TP2 - OpenMP (Introduction)

Imad Kissami

June, 2025

Exercise 1

- Write an OpenMP program displaying the number of threads used for the execution and the rank of each of the threads.
- Compile the code manually to create a monoprocessor executable and a parallel executable.
- 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 :

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

Exercise 2 : Parallelizing of PI calculation

```c
static long num_steps = 100000;
double step;

int main () {
  int i; double x, pi, sum = 0.0;
  step = 1.0 / (double) num_steps;
  for (i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  pi = step * sum;
}
```

- 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

```c
// Allocate memory dynamically
double *a = (double *)malloc(m * n * sizeof(double));
double *b = (double *)malloc(n * m * sizeof(double));
double *c = (double *)malloc(m * m * sizeof(double));

// Initialize matrices
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    a[i * n + j] = (i + 1) + (j + 1);
  }
}

for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    b[i * m + j] = (i + 1) - (j + 1);
  }
}

for (int i = 0; i < m; i++) {
  for (int j = 0; j < m; j++) {
    c[i * m + j] = 0;
  }
}

// Matrix multiplication
for (int i = 0; i < m; i++) {
  for (int j = 0; j < m; j++) {
    for (int k = 0; k < n; k++) {
      c[i * m + j] += a[i * n + k] * b[k * m + j];
    }
  }
}
```

— 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <math.h>
#include <sys/time.h>
#include <omp.h>

#ifndef VAL_N
#define VAL_N 120
#endif
#ifndef VAL_D
#define VAL_D 80
#endif

```

```c
void random_number(double* array, int size) {
  for (int i = 0; i < size; i++) {
    array[i] = (double)rand() / (double)(RAND_MAX - 1);
  }
}

int main() {
  int n = VAL_N, diag = VAL_D;
  int i, j, iteration = 0;
  double norme;

  double *a = (double*)malloc(n * n * sizeof(double));
  double *x = (double*)malloc(n * sizeof(double));
  double *x_courant = (double*)malloc(n * sizeof(double));
  double *b = (double*)malloc(n * sizeof(double));

  if (!a || !x || !x_courant || !b) {
    fprintf(stderr, "Memory␣allocation␣failed!\n");
    exit(EXIT_FAILURE);
  }

  struct timeval t_elapsed_0, t_elapsed_1;
  double t_elapsed;

  double t_cpu_0, t_cpu_1, t_cpu;

  srand(421);
  random_number(a, n * n);
  random_number(b, n);

  for (i = 0; i < n; i++) {
    a[i * n + i] += diag;
  }

  for (i = 0; i < n; i++) {
    x[i] = 1.0;
  }

  t_cpu_0 = omp_get_wtime();
  gettimeofday(&t_elapsed_0, NULL);

  while (1) {
    iteration++;

    for (i = 0; i < n; i++) {
      x_courant[i] = 0;
      for (j = 0; j < i; j++) {
        x_courant[i] += a[j * n + i] * x[j];
      }
      for (j = i + 1; j < n; j++) {
        x_courant[i] += a[j * n + i] * x[j];
      }
      x_courant[i] = (b[i] - x_courant[i]) / a[i * n + i];
    }

    double absmax = 0;
    for (i = 0; i < n; i++) {
      double curr = fabs(x[i] - x_courant[i]);
      if (curr > absmax)
      absmax = curr;
    }
    norme = absmax / n;

    if ((norme <= DBL_EPSILON) || (iteration >= n)) break;

    memcpy(x, x_courant, n * sizeof(double));
  }

  gettimeofday(&t_elapsed_1, NULL);
  t_elapsed = (t_elapsed_1.tv_sec - t_elapsed_0.tv_sec) +
  (t_elapsed_1.tv_usec - t_elapsed_0.tv_usec) / 1e6;

  t_cpu_1 = omp_get_wtime();
```

```c
  t_cpu = t_cpu_1 - t_cpu_0;

  fprintf(stdout, "\n\nSystem size        : %5d\n"
  "Iterations         : %4d\n"
  "Norme              : %10.3E\n"
  "Elapsed time       : %10.3E sec.\n"
  "CPU time           : %10.3E sec.\n",
  n, iteration, norme, t_elapsed, t_cpu);

  free(a); free(x); free(x_courant); free(b);
  return EXIT_SUCCESS;
}
```

- 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.