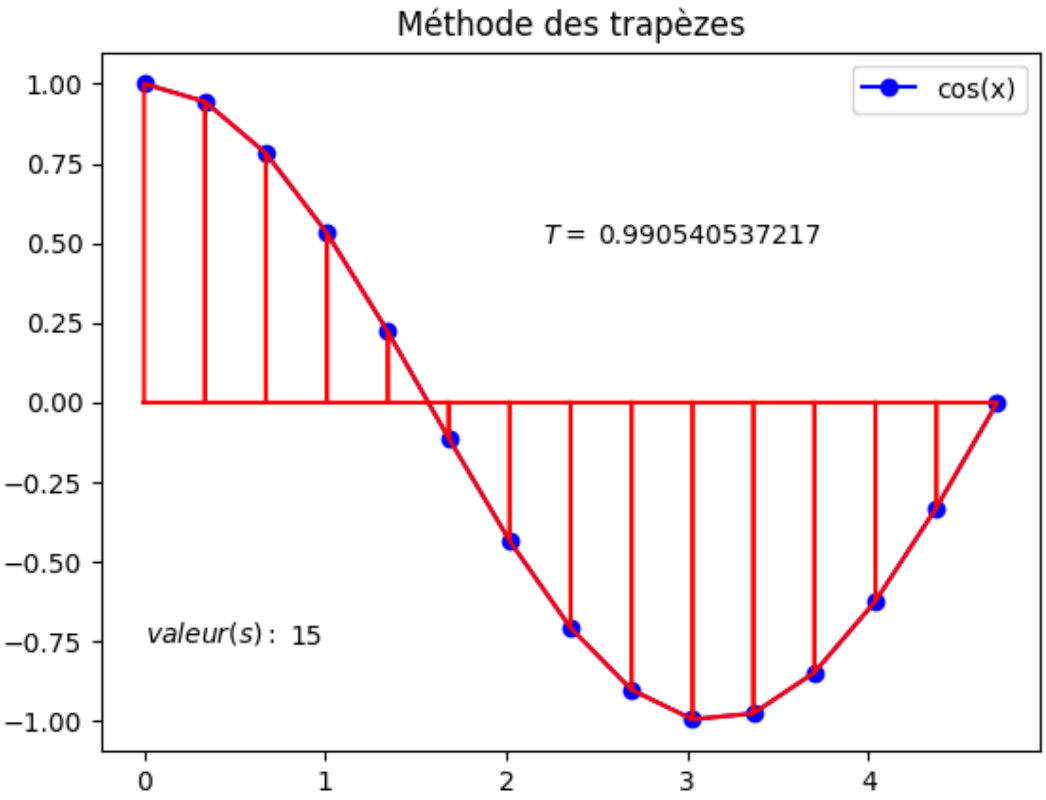
Institut Universitaire de Technologie
Aix*Marseille Université
Site d'Arles

Projet de programmation parallèle, M3101
Dans le cadre du Diplôme Universitaire de Technologie

DECOOPMAN Clément
DAMIENS Lucas

Calcul d'intégrale
par la méthode des trapèzes



Test préalable généré depuis un programme python personnel

Table des matières

Introduction et contexte

Présentation du sujet

Le but du projet est de calculer l'intégrale d'un polynôme entre 2 bornes par la méthode des trapèzes, pour un nombre de threads et d'itérations donnés.
La formule de la courbe est stockée dans une structure par le biais d'un simple tableau d'entier.
Ex : coef[0] = a ; coef[1] = b ; coef[2] = c ;

Le programme résulte par l'affichage des résultats en console, ainsi que la création d'un fichier txt contenant ces mêmes résultats.

Description de la machine de test

Pour les besoins des tests, le programme 'plot_curve.cpp' sera utilisé car il illustre bien l'utilisation du processeur pendant les calculs, puisqu'il lance de manière successive le programme de calcul principal avec différents paramètres.

