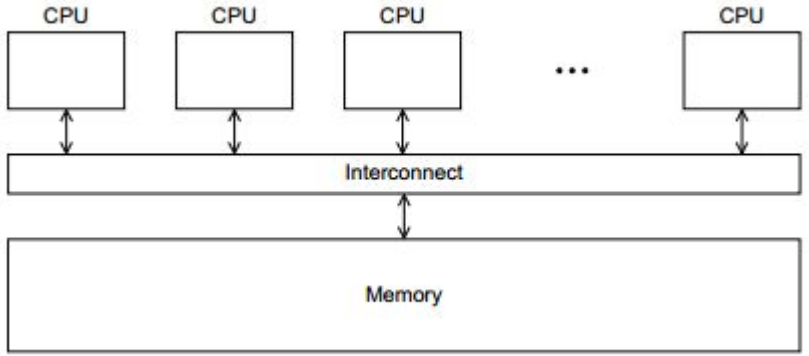
Parallel Computing

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
  - Porqué programas de gran escala en pthread se vuelven bastante complejos
  - Permite paralelizar programas secuenciales

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

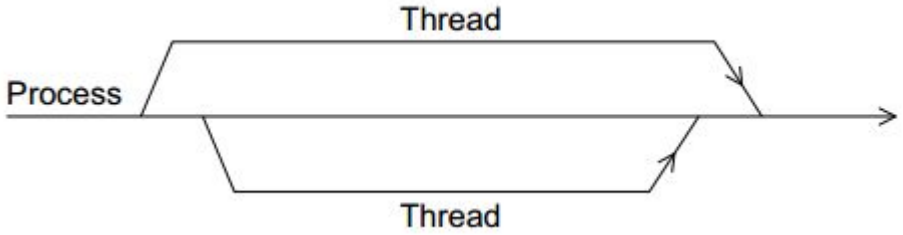
- Hello from thread 1 of 4
- Hello from thread 0 of 4
- Hello from thread 3 of 4
- Hello from thread 2 of 4

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



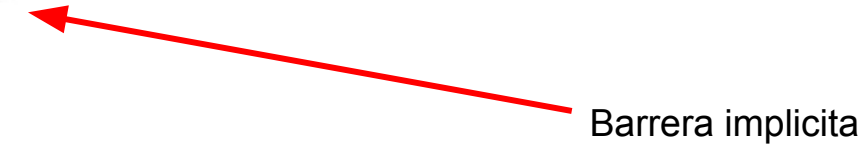
Programar con OpenMP

```
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>

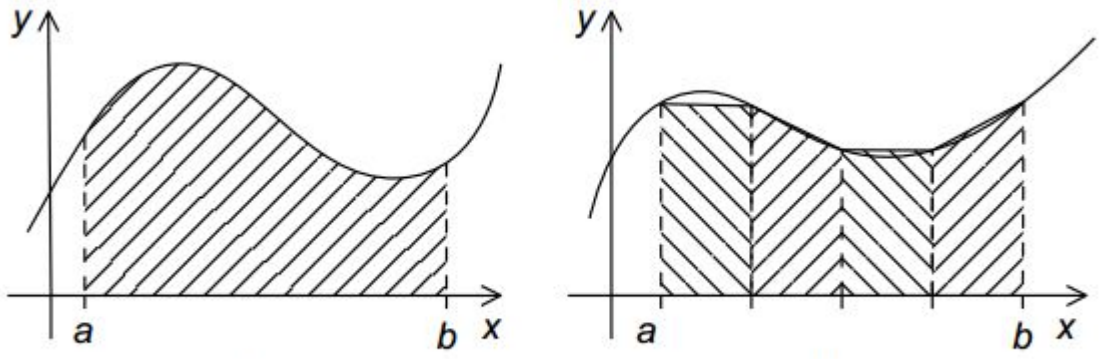
```
#ifdef OPENMP
# include <omp.h>
#endif
```

