Grai2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Laura Sánchez Parra
Grupo de prácticas: Miércoles
Fecha de entrega: 25 Abril 2018
Fecha evaluación en clase: 25 Abril 2018

Ejercicios basados en los ejemplos del seminario práctico

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? (añada una captura de pantalla que muestre lo que ocurre) **(b)** Resuelva el problema generado sin eliminar `default(none)`.Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA:

La cláusula default(none) obliga al programador a especificar el alcance de cada una de las variables del fragmento de código paralelizado. Como veremos, el alcance de la variable a sí que está especificado mediante la clausula “shared(a)” pero sin embargo, no conocemos el alcance de la variable n.

CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
main(){
  int i, n = 7;
  int a[n];

  for (i=0; i<n; i++)
  a[i] = i+1;

  #pragma omp parallel for shared(a) default(none)
  for (i=0; i<n; i++)
    a[i] += i;

  printf("Después de parallel for:\n");

  for (i=0; i<n; i++)
    printf("a[%d] = %d\n",i,a[i]);
}
```

CAPTURAS DE PANTALLA:

Se produce el siguiente error:

```
laura@laura-portatil:~/Escritorio/Uni/AC/practica3$ gcc -O2 -fopenmp shared-clau
se-modificado.c
shared-clause-modificado.c:5:1: warning: return type defaults to ‘int’ [-Wimplic
it-int]
 main(){
 ^~~~
shared-clause-modificado.c: In function ‘main’:
shared-clause-modificado.c:12:11: error: ‘n’ not specified in enclosing ‘paralle
l’
   #pragma omp parallel for shared(a) default(none)
           ^~~
shared-clause-modificado.c:12:11: error: enclosing ‘parallel’
```

Como ya habíamos deducido, el compilador de openMP no conoce el alcance de la variable n y por tanto produce un error.

ara resolver el problema sin eliminar el default(none), solo tenemos que indicarle al compilador el alcance de la variable de entorno n, de la siguiente manera:

```
1  #include <stdio.h>
2  #ifdef _OPENMP
3  #include <omp.h>
4  #endif
5  main(){
6    int i, n = 7;
7    int a[n];
8
9    for (i=0; i<n; i++)
10   a[i] = i+1;
11
12   #pragma omp parallel for shared(a,n) default(none)
13   for (i=0; i<n; i++)
14     a[i] += i;
15
16   printf("Después de parallel for:\n");
17
18   for (i=0; i<n; i++)
19     printf("a[%d] = %d\n",i,a[i]);
20 }
```

Al intentar compilar nuestro código:

```
laura@laura-portatil:~/Escritorio/Uni/AC/practica3$ gcc -O2 -fopenmp shared-clau
se-arreglado.c
shared-clause-arreglado.c:5:1: warning: return type defaults to ‘int’ [-Wimplici
t-int]
 main(){
 ^~~~
```

Comprobamos que no hay errores, el warning que aparece se debe al tipo devuelto del main, que no tiene ningún tipo de relación con las cláusulas o con OpenMP.

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? (inicialice suma a un valor distinto de 0

dentro y fuera de `parallel`) Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA:

El código no produce fallo a la hora de compilar, sin embargo, si inicializamos dentro y fuera del bucle, el valor que se utiliza es el que definimos dentro del código paralelizado.
Si quitamos la inicialización dentro del bucle, vamos a ver que el resultado puede no ser correcto. Esto se debe a que la cláusula private no toma como propios los valores de las variables de entorno privadas que se hayan inicializado previamente, por tanto, si no se redefine dentro de la sección paralelizada, se inicializará con basura.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado.c`

```
aura/Escritorio/Uni/AC/practica3/private-clause.c
#include <omp.h>
  #else
#define omp_get_thread_num() 0
  #endif

main(){
  int i, n = 7;
  int a[n], suma;
  for(i=0; i<n; i++){
    a[i] = i;
  }

  suma=1;
  printf("suma inicial:%d \n",suma);
  #pragma omp parallel private(suma)
  {
    suma=10;
    printf("suma private:%d \n",suma);
    #pragma omp for
    for(i=0; i<n; i++){
        suma = suma + a[i];
        printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
      }
    printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
  }
}
```

CAPTURAS DE PANTALLA:

```
laura@laura-portatil:~/Escritorio/Uni/AC/practica3$ ./private-clause
suma inicial:1
suma private:10
thread 1 suma a[2] / thread 1 suma a[3] / suma private:10
thread 0 suma a[0] / thread 0 suma a[1] / suma private:10
thread 2 suma a[4] / thread 2 suma a[5] / suma private:10
thread 3 suma a[6] /
* thread 0 suma= 11
* thread 3 suma= 16
* thread 1 suma= 15
* thread 2 suma= 19laura@laura-portatil:~/Escritorio/Uni/AC/practica3$
```

3. ¿Qué ocurre si en `private-clause.c` se elimina la cláusula `private(suma)`? ¿A qué cree que es debido?

RESPUESTA: Eliminar la cláusula private hace que la variable suma sea compartida por todas las hebras. A nivel práctico esto influirá en los resultados, pues el valor de suma se va actualizando en cada iteración de bucle, de manera que el valor de suma compartido sea, al final, el de la suma acumulada.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado3.c`

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
  #else
#define omp_get_thread_num() 0
  #endif

main(){
  int i, n = 7;
  int a[n], suma;
  for(i=0; i<n; i++){
    a[i] = i;
  }

  suma=1;
  printf("suma inicial:%d \n",suma);
  #pragma omp parallel
  {
    suma=10;
    printf("suma private:%d \n",suma);
    #pragma omp for
    for(i=0; i<n; i++){
        suma = suma + a[i];
        printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
      }
    printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
  }
}
```

CAPTURAS DE PANTALLA:

```
laura@laura-portatil:~/Escritorio/Uni/AC/practica3$ ./private-clause
suma inicial:1
suma private:10
thread 0 suma a[0] / thread 0 suma a[1] / suma private:10
thread 2 suma a[4] / thread 2 suma a[5] / suma private:10
thread 1 suma a[2] / thread 1 suma a[3] / suma private:10
thread 3 suma a[6] /
* thread 0 suma= 31
* thread 1 suma= 31
* thread 2 suma= 31
* thread 3 suma= 31laura@laura-portatil:~/Escritorio/Uni/AC/practica3$
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta.

RESPUESTA: Recordemos como funciona cada una de las cláusulas que se usan en este programa. Primeramente, la cláusula lastprivate almacena en la variable de nombre suma externa al código paralelizado la versión privada que ejecuta la iteración final. Por otra parte, firstprivate, especifica en cada subproceso que la variable privada suma de cada uno de ellos se inicializa con el mismo valor que tiene la variable suma antes de la región paralelizada. Sabiendo esto, deducimos que al realizarse siempre la mismas operaciones, el valor de suma externo final variará si modificamos el valor de suma inicial (antes de la región paralelizada).

CAPTURAS DE PANTALLA:

```c
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

main() {
  int i, n = 7;
  int a[n], suma=0;
  for (i=0; i<n; i++)
    a[i] = i;

  #pragma omp parallel for firstprivate(suma) lastprivate(suma)
    for (i=0; i<n; i++)
    {
      suma = suma + a[i];
      printf(" thread %d suma a[%d] suma=%d \n",omp_get_thread_num(),i,suma);
    }
  printf("\nFuera de la construcción parallel suma=%d\n",suma);
}
```

Al crear el ejecutable y correr el programa se produce la siguiente salida:

```
laura@laura-portatil:~/Uni/AC/practica3$ gcc -O2 -fopenmp firstlastprivate-clause.c -o firstlastprivate-clause
firstlastprivate-clause.c:8:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main() {
 ^~~~
laura@laura-portatil:~/Uni/AC/practica3$ ./firstlastprivate-clause
 thread 1 suma a[2] suma=2
 thread 1 suma a[3] suma=5
 thread 3 suma a[6] suma=6
 thread 0 suma a[0] suma=0
 thread 0 suma a[1] suma=1
 thread 2 suma a[4] suma=4
 thread 2 suma a[5] suma=9

