Algoritmos Paralelos

(clase 03.11.15)

Prof. J.Fiestas

Ejemplo 1: hello_omp.cpp

```
#include <iostream>
#include <omp.h>
using namespace std;

int main ()
{
  int nthreads = 4;
  omp_set_num_threads(nthreads);

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    cout<<"Hello World from thread = "<<id<<endl;;
    cout<<" with "<<omp_get_num_threads()<<" threads"<<endl;
  }
  cout<<"all done, with hopefully "<< nthreads<<" threads"<<endl;;
}
```

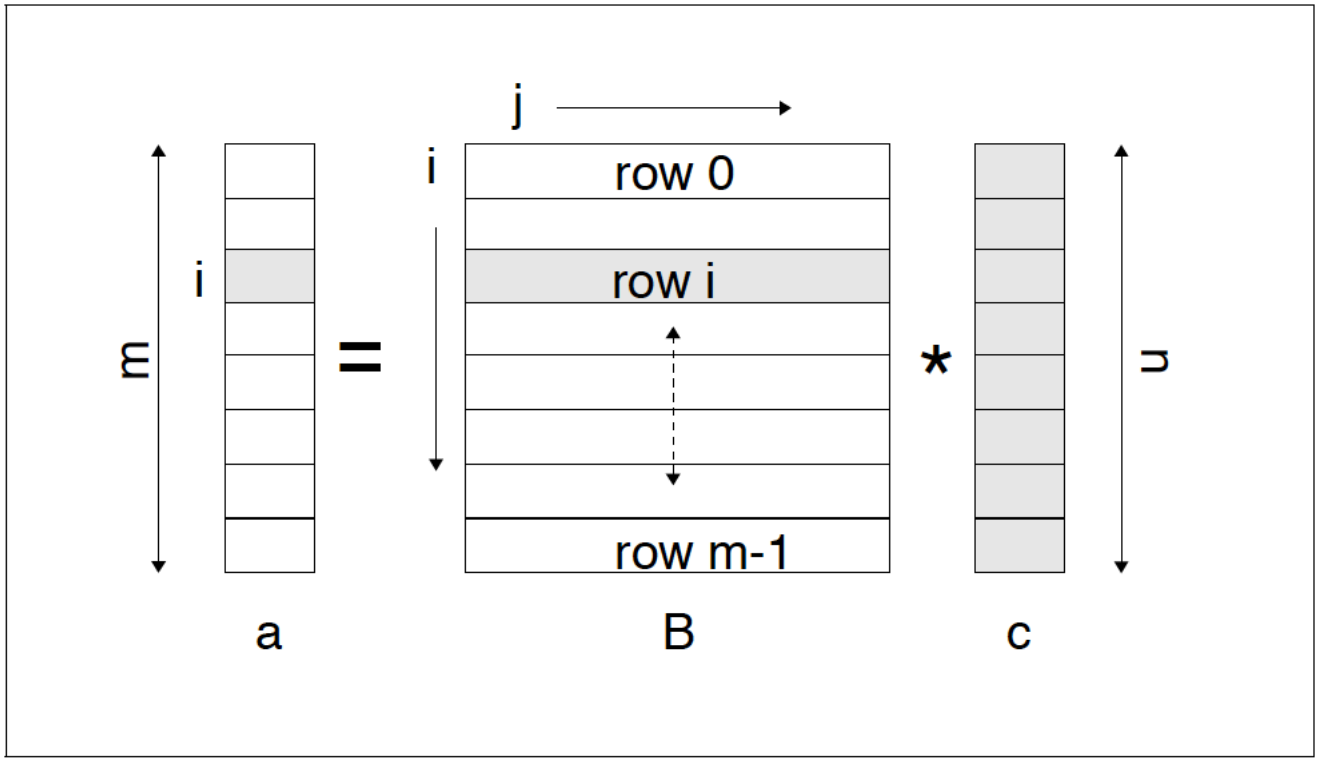
Ejemplo 2: conditional_omp.cpp

```
#include <iostream>
using namespace std;
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif
```

