

Algoritmos Paralelos

Ejemplos prácticos de OPENMP (2)
(clase 17.11.15)

Prof. J.Fiestas

Constructor master:

Especifica un bloque que es ejecutado por el *master thread*, dentro de la región en paralelo correspondiente.

En el ejemplo, el *master* controla el número de iteraciones e imprime un reporte de avance de la ejecución. Los demás *threads* evitan la región *master* sin necesidad de esperar

```
#pragma omp master new-line
structured-block
```

void master_example

```
( float* x, float* xold, int n, float tol ) {
c = 0;
#pragma omp parallel {
do{
#pragma omp for private(i)
for( i = 1; i < n-1; ++i ){
xold[i] = x[i]; }
#pragma omp single {
toobig = 0; }

#pragma omp for private(i,y,error)
reduction(+:toobig)
for( i = 1; i < n-1; ++i ){
y = x[i];
x[i] = average( xold[i-1], x[i], xold[i+1] );
error = y - x[i];
if( error > tol || error < -tol ) ++toobig; }

#pragma omp master {
++c;
Printf("iteration %d, toobig=%d\n", c, toobig );
}

}while( toobig > 0 );
} }
```

Constructor **critical**:

Restringe la ejecución del bloque a un *thread* a la vez. Influencia a todos los *threads* dentro de la región en paralelo. Se le puede asignar un nombre. Un *thread* esperará el inicio de una región crítica, hasta que no haya *thread* ejecutando esa región.

En el ejemplo, una tarea es sacada de la cola (`dequeue`) y trabajada. La tarea debe ser crítica para evitar que la misma tarea se execute por diferentes threads. Aquí, dos colas independientes se nombran *xaxis* e *yaxis*.

```
int dequeue(float *a);
void work(int i, float *a);
void critical_example(float *x, float *y)
{
    int ix_next, iy_next;
    #pragma omp parallel shared(x, y)
    private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
        ix_next = dequeue(x);
        work(ix_next, x);
        #pragma omp critical (yaxis)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```

**#pragma omp critical [(name)] new-line
structured-block**

Regiones **barrier** dependientes

Un constructor **barrier** enlazará la región en paralelo mas inmediata. Todos los *threads* deben ejecutar **barrier** y las tareas previas antes de poder continuar.

En el ejemplo, el barrier en *sub3* esta ligada a la región paralela en *sub2*.

```
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}
```

```
void work(int n) {}
void sub3(int n) {
    work(n);
    #pragma omp barrier
    work(n);
}
void sub2(int k) {
    #pragma omp parallel shared(k)
    sub3(k);
}
void sub1(int n) {
    int i;
    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
```

Constructor **atomic**:

Asegura que la locación de memoria será accedida atomicamente, evitando lectura o escritura simultánea de la misma, lo que podría conducir a valores erróneos. **Atomic** fuerza acceso exclusivo con respecto a otras regiones **atomic** que acceden a la misma locación de memoria **x**, entre todos los *threads* del programa, independientemente de los bloques a los que pertenecen

```
#pragma omp atomic capture new-line  
structured-block
```

Constructor **atomic**:

```
#pragma omp atomic [read | write | update | capture ] new-line  
expression-stmt
```

La cláusula **read** fuerza una lectura atómica de la ubicación de almacenamiento de **x**

La cláusula **write** fuerza una escritura atómica en la ubicación de almacenamiento de **x**

La cláusula **update** fuerza una actualización atómica en la ubicación de almacenamiento de **x**

La cláusula **capture** fuerza una actualización atómica en la ubicación de almacenamiento de **x**, a la vez que la captura del valor original o final de **x**, con respecto al **atomic update**

Constructor **atomic**:

En el ejemplo, se evitan condiciones de competencia (actualización simultánea de **x** por múltiples *threads*) usando el constructor **atomic**. Aquí, **atomic** permite updates de dos diferentes elementos en paralelo. Si se usara **critical**, los updates se ejecutarían en serie (e incluso en un orden indeterminado)

```
float work1(int i) {
    return 1.0 * i;
}

float work2(int i) {
    return 2.0 * i;
}

void atomic_example(float *x, float *y, int
*index, int n) {
    int i;
    #pragma omp parallel for shared(x, y, index,
n)
    for (i=0; i<n; i++) {
        #pragma omp atomic update
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```

