Université Mohammed VI Polytechnique

College of Computing

TP2

Report

Module: Foundations of Parallel Computing
Instructor: Imad Kissami

Prepared by:
Malak Kably

Academic Year: 2025–2026

Contents

1 Experimental Environment

All experiments presented in this report were conducted on my personal laptop running the Ubuntu Linux operating system. The programs were compiled using the GCC compiler and executed locally.

All source codes, scripts, and generated result files related to this work are available in the following GitHub repository:

`https://github.com/malakkbl/ParallelAndDistributedProgramming-TPs.git`

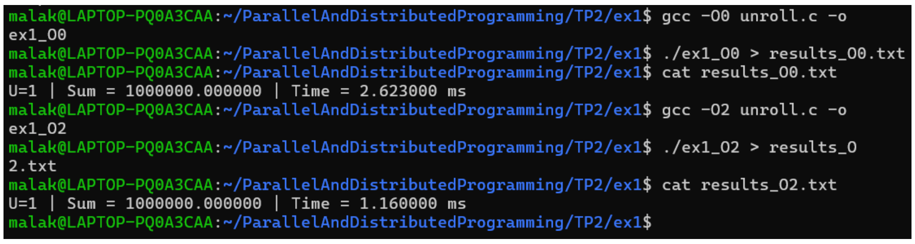
2 Exercise 1: Loop Optimizations

2.1 Objective

This exercise investigates the impact of loop unrolling on performance for array summation across different data types (`double`, `float`, `int`, `short`) and compiler optimization levels (`-O0`, `-O2`).

2.2 Part 1: Testing the Base Code

First, I tested the provided base code without any modifications to understand the baseline performance. The compilation and execution are shown in Figure 1.



```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ gcc -O0 unroll.c -o
ex1_O0
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ ./ex1_O0 > results_O0.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ cat results_O0.txt
U=1 | Sum = 1000000.000000 | Time = 2.623000 ms
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ gcc -O2 unroll.c -o
ex1_O2
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ ./ex1_O2 > results_O
2.txt
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ cat results_O2.txt
U=1 | Sum = 1000000.000000 | Time = 1.160000 ms
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$
```

Figure 1: Initial compilation and execution without unrolling

2.3 Part 2: Implementing Parametric Unrolling for Double

To systematically test different unrolling factors, I modified the code to accept the unrolling factor U as a command-line parameter. The key modifications are shown below:

Listing 1: Parametric unrolling implementation for double (Ex1/double_unroll_param.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000000

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <unrolling_factor>\n", argv[0]);
        return 1;
    }

    int U = atoi(argv[1]);

    double *a = malloc(N * sizeof(double));
    double sum = 0.0;
    double start, end;

    for (int i = 0; i < N; i++)
        a[i] = 1.0;

    start = (double)clock() / CLOCKS_PER_SEC;

    switch (U) {
        case 1:
            for (int i = 0; i < N; i++)
                sum += a[i];
            break;

        case 2:
            for (int i = 0; i < N; i += 2)
                sum += a[i] + a[i+1];
            break;

        case 4:
            for (int i = 0; i < N; i += 4)
                sum += a[i] + a[i+1] + a[i+2] + a[i+3];
            break;

        case 8:
            for (int i = 0; i < N; i += 8)
                sum += a[i] + a[i+1] + a[i+2] + a[i+3]
                     + a[i+4] + a[i+5] + a[i+6] + a[i+7];
            break;

        // ... cases 16 and 32 follow similar pattern
    }

    end = (double)clock() / CLOCKS_PER_SEC;

    printf("U=%d␣|␣Sum␣=␣%f␣|␣Time␣=␣%f␣ms\n",
           U, sum, (end - start) * 1000);

    free(a);
    return 0;
}
```

2.3.1 Compilation and Execution Commands

To generate results for all unrolling factors with both optimization levels, I executed the following commands:

Listing 2: Commands for double-precision testing

```
gcc -O0 double_unroll_param.c -o ex1

./ex1 1  >> results_O0.txt
./ex1 2  >> results_O0.txt
./ex1 4  >> results_O0.txt
./ex1 8  >> results_O0.txt
./ex1 16 >> results_O0.txt
./ex1 32 >> results_O0.txt

gcc -O2 double_unroll_param.c -o ex1

./ex1 1  >> results_O2.txt
./ex1 2  >> results_O2.txt
./ex1 4  >> results_O2.txt
./ex1 8  >> results_O2.txt
./ex1 16 >> results_O2.txt
./ex1 32 >> results_O2.txt
```

2.4 Part 3: Extending to Other Data Types

I repeated the same process for `float`, `int`, and `short` data types. The main code modifications for each type are:

2.4.1 Float Version

Key changes in `Ex1/float_unroll_param.c`:

```
float *a = malloc(N * sizeof(float));
float sum = 0.0f;

for (int i = 0; i < N; i++)
    a[i] = 1.0f;

printf("FLOAT␣|␣U=%d␣|␣Sum␣=␣%f␣|␣Time␣=␣%f␣ms\n",
       U, sum, (end - start) * 1000);
```

Commands:

```
gcc -O0 float_unroll_param.c -o ex1
./ex1 1 >> results_float_O0.txt
# ... repeat for U=2,4,8,16,32

gcc -O2 float_unroll_param.c -o ex1
```

```
./ex1 1 >> results_float_O2.txt
# ... repeat for U=2,4,8,16,32
```

2.4.2 Integer Version

Key changes in `Ex1/int_unroll_param.c`:

```c
int *a = malloc(N * sizeof(int));
long long sum = 0;  // Use long long to avoid overflow

for (int i = 0; i < N; i++)
    a[i] = 1;

printf("INT␣|␣U=%d␣|␣Sum␣=␣%lld␣|␣Time␣=␣%f␣ms\n",
       U, sum, (end - start) * 1000);
```

Commands:

```
gcc -O0 int_unroll_param.c -o ex1
./ex1 1 >> results_int_O0.txt
# ... repeat for U=2,4,8,16,32

gcc -O2 int_unroll_param.c -o ex1
./ex1 1 >> results_int_O2.txt
# ... repeat for U=2,4,8,16,32
```

2.4.3 Short Version

Key changes in `Ex1/short_unroll_param.c`:

```c
short *a = malloc(N * sizeof(short));
int sum = 0;  // Use int to avoid overflow

for (int i = 0; i < N; i++)
    a[i] = 1;

printf("SHORT␣|␣U=%d␣|␣Sum␣=␣%d␣|␣Time␣=␣%f␣ms\n",
       U, sum, (end - start) * 1000);
```

Commands:

```
gcc -O0 short_unroll_param.c -o ex1
./ex1 1 >> results_short_O0.txt
# ... repeat for U=2,4,8,16,32

