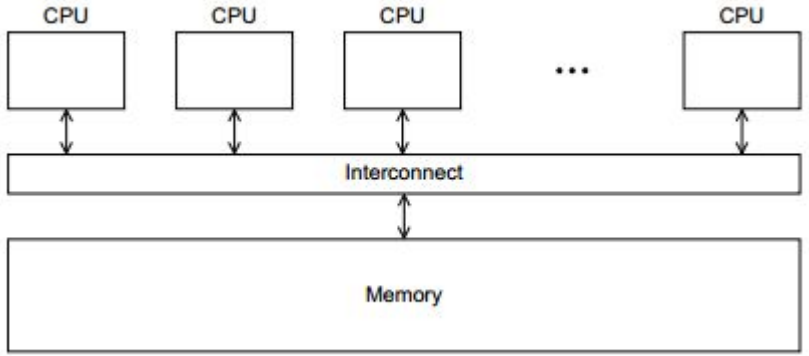
Parallel Computing

OpenMP

OpenMP

- MP -> **Multiprocessing**
- Diseñado para los sistemas donde cada thread o proceso tiene acceso a la toda la memoria disponible
- Vemos nuestro sistema como un **conjunto de núcleos donde cada uno tiene acceso a la main memory**



OpenMP

- Shared-memory system (como threads C++11 o pthread)
- La gran diferencia con pthread es que en pthread tenemos que **especificar el comportamiento** de cada thread
- OpenMP **decidimos si algún bloque** de código tiene que ser ejecutado en paralelo.
- Dejamos muchas cosas en manos del sistema y **compilador**
- **Esto implica tener un compilador que acepta OpenMP (no es el caso de MPI o Pthread)**

OpenMP

- OpenMP high-level vs Pthreads low-level
- Porque haber creado OpenMP teniendo Pthread
  - Porque programas de gran escala en pthread se vuelven bastante complejos
  - Permite paralelizar programas seriales

Objetivos

- Escribir un programa con OpenMP
- Compilar y ejecutar un pograma con OpenMP
- Paralelizar bucles for
- Entender otras características de OpenMP (task parallism, etc)
- Entender los problemas clásicos de shared-memory

Programar con OpenMP

- API basada sobre directivas
- C/C++
- Preprocessor instruccion pragmas
- Compilador puede ignorar los pragmas
- # columna 1 y pragma alineado con el código

#pragma

Programar con OpenMP

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void);  /* Thread function */
6
7  int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 #  pragma omp parallel num_threads(thread_count)
12    Hello();
13
14    return 0;
15 }  /* main */
16
17 void Hello(void) {
18    int my_rank = omp_get_thread_num();
19    int thread_count = omp_get_num_threads();
20
21    printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23 }  /* Hello */
```

Programar con OpenMP

- compilar con la opcion -fopenmp
  - $ gcc −g −Wall −fopenmp −o omp_hello omp_hello.c

- lanzar el programa
  - $ ./omp_hello 4

Programar con OpenMP

- $ ./omp_hello 4

```
10
11  #  pragma omp parallel num_threads(thread_count)
12     Hello();
```

```
17  void Hello(void) {
18     int my_rank = omp_get_thread_num();
19     int thread_count = omp_get_num_threads();
20
21     printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23  }  /* Hello */
```

- Hello from thread 0 of 4
- Hello from thread 1 of 4
- Hello from thread 2 of 4
- Hello from thread 3 of 4

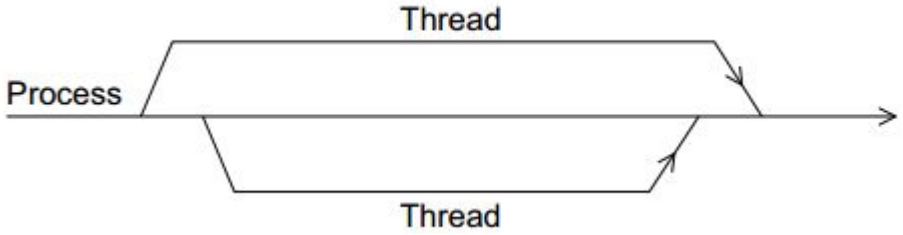
- Hello from thread 1 of 4
- Hello from thread 0 of 4
- Hello from thread 4 of 4
- Hello from thread 3 of 4

el resultado es nondeterministic.

Programar con OpenMP