- Description de la machine : PC fixe (personnel)

Vitesse de base :	3,00 GHz
Sockets :	1
Cœurs :	4
Processeurs logiques :	4
Virtualisation :	Activé
Cache de niveau 1 :	256 Ko
Cache de niveau 2 :	1,0 Mo
Cache de niveau 3 :	6,0 Mo

- Évolution du processeur en temps réel pendant le programme (plotCurveExec) :

process_evolution.mp4

(double clic pour ouvrir la vidéo)

- RAM : Obtenue avec la commande « systeminfo » sous Windows

Avant le lancement du programme

```
Mémoire physique totale:                    8 096 Mo
Mémoire physique disponible:                3 366 Mo
Mémoire virtuelle : taille maximale:        12 950 Mo
Mémoire virtuelle : disponible:             6 815 Mo
Mémoire virtuelle : en cours d'utilisation: 6 135 Mo
```

Pendant le programme

```
Mémoire physique totale:                    8 096 Mo
Mémoire physique disponible:                3 372 Mo
Mémoire virtuelle : taille maximale:        12 950 Mo
Mémoire virtuelle : disponible:             6 347 Mo
Mémoire virtuelle : en cours d'utilisation: 6 603 Mo
```

Documentations

- **Calcul d'intégrales par la méthode des trapèzes :**

Approximation d'une intégrale, « Algorithmique et calcul numérique », José OUIN

https://fr.wikipedia.org/wiki/M%C3%A9thode_des_trap%C3%A8zes

http://serge.mehl.free.fr/anx/meth_trap.html

http://www.courspython.com/integration-rectangle-trapeze.html

- **Parallélisation du programme :**

Cours sur Amétice

https://www.lri.fr/~falcou/teaching/par/OpenMP-cours.pdf

Algorithme

Principe et fonctionnement séquentiel

- **La fonction** :

```
typedef struct
{
  long double borne_min, borne_max;
  int coef[3];
  long double integrale;
}Function;
```

La fonction polynomiale de second degré est définie telle une structure comportant son intervalle de définition, ses coefficients ainsi que la valeur de son intégrale.

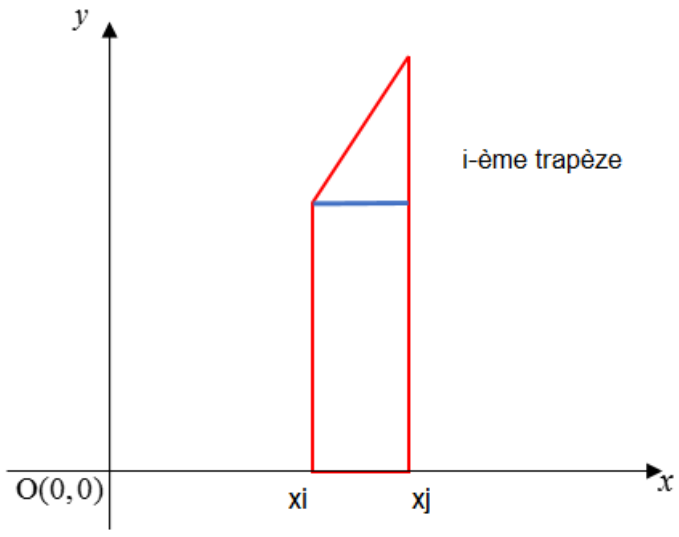
```
long double fn(long double x, Function * p)
{
  return p->coef[0]*(x*x) + p->coef[1]*x + p->coef[2];
}
```

Ci-dessus la fonction permettant de calculer le polynome.
Cette dernière prend en paramètre la valeur de l'inconnue x et un pointeur sur une instance de la structure.