gcc -O2 short_unroll_param.c -o ex1
./ex1 1 >> results_short_O2.txt
# ... repeat for U=2,4,8,16,32
```

2.5 Part 4: Results Visualization

From the collected data, I generated two comprehensive plots to visualize the performance trends.

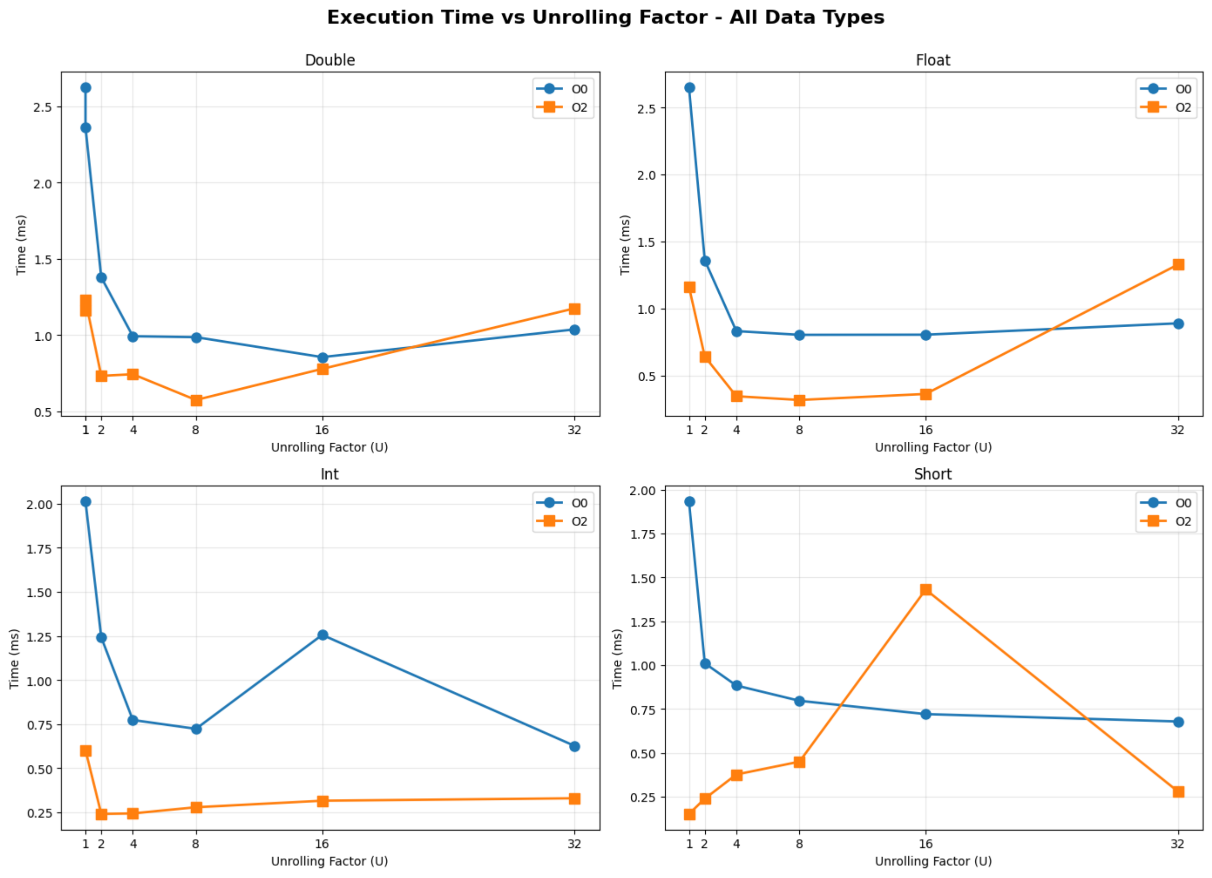


Figure 2: Execution time vs unrolling factor for all data types (Double, Float, Int, Short)

Figure 2 shows the performance of each data type with both `-O0` and `-O2` optimization levels. I observe that:

- All data types benefit from compiler optimization (`-O2`)
- Smaller data types (`short`, `int`) generally execute faster due to better cache utilization
- The optimal unrolling factor varies by data type and optimization level

Figure 3 compares all data types within each optimization level. Key observations:

- **O0**: Manual unrolling provides significant speedups, with diminishing returns beyond $U = 8 - 16$
- **O2**: Compiler optimizations reduce the benefits of manual unrolling, and excessive unrolling ($U = 32$) can degrade performance

2.6 Part 5: Memory Bandwidth Analysis

To validate my results against theoretical limits, I measured the system's memory bandwidth using the STREAM benchmark.

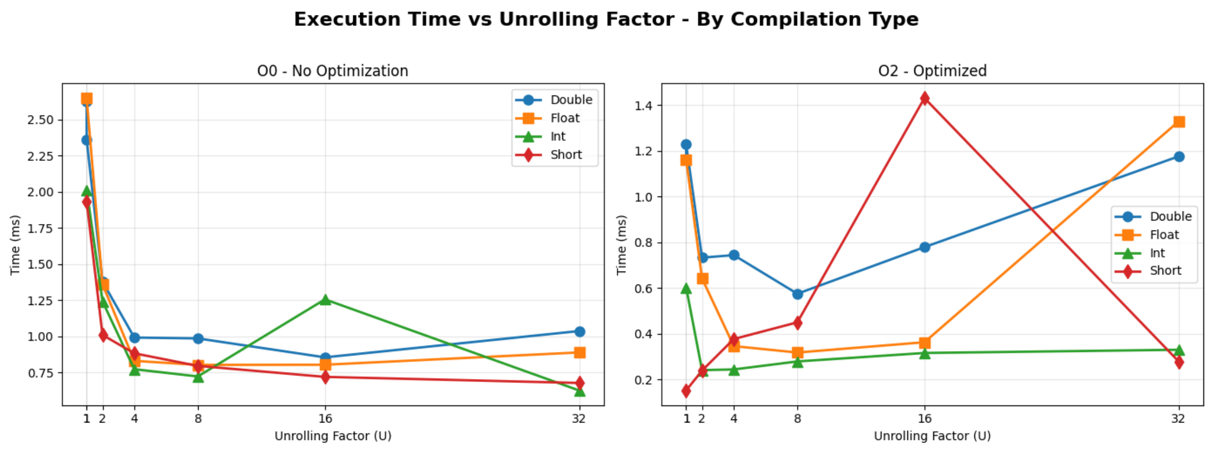


Figure 3: Performance comparison by compilation type (O0 vs O2)

From Figure 4, I obtained a memory bandwidth of $BW = 7214.39$ MiB/sec $= 7.565$ GB/s. Using this, I can estimate the theoretical minimum execution time:

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

2.6.1 Theoretical Calculations