Fuera de la construcción parallel suma=6
laura@laura-portatil:~/Uni/AC/practica3$
```

Cambiamos el valor de suma antes de la región paralelizada:

```c
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

main() {
  int i, n = 7;
  int a[n], suma=10;
  for (i=0; i<n; i++)
    a[i] = i;

  #pragma omp parallel for firstprivate(suma) lastprivate(suma)
    for (i=0; i<n; i++)
    {
      suma = suma + a[i];
      printf(" thread %d suma a[%d] suma=%d \n",omp_get_thread_num(),i,suma);
    }
  printf("\nFuera de la construcción parallel suma=%d\n",suma);
}

```

Veamos que se modifica el valor de la variable suma final a 16.

```
laura@laura-portatil:~/Uni/AC/practica3$ gcc -O2 -fopenmp firstlastprivate-claus
e.c -o firstlastprivate-clause
firstlastprivate-clause.c:8:1: warning: return type defaults to ‘int’ [-Wimplici
t-int]
 main() {
 ^~~~
laura@laura-portatil:~/Uni/AC/practica3$ ./firstlastprivate-clause
 thread 0 suma a[0] suma=10
 thread 0 suma a[1] suma=11
 thread 3 suma a[6] suma=16
 thread 2 suma a[4] suma=14
 thread 2 suma a[5] suma=19
 thread 1 suma a[2] suma=12
 thread 1 suma a[3] suma=15

Fuera de la construcción parallel suma=16
laura@laura-portatil:~/Uni/AC/practica3$
```

5. ¿Qué se observa en los resultados de ejecución de `copyprivate-clause.c` cuando se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

RESPUESTA:
Si eliminaos la cláusula copyprivate(a), el valor de la variable a no se copia en cada una de las instancias de a en los subprocesos, sino que se asigna a la instancia de la hebra en la que se haya inicializado la variable, como en nuestro bucle cada hebra realiza dos asignaciones, entonces dos hebras estarán inicializadas al valor de a.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```c
#include <stdio.h>
#include <omp.h>

int main() {
int n = 9, i, b[n];
for (i=0; i<n; i++)
b[i] = -1;
#pragma omp parallel
{ int a;
#pragma omp single copyprivate(a)
{
  printf("\nIntroduce valor de inicialización a:");
  scanf("%d", &a );
  printf("\n Single ejecutada por el thread %d \n",omp_get_thread_num());
}
#pragma omp for
for (i=0; i<n; i++) b[i] = a;
}
printf("Depués de la región parallel:\n");
for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
printf("\n");
}
```

CAPTURAS DE PANTALLA:

Ejecución sin modificar:

```
laura@laura-portatil:~/Uni/AC/practica3$ cc -O2 -fopenmp copyprivate-clause.c -o
 copyprivate-clause
laura@laura-portatil:~/Uni/AC/practica3$ ./copyprivate-clause

Introduce valor de inicialización a:10

 Single ejecutada por el thread 3
Depués de la región parallel:
b[0] = 10       b[1] = 10       b[2] = 10       b[3] = 10       b[4] = 10       b
[5] = 10        b[6] = 10       b[7] = 10       b[8] = 10
laura@laura-portatil:~/Uni/AC/practica3$
```