```
10
11  #  pragma omp parallel num_threads(thread_count)
12     Hello();
```

- # pragma omp
- parallel
  - Especifica que el siguiente bloque siguiente bloque de código tiene que ser ejecutado de forma paralela
  - Cada thread dispone de su propio stack



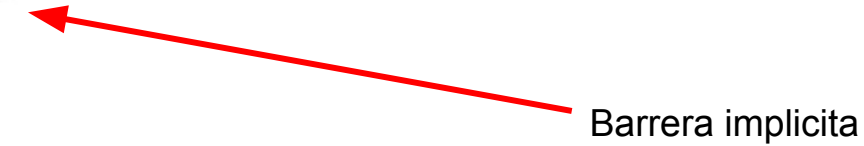
Programar con OpenMP

```
10
11  #  pragma omp parallel num_threads(thread_count)
12     Hello();
```

- num_thread puede ser agregado a parallel para definir cuántos thread queremos
- thread_count es la cantidad de thread que pedimos, estamos limitados pero un sistema clásico permite lanzar cientos o miles de threads
- Se agrega thread_count - 1 al programa al llegar a la directiva parallel
- Todos los threads se llama **team,** el principal **master,** el resto **salves ...**

Programar con OpenMP

```
10
11  #  pragma omp parallel num_threads(thread_count)
12     Hello();
```



- Barrera implícita **obliga** a los threads de **esperar a los demás hasta que todos terminen**
- Después el **master vuelve a su trabajo normal**

Error Checking

- En el ejemplo anterior deberíamos de verificar el valor de thread_count antes de llegar al pragma
- El verdadero problema es el compilador que no necesariamente será capaz de compilar con OpenMP y generará errores con el #include<omp.h>

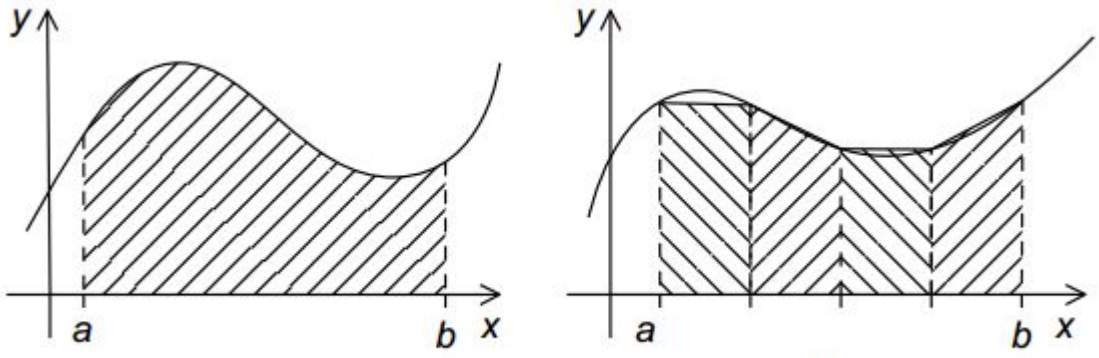
```
#ifdef OPENMP
# include <omp.h>
#endif
```