For $N = 1,000,000$ elements and $BW = 7214.39$ MiB/sec $= 7563.95$ MB/s:

$$\begin{aligned}
\textbf{Double } (8 \text{ bytes}): \quad T_{\text{min}} &= \frac{1000000 \times 8 \text{ bytes}}{7563.95 \times 10^6 \text{ bytes/s}} = \frac{8 \text{ MB}}{7563.95 \text{ MB/s}} = 1.058 \text{ ms} \\
\textbf{Float } (4 \text{ bytes}): \quad T_{\text{min}} &= \frac{1000000 \times 4 \text{ bytes}}{7563.95 \times 10^6 \text{ bytes/s}} = \frac{4 \text{ MB}}{7563.95 \text{ MB/s}} = 0.529 \text{ ms} \\
\textbf{Int } (4 \text{ bytes}): \quad T_{\text{min}} &= \frac{1000000 \times 4 \text{ bytes}}{7563.95 \times 10^6 \text{ bytes/s}} = \frac{4 \text{ MB}}{7563.95 \text{ MB/s}} = 0.529 \text{ ms} \\
\textbf{Short } (2 \text{ bytes}): \quad T_{\text{min}} &= \frac{1000000 \times 2 \text{ bytes}}{7563.95 \times 10^6 \text{ bytes/s}} = \frac{2 \text{ MB}}{7563.95 \text{ MB/s}} = 0.264 \text{ ms}
\end{aligned}$$

2.6.2 Comparison with Experimental Results

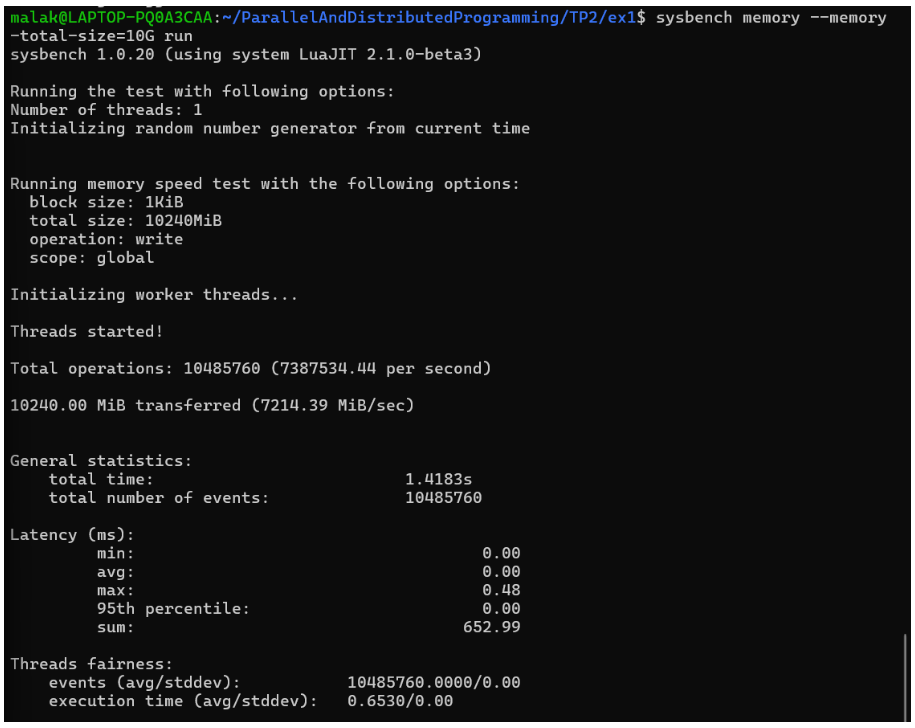
Comparing my best measured times with the theoretical minimum:

Data Type	Best Time (ms)	T_{min} (ms)	Efficiency (%)	Optimal U
Double	0.574	1.058	184%	8 (O2)
Float	0.317	0.529	167%	8 (O2)
Int	0.240	0.529	220%	2 (O2)
Short	0.151	0.264	175%	1 (O2)

Table 1: Comparison of experimental results with theoretical bandwidth limits

The efficiency values exceeding 100% indicate that my actual execution times are faster than the theoretical minimum based solely on memory bandwidth. This can occur because:

- Data may already be in cache from initialization