```
# 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].**

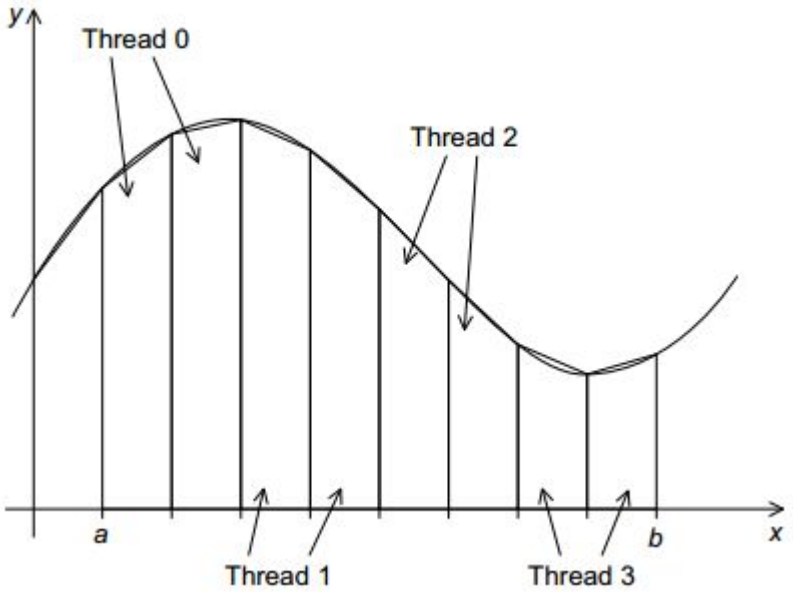
Ejemplo: la regla de trapecio

- el $h = (b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$
- Approx : $h[\, f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$.

```
/* 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 */
```

```
/* 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;
```

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;
```

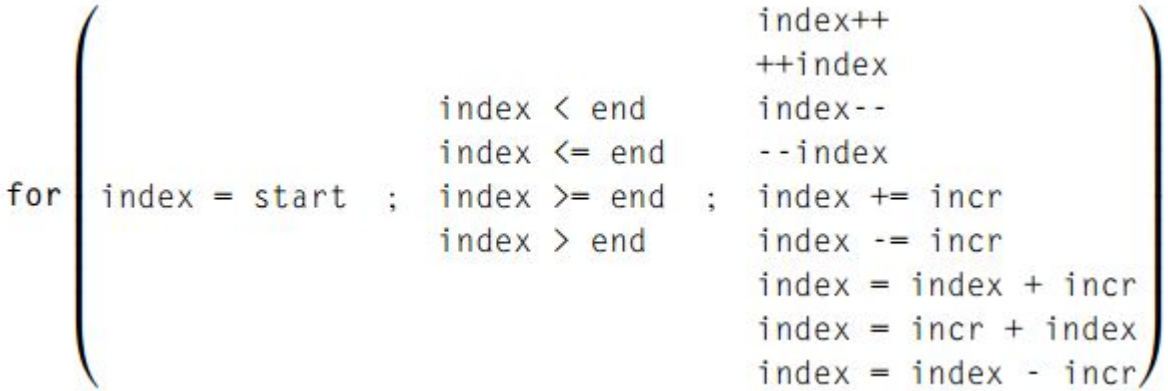
Directiva *parallel for*

- 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 iteraciónes **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

Loop Scheduling

- La cantidad de iteraciones por thread atribuida a través del *parallel for* depende del sistema
- En general es ite=n/thread_count
- A veces **no es optimal**

```
sum = 0.0;
for (i = 0; i <= n; i++)
        sum += f(i);
```

- Si la función f demora más mientras i crece

Loop Scheduling

