TP2 - Foundations of Parallel Computing

Oussama Laaroussi

February 4, 2026

Exercise 1: Loop Optimizations

We analyzed the impact of manual loop unrolling on execution time for summation loops using `int` and `float` types, compiled with `-O0` and `-O2`.

Code Analysis

The loop was unrolled with factors $U = 1, 2, 4, \ldots, 32$.

```
// Example U=4
for (int i = 0; i < N; i+=4)
    sum += a[i] + a[i+1] + a[i+2] + a[i+3];
```

Listing 1: Manual Unrolling Structure

Results

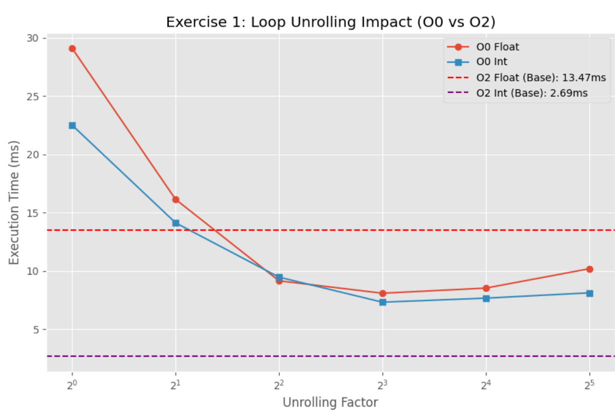

Figure 1: Execution Time vs Unrolling Factor

- **Float (O0):** Manual unrolling significantly reduces time (from $\approx$ 29ms to $\approx$ 9ms at $U = 4$). This is due to reduced loop overhead (fewer comparisons/increments) and better instruction scheduling potential.
- **Int (O0):** Similar trend, with optimized unrolling reaching $\approx$ 7.5ms.
- **Comparison with O2:**
  - For `int`, the compiler optimization (`-O2`) achieves $\approx$ 2.7ms, beating the best manual unrolling.
  - For `float`, curiously, the manual unrolling at `-O0` (9ms) performed slightly better than the baseline `-O2` (13.5ms). This might indicate that the compiler prioritized strict floating-point associativity over aggressive vectorization in this specific simple case.

Exercise 2: Instruction Scheduling

We improved performance by manually breaking dependency chains to allow the CPU pipelined execution units to work in parallel.

Optimization Technique

The original code accumulated results into variables `x` and `y` sequentially. In the optimized version, we compute a common term `res = a * b` and then accumulate into `x` and `y` in a manner that exposes more independent operations for the CPU pipeline.

```c
int main() {
    double a = 1.1, b = 1.2;
    double x = 0.0, y = 0.0;
    clock_t start, end;

    start = clock();
    double res = a * b; // Calculate once

    // Loop unrolling implicitly handled or simple iteration
    for (int i = 0; i < N/4; i++) {
        // Breaking dependencies:
        // Using 'res' allows independent adds if the compiler
        // or hardware can pipeline it.
        x = res + res + res + res + x;
        y = res + res + res + res + y;
    }
    end = clock();
    // ...
}
```

Listing 2: Manually Optimized Version (ex2_manually_optimized.c)

Performance Comparison

- **Original -O0:** 0.231s
- **Original -O2:** 0.103s
- **Manually Optimized:** 0.066s

The manual optimization beats the compiler optimization (`-O2`) because it fundamentally changes the algorithm to reduce the number of multiplications and exposes 4 independent additions per loop iteration.

Exercise 3: Mixed Workload (Scaling)

We profiled a program containing sequential (noise generation, reduction) and parallelizable (initialization, addition) parts.

Profiling Data (Callgrind)

For $N = 10^8$:

- `compute_addition` (Parallel): 33.85%
- `add_noise` (Sequential): 27.69%
- `init_b` (Parallel): 20.00%
- `reduction` (Sequential): 18.46%

Sequential Fraction (f_s): $\approx 27.7\% + 18.5\% = 46.2\%$.

We analyzed a program where the sequential fraction f_s remains constant regardless of problem size.

Data Analysis ($f_s \approx 46\%$)

Since all parts of the algorithm (initialization, noise, addition, reduction) scale linearly as $O(N)$, the ratio between parallel and sequential work remains constant at $f_s \approx 0.46$.