- **Calcul de l'intégrale :**

```
void Ttrapeze(int n, Function * p)
{
  long double T = 0;
  long double xi, xj;
  long double a = p->borne_min;
  long double b = p->borne_max;


  for(int i=0; i<n; i++)
  {
      xi = a+(b-a)*i/(float)n;
      xj = a+(b-a)*(i+1)/(float)n;
      T += (xj-xi)/2.0f * (fn(xi,p)+fn(xj,p));
  }

  if(T<0 || a>b)
    p->integrale = -T;

  p->integrale = T;
}
```

Sur la fonction ci-dessus, on effectue le calcul de l'intégrale pour chaque trapèze contenu dans l'intervalle de la fonction.
Pour cela, deux variables xi et xj définissent le début et la fin de chaque trapèze tel que :



Au fur et à mesure de la boucle for, les valeurs de xi et xj se rapprochent de la borne majorante en conservant la distance xixj.

A chaque itération, on avance dans les trapèzes

Parallélisation sans OpenMP

- **La fonction :**

```
typedef struct
{
  long double min, max;
  double coef[3];
  float bmin, bmax;
  int ITERATION;
}Function;
```

Pour le programme threads, la structure de données pour la fonction compte des bornes internes afin de calculer l'intégrale sur un intervalle différent (compris dans l'intervalle global de la fonction) pour chaque thread.

La valeur de l'intégrale correspond à une variable globale partagée entre tous les threads.

- **Parallélisation :**

```
// VARIABLES
pthread_t thtab[NBTHREADS];
Function * myFunctions = new Function[NBTHREADS];
```

Les threads (dont le nombre est entré en paramètre du programme) sont stockés dans un tableau.
Le calcul de l'intégrale se divise donc entre tous ces threads, et cela en créant une structure de données pour chaque thread.

```
for(size_t i=0; i<NBTHREADS; ++i)
{
  myFunctions[i] . min       = 0;
  myFunctions[i] . max       = 5;
  myFunctions[i] . bmin      = partSize*i;
  myFunctions[i] . bmax      = partSize*(i+1);
  myFunctions[i] . coef[0]   = 5;
  myFunctions[i] . coef[1]   = 2;
  myFunctions[i] . coef[2]   = 20;
  myFunctions[i] . ITERATION = NBITERATION;
}
```

Les données nécessaires au calcul sont réparties dans tous les threads à travers cette boucle. Il est observable que les bornes internes aux threads, bmin et bmax, sont bien divisées entre tous les threads, et ce sur l'ensemble de l'intervalle global de la fonction. De ce fait, aucun thread ne calcul la même chose dans l'intervalle.

PartSize correspond à la division de NBITERATION par NBTHREADS, soit au nombre de trapèzes qui doivent être calculés par threads.

```
for(size_t th=0; th<NBTHREADS; ++th)
{
  cerr << "lancement du thread " << th << endl;
  pthread_create(&thtab[th], NULL, Ttrapeze, (void *) &myFunctions[th]);
}
```

Le départ des threads s'organise de manière successive. On créé un thread et on l'affecte à la fonction de calcul en prenant en paramètre la structure de données. L'opération est répétée autant de fois qu'il y a de nombre de threads.

```
for(size_t th=0; th<NBTHREADS; ++th)
{
  pthread_join(thtab[th], NULL);
  printf("arrivée du thread %zu \n", th);
}
```

L'arrivée des threads est sensiblement structurée de la même manière que le départ. On attend que chaque thread contenu dans thtab ait fini de calculer.

```
void * Ttrapeze(void * _args)
{
  Function * _p = (Function *) _args;

  long double T = 0.0f;
  long double localSum = 0.0f;
  long double xi, xj;
  long double a = _p->min;
  long double b = _p->max;
  const int N = _p->ITERATION;


  for(size_t i=_p->bmin; i < size_t(_p->bmax); ++i)
  {
      xi = a+(b-a)*i / double(N);              // On
      xj = a+(b-a)*(i+1) / double(N);          // pou
      T  = (xj-xi)/2.0f * (fn(xi,_p)+fn(xj,_p));

      if(T<0 || a>b)
        T = -T;

      localSum += T;
  }

  IntegralSum(localSum);    // Ajout du résultat local
  return(NULL);
}
```