```
sum = 0.0;
for (i = 0; i <= n; i++)
        sum += f(i);
```

- Quisiéramos hacer una atribución cíclica de las iteraciones

Thread	Iterations
0	0, n/t, 2n/t, …
1	1, n/t + 1, 2n/t + 1, …
⋮	⋮
t − 1	t − 1, n/t + t − 1, 2n/t + t − 1, …

Loop Scheduling

- si f(2i) demora dos veces f(i)
- Ej: n=10 000,
- Sin threads 3.67s
- 2 threads block 2.76s speedup 1.33
- 2 threads ciclos 1.86s 1 .99

Thread	Iterations
0	0, n/t, 2n/t, …
1	1, n/t + 1, 2n/t + 1, …
⋮	⋮
t − 1	t − 1, n/t + t − 1, 2n/t + t − 1, …

Loop Scheduling

- schedule(<type> ,chunck size)

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- tipo : static, dynamic, auto, runtime
- chunk : la cantidad de iteración por ciclo (pero depende del tipo)

Loop Scheduling

- Static
- 12 iteraciones
- schedule(static,1)

Thread 0:	0, 3, 6, 9
Thread 1:	1, 4, 7, 10
Thread 2:	2, 5, 8, 11

- f schedule(static,2)

Thread 0:	0, 1, 6, 7
Thread 1:	2, 3, 8, 9
Thread 2:	4, 5, 10, 11

Loop Scheduling

- Dynamic
  - Mismo que static pero el thread pide un nuevo chunk a cada vez que termina un chunk
  - Entonces no hay un orden predefinido
- Guided
  - La cantidad de chunk es divido por dos a cada reatribucion
  - n=10000
  - Chunck n°1 = 10000/2 -> 5000
  - Chunck n°2 = 5000/2 -> 2500
  - ...
  - Chunk n°N = 1
  - No se especifica el chuncksize
  - pero si lo haces, determinara el chuncksize minimo

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Loop Scheduling

- Runtime
  - environment variables
  - OMP_SCHEDULE
  - Puede ser para *static, dynamic, guided*
  - *$ export OMP SCHEDULE="static,1"*
- Permite modificar el comportamiento paralelo del programa sin volver a compilarlo

Loop Scheduling

- Cual escoger ?
  - Overhead
  - static < dynamic < guided
- Una solución es **experimentar** si existe una duda
- En el ejemplo anterior es muy poco probable mejorar el speedup a más de 1.99 asi que **no hay** buscar más
- Aunque si **cambiamos** la cantidad de threads deberíamos de volver a experimentar
- El **speedup** depende de la **cantidad de thread e iteraciones**
- Si cada iteración lleva a la misma cantidad de cálculos se puede dejar todo por defecto
- Si el costo por iteración incrementa deberíamos usar *static*
- Si no se sabe **experimentar** con **runtime** es lo mejor

Productores/clientes - Cola

- Elemento donde los nuevos se agregan al final (cola)
- Elementos se quiten de la cabeza
- Analogía hacia una cola de clientes
  - enqueued
  - dequeued
- Situación común en ciencia de la computación ej.
  - cola de procesos que quieren escribir en el DD.
- En aplicaciones multi-threads
  - threads productores (enqueued)
    - producen requests servidor
  - threads clientes (dequeued)
    - Cosumen requests

Productores/clientes - Message passing

- Aplicación de transmisión de mensajes
- Cada thread tiene una cola compartida de mensaje
- Cuando un thread quiere mandar un mensaje a otro
  - puede poner su mensaje en la cola de la destinación
- Un thread recibe un mensaje
  - Quitando de su cola un mensaje (cabeza)

Productores/clientes - Message passing

- Creamos un aplicación de transmisión de mensajes
- Después de crear el mensaje, el thread lo **agrega en la cola del destinatario**
- Después de mandar un mensaje un thread mira su cola **si ha recibido** un mensaje
  - **Si tiene lo quita de su cola y lo imprime**
- Los threads **alternan** entre mandar mensajes y recibir mensajes
- La cantidad de mensajes a mandar por thread es definida por el usuario
- Cuando un thread **terminó** se dedica a recibir mensajes

Productores/clientes - Message passing

```
for (sent_msgs = 0; sent_msgs < send_max; sent _msgs++) {
        Send msg();
        Try receive();
}
while (!Done())
        Try receive();
```

Productores/clientes - Sending Messages

- Acceder un cola de mensajes para poner en cola un mensaje debe ser **crítico**
  - **Varios threads pueden querer enviar un mensaje a un mismo thread**
- Si utilizamos un lista enlazada necesitamos un puntero hacia la cola para poder insertar fácilmente los mensajes (sino tendremos que ir desde la cabeza a la cola)
- Al agregar un mensaje tenemos que **poner al día** este puntero (cola)
  - Si dos threads quieren hacer eso, se podría **perder un mensaje**

```
Send_msg()
    mesg = random();
    dest = random() % thread_count;
    # pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

Productores/clientes - Receiving Messages

- Solamente un dueño de la cola quita mensajes de la cola
- Mientras existen más de 1 elemento en la cola no creará conflictos
  - Tenemos que saber **cuántos elementos** hay en la cola
  - **Conflicto** para poner al día esta variable de tamaño
  - Guardar dos variables **enqueued**, **dequeued**
  - size= enqueued - dequeued **//puede haber un pequeño error aqui**
  - **dequeued** solamente será modificado por el **dueño de cola**
  - **enqueued** solamente ser modificado por los **otros threads (critico)**

```
Receive_msg()
        queue size = enqueued − dequeued;
        if (queue size == 0) return;
        else if (queue size == 1)
#               pragma omp critical
                Dequeue(queue, &src, &mesg);
        else
                Dequeue(queue, &src, &mesg);
                Print message(src, mesg);
```

