Pontificia Universidad Católica de Chile
Facultad de Ingeniería
Instituto de Ingeniería Matemática y Computacional
Segundo Semestre 2021
Docente: Elwin Van T Wout

IMT2112 - Algoritmos Paralelos en Computación Científica

Tarea Nº2

Nicolás Sumonte

Esta tarea contempló la implementación en OpenMP del cálculo de la supericie del conjunto de Mandelbrot, el cual, se define como todos los números complejos c tal que la sucesión $x_0 = 0, x_n = x_{n-1}^2 + c, n = 1, 2, \ldots$ se mantiene acotada en valor absoluto. Hay un teorema que dice que c pertenece al conjunto de Mandelbrot si $|x_n| < 2$ para todos n

Para la implementación se han creado dos archivos: `parallel.cpp` y `parallel_p.cpp`. El primero, contiene el método de *pixel counting* sin múltiples procesos, mientras que el segundo permite calcular el algoritmo con múltiple numero de *Threads*. Estos métodos se explican a continuación:

1. Parallel.cpp

Para ambos archivos se utilizó la librería `complex` de `C++` la cual permite instanciar números complejos como también, encontrar parte real e imaginaria del mismo de forma rápida. También se utilizó la función `decltype` la cual nos permite mantener la estructura de ciertas constantes otorgando nuevos valores a la misma.

En ambos archivos se utilizó la función `perteneceA` la cual recibe un numero complejo y un numero máximo de iteraciones para finalmente retornar 1 si el modulo del numero es menor a 2 o 0 en otro caso. Esta función va iterando sobre la sucesión retornando cada vez que hay una iteración.

En el `main()` se declaran el numero máximo de puntos y la cuenta de los números de la sucesión que cumplen con las condiciones, así como también los tiempos y la parte real e imaginaria de un numero cualquiera.

Luego se comienza a iterar sobre el máximo de punto y se generan dos números aleatorios, uno para la parte real y otro para la parte imaginaria, finalmente se define el numero complejo con `decltype` y se le entrega el numero a la función `perteneceA` para hacer el calculo de numeros que cumplen las condiciones del *Pixel Counting*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <complex.h>
4 #include<iostream>
5 #include <math.h>
6 #include <omp.h>
7 // se utiliza el namespace de complex
```

```
8  using namespace std;
9
10 int perteneceA(int maxiteration, complex<float> complexnum) {
11     // INPUT: numero complejo inicial y numero de iteraciones,
          OUTPUT: 0 o 1.
12     // Esta funcion calcula la sucesi n mandelbrot y retorna 1 si
          la iteracion se encuentra en el conjunto 0 cero en e caso
          contrario
13     float realx, imagx, realc, imagc;
14     // se inicializa un numero complejo igual al entregado a la
          funcion
15     complex<float> xn = complexnum;
16     // se calcula la parte real e imaginaria
17     realc = real(complexnum);
18     imagc = imag(complexnum);
19     for (int iteration = 0; iteration<maxiteration; iteration++){
20         // se comienza a iterar sobre la sucesi n
21         xn = pow(xn,2);
22         // se calcula la parte real e imaginaria de c
23         realx = real(xn);
24         imagx = imag(xn);
25         // se suma xn^2 + c
26         xn = decltype(xn)((float) realc + realx, (float) imagc +
              imagx);
27         // se calcula la norma de la sucesi n y se retorna 0 o 1
28         if( abs(xn) > 2 ) return 0;
29     }
30     return 1;
31 }
32
33 int main(){
34     // se define el numero de iteraciones y la cuenta de los
          valores que se encuentran en determinada area
35     int max_n, count;
36     // se define la parte real e imaginaria de un numero c
          cualquiera
37     float real,imag, max_points;
38     // se definen los tiempos de inicio y termino del algoritmo
39     double start,end,times;
40     // se calcula el tiempo de inicio y se setean valores para
          comenzar el algoritmo
41     int max_n, count;
42     float real,imag, max_points;
43     double start,end,times;
44     // se inicializa el tiempo
45     start =  omp_get_wtime();
46     count = 0;
47     max_points = 10000000;
48     max_n = 1640;
49     // se intancia un numero complejo cualquiera "c"
50     complex<float> complexnum;
```

