Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Grado en Ingeniería de Computadores
COMPUTACIÓN DE ALTAS PRESTACIONES

Práctica 2-2

Implementación del método de Gauß-Seidel

OBJETIVO

Implementar el *método iterativo de Gauß-Seidel*. Este método, junto con el de Jacobi, están en la categoría de los métodos estacionarios, que permiten resolver de forma iterativa un sistema de ecuaciones lineales simultáneas.

Método de Jacobi

Dado un sistema lineal $Ax = B$, donde A y B son matriz $n \times n$ y vector de longitud n, la i-ésima ecuación se escribe

$$\sum_{j=1}^{n} a_{ij} x_j = b_i.$$

Si, en esa ecuación, despejamos la variable x_i suponiendo el resto de los valores de x fijados, tenemos

$$x_i = \frac{b_i - \sum_{j \neq i} a_{ij} x_j}{a_{ii}}. \tag{1}$$

La ecuación (1) sugiere el siguiente método iterativo, conocido como *método de Jacobi*:

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}}{a_{ii}}.$$

Este método es inmediatamente paralelizable, pues el cómputo de cada x_i en la iteración k-ésima depende exclusivamente de los valores de x en la iteración anterior.

Método de Gauß-Seidel

El método de Jacobi converge muy lentamente por lo que se han propuesto algunas variantes que resuelven el sistema con un número menor de iteraciones. Entre ellas está el *método de Gauß-Seidel*. Si procedemos igual que en el método de Jacobi pero evaluando las componentes de x en la iteración k-ésima por orden y usando los valores de x tan pronto como están disponibles, llegamos a la ecuación siguiente:

$$x_i^{(k)} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)}}{a_{ii}}. \tag{2}$$

Observemos que este método no es paralelizable en principio, ya que la evaluación de una componente de x depende de la evaluación previa de las componentes de x. Además el orden en que esta evaluación se realice modificará el valor de los componentes.

En la práctica, puesto que el método es iterativo, dos soluciones obtenidas cambiando el orden estarán tan cerca una de otra como se quiera: basta incrementar el número de iteraciones.

El método de Gauß-Seidel será aplicado en nuestro caso para resolver una una instancia de la ecuación diferencial de Laplace:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,$$

donde $u(x,y)$ es una función diferenciable al menos dos veces en un abierto $\Omega \subset \mathbb{R}^2$, y con la condición de contorno $u(x,y) = g(x,y)$ para $(x,y) \in \partial\Omega$, siendo g una función conocida, tal que $\nabla^2 g = 0$. Si discretizamos la función u en el dominio Ω, generando por ejemplo una malla de $(n+2) \times (n+2)$ puntos con un paso h, y aplicamos un desarrollo en serie de Taylor, obtenemos la siguiente aproximación de u en un punto de la malla, (x_i, y_j):

$$-4u(x_i,y_j) + u(x_i+h,y_j) + u(x_i-h,y_j) + u(x_i,y_j+h) + u(x_i,y_j-h) = 0, \tag{3}$$

en donde h, el paso de la malla, es un valor pequeño.

Si denotamos $x_i \pm h = x_{i\pm1}$ e $y_j \pm h = y_{j\pm1}$, podremos escribir la ecuación (3) como

$$u(x_i,y_j) = \frac{1}{4}\left(u(x_{i+1},y_j) + u(x_{i-1},y_j) + u(x_i,y_{j+1}) + u(x_i,y_{j-1})\right).$$

Observemos que para $1 \leq i \leq n$, $1 \leq j \leq n$, tenemos un sistema de n^2 incógnitas, a saber, los valores $u(x_i,y_j)$, al que podemos aplicar fácilmente el método de Gauß-Seidel. En efecto, en total analogía con la ecuación (2), podemos reescribir la ecuación anterior como

$$u^{(k)}(x_i,y_j) = \frac{1}{4}\left(u^{(k-1)}(x_{i+1},y_j) + u^{(k)}(x_{i-1},y_j) + u^{(k-1)}(x_i,y_{j+1}) + u^{(k)}(x_i,y_{j-1})\right).$$

En nuestro caso, vamos a considerar un código secuencial, que nos proporcionan para resolver una instancia del anterior problema. Observando el código, es fácil darse cuenta de que se trata de descomponer el dominio $\Omega = [-10,10] \times [-10,10] \subset \mathbb{R}^2$ en una malla de, como máximo, 200×200 y la función que se usa para calcular las condiciones de contorno es $g(x,y) = x^2 - y^2$ (en realidad, podemos usar cualquier función $g^*(x,y)$ para las condiciones de contorno con tal que $\nabla^2 g^* = 0$).