Productores/clientes - Termination detection

- Tenemos que saber cuando un thread terminó de mandar sus mensajes

```
queue_size = enqueued − dequeued;
if (queue_size == 0 && done_sending == thread_count)
        return TRUE;
else
        return FALSE;
```

- **done_sending** se incrementa después de cada for
  - cuando llega a la cantidad de thread significa que el trabajo de intercambio de mensajes se terminó

Productores/clientes - Startup

- Al inicio el master viene a crear un array de colas de mensajes por thread
- Este array debe ser compartido para los threads
  - Un thread puede mandar un mensaje a cualquier otro thread
- Cola
  - lista de mensajes
  - puntero hacia el primer elemento de la lista
  - puntero hacia el utlimo elemento de la lista
  - Un contador para los mensajes puestos en cola
  - Un contador para los mensajes retirados de la cola
- Cada thread asignara memoria para su cola
  - Es necesario que todos terminen esta fase antes de empezar a mandar mensajes
  - # pragma omp barrier

Productores/clientes - La directiva atomic

- Después de mandar los mensajes un thread incrementa
  - done_sending++ (critical)
  - Después de eso solamente espera mensajes
- Podemos utilizar algo mejor que atomic
  - #pragma omp atomic
  - x <op>=<expression>; x++; ++x; x−−; −−x;
  - op +, *, −, /, &, ˆ, | , <<, o >>
  - Protección de tipo **load-modify-store**

Productores/clientes - Secciones críticas y locks

- Critical
  - done_sending++;
  - Enqueue(q_p, my_rank, mesg);
  - Dequeue(q_p, &src, &mesg);
- **Enqueue** -> No es completamente necesario porque ej.
  - Thread 0 puede mandar un mensaje a thread 1 mientras que thread 1 manda a thread 2...
- OpenMP no toma en cuenta eso así que se deteriora el rendimiento del sistema
- Solucion ? # pragma omp critical(name)
- Secciones críticas con un nombre distinto pueden ser ejecutados simultáneamente

Productores/clientes - Secciones críticas y locks

- # pragma omp critical(name)
- Secciones críticas con un nombre distinto pueden ser ejecutados simultáneamente
- los nombres están definidos al momento de compilar
- pero nosotros queremos que los nombres dependen de los threads

Productores/clientes - Secciones críticas y locks

- Alternativa : Locks

```
/* Executed by one thread */
Initialize the lock data structure;
 . . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure
;
 . . .
/* Executed by one thread */
 Destroy the lock data structure;
```

Productores/clientes - Secciones críticas y locks

- Locks
- Structura compartida entre los threads
- el master inicializa el lock y una vez terminado los destruirá
- Antes de que un thread entre en la sección crítica intenta
  - **bloquear** el lock (**set**)
  - **Si** no existe un thread que está en la sección crítica -> lo logra
    - Entra en la sección crítica y realiza su trabajo et **desbloquea** el lock (**unset**)
  - **sino** esperará **hasta** que se libere el lock (válido para todo los threads que lo intentan)
- Simple lock
  - puede ser set solamente una vez por un thread antes de unset
- Nested lock
  - puede ser set varias una veces por un thread antes de unset

Productores/clientes - Secciones críticas y locks

- Locks
- void omp_init lock(omp lock t* lock p /* out */);
- void omp_set lock(omp lock t* lock p /* in/out */);
- void omp_unset lock(omp lock t* lock p /* in/out */);
- void omp_destroy lock(omp lock t* lock p /* in/out */);

- usaremos locks **simples**

Productores/clientes - Secciones críticas y locks

- Locks

```
q_p = msg queues[dest]
# pragma omp critical
Enqueue(q_p, my rank, mesg);
```

```
q_p = msg_queues[dest]
omp_set lock(&q_p−>lock);
Enqueue(q_p, my rank, mesg);
omp_unset lock(&q_p−>lock);
```

Productores/clientes - Secciones críticas y locks

- Locks

```
q_p = msg_queues[my_rank]
# pragma omp critical
Dequeue(q_p, &src, &mesg);
```

```
q_p = msg_queues[my_rank]
omp set_lock(&q_p−>lock);
Dequeue(q_p, &src, &mesg);
omp_unset lock(&q_p−>lock);
```

Productores/clientes - Secciones críticas y locks

- Locks
- Tenemos que inicializar el lock dentro la inicialización de la cola
- Ahora ya que cada cola tiene su lock
  - Un thread está bloqueado solamente si quiere acceder una misma cola que otro thread
- Antes un solo thread podía acceder a una cola a la vez (secuencial :-( )

Critical directives, atomic directives, or locks?

