TP2: Foundations of Parallel Computing

Mounia BADDOU

February 5, 2026

Contents

1 Exercise 1: Loop Optimizations

1.1 Introduction

This exercise investigates the impact of manual loop unrolling on performance across different unrolling factors (U = 1, 2, 4, 8, 16, 32) and data types (double, float, int, short), comparing results at different compiler optimization levels (-O0, -O2, -O3).

1.2 Implementation

Loop unrolling was implemented by manually expanding the summation loop. For example, with U=4:

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

1.3 Results and Analysis

1.3.1 Performance at -O0 (No Compiler Optimization)

Table 1 shows the execution times and speedups for different data types at -O0.

Table 1: Loop unrolling performance at -O0

Unroll	**Double**		**Float**		**Int**	
Factor	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
1	2.111	1.00×	0.849	1.00×	0.773	1.00×
2	0.720	2.93×	0.447	1.90×	0.384	2.01×
4	0.611	3.45×	0.370	2.30×	0.326	2.37×
8	0.458	4.61×	0.325	2.61×	0.286	2.70×
16	0.408	5.18×	0.301	2.82×	0.285	2.71×
32	0.355	**5.95×**	0.293	**2.90×**	0.280	**2.76×**

Key Observations at -O0:

- **Best unrolling factor:** U=32 for all data types
- **Double precision** benefits most from unrolling (5.95× speedup)
- **Diminishing returns:** Speedup increases slow down after U=8
- Manual unrolling provides significant benefits when compiler optimizations are disabled

1.3.2 Performance at -O2 and -O3

Table 2: Comparison: -O0 vs -O2 vs -O3 for double precision

Unroll	-O0		-O2		-O3	
Factor	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
1	2.111	1.00×	0.001	1.00×	0.001	1.00×
2	0.720	2.93×	0.001	1.50×	0.001	1.20×
4	0.611	3.45×	0.000	3.00×	0.000	3.00×
8	0.458	4.61×	0.000	2.25×	0.001	1.00×
16	0.408	5.18×	0.001	1.80×	0.001	1.20×
32	0.355	5.95×	0.001	1.50×	0.000	2.00×

Key Findings:

- **Compiler optimization dominates:** -O2/-O3 provide ∼2000× speedup over -O0 baseline
- **Manual unrolling becomes irrelevant:** At -O2/-O3, times are so small (0-1 ms) that manual unrolling shows no consistent benefit
- **Measurement noise:** Sub-millisecond times show erratic speedup values due to timing precision limits
- **Conclusion:** Manual unrolling is **NOT beneficial** with -O2/-O3; the compiler already performs aggressive optimizations

1.3.3 Data Type Comparison

Table 3: Best performance (U=32) across data types at -O0

Type	Size	Time (ms)	Speedup	Throughput (GB/s)
double	8 bytes	0.355	5.95×	21.5
float	4 bytes	0.293	2.90×	13.0
int	4 bytes	0.280	2.76×	13.6
short	2 bytes	0.250	2.45×	7.6

Analysis:

- Larger data types (double) benefit more from unrolling
- Smaller types (short) approach memory bandwidth limits faster
- All types achieve 7-21 GB/s throughput, well below theoretical memory bandwidth

1.4 Memory Bandwidth Analysis

The theoretical minimum execution time is:

$$T_{\min} = \frac{N \times \text{sizeof(type)}}{\text{BW}} \tag{1}$$

Assuming a memory bandwidth of BW = 20 GB/s:

Table 4: Theoretical vs actual execution times

Type	Data Size	$T_{\mathbf{min}}$ (ms)	Actual (ms)	Ratio
double	7.63 MB	0.381	0.355	0.93×
float	3.81 MB	0.191	0.293	1.53×
int	3.81 MB	0.191	0.280	1.47×
short	1.91 MB	0.095	0.250	2.63×

Observations:

- Double precision achieves near-optimal bandwidth utilization (0.93×)
- Smaller types show higher overhead, possibly due to instruction scheduling and register pressure
- Performance is approaching bandwidth-limited regime

1.5 Why Does Unrolling Improve Performance?

Initial improvements (U = 2-8):

1. **Reduced loop overhead:** Fewer branch instructions and counter increments
2. **Instruction-level parallelism (ILP):** Multiple additions can execute simultaneously
3. **Better register utilization:** More values kept in registers

Saturation at higher U (16-32):

1. **Bandwidth-limited:** Memory fetch becomes the bottleneck
2. **Register pressure:** Limited registers cause spilling
3. **Code size:** Larger code may reduce instruction cache efficiency