Constructor **atomic**:

Note que las directivas **atomic** se aplican solo a los bloques que le siguen. Como consecuencia, los elementos de **y** no son actualizados atómicamente en este ejemplo.

```
int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;
    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }
    for (i = 0; i < 1000; i++)
        x[i] = 0.0;
    atomic_example(x, y, index,
    10000);
    return 0;
}
```

Constructor **ordered**

Especifica un bloque en un ciclo que será ejecutado en el orden de las iteraciones en el ciclo, mientras el programa se ejecuta en paralelo fuera de la región

```
#pragma omp ordered new-line  
structured-block
```

ordered influencia todos los *threads* de la región, en el ciclo interior inmediato. Mientras regiones **ordered** en distintos ciclos se ejecutan independientemente. Cuando un *thread* ejecutando la primera iteración encuentra un constructor **ordered**, ingresa sin esperar. Cuando un *thread* ejecutando una iteración posterior encuentra una región **ordered**, espera a que las iteraciones anteriores han sido completadas.

El ciclo perteneciente a una región **ordered** debe poseer una cláusula **ordered** declarada en el constructor correspondiente

Constructor y cláusula ordered

Constructores **ordered** son útiles para ordenar secuencialmente la salida de una región en paralelo. El ejemplo imprime los índices en orden secuencial.

```
#include <stdio.h>
void work(int k)
{
#pragma omp ordered
printf(" %d\n", k);
}
void ordered_example(int lb, int ub, int stride)
{
int i;
#pragma omp parallel for
ordered schedule(dynamic)
for (i=lb; i<ub; i+=stride)
work(i);
}
int main() {
ordered_example(0, 100, 5);
return 0;
}
```

Constructor y cláusula ordered

Es posible tener múltiples constructores **ordered** en un ciclo con la cláusula **ordered** declarada.

Sin embargo, en este caso todas las iteraciones ejecutan dos regiones **ordered**, por lo que el código **no es válido**.

Una iteración en un ciclo no debe ejecutar mas de una región **ordered**.

```
void work(int i) {}  
void ordered_wrong(int n)  
{  
    int i;  
    #pragma omp for ordered  
    for (i=0; i<n; i++) {  
        /* incorrect because an iteration may  
        not execute more than one ordered region */  
        #pragma omp ordered  
        work(i);  
        #pragma omp ordered  
        work(i+1);  
    }  
}
```

Constructor y cláusula ordered

Es posible tener múltiples constructores **ordered** en un ciclo con la cláusula **ordered** declarada.

Sin embargo, en este caso todas las iteraciones ejecutan dos regiones **ordered**, por lo que el código **no es válido**.

Una iteración en un ciclo **for** debe ejecutar mas de una región **ordered**.

```
void work(int i) {}  
void ordered_wrong(int n)  
{  
    int i;  
    #pragma omp for ordered  
    for (i=0; i<n; i++) {  
        /* incorrect because an iteration may  
        not execute more than one ordered region */  
        #pragma omp ordered  
        work(i);  
        #pragma omp ordered  
        work(i+1);  
    }  
}
```



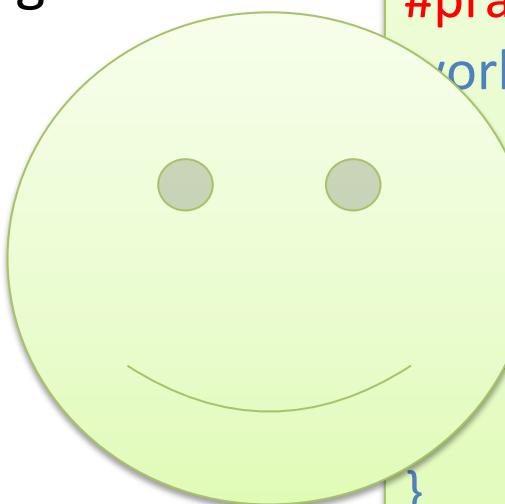
Constructor y cláusula ordered

Este código (con más de un constructor **ordered**) si es válido, ya que cada iteración ejecutará solo una región **ordered**

```
void work(int i) {}  
void ordered_good(int n)  
{  
    int i;  
    #pragma omp for ordered  
    for (i=0; i<n; i++) {  
        if (i <= 10) {  
            #pragma omp ordered  
            work(i);  
        }  
        if (i > 10) {  
            #pragma omp ordered  
            work(i+1);  
        }  
    }  
}
```