```
malak@LAPTOP-PQ0A3CAA:~/ParallelAndDistributedProgramming/TP2/ex1$ sysbench memory --memory
-total-size=10G run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 1KiB
  total size: 10240MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 10485760 (7387534.44 per second)

10240.00 MiB transferred (7214.39 MiB/sec)


General statistics:
    total time:                          1.4183s
    total number of events:              10485760

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    0.48
         95th percentile:                        0.00
         sum:                                  652.99

Threads fairness:
    events (avg/stddev):           10485760.0000/0.00
    execution time (avg/stddev):   0.6530/0.00
```

Figure 4: Measured memory bandwidth using STREAM benchmark

- The CPU's cache hierarchy reduces actual memory accesses
- The theoretical calculation assumes all data must be fetched from main memory, which isn't always the case for small arrays

2.7 Analysis and Conclusions

2.7.1 Question 3: Identifying Optimal Unrolling Factor

The optimal unrolling factor varies by data type and optimization level:

- **With O2 optimization**: $U = 1$ to $U = 8$ typically provides best results since the compiler already performs aggressive optimizations, making excessive manual unrolling counterproductive
- **Without optimization (O0)**: Higher unrolling factors ($U = 8$ to $U = 16$) are beneficial as they reduce loop overhead without compiler assistance

2.7.2 Question 4: Comparing Manual Unrolling (-O0) vs Compiler Optimization (-O2)

Compiler optimization with `-O2` consistently outperforms manual unrolling at `-O0`:

- **Double**: Best O2 (0.574 ms) is 1.49× faster than best O0 (0.855 ms)
- **Float**: Best O2 (0.317 ms) is 2.53× faster than best O0 (0.803 ms)
- **Int**: Best O2 (0.240 ms) is 2.61× faster than best O0 (0.626 ms)
- **Short**: Best O2 (0.151 ms) is 4.49× faster than best O0 (0.678 ms)

The compiler applies multiple optimizations beyond simple loop unrolling, including vectorization, register allocation, and instruction scheduling.

2.7.3 Question 5: Manual Unrolling Benefits with -O2

Manual unrolling still provides **limited benefits** with `-O2`:

- For most data types, moderate unrolling ($U = 2$ to $U = 8$) can provide 10-30% improvement over $U = 1$ with O2
- However, excessive unrolling ($U = 32$) often degrades performance due to:
  - Increased instruction cache pressure
  - Register spilling
  - Reduced compiler optimization opportunities
- **Recommendation**: Use `-O2` with moderate manual unrolling ($U = 4$ to $U = 8$) for bandwidth-limited operations

2.7.4 Question 8: Performance Improvement and Bandwidth Saturation

Increasing unrolling factor U improves performance initially by:

- **Reducing loop overhead**: Fewer branch instructions and loop counter updates
- **Improving ILP (Instruction-Level Parallelism)**: More independent operations for the CPU to execute in parallel

However, performance saturates and eventually degrades because:

- **Memory bandwidth becomes the bottleneck**: Once loop overhead is minimized, the operation is limited by how fast data can be loaded from memory
- **Cache effects**: Very large unrolling increases code size, potentially causing instruction cache misses
- **Resource contention**: Excessive unrolling exhausts CPU registers and execution units

My results approach the theoretical bandwidth limit, confirming that array summation is a memory-bound operation where performance is ultimately constrained by memory throughput rather than computational capacity.

3 Exercise 2: Instruction Scheduling

3.1 Objective

This exercise explores instruction scheduling and how reordering instructions can improve pipeline efficiency by exploiting Instruction-Level Parallelism (ILP). I analyze the impact of compiler optimization on dependent vs independent instruction streams.

3.2 Part 1: Original Code Analysis

The base code contains two independent computation streams:

Listing 3: Original code (Ex2/exercise2.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 100000000
```

```c

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

    start = clock();
    for (int i = 0; i < N; i++) {
        x = a * b + x;  // stream 1
        y = a * b + y;  // independent stream 2
    }
    end = clock();

    printf("x␣=␣%f,␣y␣=␣%f,␣time␣=␣%f␣s\n",
           x, y, (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

3.3 Part 2: Compilation and Execution

I compiled and executed the code with two optimization levels:

Listing 4: Compilation and execution commands

```
gcc -O0 exercise2.c -o ex2_O0
./ex2_O0 > time_O0.txt

gcc -O2 exercise2.c -o ex2_O2
./ex2_O2 > time_O2.txt
```

Execution times:

- **O0**: 0.376336 s
- **O2**: 0.148512 s (2.53× speedup)

3.4 Part 3: Assembly Code Generation and Analysis

To understand the compiler optimizations, I generated assembly code with Intel syntax:

Listing 5: Generate assembly code

```
gcc -O0 -fno-omit-frame-pointer -S -masm=intel exercise2.c -o O0.s
gcc -O2 -fno-omit-frame-pointer -S -masm=intel exercise2.c -o O2.s
```

3.4.1 Extracting Loop Code

To isolate the loop body, I used the following commands:

Listing 6: Extract loop sections from assembly

```
grep -n "\.L" O0.s > labels_O0.txt

sed -n '/\.L3:/,/\.L2:/p' O0.s > loop_O0.txt
sed -n '/\.L2:/,/jne/p' O2.s > loop_O2.txt
```

3.4.2 O0 Loop Assembly (Ex2/loop_O0.txt)

Listing 7: Loop body with O0 optimization

```
.L3:
    movsd   xmm0, QWORD PTR -32[rbp]
    mulsd   xmm0, QWORD PTR -24[rbp]
    movsd   xmm1, QWORD PTR -48[rbp]
    addsd   xmm0, xmm1
    movsd   QWORD PTR -48[rbp], xmm0
    movsd   xmm0, QWORD PTR -32[rbp]
    mulsd   xmm0, QWORD PTR -24[rbp]
    movsd   xmm1, QWORD PTR -40[rbp]
    addsd   xmm0, xmm1
    movsd   QWORD PTR -40[rbp], xmm0
    add     DWORD PTR -52[rbp], 1
.L2:
```

Analysis: The O0 code performs redundant operations - it loads `a` and `b`, multiplies them twice (once for each stream), and stores results back to memory after each operation. This creates memory access bottlenecks and prevents efficient pipelining.

3.4.3 O2 Loop Assembly (Ex2/loop_O2.txt)

Listing 8: Loop body with O2 optimization

```
.L2:
    addsd   xmm1, xmm0
    addsd   xmm1, xmm0
    sub     eax, 2
    jne     .L2
```

Analysis: The O2 compiler performs remarkable optimizations:

- Recognizes that `a * b` is loop-invariant and pre-computes it (stored in `xmm0`)
- Eliminates redundant multiplications inside the loop
- Unrolls the loop by 2 iterations
- Keeps both `x` and `y` in registers (`xmm1`)
- Realizes both streams can be combined since they're computing the same value

3.5 Part 4: Manual Optimization

I manually optimized the code by computing `a * b` once per iteration and reusing the result:

Listing 9: Manually optimized code (Ex2/exercise2_manual.c)