```
# ifdef OPENMP
     int my rank = omp get thread num();
     int thread count = omp get num threads();
# else
     int my rank = 0; int thread count = 1;
# endif
```

Ejemplo: la regla de trapecio

- y=f(x), a < b queremos integrar esta funcion con la regla de trapecio
- Subdividir el espacio entre a y b en n partes
- el ***h = (b − a)/n***, ***xi = a + ih, i = 0, 1,...,n***

$$T = (b-a)\frac{f(a)+f(b)}{2}.$$



Ejemplo: la regla de trapecio

- el ***h = (b − a)/n***, ***xi = a + ih***, ***i = 0, 1,...,n***

$$\int_a^b f(x)\,dx = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\frac{b-a}{n}\right)\right) + R_n(f)$$

- Approx : **h[f(x0)/2 + f(x1) + f(x2) + ⋯ + f(xn−1) + f(xn)/2].**

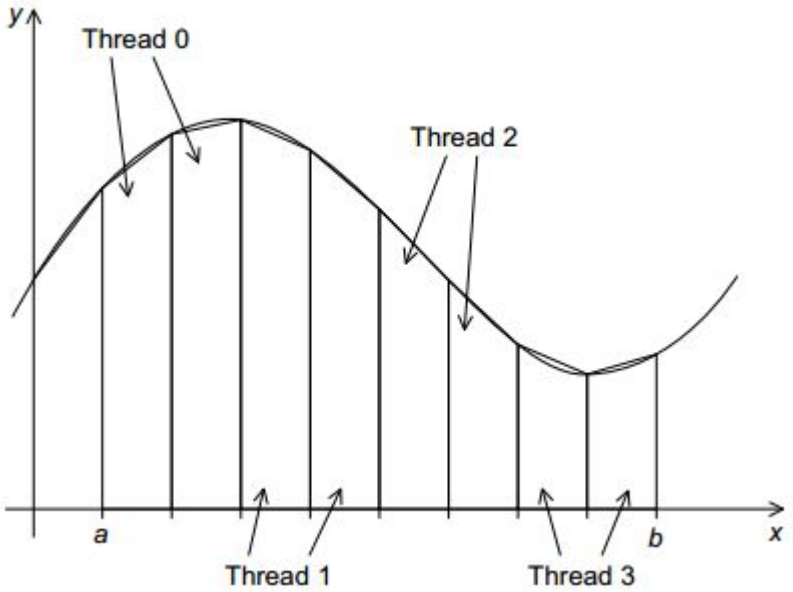
Ejemplo: la regla de trapecio

- el ***h = (b − a)/n***, ***xi = a + ih***, ***i = 0, 1,...,n***
- Approx : **h[f(x0)/2 + f(x1) + f(x2) + ⋯ + f(xn−1) + f(xn)/2].**

```
/* Input: a, b, n */
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n−1; i++) {
     x i = a + i*h;
     approx += f(x_i);
}
approx = h*approx;
```

Ejemplo: la regla de trapecio

- Ahora con OpenMP



Ejemplo: la regla de trapecio

- Dos tareas
  - Calcular las superficies de los trapecios (a)
  - Agregar las superficies a la suma global (b)
- No hay necesidad de comunicar entre las tareas (a)
- Hay necesidad de comunicar entre las tareas (b)
- Asumamos que tenemos más taprecios que de nucleos
  - Cada núcleo tendrá que hacer varios trapecios
  - Daremos un intervalo a cada core donde él aplicará la regla del trapecio
- Al final cada core debara agregar sus resultados
  - **global_result += my_result;**
  - **Error no previsible porque todos los threads querrán acceder al gloabl_result a la vez**

Ejemplo: la regla de trapecio

- Al final cada core debara agregar sus resultados
  - **global_result += my_result; <- critical section**
  - Error no previsible porque todos los threads querrán acceder al gloabl_result a la vez
  - **Race condition**

Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

Ejemplo: la regla de trapecio

- Al final cada core deberá agregar sus resultados

 # pragma omp critical
global result += my result;

- **opm critial**
  - permite pedir la exclusion mutual de los threads para acceder a este bloque de codigo

Ejemplo: la regla de trapecio

