

# TP3 - OpenMP (Introduction)

Imad Kissami

February, 2026

## Exercise 1

- Write an OpenMP program displaying the number of threads used for the execution and the rank of each of the threads.
- Test the programs obtained with different numbers of threads for the parallel program, without submitting in batch.

**Output example for the parallel program with 4 threads :**

```
1 // Possible output
2 Hello from the rank 2 thread
3 Hello from the rank 1 thread
4 Hello from the rank 3 thread
5 Hello from the rank 0 thread
6 Parallel execution of hello_world with 4 threads
```

## Exercise 2 : Parallelizing of PI calculation

```
1 static long num_steps = 100000;
2 double step;
3
4 int main () {
5     int i; double x, pi, sum = 0.0;
6     step = 1.0 / (double) num_steps;
7     for (i = 0; i < num_steps; i++) {
8         x = (i + 0.5) * step;
9         sum = sum + 4.0 / (1.0 + x * x);
10    }
11    pi = step * sum;
12 }
```

- Create a parallel version of the pi program using a parallel construct.
- Don't use `#pragma omp parallel for`.
- Pay close attention to shared versus private variables.
- Use `double omp_get_wtime()` to calculate the CPU time.

## Exercise 3 : Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct.
- Your goal is to minimize the number of changes made to the serial program (add only 1 line).

## Exercise 4 : Parallelizing Matrix Multiplication with OpenMP

```

1  // Allocate memory dynamically
2  double *a = (double *)malloc(m * n * sizeof(double));
3  double *b = (double *)malloc(n * m * sizeof(double));
4  double *c = (double *)malloc(m * m * sizeof(double));
5
6  // Initialize matrices
7  for (int i = 0; i < m; i++) {
8      for (int j = 0; j < n; j++) {
9          a[i * n + j] = (i + 1) + (j + 1);
10     }
11 }
12
13 for (int i = 0; i < n; i++) {
14     for (int j = 0; j < m; j++) {
15         b[i * m + j] = (i + 1) - (j + 1);
16     }
17 }
18
19 for (int i = 0; i < m; i++) {
20     for (int j = 0; j < m; j++) {
21         c[i * m + j] = 0;
22     }
23 }
24
25 // Matrix multiplication
26 for (int i = 0; i < m; i++) {
27     for (int j = 0; j < m; j++) {
28         for (int k = 0; k < n; k++) {
29             c[i * m + j] += a[i * n + k] * b[k * m + j];
30         }
31     }
32 }

```

- Insert the appropriate OpenMP directives and analyze the code performance.
- Use `collapse` directive to parallelize this matrix multiplication code.
- Run the code using 1, 2, 4, 8, 16 threads and plot the speedup and efficiency.
- Test the loop iteration repartition modes (STATIC, DYNAMIC, GUIDED) and vary the chunk sizes.

## Exercise 5 : Parallelizing of Jacobi Method with OpenMP

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <float.h>
5  #include <math.h>
6  #include <sys/time.h>
7  #include <omp.h>
8
9  #ifndef VAL_N
10 #define VAL_N 120
11 #endif
12 #ifndef VAL_D
13 #define VAL_D 80
14 #endif
15
16 void random_number(double* array, int size) {
17     for (int i = 0; i < size; i++) {
18         array[i] = (double)rand() / (double)(RAND_MAX - 1);
19     }
20 }
21

```

```

22 int main() {
23     int n = VAL_N, diag = VAL_D;
24     int i, j, iteration = 0;
25     double norme;
26
27     double *a = (double*)malloc(n * n * sizeof(double));
28     double *x = (double*)malloc(n * sizeof(double));
29     double *x_courant = (double*)malloc(n * sizeof(double));
30     double *b = (double*)malloc(n * sizeof(double));
31
32     if (!a || !x || !x_courant || !b) {
33         fprintf(stderr, "Memory allocation failed!\n");
34         exit(EXIT_FAILURE);
35     }
36
37     struct timeval t_elapsed_0, t_elapsed_1;
38     double t_elapsed;
39
40     double t_cpu_0, t_cpu_1, t_cpu;
41
42     srand(421);
43     random_number(a, n * n);
44     random_number(b, n);
45
46     for (i = 0; i < n; i++) {
47         a[i * n + i] += diag;
48     }
49
50     for (i = 0; i < n; i++) {
51         x[i] = 1.0;
52     }
53
54     t_cpu_0 = omp_get_wtime();
55     gettimeofday(&t_elapsed_0, NULL);
56
57     while (1) {
58         iteration++;
59
60         for (i = 0; i < n; i++) {
61             x_courant[i] = 0;
62             for (j = 0; j < i; j++) {
63                 x_courant[i] += a[j * n + i] * x[j];
64             }
65             for (j = i + 1; j < n; j++) {
66                 x_courant[i] += a[j * n + i] * x[j];
67             }
68             x_courant[i] = (b[i] - x_courant[i]) / a[i * n + i];
69         }
70
71         double absmax = 0;
72         for (i = 0; i < n; i++) {
73             double curr = fabs(x[i] - x_courant[i]);
74             if (curr > absmax)
75                 absmax = curr;
76         }
77         norme = absmax / n;
78
79         if ((norme <= DBL_EPSILON) || (iteration >= n)) break;
80
81         memcpy(x, x_courant, n * sizeof(double));
82     }
83
84     gettimeofday(&t_elapsed_1, NULL);
85     t_elapsed = (t_elapsed_1.tv_sec - t_elapsed_0.tv_sec) +
86     (t_elapsed_1.tv_usec - t_elapsed_0.tv_usec) / 1e6;
87
88     t_cpu_1 = omp_get_wtime();
89     t_cpu = t_cpu_1 - t_cpu_0;
90
91     fprintf(stdout, "\n\nSystem size: %5d\n"
92     "Iterations: %4d\n"
93     "Norme: %10.3E\n"
94     "Elapsed time: %10.3E sec.\n"

```

```
95  "CPU_time: %10.3E sec.\n",  
96  n, iteration, norme, t_elapsed, t_cpu);  
97  
98  free(a); free(x); free(x_courant); free(b);  
99  return EXIT_SUCCESS;  
100 }
```

- Solve the linear system using Jacobi iterative method in parallel.
- Run the code using 1, 2, 4, 8, 16 threads and plot the speedup and efficiency.