La fonction de calcul possède une variable locale « localSum » correspondant à la valeur de l'intégrale qu'un thread a calculé. La valeur de cette variable est ajoutée à la variable globale par le biais de la fonction « IntegralSum

```
//SOMME DES INTEGRALES : avec la variable globale temp_integral
// La variable temp_integrale étant globale, elle est partagée entre tous les threads
// on a donc une zone d'exclusion mutuelle avec un mutex
void IntegralSum(long double localDataSum)
{
  mtx.lock();
  temp_integral += localDataSum;
  mtx.unlock();
}
```

Ci-dessus la fonction permettant l'ajout des résultats des threads pour former la valeur globale de l'intégrale de la fonction.
Comme la variable « final_integral » est commune à tous les threads, une zone d'exclusion mutuelle ou mutex est mise en place. Un seul thread à la fois peut accéder à cette variable.

Parallélisation avec OpenMP

La parralélisation commence à la ligne 1 du code ci-dessous. « num_threads() » permet de définir le nombre de threads utilisés pour le programme. Ici on découpe le programme en un certain nombre de threads donnés par l'utilisateur lors de l'exécution stocké dans la variable « NBTHREADS ». Ensuite les threads vont tous faire la boucle for pour différentes bornes en calculant l'aire des trapèzes et additionner le résultat dans la variable Sum. A la fin de la boucle chaque threads ce stop jusqu'à ce que la parallélisation soit fini.

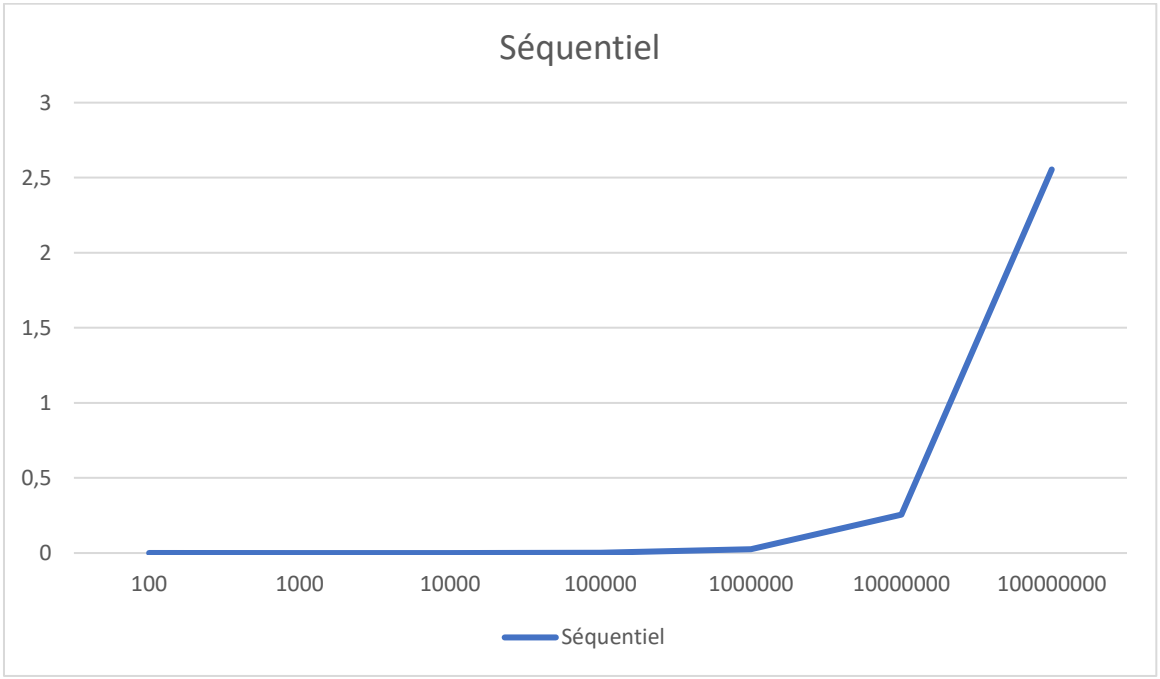
```
#pragma omp parallel sections num_threads(NBTHREADS)
{

  #pragma omp section
  for(size_t i=0; i < size_t(NBITERATION); ++i)
  {
    myFunction->bmin = (partSize)*i;
    myFunction->bmax = (partSize)*(i+1);
    xi = a+(b-a)*i / double(NBITERATION);
    xj = a+(b-a)*(i+1) / double(NBITERATION);
    long double fxi = myFunction->coef[0]*(xi*xi) + myFunction->coef[1]*xi + myFunction->coef[2];
    long double fxj = myFunction->coef[0]*(xj*xj) + myFunction->coef[1]*xj + myFunction->coef[2];
    T = (xj-xi)/2.0f * (fxi+fxj);

    if(T<0 || a>b)
      T = -T;
    myFunction->Sum += T;
  }
}
```