Ejecución tras la modificación:

```
laura@laura-portatil:~/Uni/AC/practica3$ cc -O2 -fopenmp copyprivate-modificado.
c -o copyprivate-modificado
laura@laura-portatil:~/Uni/AC/practica3$ ./copyprivate-modificado

Introduce valor de inicialización a:10

 Single ejecutada por el thread 2
Depués de la región parallel:
b[0] = 0        b[1] = 0        b[2] = 0        b[3] = 0        b[4] = 0        b
[5] = 10        b[6] = 10       b[7] = 0        b[8] = 0
laura@laura-portatil:~/Uni/AC/practica3$
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado

RESPUESTA:

Como podemos observar, al cambiar el valor de suma de 0 a 10, independientemente del número de iteraciones que haga el bucle, la suma se incrementa en 10 unidades. Esto se debe a que en la cláusula reduction el valor inicial de la variable se suma al final de haber "ejecutado" la acción de reducir. En este caso, tras sumas las n veces indicadas por el bucle, se suman 10 unidades a suma.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```c
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
 int main(int argc, char **argv) {
int i, n=20, a[n],suma=10;
if(argc < 2) {
fprintf(stderr,"Falta iteraciones\n");
exit(-1);
}
n = atoi(argv[1]);
if (n>20) {
  n=20;
 printf("n=%d",n);
}
for (i=0; i<n; i++) a[i] = i;
#pragma omp parallel for reduction(+:suma)
for (i=0; i<n; i++) suma += a[i];
printf("Tras 'parallel' suma=%d\n",suma);
}
```

CAPTURAS DE PANTALLA:

```
laura@laura-portatil:~/Uni/AC/practica3$ ./reduction-clause 2
Tras 'parallel' suma=1
laura@laura-portatil:~/Uni/AC/practica3$ ./reduction-clause 10
Tras 'parallel' suma=45
laura@laura-portatil:~/Uni/AC/practica3$ cc -O2 -fopenmp reduction-modificado.c
-o reduction-modificado
laura@laura-portatil:~/Uni/AC/practica3$ ./reduction-modificado 2
Tras 'parallel' suma=11
laura@laura-portatil:~/Uni/AC/practica3$ ./reduction-modificado 10
Tras 'parallel' suma=55
```

7. En el ejemplo `reduction-clause.c`, elimine `reduction()` de `#pragma omp parallel for reduction(+:suma)` y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector `a` en paralelo sin usar directivas de trabajo compartido .

RESPUESTA:

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

CAPTURAS DE PANTALLA:

Resto de ejercicios

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1 (implemente una versión para variables globales y otra para variables dinámicas, use una de estas versiones en los siguientes ejercicios):

$$v2 = M \bullet v1;\ v2(i) = \sum_{k=0}^{N-1} M(i,k) \bullet v(k),\ i = 0,\ldots N-1$$

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código paralelo que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CAPTURA CÓDIGO FUENTE: `pmv-secuencial.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include<time.h>


#define VECTOR_GLOBAL
//#define VECTOR_DYNAMIC

#ifdef VECTOR_GLOBAL
#define MAX 32768
double v[MAX], m[MAX][MAX], r[MAX];
#endif

int main(int argc,char** argv){

  if  (argc<2){
    printf("Faltan nº componentes del vector \n");
    exit(-1);
  }


  struct  timespec cgt1,cgt2;
  double  ncgt;   //para tiempo de ejecución
  int i, j;
  unsigned int N = atoi(argv[1]);   // Máximo N =2^32  -1=4294967295

  #ifdef VECTOR_GLOBAL
  if (N>MAX)
    N=MAX;
  #endif


  #ifdef VECTOR_DYNAMIC
  double *v, **m, *r;
  v = (double*) malloc(N*sizeof(double));
  m = (double**) malloc(N*sizeof(double*));
  for (i=0; i<N; i++)
    m[i] = (double*) malloc(N*sizeof(double));
  r = (double*) malloc(N*sizeof(double));
  if  ((v==NULL) || (m==NULL) || (r==NULL)) {
    printf("Error en la reserva de espacio para los vectores\n");
    exit(-2);
  }
  #endif

  //Inicializar vector y matriz
  #pragma omp parallel for
  for  (i=0; i<N; i++) {
    v[i] = N*0.1+  i*0.1;
    for (j=0; j<N; j++)
      m[i][j] = v[i]*0.1+j*0.1;
  }
```