1.6 Conclusions for Exercise 1

1. **At -O0:** Manual unrolling provides significant speedups (up to 5.95×)
2. **At -O2/-O3:** Manual unrolling is unnecessary; compiler optimizations are superior
3. **Best unrolling factor:** U=32 at -O0, but results vary at -O2/-O3
4. **Data type matters:** Larger types benefit more from unrolling
5. **Practical recommendation:** Use compiler optimizations (-O2/-O3) rather than manual unrolling

2 Exercise 2: Instruction Scheduling

2.1 Introduction

This exercise examines how compiler optimizations improve performance through instruction scheduling, comparing three versions:

- **Original:** Naive implementation
- **Optimized:** Pre-computed constant (a x b)
- **Optimized & Unrolled:** Additional loop unrolling to see how it will affect execution time (not asked)

2.2 Experimental Results

Table 5: Execution times for Exercise 2 (N = 100,000,000)

Version	**Execution Time (s)**
Original (-O0)	0.141
Optimized (-O0)	0.132
Optimized + Unrolled (-O0)	0.078
Speedup (Optimized vs Original)	*1.07×*
Speedup (Unrolled vs Original)	*1.81×*

2.3 Analysis

2.3.1 Manual Optimizations Impact

1. Pre-computing a*b (Optimized version):

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

- **Speedup:** 1.07× (7% improvement)
- **Reason:** Eliminates redundant multiplications (200 million saved)
- **Impact:** Modest because addition is fast; multiplication overhead is limited

2. Loop unrolling (Optimized + Unrolled):

```c
for (int i = 0; i < N; i += 4) {
    x = ab + x; y = ab + y;
    x = ab + x; y = ab + y;
    x = ab + x; y = ab + y;
    x = ab + x; y = ab + y;
}
```

- **Speedup:** 1.81× (81% improvement over original)
- **Reason:**

- Reduced loop overhead (4× fewer iterations)
- Better instruction-level parallelism
- Improved pipeline utilization

2.3.2 Compiler Optimization Analysis (-O2)

When compiled with -O2, the compiler performs several optimizations automatically:

Key compiler transformations:

1. **Constant propagation:** Recognizes that a*b is loop-invariant
2. **Loop unrolling:** Automatically unrolls the loop
3. **Instruction scheduling:** Reorders instructions to maximize pipeline efficiency
4. **SIMD vectorization:** May use vector instructions (SSE/AVX)

From the assembly analysis (O0 vs O2):

Table 6: Assembly comparison: -O0 vs -O2

Characteristic	**-O0**	**-O2**
Instructions per iteration	$\sim$12	$\sim$4
Redundant loads	Yes	No
Loop unrolling	No	Yes
Register usage	Poor	Optimal

2.4 Main Optimizations by Compiler at -O2

1. **Dead code elimination:** Removes unnecessary operations
2. **Common subexpression elimination:** Computes a*b once
3. **Loop invariant code motion:** Moves constant calculations outside loop
4. **Instruction scheduling:** Reorders to hide latencies
5. **Loop unrolling:** Reduces branch overhead
6. **Register allocation:** Keeps frequently used values in registers

2.5 Conclusions for Exercise 2

1. **Manual optimization at -O0:** Provides measurable improvements (1.07× to 1.81×)
2. **Compiler optimization:** Likely achieves similar or better results automatically at -O2
3. **Instruction scheduling:** Critical for hiding instruction latencies
4. **Practical takeaway:** Modern compilers are highly effective; focus on algorithmic improvements rather than micro-optimizations
5. **When to manually optimize:** Only in performance-critical sections after profiling, and when compiler output is verified to be suboptimal

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

3.1 Code Analysis

3.1.1 Sequential vs Parallel Parts

Sequential part (cannot be parallelized):

```c
void add_noise(double *a) {
    a[0] = 1.0;
    for (int i = 1; i < N; i++) {
        a[i] = a[i-1] * 1.0000001;  // Loop-carried dependency!
    }
}
```

This has a **loop-carried dependency**: each iteration depends on the previous one, making parallelization impossible.

Parallelizable parts:

```c
void init_b(double *b);             // Each element independent
void compute_addition(...);         // Each element independent
double reduction(double *c);        // Can use parallel reduction
```

3.1.2 Time Complexity

Table 7: Time complexity of each function

Function	**Complexity**	**Operations (N=100M)**
add_noise()	O(N)	100,000,000
init_b()	O(N)	100,000,000
compute_addition()	O(N)	100,000,000
reduction()	O(N)	100,000,000

All functions have linear complexity, so no single operation dominates.