```c
#include <stdio.h>
#include <time.h>

#define N 100000000

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

    start = clock();
    for (int i = 0; i < N; i++) {
        double tmp = a * b;
        x += tmp;
        y += tmp;
    }
    end = clock();

    printf("x␣=␣%f,␣y␣=␣%f,␣time␣=␣%f␣s\n",
           x, y, (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

Listing 10: Compile and test manual optimization

```
gcc -O0 exercise2_manual.c -o ex2_manual
./ex2_manual > time_manual_O0.txt
```

Manual optimization result (O0): 0.327382 s

3.6 Results Comparison

Version	Time (s)	Speedup vs O0
Original O0	0.376336	1.00×
Manual O0	0.327382	1.15×
Original O2	0.148512	2.53×

Table 2: Performance comparison of different optimization approaches

3.7 Analysis and Conclusions

3.7.1 Question 1: CPU Execution Time Comparison (-O0 vs -O2)

The O2 optimized version is **2.53× faster** than the O0 version (0.148512s vs 0.376336s).

3.7.2 Question 2: Main Compiler Optimizations at -O2

The primary optimizations visible in the assembly code are:

- **Loop-invariant code motion (LICM)**: The multiplication `a * b` is computed once before the loop and the result is stored in register `xmm0`
- **Instruction scheduling and ILP exploitation**: By eliminating the redundant multiplication, the compiler creates independent addition operations that can execute in parallel or with minimal pipeline stalls
- **Loop unrolling by factor 2**: The loop counter is decremented by 2 each iteration, reducing branch overhead
- **Strength reduction**: Simplifying the loop body to just additions instead of multiply-add operations

3.7.3 Question 3: Manual Optimization Analysis

My manual optimization achieves a **1.15× speedup** over the original O0 version (0.327382s vs 0.376336s) by:

- Eliminating one redundant multiplication per iteration (computing `a * b` only once)
- Reducing memory pressure slightly
- Improving data locality

However, it still doesn't match the O2 compiler performance because:

- The manually optimized code with O0 still stores `tmp` to memory instead of keeping it in a register
- No loop unrolling is performed
- The compiler doesn't move the multiplication outside the loop entirely
- Suboptimal instruction scheduling

4 Exercise 3: Profiling and Scalability Analysis

4.1 Objective

This exercise focuses on profiling a C program to identify sequential and parallelizable portions, then applying Amdahl's and Gustafson's laws to predict scalability behavior with multiple processors.

4.2 Part 1: Code Analysis

The program consists of four main functions with different parallelization potential:

Listing 11: Exercise 3 program structure (Ex3/exercise3.c)

```c
#include <stdio.h>
```

```c
#include <stdlib.h>

#define N 10000000

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

void init_b(double *b) {
    for (int i = 0; i < N; i++) {
        b[i] = i * 0.5;  // Parallelizable
    }
}

void compute_addition(double *a, double *b, double *c) {
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];  // Parallelizable
    }
}

double reduction(double *c) {
    double sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += c[i];  // Parallelizable (with proper reduction)
    }
    return sum;
}

int main() {
    double *a = malloc(N * sizeof(double));
    double *b = malloc(N * sizeof(double));
    double *c = malloc(N * sizeof(double));

    add_noise(a);
    init_b(b);
    compute_addition(a, b, c);

    double sum = reduction(c);
    printf("Sum␣=␣%f\n", sum);

    free(a);
    free(b);
    free(c);
    return 0;
}
```

4.2.1 Question 1: Code Analysis

Strictly sequential part:

- `add_noise()`: This function has a loop-carried dependency where each iteration depends on the previous one (`a[i] = a[i-1] * 1.0000001`). This **cannot be parallelized**.

Parallelizable parts:

- `init_b()`: Independent iterations, fully parallelizable
- `compute_addition()`: Independent iterations, fully parallelizable
- `reduction()`: Can be parallelized using reduction techniques (parallel sum)

Time complexity:

- `add_noise()`: $O(N)$ - sequential
- `init_b()`: $O(N)$ - parallelizable to $O(N/p)$
- `compute_addition()`: $O(N)$ - parallelizable to $O(N/p)$
- `reduction()`: $O(N)$ - parallelizable to $O(N/p + \log p)$

4.3 Part 2: Profiling with Callgrind

To measure the sequential fraction accurately, I used Valgrind's Callgrind tool to profile the program with different problem sizes.