- atomic es la mas rapida para obtener una exclusión mutua
  - Problema es que excluye todas los bloques con atomic en todo el programa
  - # pragma omp atomic
    - x++;
  - # pragma omp atomic
  - y++;
  - Se **debería** de usar **critical** con **nombres**
- Critical con o sin nombres se deberían de usar cuando no se puede usar atomic
- **locks** cuando debemos de proteger estructuras de datos

Advertencias

- Hay que tener cuidado con las exclusiones mutuas
- No mezclar atomic y critical

```
# pragma omp atomic
x += f(y);
```

```
# pragma omp critical
x = g(x);
```

- el criticos de la derecha no protege el atomic la parte de la izquierda
  - un thread puede hacer el atomic mientras otro hace el critical
- Solucion proteger ambos con atomic entonces reescribir g para usar atomic
- Usar critical en ambos

Advertencias

- La distribución no es equitativa

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

- El thread 1 puede estar bloqueado para siempre mientras el thread 0 ejecuta una y otra vez x=g(my_rank);
- Algunos sistemas distribuyen el acceso en orden de llegada.

Advertencias

- Anidar las exclusiones mutuas

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

  - Deadlock !!
  - Utilizar critical con nombres !!

Advertencias

- Anidar las exclusiones mutua
- Deadlock !!
- Aquí los nombres no ayudan !

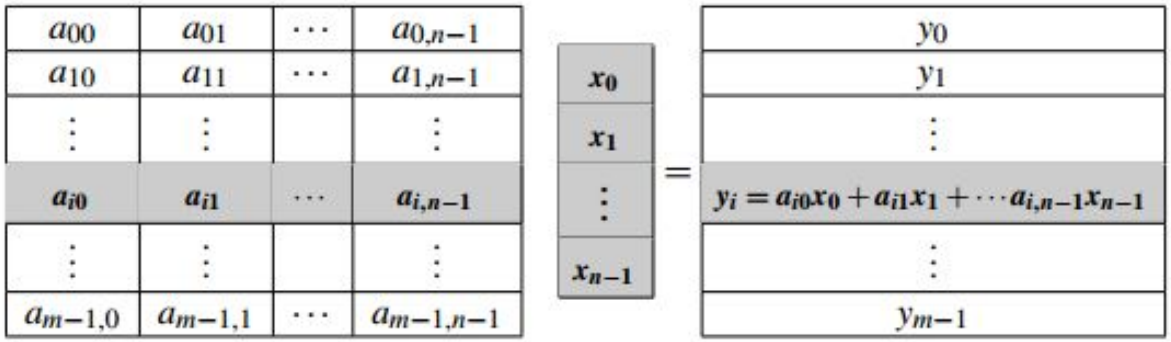
Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Cache !

- Cache permite almacenar datos cerca al procesador
  - Spatial and temporal locality
  - Cache line
  - **cache coherence**
    - variable présente en diferentes cachés
    - impacto dramático sobre el rendimiento

Cache !

- Multiplicación Matriz-Vector
- matriz **A** = (aij) m × n, **x** is a vector with **n** componentes
- Resultado y = A*x sería un vector de m componentes
- Producto punto entre un línea de la matriz y x
  - $y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$



Cache !

```
for (i = 0; i < m; i++) {
     y[i] = 0.0;
     for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
}
```

- **No existe interdependencia en el bucle exterior**
  - **pragma for**

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

Cache !

- **Cache coherence**

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- **Eficiencia**

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Cache !

- **Con un solo thread**
- **8M x 8 -> 22%**
  - **Y** tiene **8M** elementos *VS* **8K** o **8**
  - Cache **write miss** para **Y** -> Inicializar los 8M a 0 (line 4)

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

Cache !

- **Con un solo thread**
- **8 x 8M -> 26%**
  - **X** tiene **8M** elementos *VS* **8K** o **8**
  - Cache **read miss** para **X** -> leer los 8M a 0 (line 6)

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

Cache !