Code de parallélisation avec OpenMP

Comportement et évolution du programme

Les courbes et les explications ci-après concernent les programmes integrale_sequentiel.cpp / integrale_thread.cpp / Projet_OpenMP.cpp

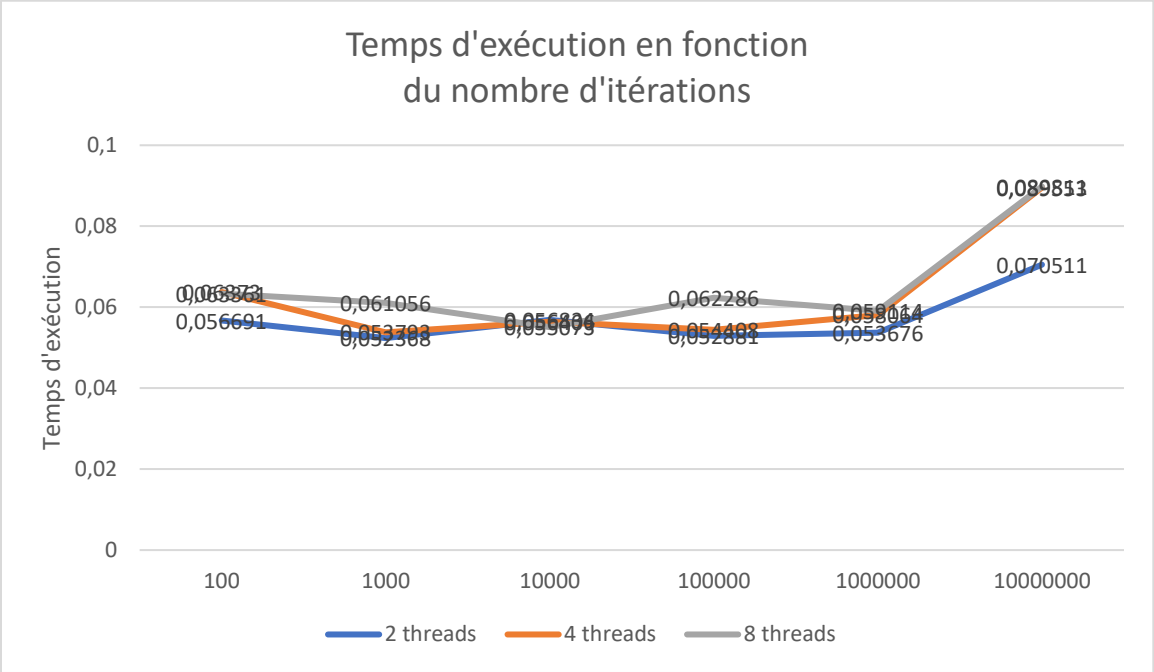
Évolution séquentielle



D'après la courbe ci-dessus, il semble que le programme séquentiel perd en efficacité lorsque le nombre d'itérations tend à augmenter.