Constructor y cláusula ordered

Este código (con más de un constructor **ordered**) si es válido, ya que cada iteración ejecutará solo una región **ordered**



```
void work(int i) {}  
void ordered_good(int n)  
{  
    int i;  
    #pragma omp for ordered  
    for (i=0; i<n; i++) {  
        if (i <= 10) {  
            #pragma omp ordered  
            work(i);  
        }  
        if (i > 10) {  
            #pragma omp ordered  
            work(i+1);  
        }  
    }  
}
```

Directiva **threadprivate**:

Especifica que variables serán replicadas, con cada thread teniendo su propia copia

```
#pragma omp threadprivate(list) new-line
```

Cada copia de una variable **threadprivate** es inicializada una vez

Se usa para mantener una variable global en cada thread durante la ejecución de regiones en paralelo

Algoritmos Paralelos

```
#include <omp.h>
int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x);
main () {
/* Explicitly turn off dynamic threads */
omp_set_dynamic(0);
printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid) {
tid = omp_get_thread_num();
a = tid;
b = tid;
x = 1.1 * tid +1.0;
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */
printf("Master thread doing serial work here\n");
printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid) {
tid = omp_get_thread_num();
printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */ }
```

THREADPRIVATE:

Se usa para mantener una variable global en cada thread durante la ejecución de regiones en paralelo

#pragma omp threadprivate (list)

1st Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 2: a,b,x= 2 2 3.200000

Thread 3: a,b,x= 3 3 4.300000

Thread 1: a,b,x= 1 1 2.100000

Master thread doing serial work here

2nd Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 3: a,b,x= 3 0 4.300000

Thread 1: a,b,x= 1 0 2.100000

Thread 2: a,b,x= 2 0 3.200000

Directiva **threadprivate**:

En el ejemplo, cada *thread* tendrá un contador independiente.

```
int counter = 0;  
#pragma omp threadprivate(counter)  
int increment_counter()  
{  
    counter++;  
    return(counter);  
}
```

Uso de **threadprivate** en una variables estática

```
int increment_counter_2()  
{  
    static int counter = 0;  
    #pragma omp threadprivate(counter)  
    counter++;  
    return(counter);  
}
```

Directiva **threadprivate**:

En el ejemplo se muestra el comportamiento incierto de la inicialización de una variable **threadprivate**.

Esta se inicializa una vez en un punto aleatorio antes de su primera referencia.

Ya que **a** es construída usando el valor de **x** (que es modificado por **x++**), el valor de **a.val** al inicio de la región paralela puede ser **1 o 2**.

Este problema es evitado para **b**, que usa una declaración como **const** y un constructor copia.

```
class T {  
public:  
    int val;  
    T (int);  
    T (const T&);  
};  
T :: T (int v){  
    val = v;  
}  
T :: T (const T& t) {  
    val = t.val;  
}  
void g(T a, T b){  
    a.val += b.val;  
}
```

Directiva **threadprivate**:

En el ejemplo se muestra el comportamiento incierto de la inicialización de una variable **threadprivate**.

Esta se inicializa una vez en un punto aleatorio antes de su primera referencia.

Ya que **a** es construída usando el valor de **x** (que es modificado por **x++**), el valor de **a.val** al inicio de la región paralela puede ser **1 o 2**. Este problema es evitado para **b**, que usa una declaración como **const** y un constructor copia.

```
int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)
```

```
void f(int n) {
    x++;
    #pragma omp parallel for
    /* In each thread:
     * a is constructed from x (with value 1 or 2?)
     * b is copy-constructed from b_aux
     */
    for (int i=0; i<n; i++) {
        g(a, b); /* Value of a is unspecified. */
    }
}
```

Cláusula **private**

private(list)

Declara uno o más elementos de lista como privados para una tarea (*task*)

Cada *task* que referencia un elemento **private** recibe una lista copia del original con referencias a su nueva lista. El valor original podrá cambiar si se accesa por punteros.