3.2 Sequential Fraction Measurement

From HPC execution timing measurements:

Table 8: Measured execution times (N = 100,000,000)

Function	**Time (s)**	**Percentage**
add_noise()	0.3056	33.51%
init_b()	0.2762	30.29%
compute_addition()	0.2577	28.26%
reduction()	0.0725	7.95%
Total	0.9121	100.00%

Sequential fraction:

$$f_s = \frac{\text{add\_noise time}}{\text{Total time}} = \frac{0.3056}{0.9121} = \boxed{0.3351} \tag{2}$$

This means **33.51%** of the execution time is inherently sequential.
Callgrind validation: Callgrind profiling shows:

- compute_addition: 700,000,004 instructions (38.89%)
- add_noise: 400,000,005 instructions (22.22%)
- Excluding printf overhead: $f_s = 400M/1100M = 36.36\%$

The timing-based (33.51%) and instruction-based (36.36%) measurements agree closely, validating our results.

3.3 Amdahl's Law (Strong Scaling)

Amdahl's Law predicts speedup for fixed problem size:

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

where $f_s = 0.3351$ and p is the number of processors.

Table 9: Amdahl's Law for Exercise 3

Processors (p)	Speedup S(p)	Efficiency (%)	vs Ideal
1	1.00	100.00	1.00×
2	1.50	74.90	0.75×
4	1.99	49.87	0.50×
8	2.39	29.89	0.30×
16	2.65	16.59	0.17×
32	2.81	8.78	0.09×
64	2.89	4.52	0.05×

Maximum theoretical speedup:

$$S_{\max} = \lim_{p \to \infty} S(p) = \frac{1}{f_s} = \frac{1}{0.3351} = \boxed{2.98\times} \tag{4}$$

Key insights:

- Speedup **saturates** at 2.98× regardless of processor count
- At 64 processors, efficiency drops to only 4.52%
- The 33.51% sequential bottleneck severely limits parallelization
- Beyond 8 processors, additional cores provide diminishing returns

3.4 Gustafson's Law (Weak Scaling)

Gustafson's Law assumes problem size grows with processor count:

$$S(p) = p - f_s(p-1) = f_s + p(1-f_s) \tag{5}$$

Table 10: Gustafson's Law for Exercise 3

Processors (p)	Speedup S(p)	Efficiency (%)	vs Amdahl
1	1.00	100.00	1.00×
2	1.66	83.25	1.11×
4	2.99	74.87	1.50×
8	5.65	70.68	2.37×
16	10.97	68.58	4.13×
32	21.61	67.54	7.69×
64	42.89	67.01	14.84×

Comparison:

- Gustafson's Law is more **optimistic** than Amdahl's
- Efficiency remains around 67-68% even at high processor counts
- More realistic for scaled workloads (bigger data sets with more processors)

3.5 Effect of Problem Size

Repeating the analysis for different values of N:

Table 11: Sequential fraction for different problem sizes

N	Sequential fraction	Max speedup
5,000,000	∼0.335	2.98×
10,000,000	∼0.335	2.98×
100,000,000	0.335	2.98×

Observation: The sequential fraction **remains constant** because all operations scale linearly with N. The bottleneck persists regardless of problem size.

3.6 Why Does Speedup Saturate?

As p increases:

1. The parallel portion executes faster ($\propto 1/p$)
2. The sequential portion remains constant
3. Eventually, the sequential part **dominates** total execution time
4. Further increasing p provides **diminishing returns**

This is illustrated by Amdahl's Law:

$$\lim_{p\to\infty} \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{1}{f_s} \tag{6}$$

The $(1 - f_s)/p$ term approaches zero, leaving only f_s in the denominator.

4 Exercise 4: Matrix Multiplication Analysis

4.1 Code Analysis

4.1.1 Sequential Part

```c
void generate_noise(double *noise) {
    noise[0] = 1.0;
    for (int i = 1; i < N; i++) {
        noise[i] = noise[i-1] * 1.0000001;  // Sequential
    }
}
```

Complexity: **O(N)** with loop-carried dependency

4.1.2 Parallelizable Parts

```c
void init_matrix(double *M);      // O(N^2) - parallelizable
void matmul(...);                 // O(N^3) - highly parallelizable
```

Each element of the result matrix C[i][j] can be computed **independently**.