```
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     double  global_result = 0.0;
9     double  a, b;
10    int     n;
11    int     thread_count;
12
13    thread_count = strtol(argv[1], NULL, 10);
14    printf("Enter a, b, and n\n");
15    scanf("%lf %lf %d", &a, &b, &n);
16 #  pragma omp parallel num_threads(thread_count)
17    Trap(a, b, n, &global_result);
18
19    printf("With n = %d trapezoids, our estimate\n", n);
20    printf("of the integral from %f to %f = %.14e\n",
21       a, b, global_result);
22    return 0;
23 }  /* main */
```

Ejemplo: la regla de trapecio

```
25  void Trap(double a, double b, int n, double* global_result_p) {
26     double  h, x, my_result;
27     double  local_a, local_b;
28     int  i, local_n;
29     int my_rank = omp_get_thread_num();
30     int thread_count = omp_get_num_threads();
31
32     h = (b-a)/n;
33     local_n = n/thread_count;
34     local_a = a + my_rank*local_n*h;
35     local_b = local_a + local_n*h;
36     my_result = (f(local_a) + f(local_b))/2.0;
37     for (i = 1; i <= local_n-1; i++) {
38       x = local_a + i*h;
39       my_result += f(x);
40     }
41     my_result = my_result*h;
42
43  #  pragma omp critical
44     *global_result_p += my_result;
45  }  /* Trap */
```

Obtener la información de mi thread y de mi equipo

calcular la zona de trabajo de thread

Exclusion mutual

Ejemplo: la regla de trapecio

- local_n = n/thread count;
  - Cuántos trapecios tiene que calcular
- local_a = a + my_rank*local_n*h;

thread 0: `a + 0*local_n*h`
thread 1: `a + 1*local_n*h`
thread 2: `a + 2*local_h*h`

- local_b = local_a + local_n*h;

Variable Scope

- En OpenMP le scope significa los threads que tienen acceso a una misma variable dentro de bloque paralelo
- Una variable accesible por un solo thread tiene un **private scope**
  - **my_rank, thread_count asignada en el stack de cada thread**
- Una variable accesible por un equipo de threads tiene un **shared scope**
  - **global_result thread_count**
- Si declaras tu variable antes de parallel entonces **shared** sino **private**
- **OpenMP permite cambiar el scope por defecto**

It’s a trap !

- void Trap(double a, double b, int n, double* global_result_p);
- Si te gusta los punteros lo dejaras asi,
- si quieres ser más amigable lo pondras asi:
  - **double Trap(double a, double b, int n);**
  - global_result = Trap(a, b, n);
- En realidad ya no podremos realizar el cúmulo de las sumas individuales en global_result dentro de *Trap*
  - double Local_trap(double a, double b, int n);

```
     global result = 0.0;
#    pragma omp parallel num threads(thread count)
     {
#         pragma omp critical
          global_result += Local_trap(double a, double b, int n);
     }
```

It’s a trap !

```
     global result = 0.0;
#    pragma omp parallel num threads(thread count)
     {
#         pragma omp critical
          global_result += Local_trap(double a, double b, int n);
     }
```

```
     global result = 0.0;
#    pragma omp parallel num threads(thread count)
     {

          double my_res = Local_trap(double a, double b, int n);
#         pragma omp critical
          global_result +=my_res

     }
```

It’s a trap !

- Operador de reducción
  - Pasar de un array de datos a un escalar (addition, multiplication etc.)

```
     global result = 0.0;
#    pragma omp parallel num threads(thread count) \
          reduction(+: global_result)
     global_result += Local_trap(double a, double b, int n);
```

- Operador de reducción
  - **reduction(<operador>: <variable>)**

It’s a trap !

- Operador de reducción
  - tener cuidado con los float o double ya que una operación no es asociativa

Directiva *parallel for*

- Regla del trapecio

```
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n−1; i++)
        approx += f(a + i*h);
approx = h*approx;
```

- parallel for

