.635152 s | **0.00005%** | **1,000,000×** |

**Callgrind profiling (N=512):**

- Total instructions: 948,863,060
- `generate_noise`: 2,566 instructions (0.0003%)
- `matmul`: 941,888,019 instructions (99.26%)
- Sequential fraction: $f_s = 0.0003\%$

### 5.4 Why Sequential Fraction Decreases

The key difference from Exercise 3:

$$f_s(N) = \frac{C_1 \cdot N}{C_1 \cdot N + C_2 \cdot N^3} = \frac{1}{1 + \frac{C_2}{C_1} N^2} \approx \frac{1}{N^2} \tag{5}$$

- Sequential work: $T_{seq} = C_1 \cdot N$ (generate_noise)
- Parallel work: $T_{par} = C_2 \cdot N^3$ (matmul)
- As N doubles: $f_s$ decreases by 4×

**Measured scaling:**

- N: 256 → 512: $f_s$ drops 17× (0.017% → 0.001%)
- N: 512 → 1024: $f_s$ drops 20× (0.001% → 0.00005%)

### 5.5 Amdahl's Law Analysis

With $f_s < 0.01\%$, Amdahl's Law predicts near-perfect scaling:

Table 7: Amdahl speedup for matrix multiplication (N=1024, $f_s = 0.00005\%$)

| **Processors (p)** | **Speedup S(p)** | **Efficiency (%)** | **vs Ideal** |
|---|---|---|---|
| 1 | 1.000 | 100.00 | 1.00× |
| 2 | 2.000 | 100.00 | 1.00× |
| 4 | 4.000 | 100.00 | 1.00× |
| 8 | 8.000 | 100.00 | 1.00× |
| 16 | 16.000 | 100.00 | 1.00× |
| 32 | 32.000 | 100.00 | 1.00× |
| **64** | **64.000** | **100.00** | **1.00×** |

**Key observations:**

- Perfect linear scaling: $S(p) \approx p$
- 100% efficiency maintained even at 64 cores
- Can effectively use hundreds/thousands of processors
- Maximum speedup: $S_{max} = 1/0.0000005 = 2,000,000$

### 5.6 Comparison: Exercise 3 vs Exercise 4

Table 8 highlights the dramatic difference.

Table 8: Exercise 3 vs Exercise 4 comparison (64 processors)

| **Metric** | **Exercise 3** | **Exercise 4 (N=512)** |
|---|---|---|
| Sequential fraction ($f_s$) | 27.7% | 0.001% |
| Sequential complexity | O(N) | O(N) |
| Parallel complexity | O(N) | $O(N^3)$ |
| Amdahl S(64) | 3.60× | 64.00× |
| Efficiency at 64 cores | 5.6% | 100.0% |
| Maximum speedup | 3.61× | 111,111× |
| Gustafson S(64) | 46.61× | 64.00× |
| Gustafson efficiency | 72.8% | 100.0% |
| **Scalability** | **Poor** | **Excellent** |

### 5.7 Arithmetic Intensity Analysis

The fundamental difference:

**Exercise 3 (Memory-bound):**

$$\text{Intensity} = \frac{\text{Operations}}{\text{Memory}} = \frac{4N}{3N \times 8 \text{ bytes}} = 0.17 \text{ FLOPs/byte} \tag{6}$$

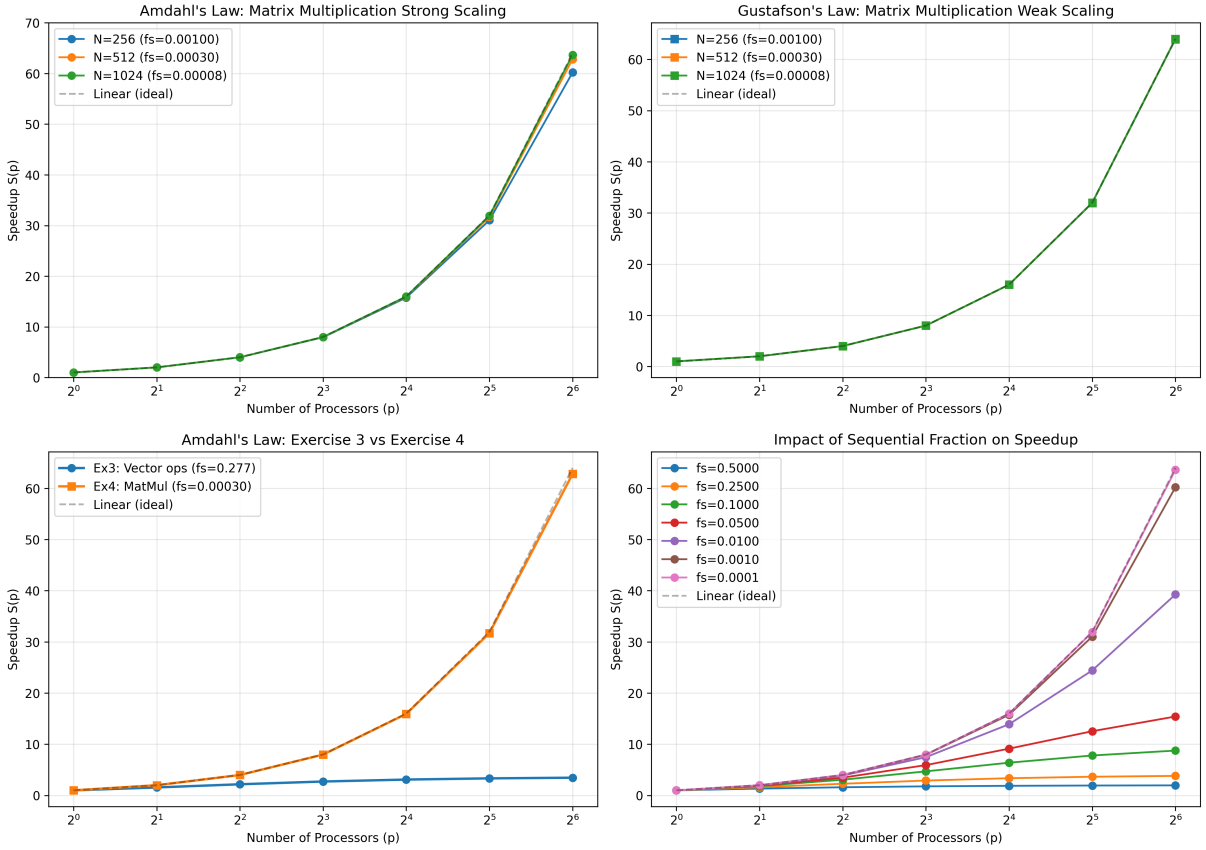
**Exercise 4 (Compute-bound):**

$$\text{Intensity} = \frac{2N^3}{3N^2 \times 8 \text{ bytes}} = \frac{N}{12} \text{ FLOPs/byte} \tag{7}$$

For N=512: Intensity = 42.7 FLOPs/byte (250× higher than Exercise 3)

## 5.8 Visualization

Figure 2 presents the complete scaling analysis for matrix multiplication and comparison with Exercise 3.




scalability.

## 5.9 Conclusion

**Key Finding:** Matrix multiplication achieves *near-perfect parallel scalability* because $O(N^3)$ computational work dominates $O(N)$ sequential overhead. As N increases, $f_s \to 0$, enabling effective use of thousands of processors.

**Fundamental Lesson:** It's not enough for code to be parallelizable - the *ratio of parallel to sequential work* determines scalability. Algorithms with higher computational complexity ($O(N^3)$ vs $O(N)$) scale dramatically better.