High Performance Computing

Calcul de π

Rémi Garde

Février 2018

Contents

1 Pre-Processing

1.1 Définition du problème

Le but de ce problème est d'obtenir une approximation de π, en se basant sur l'égalité

$$\pi = \int_0^1 \frac{4}{1+x^2} \mathrm{d}x$$

Ainsi en calculant une intégrale il est possible d'avoir une valeur approchée de π. Le calcul d'intégrale est un classique algorithmique, nous allons utiliser la méthode des trapèzes: on subdivise le domaine de l'intégrale $[a, b]$ en n segments $[a_i, a_{i+1}]_{i \in [\![0,n]\!]}$ avec $a_0 = a$, $a_{n-1} = b$ soit

$$a_i = a + i\frac{b-a}{n} \text{ et } \int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1} \left(\int_{a_i}^{a_{i+1}} f(x)\mathrm{d}x \right)$$

Chaque petite intégrale est ensuite approchée par l'aire du trapèze de sommet $(a_i, f(a_i), f(a_{i+1}), a_{i+1})$ soit

$$\int_{a_i}^{a_{i+1}} f(x)\mathrm{d}x \approx (a_{i+1} - a_i)) \frac{f(a_{i+1}) + f(a_i)}{2} = \frac{b-a}{n} \frac{f(a_{i+1}) + f(a_i)}{2}$$

D'où le code:

```
float trapeze_sequential (float a, float b, int n) {
    float integral = 0; // result
    float h = (b-a)/n; // step
    for ( int i = 1 ; i < n ; i++ ) {
        integral += h * (f(a + i*h) + f(a + (i+1)*h) / 2;
    }

    return integral;
}
```

1.2 Optimisations

On remarque que chaque point sera évalué 2 fois, hormis $f(a)$ et $f(b)$, et que l'on peut factoriser par h. On peut donc simplifier en:

```
float trapeze_sequential (float a, float b, int n) {
    float integral = (f(b) - f(a)) / 2; // result
    float h = (b-a)/n; // step
    for ( int i = 1 ; i < n ; i++ ) {
        integral += (f(a + i*h));
    }
    integral *= h;

    return integral;
}
```

2 Processing

2.1 Parallélisation

Intéressons nous maintenant à paralléliser ce programme. À première vue, la boucle ne présente aucune dépendance est peut donc être distribuée facilement. Cependant, le nombre d'itération n va être le facteur principal de la précision du résultat, il serait donc préférable de le garder indépendant du nombre de processus. Ainsi l'on va découper l'intégrale en k sous intégrales, k étant le nombre de noeuds de calcul voulus. Chaque processus va calculer sa part d'intégrale avec la méthode précédente. Le thread principal va ensuite sommer chacune des valeurs retournées par les autres processus.

```
int rank, size;
MPI_Init(&argc, &argv);                /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &size); /* get number of processes */
if (rank == 0) {
    float integral, sub_integral; // result
    float h = (b - a) / n;        // step
    // Evaluate the first part of the integral
    integral = trapeze_sequential(a, a + (b - a) / size, n / size);
    // Sum with the other sub integrals
    for (int k = 1; k < size; k++)
    {
        printf("> Waiting for process %d to compute...\n", k);
        MPI_Recv(&sub_integral, 1, MPI_FLOAT, k, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        integral += sub_integral;
    }
    // Printing the results
    printf("> Result: \n%f\n", integral);
    printf("> Error (%%): \n%f", 100*(M_PI - integral)/M_PI);
} else {
    // Other process use directly the sequential computation
    printf("> Calculation from process %d\n", rank);
    float local_a = a + rank * (b - a) / size,
            local_b = a + (rank + 1) * (b - a) / size;

    // Evaluate the sub integral
    float sub_integral =
        trapeze_sequential(local_a, local_b, n / size);
    // Send the result to the main process
    MPI_Send(&sub_integral, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
```

On notera que lancer MPI avec 1 seul thread va être équivalent avec un calcul séquentiel (en rajoutant un overhead pour MPI).

3 Post-Processing