Una variable que es parte de otra (como un *array* o *structure*) no puede ser parte de la lista de una cláusula **private**

Cláusula **private**

En el ejemplo, los valores de la lista de elementos originales **i** y **j** se mantienen al salir de la region paralela, mientras que los elementos privados **i** y **j** son modificados dentro del constructor parallel.

```
#include <stdio.h>
#include <assert.h>
int main()
{
    int i, j;
    int *ptr_i, *ptr_j;
    i = 1;
    j = 2;
    ptr_i = &i;
    ptr_j = &j;
#pragma omp parallel private(i) firstprivate(j)
{
    i = 3;
    j = j + 2;
    assert (*ptr_i == 1 && *ptr_j == 2);
}
assert(i == 1 && j == 2);
return 0;
}
```

Cláusula **private**

En el ejemplo, los usos de la variable **a** en el ciclo en la función **f()** utilizan el elemento **a** de la lista, mientras que es incierto si las referencias a **a** en la función **g()** pertenecen a un elemento privado u original.

```
int a;

void g(int k) {
    a = k; /* Accessed in the region but outside
             * of the construct; therefore unspecified whether
             * original or private list item is modified. */
}

void f(int n) {
    int a = 0;
    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {
        a = i;
        g(a*2); /* Private copy of "a" */
    }
}
```

Cláusula **private**

En el ejemplo, la lista de la cláusula **private** en un constructor **parallel** puede también aparecer en una cláusula **private** en un constructor anidado, lo que resulta en una copia adicional de la variable.

```
#include <assert.h>
void priv_example3()
{
    int i, a;
    #pragma omp parallel private(a)
    {
        a = 1;
        #pragma omp parallel for private(a)
        for (i=0; i<10; i++)
        {
            a = 2;
        }
        assert(a == 1);
    }
}
```

Cláusula **firstprivate**

firstprivate(list)

Declara una lista privada a una tarea e inicializa cada elemento de la lista con el valor correspondiente de la variable original al momento que el constructor es inicializado. Se aplican restricciones de la cláusula private.

Cláusula **lastprivate**

lastprivate(list)

Declara una lista privada a una tarea e inicializa cada elemento de la lista con el valor correspondiente de la variable original actualizada luego del final de la región. Se aplican restricciones de la cláusula private.

Cláusula **firstprivate**

En el ejemplo, se muestra el tamaño y valor de la lista de elementos de un array o puntero en una cláusula **firstprivate**. El tamaño del elemento es el del original. El tipo de ...

- A: array (de dos vectores de dos enteros)
- B: puntero a array de n ints
- C: puntero a entero
- D: array (de dos vectores de dos enteros)
- E: array de n vectores de n enteros

Los nuevos elementos de tipo *array* son inicializados con los valores de las variables originales, mientras que las variables de tipo puntero apuntan a la variable original.

```
#include <assert.h>
int A[2][2] = {1, 2, 3, 4};
void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];
    assert(n >= 2);
    E[1][1] = 4;
#pragma omp parallel firstprivate(B, C, D, E)
{
    assert(sizeof(B) == sizeof(int (*)[n]));
    assert(sizeof(C) == sizeof(int *));
    assert(sizeof(D) == 4 * sizeof(int));
    assert(sizeof(E) == n * n * sizeof(int));
    /* Private B and C have values of original B and C. */
    assert(&B[1][1] == &A[1][1]);
    assert(&C[3] == &A[1][1]);
    assert(D[1][1] == 4);
    assert(E[1][1] == 4);
}
int main() {
f(2, A, A[0]);
return 0;
}
```

Cláusula **lastprivate**

Programas en los que la ejecución depende del valor en la última iteración, utilizan un listado de variables en una cláusula **lastprivate**, para que los valores sean equivalentes a los obtenidos en una ejecución secuencial.

```
void lastpriv (int n, float *a, float *b)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for lastprivate(i)
        for (i=0; i<n-1; i++)
            a[i] = b[i] + b[i+1];
    }
    a[i]=b[i]; /* i == n-1 here */
}
```

Cláusula reduction

Especifica un operador y una lista de elementos, para cada uno de los cuales una copia privada es creada en cada tarea (task) e inicializada para el operador. Al final de la región, el valor del elemento original es actualizado con los valores de las copias privadas utilizando el operador.

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0
max	Least representable value in the reduction list item type
min	Largest representable value in the reduction list item type

reduction(operator :list)

Cláusula reduction

El código no es correcto, ya que la inicialización (**a=0**) de la variable **a** original no está sincronizada con la actualización de **a** como resultado de la cláusula **reduction** en el ciclo **for**.