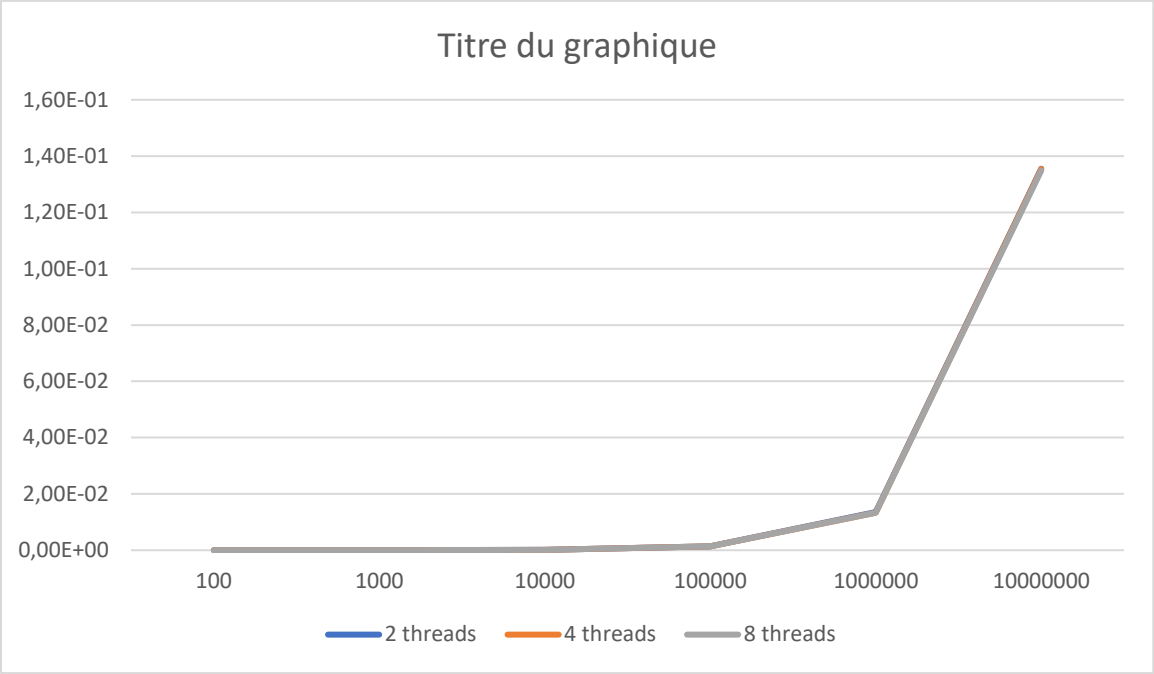
En effet, on se trouve plutôt en $O(n^2)$ au niveau du temps d'exécution.

Évolution parallèle sans OpenMP



Cette fois-ci, la courbe indique une certaine stabilité au niveau des temps d'exécution (même si certains sont plus ou moins étranges).
Il est possible de dire une telle chose si l'on prend en compte les unités (ordre de grandeur), et non de l'allure de la courbe.

Évolution parallèle avec OpenMP



Pour la version OpenMP, les courbes sont quasiment confondues.
Il est remarquable, bien que l'allure de la courbe montre une hausse du temps d'exécution, que les unités ont un ordre de grandeur bas et offrent malgré tout plus de performance que les autres programmes.

Conclusion

Avantages et inconvénients de la méthode

Les avantages de l'OpenMP sont que les temps d'exécution, quelques soit le nombre de threads et d'itérations, restent assez similaires, que le code est plus court et compréhensible. (196 lignes pour OpenMP contre 288 lignes pour les threads).