

TP2 - Foundations of Parallel Computing

Imad Kissami

January, 2026

Exercise 1 : Loop Optimizations

Consider the following program that computes the sum of an array of double-precision values :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define N 1000000
6
7 int main() {
8     double *a = malloc(N * sizeof(double));
9     double sum = 0.0;
10    double start, end;
11
12    for (int i = 0; i < N; i++)
13        a[i] = 1.0;
14
15    start = (double)clock() / CLOCKS_PER_SEC;
16    for (int i = 0; i < N; i++)
17        sum += a[i];
18    end = (double)clock() / CLOCKS_PER_SEC;
19
20    printf("Sum=%f, Time=%f ms\n", sum, (end - start) * 1000);
21
22    free(a);
23    return 0;
24 }
```

— Manually unroll the summation loop using different unrolling factors :

$$U = 1, 2, 4, \dots, 32$$

— Example (unrolling factor $U = 4$) :

```

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

Execution and Analysis

1. Implement unrolling factors $U = 1, 2, 3, 4, \dots, 32$
2. Measure execution time for each version

3. Identify the unrolling factor that gives the best performance
4. Compare manual unrolling at `-O0` with compiler optimization at `-O2`
5. Determine whether manual unrolling is still beneficial with `-O2`
6. Repeat 1, 2, 3 by replacing the array type and accumulator with (float, int, short)
7. Assuming a sustained memory bandwidth BW (measured or given), estimate a lower bound on execution time :

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

Compare this bound with your measured times and comment.

8. Explain why increasing U may improve performance at first (reduced loop overhead, more ILP), then saturate when the kernel becomes bandwidth-limited.

Exercise 2 : Instruction Scheduling

— Instruction scheduling : Reordering instructions to improve pipeline efficiency.

```
1 MUL R1, R2, R3 ; Multiply (long latency)
2 ADD R4, R1, R5 ; Add (depends on R1)
3 SUB R6, R7, R8 ; Independent subtraction
```

After reordering :

```
1 MUL R1, R2, R3 ; Multiply (long latency)
2 SUB R6, R7, R8 ; Independent subtraction (executed while MUL is running)
3 ADD R4, R1, R5 ; Add (by now, R1 is ready)
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define N 100000000
6
7 int main() {
8     double a = 1.1, b = 1.2;
9     double x = 0.0, y = 0.0;
10    clock_t start, end;
11
12    start = clock();
13    for (int i = 0; i < N; i++) {
14        x = a * b + x; // stream 1
15        y = a * b + y; // independent stream 2
16    }
17    end = clock();
18
19    printf("x=%f, y=%f, time=%f us\n",
20          x, y, (double)(end - start) / CLOCKS_PER_SEC);
21
22 }
```

```
1 gcc -O0 -S exercise3.c -o exercise3.s
2
3 cat exercise3.s
```

— O0 compilation :

```
1 gcc -O0 -fno-omit-frame-pointer -S -masm=intel exercise3.c -o 00.s
2
3 cat 00.s
```

```
.L3:
movsd xmm0, QWORD PTR -32[rbp]      ; load a
mulsd xmm0, QWORD PTR -24[rbp]      ; xmm0 = a*b
movsd xmm1, QWORD PTR -48[rbp]      ; load x
addsd xmm0, xmm1                  ; xmm0 = a*b + x
movsd QWORD PTR -48[rbp], xmm0     ; store x

movsd xmm0, QWORD PTR -32[rbp]      ; load a (encore)
mulsd xmm0, QWORD PTR -24[rbp]      ; xmm0 = a*b (encore)
movsd xmm1, QWORD PTR -40[rbp]      ; load y
addsd xmm0, xmm1                  ; xmm0 = a*b + y
movsd QWORD PTR -40[rbp], xmm0     ; store y

add    DWORD PTR -52[rbp], 1        ; i++
```

```
1 for (i = 0; i < 100000000; i++) {
2     corps;
3 }
```

```
1 i = 0;
2 goto L2;      // go test the condition first
3
4 L3:          // loop body
5     corps;
6     i = i + 1;
7
8 L2:          // // loop test
9     if (i <= 99999999) goto L3;
10    // otherwise exit the loop
```