```
51      // por cada punto se genera un numero complejo aleatorio y en
            base a este, se utiliza la funcion perteneceA
52      for(int p = 0; p < max_points; ++p){
53              real = (float)rand() / (float)RAND_MAX;
54              imag = (float)rand() / (float)RAND_MAX;
55              complexnum = decltype(complexnum)((float)(real * 4) -
                    2, (float)(imag * 4) - 2);
56              count += perteneceA(max_n, complexnum);
57      }
58      // se calculan los tiempos finales del algoritmo
59      end =  omp_get_wtime();
60      times = end - start;
61      // se muestran los resultados
62      printf("La aproximacion de Pixel Counting es : %f\n", (count /
            max_points) * 16);
63      printf("Este proceso demor  %f segundos.\n", times);
64      return count;
65  }
```

2. Parallel p.cpp

En este archivo se realiza de forma similar al anterior, sin embargo, se genera una lista de números complejos aleatorios para que el algoritmo itere sobre ella y se pueda calcular la cantidad de números de la sucesión que se encuentran en el conjunto. Esta lista se genera antes de paralelizar debido a que si inclusive ponemos el mismo *seed* en el algoritmo, este genera distintos resultados. Todo esto se debe a que en los procesos que se ejecutan en paralelo, un *Thread* puede modificar los números aleatorios en momentos inconsistentes, y el número aleatorio resultante se utiliza como semilla para otro subproceso.

Sobre esta lista se realiza la paralelización con *OpenMP* según el numero de *Threads* que hayan sido definidos en el archivo y se entregan los números complejos junto con el numero maximo de iteraciones a `perteneceA`.

Para evitar las **Condiciones de carrera** se genera una lista con espacios vacíos según la cantidad de *Threads* definidas en el archivo. Esto con el fin de que cada proceso sume sobre cada espacio de la lista en particular y que no se produzca una suma de elementos al mismo tiempo.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <complex.h>
4   #include<iostream>
5   #include <math.h>
6   #include <omp.h>
7   // se utiliza el namespace de complex
8   using namespace std;
9
10  int perteneceA(int maxiteration, complex<float> complexnum) {
11      // INPUT: numero complejo inicial y numero de iteraciones,
            OUTPUT: 0 o 1.
12      // Esta funcion calcula la sucesion mandelbrot y retorna 1 si
            la iteracion se encuentra en el conjunto 0 cero en e caso
```

```
        contrario
13      float realx, imagx, realc, imagc;
14      // se inicializa un numero complejo igual al entregado a la
            funcion
15      complex<float> xn = complexnum;
16      // se calcula la parte real e imaginaria
17      realc = real(complexnum);
18      imagc = imag(complexnum);
19      for (int iteration = 0; iteration<maxiteration; iteration++){
20          // se comienza a iterar sobre la sucesion
21          xn = pow(xn,2);
22          // se calcula la parte real e imaginaria de c
23          realx = real(xn);
24          imagx = imag(xn);
25          // se suma xn^2 + c
26          xn = decltype(xn)((float) realc + realx, (float) imagc +
                imagx);
27          // se calcula la norma de la sucesion y se retorna 0 o 1
28          if( abs(xn) > 2 ) return 0;
29      }
30      return 1;
31  }
32
33  int main(){
34      // se define el numero de iteraciones, threads y la cuenta de
            los valores que se encuentran en determinada area
35      int max_n, count,threads;
36      // se define la parte real e imaginaria de un numero c
            cualquiera
37      float real,imag, max_points;
38      // se definen los tiempos de inicio y termino del algoritmo
39      double start,end,times;
40      // se calcula el tiempo de inicio y se setean valores para
            comenzar el algoritmo
41      start =  omp_get_wtime();
42      count = 0;
43      threads = 24;
44      max_points = 10000000;
45      max_n = 1640;
46      // se genera un vector por cada thread, para que cada uno pueda
            trabajar sobre una slice especifica y no tener condicion de
            carrera
47      int* array = (int*) calloc(threads, sizeof(int));
48      // se instancia un numero complejo cualquiera "c"
49      complex<float> complexnum;
50      // se crea un vector de numeros complejos entre -2 y 2 de forma
            aleatoria
51      complex<float>* c = (complex<float>*) calloc(max_points,sizeof(
            complex<float>));
52      for(int p = 0; p < max_points; ++p){
53              real = (float)rand() / (float)RAND_MAX;
```

```
54                imag = (float)rand() / (float)RAND_MAX;
55                complexnum = decltype(complexnum)((float)(real * 4) -
                      2, (float)(imag * 4) - 2);
56                c[p] = complexnum;
57                }
58        // se paraleliza la funcion perteneceA definida anteriormente
             con el vector de numeros complejos creado recien
59        #pragma omp parallel for num_threads(threads)
60        for (int i= 0; i < max_points; ++i){
61                int id = omp_get_thread_num();
62                array[id] += perteneceA(max_n, c[i]);
63                }
64        // se suman los resultados de cada thread para encontrar el
             area final
65        for (int k=0;k<threads;k++) {
66            count+= array[k];
67        }
68        // se calcula el tiempo que tardo el algoritmo
69        end =  omp_get_wtime();
70        times = end - start;
71        // se muestran los resultados obtenidos
72        printf("La aproximacion de Pixel Counting es : %f\n", (count /
             max_points) * 16);
73        printf("Este proceso demoro %f segundos.\n", times);
74        free(array);
75        free(c);
76        return count;
77    }
```

En base a todo lo anterior se obtuvieron los resultados mostrados en la tabla 1, donde se alcanza una aproximación de 1,506589 y el mejor tiempo se obtiene usando 4 procesos, los que demoran 99,9704 segundos.

Numero de procesos	1	2	4	6	8	16	24
Pixel counting	1,506589	1,506589	1,506589	1,506589	1,506589	1,506589	1,506589
Tiempo	248,6892	131,2616	99,9704	101,3451	103,0770	101,5190	101,2974

Tabla Nº1: Desempeño Pixel Counting según numero de procesos