Nuestro objetivo es paralelizar dicho código usando el paradigma de memoria compartida.

DESARROLLO DE LA PRÁCTICA

Presentarse en la máquina

Como en las demás prácticas, hemos de presentarnos, en cualquiera de las máquinas y abrir un terminal alfanumérico.

Descarga del código

El código del programa puede descargarse de la página web de la asignatura, en el enlace `gauss-seidel.c`.

Edición de un programa

Para editar un programa, hay que utilizar algún editor de texto de los proporcionados en el menú de Aplicaciones.

Compilación y ejecución

La compilación se puede realizar editando el fichero `Makefile` y cambiando lo necesario de acuerdo al modelo ya existente.

Código secuencial

```
# include <stdio.h>
# include <stdlib.h>
# include <sys/time.h>
# include <sys/resource.h>
# include <unistd.h>
# include <math.h>

# define MAX     202  /* Tamaño máximo de la matriz           */
# define LAMBDA  0.25 /* Inversa del número de sumandos       */
                      /* en el cálculo de la nueva componente */
# define COORDMAX    10.0
# define COORDMIN   -10.0
# define INTERNO      0.0

struct  rusage   ru_init, ru_end;
double           a[MAX][MAX], TOL;
int              n;

double g(double x, double y) { return x*x - y*y; }

void Inicializa()
{
int i,j;
double h = (COORDMAX - COORDMIN)/(double)(n+1);

   for (i = 0; i < n+2; i++)
      for (j = 0; j < n+2; j++)
         a[i][j] = INTERNO;

   for (i=0; i<n+2; i++)
   {
      a[0][i]   = g(COORDMIN, COORDMIN + (double)i*h);
      a[n+1][i] = g(COORDMAX, COORDMIN + (double)i*h);
      a[i][0]   = g(COORDMIN + (double)i*h, COORDMIN);
      a[i][n+1] = g(COORDMIN + (double)i*h, COORDMAX);
   }
}

void PrintData(FILE *fpc)
{
int i, j;
double h = (COORDMAX - COORDMIN)/(double)(n+1);

   /* Imprime los resultados de salida */
   for (i = 1; i < n+1; i++)
   {
      for (j = 1; j < n+1; j++)
         fprintf(fpc, "%.3f %.3f %.3f\n", COORDMIN + (double)i*h,
                                          COORDMIN + (double)j*h,
                                          a[i][j]);
      fprintf(fpc, "\n");
   }
}

void GaussSeidel()
{
int    i, j, flag = 0;
double diff = 0.0, temp;
  /* Algoritmo Gauß-Seidel */
   do
   {
      diff = 0.0;
      flag = 0;
      for (i = 1; i < n+1; i++)
      {
         for (j = 1; j < n+1; j++)
         {
            temp    = a[i][j]; /* Guarda los valores antiguos */

            a[i][j] = LAMBDA*(a[i][j-1] + a[i-1][j] +
                              a[i][j+1] + a[i+1][j]);
            diff += fabs(a[i][j] - temp);
```

```
         }
      }
      if (diff/n/n > TOL)
         flag++;
   } while (flag != 0);
}

void InitTime() { getrusage(RUSAGE_SELF, &ru_init); }

void PrintTime()
{
double elapsed_time = 0.0;
int    tv_sec       = 0;
int    tv_usec      = 0;

   getrusage(RUSAGE_SELF, &ru_end);

   if (ru_init.ru_utime.tv_usec > ru_end.ru_utime.tv_usec)
   {
      ru_end.ru_utime.tv_usec += 1000000;
      --ru_end.ru_utime.tv_sec;
   }

   tv_sec  = ru_end.ru_utime.tv_sec  - ru_init.ru_utime.tv_sec;
   tv_usec = ru_end.ru_utime.tv_usec - ru_init.ru_utime.tv_usec;

   elapsed_time = (double)tv_sec + (double)tv_usec/1.0e6;
   fprintf(stdout, "Tiempo de cálculo (modo user) = %.3lf segundos.\n", elapsed_time);

   if (ru_init.ru_stime.tv_usec > ru_end.ru_stime.tv_usec)
   {
      ru_end.ru_stime.tv_usec += 1000000;
      --ru_end.ru_stime.tv_sec;
   }

   tv_sec  = ru_end.ru_stime.tv_sec  - ru_init.ru_stime.tv_sec;
   tv_usec = ru_end.ru_stime.tv_usec - ru_init.ru_stime.tv_usec;

   elapsed_time = (double)tv_sec + (double)tv_usec/1.0e6;
   fprintf(stdout, "Tiempo de cálculo (modo system) = %.3lf segundos.\n", elapsed_time);
}

int main(int argc, char **argv)
{
FILE *fpc = NULL;

/* obtener el tamaño de la matriz */
   if (argc != 4  &&  argc != 3)
   {
      fprintf(stderr, "Usage: %s <filas> <convergencia (ej: 0.001)> [<archivo>]\n", argv[0]);
      exit(1);
   }
   n   = atoi(argv[1]);
   TOL = atof(argv[2]);

   InitTime();
   Inicializa();
   GaussSeidel(); /* Cálculo de la aproximación */
   PrintTime();

   if (argc == 4)
   {
      if ((fpc = fopen(argv[3], "w")) == NULL)
         fpc = stdout;
   }
   else
      fpc = stdout;

   PrintData(fpc);
   return 0 ;
}
```