— O2 compilation :

```
1 gcc -O2 -fno-omit-frame-pointer -S -masm=intel exercice3.c -o 02.s
2
3 cat 02.s
```

```
mov    eax, 100000000    ; initial counter
.L2:
addsd xmm1, xmm0
addsd xmm1, xmm0      ; 2 additions per loop iteration
sub    eax, 2          ; decrement by 2
jne    .L2            ; loop while eax != 0
```

Execution and Analysis

1. Compare the CPU execution time using `-O0` and `-O2`.
2. Identify the main optimizations performed by the compiler at `-O2` compared to `-O0` (instruction scheduling, loop transformations, ILP).
3. Implement a manually optimized version of the code, compile it using `-O0`, and compare its performance with the automatically optimized version compiled at `-O2`. What conclusion can you draw ?

Exercise 3 :

We consider the following C program, composed of a strictly sequential part and a potentially parallelizable part :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 100000000
5
6 void add_noise(double *a) {
7     a[0] = 1.0;
8     for (int i = 1; i < N; i++) {
9         a[i] = a[i-1] * 1.0000001;
10    }
11 }
12
13 /* ===== Initialization ===== */
14 void init_b(double *b) {
15     for (int i = 0; i < N; i++) {
16         b[i] = i * 0.5;
17     }
18 }
19
20 /* ===== Compute addition ===== */
21 void compute_addition(double *a, double *b, double *c) {
22     for (int i = 0; i < N; i++) {
23         c[i] = a[i] + b[i];
24     }
25 }
26
27 /* ===== Reduction ===== */
28 double reduction(double *c) {
29     double sum = 0.0;
30     for (int i = 0; i < N; i++) {
31         sum += c[i];
32     }
33     return sum;
34 }
35
36 int main() {
37     double *a = malloc(N * sizeof(double));
38     double *b = malloc(N * sizeof(double));
39     double *c = malloc(N * sizeof(double));
40
41     add_noise(a);
42     init_b(b);
43     compute_addition(a, b, c);
44
45     double sum = reduction(c);
46     printf("Sum = %f\n", sum);
47
48     free(a);
49     free(b);
50     free(c);
51     return 0;
52 }
```

Question 1 — Code Analysis

1. Identify the strictly sequential part of the program.
2. Identify the parallelizable part.
3. Give the time complexity of each part as a function of N .

Question 2 — Measuring the Sequential Fraction

The program is compiled with optimization and debug symbols and analyzed using Valgrind / Callgrind.

1. Use Valgrind to profile the code and to define the sequential fraction f_s using the Callgrind measurements.

```
1 valgrind --tool=callgrind ./a.out
```

Using callgrind_annotate :

```
1 callgrind_annotate callgrind.out.244663
2
3 -----
4 Profile data file 'callgrind.out.244663' (creator: callgrind-3.22.0)
5 -----
6 I1 cache:
7 D1 cache:
8 LL cache:
9 Timerange: Basic block 0 - 400034494
10 Trigger: Program termination
11 Profiled target: ./a.out (PID 244663, part 1)
12 Events recorded: Ir
13 Events shown: Ir
14 Event sort order: Ir
15 Thresholds: 99
16 Include dirs:
17 User annotated:
18 Auto-annotation: on
19
20 -----
21 Ir
22 -----
23 6,500,157,241 (100.0%) PROGRAM TOTALS
24
25 -----
26 Ir file:function
27 -----
28 2,200,000,014 (33.85%) ????:compute_addition \
29     [/home/kissami/Documents/GITHUB/Assignments_DC_CI2_2026/TP2/a.out]
30 1,799,999,997 (27.69%) ????:add_noise \
31     [/home/kissami/Documents/GITHUB/Assignments_DC_CI2_2026/TP2/a.out]
32 1,300,000,012 (20.00%) ????:init_b \
33     [/home/kissami/Documents/GITHUB/Assignments_DC_CI2_2026/TP2/a.out]
34 1,200,000,013 (18.46%) ????:reduction \
35     [/home/kissami/Documents/GITHUB/Assignments_DC_CI2_2026/TP2/a.out]
```

Or using kcachegrind :