Por lo que el código imprimirá un valor errado de **a**.

Para evitar el problema, la inicialización debe hacerse antes de la actualización debido a la cláusula **reduction**. Para ello se puede añadir una directiva **barrier** luego de **a=0**, o incluir **a=0** en una directiva **single** (que tiene barrera intrínseca), o inicializar **a** antes de iniciar la región **parallel**.

```
#include <stdio.h>
int main (void)
{
    int a, i;
#pragma omp parallel shared(a) private(i)
{
#pragma omp master
    a = 0;
    // To avoid race conditions, add a barrier here.
#pragma omp for reduction(+:a)
    for (i = 0; i < 10; i++) {
        a += i;
    }
#pragma omp single
    printf ("Sum is %d\n", a);
}
```

Ejercicio 1:

Paralelizar con OpenMP el cálculo de **pi** utilizando la cláusula **reduction (+: sum)**

Imprima el valor calculado de **pi** con N=100,1000,10000,1000000

Calcule el tiempo de cálculo para N= 10^6 y np=2,4,8,16

```
for (i = 0; i < N; ++i)
{
    double x = x_0 + i * h + h/2;
    sum += sqrt(1 - x*x);
}
pi = sum * h * 4.0;
```

Ordenamiento de datos (sorting):

En C++:

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));
```

base: puntero al primer objeto del array a ser ordenada

num: número de elementos del array apuntada por base

size: tamaño en bytes de cada elemento en el array

compar: puntero a una función que compara dos elementos. Con punteros como argumentos, la función define el orden de los elementos retornando <0, 0 o >0