Fases de la paralelización

Una vez analizado el código, es claro que no se puede paralelizar de inmediato debido a las dependencias en el bucle que actualiza los valores de la matriz `a[i][j]`. Por ello, la estrategia debe ser ignorar parcialmente esa dependencia.

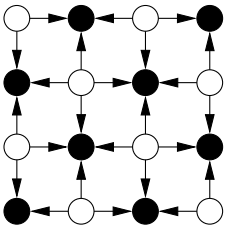

Figura 1: Se actualizan los puntos negros

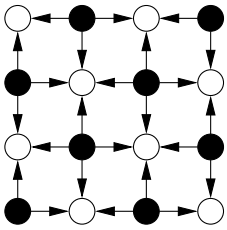

Figura 2: Se actualizan los puntos blancos

Para ello, nos imaginamos que los elementos de la matriz están marcados de colores blanco y negro, a modo de tablero de ajedrez, para que alrededor de cada punto negro sólo haya blancos y viceversa. Así las cosas, podemos dividir la tarea de actualizar la matriz en dos fases: primero actualizamos los puntos negros (Figura 1) y luego los blancos (Figura 2).

Obsérvese que estas dos tareas sí que son totalmente paralelizables, al no existir dependencia alguna de datos. Si la matriz es de tamaño $n \times n$, entonces las fases actúan sucesivamente, actualizando cada una $n^2/2$ puntos. Cada una de las fases es susceptible de descomposición en tantas tareas como puntos, es decir, $n^2/2$. Ahí es donde debemos usar el paradigma de memoria compartida para realizar la paralelización.

La siguiente figura presenta gráficamente los resultados exactos y experimentales, obtenidos mediante el código secuencial, usando una malla de 100×100 y una tolerancia de 0,0001. Obsérvese que la función harmónica de contorno elegida ha sido $g(x,y) = x^2 - y^2$.

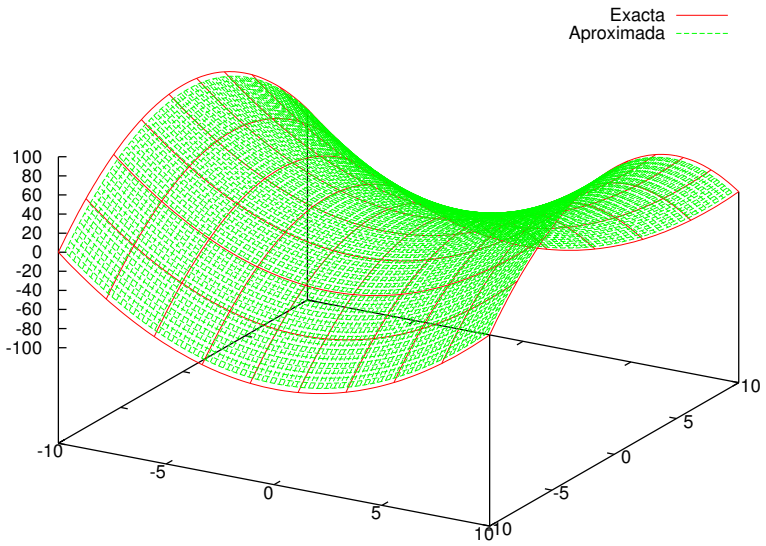

Figura 3: Resultados de la ejecución