Scaling Analysis

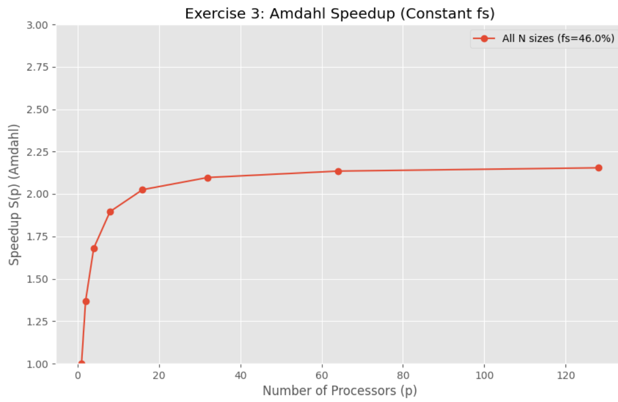

Figure 2: Strong Scaling (Amdahl's Law) for Ex3

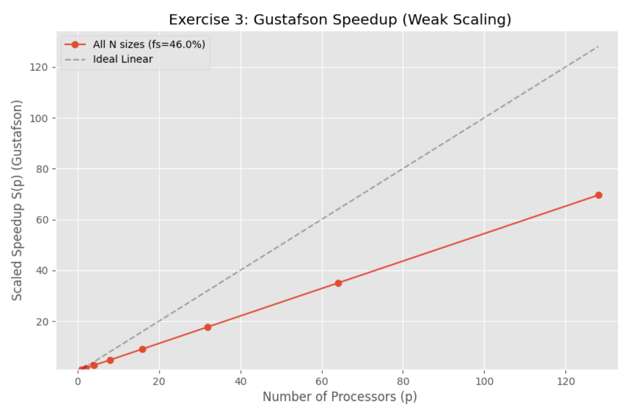

Figure 3: Weak Scaling (Gustafson's Law) for Ex3

Due to $f_s \approx 46\%$, Strong Scaling (Fig 2) is extremely limited ($S_{max} \approx 2.17$). However, Weak Scaling (Fig 3) shows linear growth but with a very shallow slope (slope $= 1 - f_s \approx 0.54$). This means even if we scale the problem size with processors, we only get about 54% efficiency.

Exercise 4: Effect of Problem Size on Parallelism

In this exercise, we observed a dramatic shift in parallel efficiency as the problem size N increased.

Data Analysis

f_s drops dramatically as N increases (36.6% $\rightarrow$ 0.35%), because the parallel part is $O(N^3)$ while sequential parts are $O(N)$.

Scaling Analysis

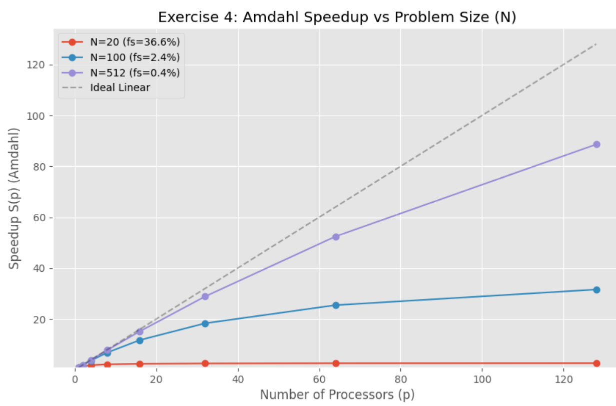

Figure 4: Strong Scaling (Amdahl) for Different N

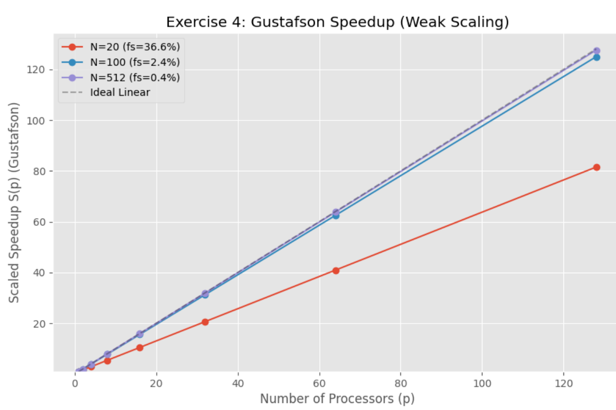

Figure 5: Weak Scaling (Gustafson) for Different N

For large $N = 512$, both Strong Scaling (Fig 4) and Weak Scaling (Fig 5) are near-ideal. The Weak Scaling plot for $N = 512$ overlaps almost perfectly with the ideal linear line, demonstrating that for algorithmically complex tasks, we can maintain high efficiency at scale.