```
int compar (const void* p1, const void* p2);
```

```
int compareMyType (const void * a, const void * b)
{
    if ( *(MyType*)a < *(MyType*)b ) return -1;
    if ( *(MyType*)a == *(MyType*)b ) return 0;
    if ( *(MyType*)a > *(MyType*)b ) return 1;
}
```

Ordenamiento para memoria distribuída

Dados n números, por ejemplo en $[0,1]$

El algoritmo usa p bloques en 2 pasos:

- Repartir los números x en p bloques, tal que
- Ordenar los p bloques $x \in [i/p, (i+1)/p] \Rightarrow x \in (i+1)$

El costo de partición de números en p bloques es $O(n \log_2(p))$

En el mejor caso, cada bloque contiene n/p números

El costo de qsort es $O(n/p \log_2(n/p))$ por bloque

Ordenando p bloques, toma $O(n \log_2(n/p))$

El costo total es $O(n(\log_2(p)) + \log_2(n/p))$

Ordenamiento para memoria distribuída

En p procesos, todos los nodos ordenan:

- Nodo madre (root) distribuye números: proceso i recibe bloque i
- Proceso i ordena bloque i
- Proceso root colecciona bloques ordenados de otros procesos

Ya que el costo en serie es $O(n(\log_2(p)) + \log_2(n/p))$

El costo en paralelo es

$n\log_2(p)$ por enviar números en bloques

$n/p \log_2(n/p)$ por ordenar bloques

Entonces la eficiencia = $n(\log_2(p)+\log_2(n/p)) / (n(\log_2(p) + \log_2(n/p)/p))$

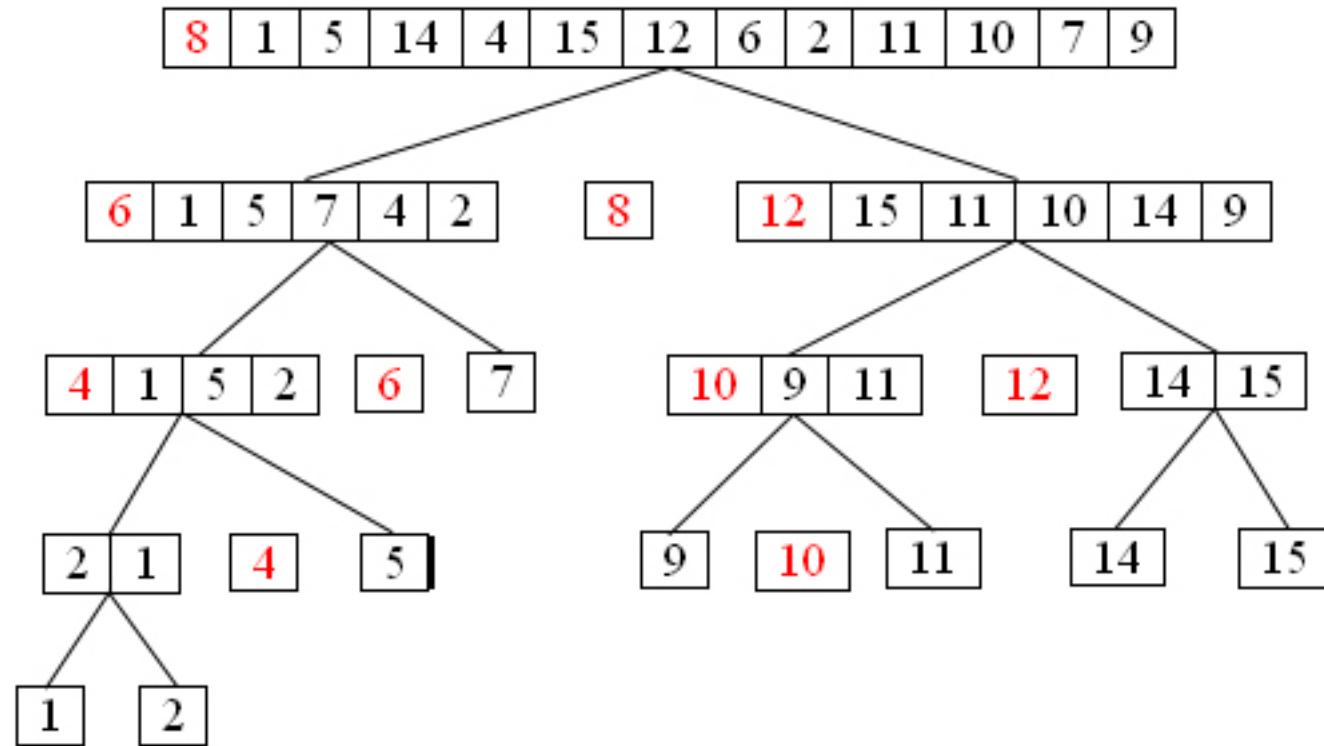
Luego de cierta álgebra ...

$$= \log_2(n) / (1/p(\log_2(n)+(1-1/p)\log_2(p)))$$

Por ejemplo, $n=2^{20}$, $\log_2(n)=20$, $p=2^2$, $\log_2(p)=2$,

$$\text{eficiencia} = 20/(5+3/2) \approx 3.1$$

Algoritmo recursivo de ordenamiento



Juntando los elementos, el arreglo quedaría ordenado

1	2	4	5	6	7	8	9	10	11	12	14	15
---	---	---	---	---	---	---	---	----	----	----	----	----

Algoritmo recursivo de ordenamiento

```
void qs(int *z, int zstart, int zend, int  
firstcall){  
    int part;  
    if(firstcall == 1) {  
        qs(z,0,zend,0);  
    } else {  
        if(zstart<zend) {  
            part = separate(z, zstart,zend);  
            qs(z, zstart, part-1,0);  
            qs(z,part+1,zend,0);  
        }  
    }  
}
```

```
int separate(int *x, int low, int high){  
    int i, pivot, last;  
    pivot = x[low];  
    swap(x+low, x+high);  
    last = low;  
    for(i = low; i<high; i++){  
        if(x[i] <= pivot){  
            swap(x+last, x+i);  
            last += 1;  
        }  
    }  
    swap(x+last, x+high);  
    return last;  
}
```

Ejercicio 2:

Programar las funciones de **quicksort** y parallelizar con OpenMP el algoritmo tomando en cuenta que:

- El algoritmo recursivo **qs()** del ejemplo puede ser ejecutado en tareas (**tasks**), dentro de un constructor **parallel**
- El condicional:

```
if(firstcall == 1) {  
    qs(z,0,zend,0); }
```

Debe ser ejecutado solo una vez (por uno de los threads)

- Los valores iniciales a ser ordenados serán generados aleatoriamente con valores entre 1 y 99
- Demostrar el funcionamiento del algoritmo imprimiendo valores para N=10 y N=20
- Medir tiempos de cálculo para N=106 y np=2,4,8 (no necesita imprimir los valores en este caso)