Utilizar misma fuente para serie y paralelo con #ifdef _OPENMP

```
int main(int argc, char *argv[])
{
  int TID = omp_get_thread_num();
  cout<<"Hilo maestro es "<<TID<<endl;
#ifdef _OPENMP
      omp_set_num_threads(4);
#endif
cout<<"Hilos paralelos: "<<endl;
#pragma omp parallel
{
  int TID = omp_get_thread_num();
  cout<<“\t”<<TID<<endl;
}
}
```

Ejemplo 3: mxv.cpp



$$a_i = \sum_{j=1}^{n} B_{i,j} * c_j \qquad i = 1, \ldots, m$$

Ejemplo 3: mxv.cpp

```
#include <iostream>
using namespace std;
#define NMAX 1000
#define MMAX 1000

int main(int argc, char *argv[])
{
  double a[NMAX],b[NMAX*MMAX],
c[NMAX];
  int i, j, m, n;
  cout<<"Ingresar m: "<<endl;
  cin>>n;
  cout<<"Ingresar n: "<<endl;
  cin>>m;
  cout<<endl;
```

```
cout<<"Initializando matriz B y vector c"<<endl;
  for (j=0; j<n; j++)
    c[j] = 2.0;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      b[i*n+j] = i;

  for (i=0; i<m; i++)
  {
    a[i] = 0.0;
    for (j=0; j<n; j++)
      a[i] += b[i*n+j]*c[j];
  }
}
```

Ejercicio 1: mxv.cpp

Decidir qué region(es) del código serán paralelizadas. Y utilizar #pragma omp parallel for para definirlas

Decidir que variables son compartidas (entre **a,b,c,m,n**).
Util para aumentar rendimiento, y evitar errores. Pero hay que pensar que variables son compartidas o privadas

DEFAULT:

Especifica el alcance por defecto de todas las variables en toda region en paralelo

default (none)

SHARED:

Declara variables en una lista para ser compartidas entre los threads
Una variable compartida existe solo en una locación de memoria, cuya dirección pueden leer todos los threads.

shared *(m,n,a,b,c,)*

PRIVATE:

Un nuevo objeto del mismo tipo es declarado para cada thread
Toda referencia al objeto original es reemplazado con referencias al nuevo objeto
Variables privadas no estan originalmente inicializadas para cada thread

private *(i,j)*

Ejercicio 2: for_loops.cpp

Paralelizar for_loops.cpp en una región en paralelo

#pragma omp parallel

con variables compartidas y privadas, que contenga dos regiones

#pragma omp for

una para cada bucle

```
#include <iostream>
using namespace std;
int main()
{
  int i, n = 9;
  int a[n], b[n];

  for (i=0; i<n; i++)
  {   a[i] = i;
      cout<<"i: "<<i<<", a_i: "<<a[i]<<endl;
  }

  for (i=0; i<n; i++)
  {
    b[i] = 2 * a[i];
      cout<<"i: "<<i<<", b_i: "<<b[i]<<endl;
  }
}
```

Ejercicio 2: for_loops.cpp

Usar
#pragma omp single
Para imprimir el número de hilos usado en cada bucle, antes de iniciarlo
#pragma omp for
una para cada bucle

Añadir una cláusula
... schedule(runtime)
a cada bucle for y comentar el resultado

```
#include <iostream>
using namespace std;
int main()
{
  int i, n = 9;
  int a[n], b[n];

  for (i=0; i<n; i++)
  {   a[i] = i;
      cout<<"i: "<<i<<", a_i: "<<a[i]<<endl;
  }

  for (i=0; i<n; i++)
  {
    b[i] = 2 * a[i];
     cout<<"i: "<<i<<", b_i: "<<b[i]<<endl;
  }
}
```