```
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+: approx)
    for (i = 1; i <= n−1; i++)
        approx += f(a + i*h);
    approx = h*approx;
```

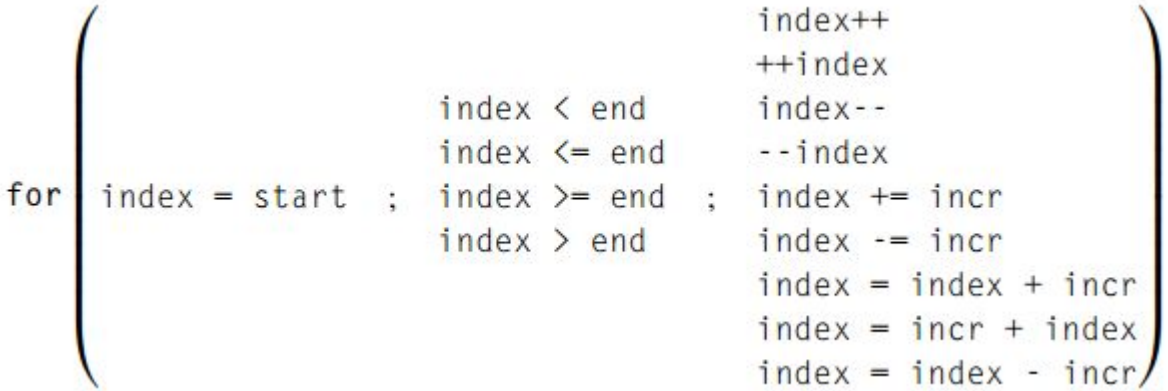
Directiva *parallel for*

- Paralelizar un bucle for
  - Distribuye las iteraciones a los diferentes threads
- Muy diferente de la directiva *parallel*
  - Distribuye un bloque codigo (tarea) a los threads
- Distribuye las iteraciones a los diferentes threads
  - m datos -> m/thread_count a cada thread
  - cada thread del grupo tiene una copia de i

Advertencias

- Parece entonces muy simple paralelizar todos los bucles for con la directiva *parallel for*
- solo **for** no **while, ni do-while**
- no funciona con un for con **break, tenemos saber cuántas iteraciones haremos**
- el contador tiene que ser **int (no float)**
- start, end, incr tienen que ser del **mismo tipo o compatible**
- start, end, incr **no tienen que cambiar** durante la ejecución
- **solamente** modificar el contador con **incr**

Advertencias



Dependencia en los datos

```
1     int Linear_search(int key, int A[], int n) {
2        int i;
3        /* thread_count is global */
4     #  pragma omp parallel for num_threads(thread_count)
5        for (i = 0; i < n; i++)
6           if (A[i] == key) return i;
7        return -1;   /* key not in list */
8     }
```

- **Line 6: error: invalid exit from OpenMP structured block**

Dependencia en los datos

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
     fibo[i] = fibo[i-1] + fibo[i-2];
```

- 1 1 2 3 5 8 13 21 34 55
- 1 1 2 3 5 8 0 0 0 0
- **unpredictable**

- **2 Threads**
  - **fibo[2], fibo[3], fibo[4], and fibo[5]**
  - **fibo[6], fibo[7], fibo[8], and fibo[9]**
- **a veces el thread 1 termina su trabajo antes que el 2 empiece**
- **a veces el thread 2 empieza mientre que los números de fibonacci siguen en 0**

Dependencia en los datos

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
     fibo[i] = fibo[i-1] + fibo[i-2];
```

- 1 1 2 3 5 8 13 21 34 55
- 1 1 2 3 5 8 0 0 0 0
- **unpredictable**

- Los compiladores **no verifican dependencia** entre los datos
- Los bucles donde hay una **interdependencia** entre las iteración **no pueden** ser correctamente paralelizados con OpenMP (**loop-carried dependence**)

Dependencia en los datos

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

- No hay problema aquí a pesar de una dependencia entre datos pero son “internos” a una iteración

Estimacion de π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

- Como vamos a resolver esto con OpenMP ?