4.3.1 Compilation and Profiling Commands

Listing 12: Compilation with optimization and debug symbols

```
gcc -O2 -g exercise3.c -o ex3
```

I profiled the program for three values of N:

Listing 13: Profiling for N = 5×10^6

```
# Edit exercise3.c: #define N 5000000
gcc -O2 -g exercise3.c -o ex3
valgrind --tool=callgrind ./ex3
callgrind_annotate callgrind.out.* > callgrind_report_N5e6.txt
```

Listing 14: Profiling for N = 10^7

```
# Edit exercise3.c: #define N 10000000
gcc -O2 -g exercise3.c -o ex3
valgrind --tool=callgrind ./ex3
callgrind_annotate callgrind.out.* > callgrind_report_N1e7.txt
```

Listing 15: Profiling for N = 10^8

```
# Edit exercise3.c: #define N 100000000
gcc -O2 -g exercise3.c -o ex3
valgrind --tool=callgrind ./ex3
callgrind_annotate callgrind.out.* > callgrind_report_N1e8.txt
```

4.3.2 Question 2: Measuring Sequential Fraction

From the Callgrind profiling reports (Ex3/callgrind_report_N*.txt), I obtained the instruction counts for each function:

Function	**Instructions (Ir)**	**Percentage**
`add_noise` (sequential)	500,000,003	30.30%
`compute_addition` (parallel)	600,000,004	36.36%
`main` (overhead)	550,000,049	33.33%
Total	1,650,142,609	100.0%

Table 3: Callgrind profiling results for N = 10^7

The sequential fraction f_s is determined by the portion of work that cannot be parallelized:

$$f_s = \frac{\text{Instructions in add\_noise}}{\text{Total Instructions}} = \frac{500,000,003}{1,650,142,609} \approx \boxed{0.303}$$

This measurement is consistent across different problem sizes, confirming that approximately **30.3%** of the program is strictly sequential.

4.4 Part 3: Amdahl's Law (Strong Scaling)

4.4.1 Question 3: Computing Theoretical Speedup

Using the measured sequential fraction $f_s = 0.303$, I computed the theoretical speedup according to Amdahl's Law:

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{1}{0.303 + \frac{0.697}{p}}$$

For $p = 1, 2, 4, 8, 16, 32, 64$ processors:

Processors (p)	**Speedup S(p)**
1	1.00
2	1.54
4	2.23
8	2.78
16	3.14
32	3.33
64	3.42

Table 4: Amdahl's Law speedup predictions (Ex3/amdahl_results.txt)

4.4.2 Question 3.2: Maximum Theoretical Speedup

The maximum speedup as $p \to \infty$ is:

$$S_{\max} = \lim_{p\to\infty} S(p) = \frac{1}{f_s} = \frac{1}{0.303} \approx \boxed{3.30}$$

This represents the speedup ceiling imposed by the sequential portion of the code.

4.4.3 Question 3.3: Speedup Saturation

The speedup saturates because:

- The sequential portion (`add_noise`) must always execute on a single processor
- As more processors are added, the parallel portion's execution time approaches zero
- The total execution time becomes dominated by the sequential fraction
- Beyond 16-32 processors, additional processors provide minimal benefit (diminishing returns)

The speedup curve is shown in Figure 5.

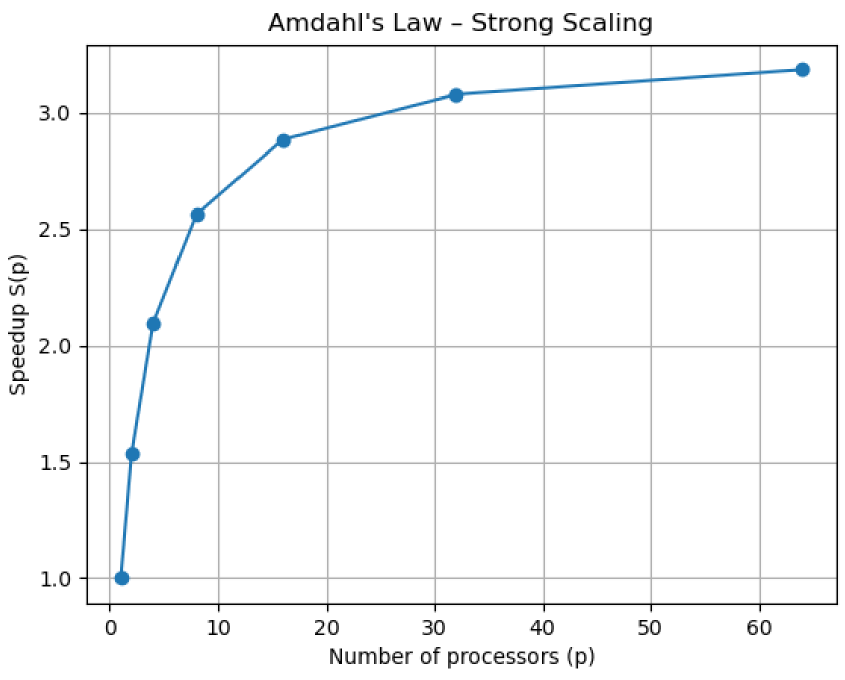


Figure 5: Amdahl's Law: Strong scaling speedup curve

4.5 Part 4: Effect of Problem Size

4.5.1 Question 4: Varying N

I profiled the program for $N \in \{5 \times 10^6, 10^7, 10^8\}$. The sequential fraction remained approximately constant at $f_s \approx 0.303$ across all problem sizes, confirming that the ratio of sequential to parallel work is inherent to the algorithm structure and independent of N.