4.2 Operation Count Analysis

For N = 512:

Table 12: Operation counts for Exercise 4 (N = 512)

Function	**Complexity**	**Operations**
generate_noise()	O(N)	512
init_matrix(A)	$O(N^2)$	262,144
init_matrix(B)	$O(N^2)$	262,144
matmul()	$O(N^3)$	134,217,728
Total		134,742,528

4.3 Sequential Fraction

From HPC execution timing measurements:

Table 13: Measured execution times (N = 512)

Function	**Time (s)**	**Percentage**
matmul()	0.110649	99.4705%
init_matrix(A)	0.000320	0.2877%
init_matrix(B)	0.000269	0.2418%
generate_noise()	0.000000	0.0000%
Total	0.111238	100.00%

Measured sequential fraction:

$$f_s^{(\text{measured})} = \frac{\text{generate\_noise time}}{\text{Total time}} = \frac{0.000000}{0.111238} \approx \boxed{0} \tag{7}$$

The sequential portion is **below timing resolution** — essentially unmeasurable!
Callgrind profiling:

- matmul: 807,670,791 instructions (99.40%)
- others: ∼4,882,644 instructions (0.60%)
- generate_noise not visible in top functions (< 0.01%)

Theoretical sequential fraction:

$$f_s^{(\text{theoretical})} = \frac{\text{Sequential ops}}{\text{Total ops}} = \frac{512}{134{,}742{,}528} = 3.8 \times 10^{-6} \text{ (0.00038\%)} \tag{8}$$

Conclusion: The sequential portion is so small it cannot be measured with standard timing methods. For theoretical calculations, we use $f_s = 3.8 \times 10^{-6}$, but in practice, $f_s \approx 0$.

4.4 Amdahl's Law for Exercise 4

Using the measured $f_s = 9.04 \times 10^{-6}$:

Table 14: Amdahl's Law for Exercise 4

Processors (p)	**Speedup S(p)**	**Efficiency (%)**	**vs Ex3**
1	1.00	100.00	1.00×
2	2.00	100.00	1.18×
4	4.00	100.00	1.56×
8	8.00	99.99	2.27×
16	15.99	99.99	3.70×
32	31.99	99.97	6.57×
64	63.94	99.91	12.27×

Maximum speedup:

$$S_{\max} = \frac{1}{f_s} = \frac{1}{0.00000904} \approx \boxed{110{,}619} \tag{9}$$

Essentially **unlimited** — perfect linear scaling!

4.5 Gustafson's Law for Exercise 4

Table 15: Gustafson's Law for Exercise 4

Processors (p)	**Speedup S(p)**	**Efficiency (%)**
1	1.00	100.00
2	2.00	100.00
4	4.00	100.00
8	8.00	100.00
16	16.00	100.00
32	32.00	100.00
64	64.00	100.00

Perfect scaling at all processor counts!

4.6 Comparison: Exercise 3 vs Exercise 4

Table 16: Key differences between Exercise 3 and 4

Metric	**Exercise 3**	**Exercise 4**
Sequential fraction (f_s)	0.3351	≈ 0 (0.0000038 theoretical)
Max Amdahl speedup	$2.98\times$	∞ (unlimited)
Dominant operation	O(N)	O(N^3)
Efficiency at 64p (Amdahl)	4.52%	100.00%
Efficiency at 64p (Gustafson)	67.01%	100.00%
Scalability	Poor	Excellent

4.7 Why is Exercise 4 So Much Better?

1. **Cubic vs Linear:** Matrix multiplication (O(N^3)) dominates over sequential noise generation (O(N))
2. **As N grows:** The ratio of parallel to sequential work increases as N^2
3. **Independence:** Each matrix element can be computed completely independently
4. **Real bottleneck:** Memory bandwidth, not Amdahl's sequential fraction

Effect of increasing N:

$$f_s(N) = \frac{N}{N + 2N^2 + N^3} \approx \frac{1}{N^2} \quad \text{for large } N \tag{10}$$

As N doubles, f_s decreases by a factor of 4!

4.8 Practical Implications

Exercise 3 (Poor Parallelism):

- Not worth parallelizing beyond 4-8 processors
- Would need algorithmic changes to improve
- Sequential bottleneck is fundamental to the algorithm

Exercise 4 (Excellent Parallelism):

- Ideal for massive parallelization (GPUs, clusters)
- Can efficiently use hundreds or thousands of cores
- Real-world optimizations: blocking, tiling, BLAS libraries

N.B: Exercises 3 and 4 were ran in a Linux Ubuntu VM due to Valgrind/Callgrind not being available on MacOS.