```
1        double factor = 1.0;
2        double sum = 0.0;
3        for (k = 0; k < n; k++) {
4           sum += factor/(2*k+1);
5           factor = -factor;
6        }
7        pi_approx = 4.0*sum;
```

```
1        double factor = 1.0;
2        double sum = 0.0;
3  #     pragma omp parallel for num_threads(thread_count) \
4           reduction(+:sum)
5        for (k = 0; k < n; k++) {
6           sum += factor/(2*k+1);
7           factor = -factor;
8        }
9        pi_approx = 4.0*sum;
```

Estimacion de π

```
1        double factor = 1.0;
2        double sum = 0.0;
3  #     pragma omp parallel for num_threads(thread_count) \
4           reduction(+:sum)
5        for (k = 0; k < n; k++) {
6           sum += factor/(2*k+1);
7           factor = -factor;
8        }
9        pi_approx = 4.0*sum;
```

loop-carried dependence

- Como eliminar la dependencia ?

```
sum += factor/(2*k+1);
factor = -factor;
```

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*k+1);
```

Estimacion de π

- Siguen los problemas ...

```
1     With n = 1000 terms and 2 threads,
2        Our estimate of pi = 2.97063289263385
3     With n = 1000 terms and 2 threads,
4        Our estimate of pi = 3.22392164798593
```

```
1     With n = 1000 terms and 1 threads,
2        Our estimate of pi = 3.14059265383979
```

- factor es compartido entre los threads

Estimacion de π

```
double sum = 0.0;
pragma omp parallel for num_threads(thread_count) \
   reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
   factor = (k % 2 == 0) ? 1.0 : -1.0;
   sum += factor/(2*k+1);
}
pi_approx = 4.0*sum;
```

- La cláusula private permite crear una copia de esta variable para todos los threads

Buenas prácticas

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Ejemplo : Bubble sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

- a un array que almacena n ints
- loop carried (bucle exterior)
  - 1 itération a = 3, 4, 1, 2
  - 2 iteration a = 3,1,2,4
- loop carried (bucle interior)
  - problema del swap
- Parece aquí bien complejo quitar loop carried sin reescribir todo
  - Generalmente es difícil a veces es imposible

Odd-even transposition sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Este algoritmo es conocida por ser una versión del buble sort mas amigable a la paralelización
- a={9,7,8,6}

	Subscript in Array						
Phase	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Odd-even transposition sort

- Bucle exterior : loop-carried
  - parece ser complicado paralelizar este bucle exterior
- Boucles interiores
  - no parece haber problemas
  - Ej. fase par: i=j,i=k
    - {j,j-1} != {k,k-1}
    - Podemos entonces comparar y swap simultáneamente ambos
- Problema, tenemos que estar seguro que los threads lanzados en la fase p terminan antes de empezar la fase p+1.

	Subscript in Array						
Phase	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Odd-even transposition sort

```
1     for (phase = 0; phase < n; phase++) {
2        if (phase % 2 == 0)
3  #        pragma omp parallel for num_threads(thread_count) \
4              default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6              if (a[i-1] > a[i]) {
7                 tmp = a[i-1];
8                 a[i-1] = a[i];
9                 a[i] = tmp;
10             }
11          }
12       else
13 #        pragma omp parallel for num_threads(thread_count) \
14             default(none) shared(a, n) private(i, tmp)
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Odd-even transposition sort

- Otro problema es el overheading
  - es decir a cada fase hacemos un thread_count forks y joins
  - Es más inteligente reutilizar los threads, entonces solamente un fork.
    - **podemos hacerlo !**

Odd-even transposition sort

```
1  #  pragma omp parallel num_threads(thread_count) \
2        default(none) shared(a, n) private(i, tmp, phase)
3     for (phase = 0; phase < n; phase++) {
4        if (phase % 2 == 0)
5  #        pragma omp for
6           for (i = 1; i < n; i += 2) {
7              if (a[i-1] > a[i]) {
8                 tmp = a[i-1];
9                 a[i-1] = a[i];
10                a[i] = tmp;
11             }
12          }
13       else
14 #        pragma omp for
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Forks

utiliza los threads

Barrera implicita