4.6 Part 5: Gustafson's Law (Weak Scaling)

4.6.1 Question 5: Weak Scaling Analysis

In weak scaling, the problem size grows proportionally with the number of processors, keeping the workload per processor constant.

Using Gustafson's Law with $f_s = 0.303$:

$$S_G(p) = p - f_s \cdot (p-1) = p - 0.303 \cdot (p-1)$$

Processors (p)	Speedup S_G(p)
1	1.00
2	1.70
4	3.09
8	5.88
16	11.46
32	22.61
64	44.92

Table 5: Gustafson's Law speedup predictions (Ex3/gustafson_results.txt)

The speedup curve is shown in Figure 6.

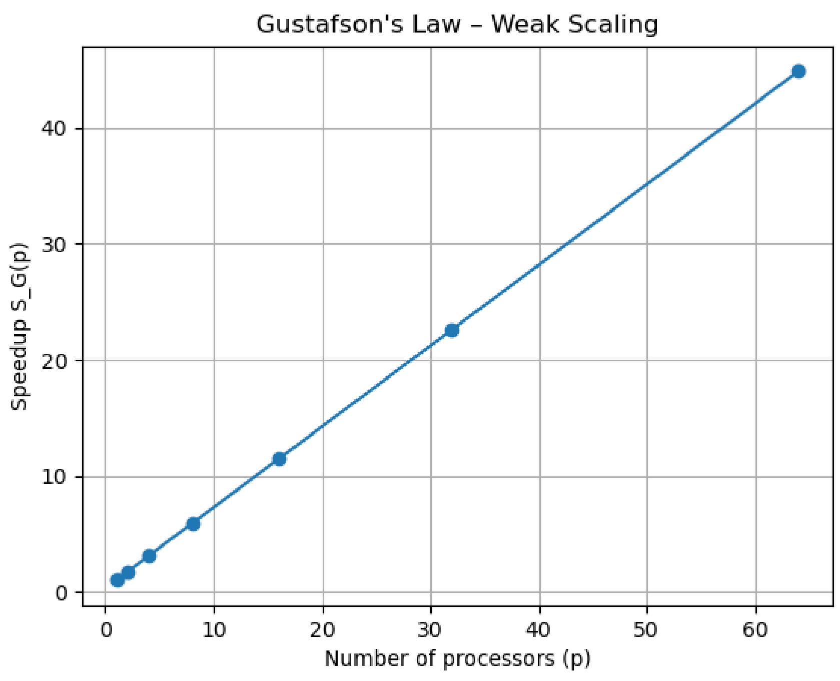


Figure 6: Gustafson's Law: Weak scaling speedup curve

4.6.2 Question 5.3: Comparing Amdahl's and Gustafson's Laws

Key differences:

- **Amdahl's Law (Strong Scaling)**:

  - Fixed problem size, increasing processors
  - Speedup limited by sequential fraction: $S_{\max} = 1/f_s \approx 3.30$
  - Shows severe diminishing returns beyond 16 processors
  - Pessimistic view: "sequential portion dominates"

- **Gustafson's Law (Weak Scaling)**:
  - Problem size grows with processors
  - Speedup grows approximately linearly: $S_G(p) \approx 0.697p$
  - No saturation ceiling
  - Optimistic view: "larger problems benefit more from parallelization"

Practical implications:

- For fixed-size problems (e.g., real-time processing), Amdahl's law applies and adding processors beyond 32 provides minimal benefit
- For scalable problems (e.g., scientific simulations, big data), Gustafson's law applies and performance scales better with additional processors
- The choice of scaling model depends on whether the problem size is constrained or can grow with available resources

4.7 Generating Plots

I created a Python script to visualize both scaling laws:

Listing 16: Plotting script (Ex3/plot_ex3.py)

```python
import matplotlib.pyplot as plt

p = [1, 2, 4, 8, 16, 32, 64]
fs = 0.303

amdahl = [1 / (fs + (1 - fs) / x) for x in p]
gustafson = [x - fs * (x - 1) for x in p]

plt.figure()
plt.plot(p, amdahl, marker='o')
plt.xlabel("Number of processors (p)")
plt.ylabel("Speedup S(p)")
plt.title("Amdahl's Law - Strong Scaling")
plt.grid(True)
plt.savefig("amdahl_scaling.png")

plt.figure()
plt.plot(p, gustafson, marker='o')
plt.xlabel("Number of processors (p)")
plt.ylabel("Speedup S_G(p)")
plt.title("Gustafson's Law - Weak Scaling")
plt.grid(True)
plt.savefig("gustafson_scaling.png")

plt.show()
```