```
53
54    clock_gettime(CLOCK_REALTIME,&cgt1);
55    //Calcular el producto
56    int sum;
57    for  (i=0; i<N; i++) {
58      sum = 0;
59      for (j=0; j<N; j++)
60        sum += m[i][j]*v[j];
61      r[i] = sum;
62    }
64    clock_gettime(CLOCK_REALTIME,&cgt2);
65    ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) +
66         (double) ((cgt2.tv_nsec - cgt1.tv_nsec)/(1.e+9));
67
68    //Imprimir resultado del producto
69    printf("\n Resultado:\n");
70    #ifdef  PRINT_ALL
71    for  (i=0; i<N; i++) {
72      printf("\t%f", r[i]);
73    }
74    printf("\n");
75    #else
76    printf("Primer valor: %f \t Último valor: %f \n", r[0], r[N-1]);
77    #endif
78    printf("\n Tiempo de ejecución(s): %11.9f\n", ncgt);
79
80    #ifdef VECTOR_DYNAMIC
81    free(v);        // libera el espacio reservado para v
82    free(m);        // libera el espacio reservado para m
83    free(r);
84    #endif
85    return  0;
86  }
```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for` . Debe implementar dos versiones del código (consulte la lección 5/Tema 2):

    a. una primera que paralelice el bucle que recorre las filas de la matriz

    b. una segunda que paralelice el bucle que recorre las columnas.

    Use las directivas que estime oportunas y las cláusulas que sean necesarias **excepto la cláusula `reduction`**. Se debe paralelizar también la inicialización de las matrices. Respecto a este ejercicio:

    - Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
    - Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CAPTURA CÓDIGO FUENTE : `pmv-OpenMP-a.c`

```
25    #ifdef VECTOR_DYNAMIC
26    double *v, **m, *r;
27    m = (double**) malloc(N*sizeof(double*));
28    v = (double*) malloc(N*sizeof(double));
29
30    for (i=0; i<N; i++)
31      m[i] = (double*) malloc(N*sizeof(double));
32    r = (double*) malloc(N*sizeof(double));
33    if  ((v==NULL) || (m==NULL) || (r==NULL)) {
34      printf("Error en la reserva de espacio para los vectores\n");
35      exit(-2);
36    }
37    #endif
38
39
40    #pragma omp parallel for
41    for  (i=0; i<N; i++) {
42      v[i] = N*0.1+  i*0.1;
43      for (j=0; j<N; j++)
44        m[i][j] = v[i]*0.1+j*0.1;
45    }
```

CAPTURA CÓDIGO FUENTE: `pmv-OpenMP-b.c`

```
clock_gettime(CLOCK_REALTIME,&cgt1);
//Calcular el producto
int sum_local;
for (i=0; i<N; i++) {
  r[i] = 0;
  #pragma omp parallel
  {
    sum_local = 0;
    #pragma omp for
    for (j=0; j<N; j++)
      sum_local += m[i][j]*v[j];
    #pragma omp atomic
      r[i] += sum_local;
  }
}

clock_gettime(CLOCK_REALTIME,&cgt2);
ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) +
     (double) ((cgt2.tv_nsec - cgt1.tv_nsec)/(1.e+9));
```

10. A partir de la segunda versión de código paralelo desarrollado en el ejercicio anterior, implementar una versión paralela del producto matriz por vector con OpenMP que use para comunicación/sincronización la cláusula `reduction`. Respecto a este ejercicio:
    - Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
    - Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

    CAPTURA CÓDIGO FUENTE: `pmv-OpenmMP-reduction.c`

```
64          clock_gettime(CLOCK_REALTIME,&cgt1);
65           //Calcular el producto
66           int sum;
67           for (i=0; i<N; i++) {
68             sum = 0;
69             #pragma omp parallel for reduction(+:sum) private(j)
70             for (j=0; j<N; j++)
71               sum += m[i][j]*v[j];
72             r[i] = sum;
73           }
74
75          clock_gettime(CLOCK_REALTIME, &cgt2);
```