```
1 kcachegrind callgrind.out.244663
```

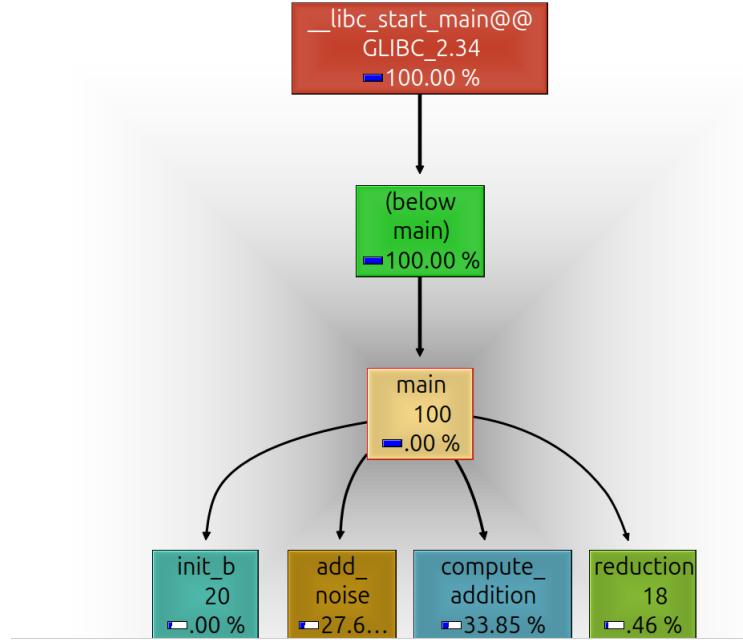
Question 3 — Strong Scaling (Amdahl's Law)

1. Using the measured sequential fraction f_s , compute the theoretical speedup $S(p)$ for $p = 1, 2, 4, 8, 16, 32, 64$.
2. Plot the speedup curve and indicate the theoretical maximum speedup.
3. Explain why the speedup saturates as p increases.

Question 4 — Effect of Problem Size

The experiment is repeated for several values of N :

$$N \in \{5 \times 10^6, 10^7, 10^8\}$$



1. Measure the sequential fraction f_s for each value of N .
2. Plot the Amdahl speedup curves for each value of N .

Question 5 — Weak Scaling (Gustafson's Law)

In weak scaling, the problem size grows proportionally with the number of processors :

1. Using the measured sequential fraction, compute the Gustafson speedup for $p = 1, 2, 4, 8, 16, 32, 64$.
2. Plot the corresponding speedup curve.
3. Compare the predictions of Amdahl's and Gustafson's laws.

Exercise 4 :

Consider the following C program, which combines a strictly sequential phase with a highly parallelizable matrix-matrix multiplication :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 512 // Matrix size
5
6 /* ===== Generate noise ===== */
7 void generate_noise(double *noise) {
8     noise[0] = 1.0;
9     for (int i = 1; i < N; i++) {
10         noise[i] = noise[i-1] * 1.0000001;
11     }
12 }
13
14 /* ===== Matrices Initialization ===== */
15 void init_matrix(double *M) {
16     for (int i = 0; i < N*N; i++) {
17         M[i] = (double)(i % 100) * 0.01;
18     }
19 }
```

```
20
21 /* ===== Matrix Multiplication ===== */
22 void matmul(double *A, double *B, double *C, double *noise) {
23     for (int i = 0; i < N; i++) {
24         for (int j = 0; j < N; j++) {
25             double sum = noise[i];
26             for (int k = 0; k < N; k++) {
27                 sum += A[i*N + k] * B[k*N + j];
28             }
29             C[i*N + j] = sum;
30         }
31     }
32 }
33
34 int main() {
35     double *A = malloc(N*N * sizeof(double));
36     double *B = malloc(N*N * sizeof(double));
37     double *C = malloc(N*N * sizeof(double));
38     double *noise = malloc(N * sizeof(double));
39
40     generate_noise(noise);
41     init_matrix(A);
42     init_matrix(B);
43
44     matmul(A, B, C, noise);
45
46     printf("C[0]=%f\n", C[0]);
47
48     free(A);
49     free(B);
50     free(C);
51     free(noise);
52     return 0;
53 }
```

- Do the same benchmarks and analysis as in Exercise 3 :
1. measure the sequential fraction using Valgrind/Callgrind,
 2. evaluate the strong scaling using Amdahl's law,
 3. evaluate the weak scaling using Gustafson's law,
 4. plot the corresponding speedup curves,
 5. and compare the results with those obtained in Exercise 3.