Ejercicio 2: for_loops.cpp

Imprimir
Hilo ... ejecuta iteración ...
en cada bucle

¿Cómo logra que la impresión sea ordenada?

```
#include <iostream>
using namespace std;
int main()
{
  int i, n = 9;
  int a[n], b[n];

  for (i=0; i<n; i++)
  {   a[i] = i;
      cout<<"i: "<<i<<", a_i: "<<a[i]<<endl;
  }

  for (i=0; i<n; i++)
  {
    b[i] = 2 * a[i];
      cout<<"i: "<<i<<", b_i: "<<b[i]<<endl;
  }
}
```

Ejemplo 4: sections

Simple aplicación de sections

```
void funcionA()
{
#pragma omp critical
      cout<<"Esta seccion es ejecutada en
la funcionA por el hilo
"<<omp_get_thread_num()<<endl;
}
void funcionB()
{
#pragma omp critical
cout<<"Esta seccion es ejecutada en la
funcionB por el hilo
"<<omp_get_thread_num()<<endl;
}
```

```
int main()
{
#ifdef _OPENMP
  omp_set_num_threads(4);
#endif
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      funcionA();
    #pragma omp section
      funcionB();
  } /*-- Fin del bloque section --*/
} /*-- Fin del bloque section --*/
}
```

Ejercicio 3: sections_omp.cpp

Paralelizar sections.cpp
Definir una sección para **ini()**
y otra para **suma()**
Asegurarse que la suma se calcule luego que el array haya sido inicializada, e.g. con una variable logica que sea TRUE solo si se ejecuta la inicialización, tal que se ejecute la suma si es FALSE

```
int main()
{
  double sum;
  double A[N];

  ini(N, A);
  sum = suma(N, A);
  cout<<"la suma es: "<<sum<<endl;
}
```

Ejercicio 3: sections_omp.cpp

Retorna el tiempo en segundos de un tiempo arbitrario en el pasado, pero constante durante la ejecución

```
double omp_get_wtime( );
```

Utilizar omp_get_wtime() para calcular el tiempo de ejecución. Asegurarse que el tiempo se mida por un solo hilo, e.g. Usando
#pragma omp master

Ejemplo 5: single_omp.cpp

```
int main()
{ ...
  for (i=0; i<n; i++)
      b[i] = -1;
#pragma omp parallel shared(a,b) private(i)
{
  #pragma omp single { // si no se usa, todos los hilos usan memoria para a: ineficiente
    a = 10;
    cout<<"constructor Single ejecutado por el hilo "<<omp_get_thread_num()<<endl;
  } // la barrera implicita de single, garantiza que a=10 para todos los hilos
  #pragma omp for
  for (i=0; i<n; i++)
      b[i] = a;
}
cout<<"Fin de la region en paralelo: "<<endl;
  for (i=0; i<n; i++)
  cout<<"b["<<i<<"]"<<"= "<<b[i]<<endl;
}
```

Ejercicio 4: single_omp.cpp

Paralelize en ejemplo 5
sections_omp.cpp
usando
#pragma omp single
y
#pragma omp for
mida y compare tiempos de
ejecución para los ejemplos
5 y 6

Ejercicio 5: Variables compartidas/privadas y reducción

Reescribir el siguiente programa con solo las directivas OPENMP siguientes:
#pragma omp parallel , antes de inicializar a=0
#pragma omp for reduction (+,a) , antes del for
Observar el resultado de la suma luego de varios intentos. Se mantiene el resultado?

```
#include <iostream>
int main () {
  int a, i;
  a = 0;.
  for (i = 0; i < 10; i++) {
        a += i;
    }
Cout<<"La suma es:
"<<a<<endl;
 }
}
```

Ejercicio 5: Variables compartidas/privadas y reducción

Reescribir el programa con las directivas de sincronización OPENMP adecuadas, tal que:
a sea una variable compartida shared(a), e
i sea una privada private(i)
La inicialización de a se haga solo por el
master thread
La impresión de la suma la haga solo un thread
directiva single
Compruebe si el resultado es ahora correcto
En caso contrario, donde sería adecuado incluir
una barrera? #pragma omp barrier
Escribir un programa similar para hallar el mínimo y el máximo de un array de 10 elementos generados aleatoriamente entre 1 y 50