- **Con 2 threads 20%**
  - **Eficiencia : 8M x 8 > 8k x 8k > 8 x 8M**
- **Con 4 threads 50%**
  - **Eficiencia : 8M x 8 > 8k x 8k > 8 x 8M**

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Cache !

- **Porque tanta diferencia entre 8Mx8 vs 8x8M**
- **8Mx8**
  - **Cada thread 2M elementos**
- **8k x 8k**
  - **Cada thread 2K elementos**
- **8x8M**
  - **Cada thread 2 elementos**
- Cache line 64 bytes
- Y double 8 bytes
  - **Cache line -> 8 double**

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Cache !

- Cache line 64 bytes
- Y double 8 bytes
  - **Cache line -> 8 double**
- **Dual Core**
  - **cada núcleo tiene su propio caché**
  - **si se escribe en el caché de uno se invalida la línea del caché del otro si presente**
  - **Threads 0-1 core 1**
  - **Threads 2-3 core 2**

Cache !

- **Cache line -> 8 double**
- **8 x 8M**
  - **Y** puede ser almacenado completamente en un cache line
  - A cada vez que se **core 1** escribe en Y la línea será invalidada en caché de **core 2**
  - **Total cada thread tiene que ejecutar 16M de veces esta operación**

```
y[i] += A[i][j]*x[j];
```

  - A pesar de y[0] será solamente tocado por thread 0, y[1] thread 0, ..., y[2] thread 1, y[2] thread1,...
  - **False sharing**

Cache !

- **Porque no pasa eso con 8k x 8k**
  - **Thread 2 core 0**
    - y[4000], y[4001], . . . , y[5999],
  - **Thread 3 core 1**
    - y[6000], y[6001], . . . , y[7999]
- **Cache line -> 8 double**
- **Cuales son las posibilidades de false sharing ?**
  - Ej. y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003]
  - Thread 2 accédera a [5996], y[5997], y[5998], y[5999] **al final** de sus iteraciones
  - Thread 3 accédera a y[6000], y[6001], y[6002], y[6003] **al inicio** de sus iteraciones
  - Así que es **poco probable** que pasé...

Cache !

```
1       # pragma omp parallel for num threads(thread count) \
2       default(none) private(i, j) shared(A, x, y, m, n)
3        for (i = 0; i < m; i++) {
4               y[i] = 0.0;
5               for (j = 0; j < n; j++)
6                       y[i] += A[i][j]*x[j];
7       }
```

Cache !

```
# pragma omp parallel for num threads(thread count) \
2       default(none) private(i, j) shared(A, x, y, m, n)
3        for (i = 0; i < m; i++) {
4                tmp = 0.0
5                for (j = 0; j < n; j++)
6                        tmp += A[i][j]*x[j];
7                y[i] = tmp;
8        }
```

Thread safe

- Ej. strtok
- separar las palabras de un archivo que contiene líneas de texto
- Cada thread se ocupa de varias líneas arregladas de forma cíclica.

```c
/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

Thread safe

- Ej. strtok
- separar las pa
- Cada thread s

ca.

```
1  void Tokenize(
2        char*  lines[]       /* in/out */,
3        int    line_count    /* in     */,
4        int    thread_count  /* in     */) {
5     int my_rank, i, j;
6     char *my_token;
7
8  #  pragma omp parallel num_threads(thread_count) \
9        default(none) private(my_rank, i, j, my_token) \
10       shared(lines, line_count)
11    {
12       my_rank = omp_get_thread_num();
13 #     pragma omp for schedule(static, 1)
14       for (i = 0; i < line_count; i++) {
15          printf("Thread %d > line %d = %s", my_rank, i,
                  lines[i]);
16          j = 0;
17          my_token = strtok(lines[i], " \t\n");
18          while ( my_token != NULL ) {
19             printf("Thread %d > token %d = %s\n", my_rank, j,
                     my_token);
20             my_token = strtok(NULL, " \t\n");
21             j++;
22          }
23       } /* for i */
24    }  /* omp parallel */
25
26 }  /* Tokenize */
```

Thread safe

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

- Ej. strtok
- separar las palabras de un archivo que contiene líneas de texto
- Cada thread se ocupa de varias líneas arregladas de forma cíclica.

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

Thread safe

- El resultado aquí es incorrecto sin embargo en otras ejecuciones el resultado hubiera podido ser correcto.

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

Resumen !

- Directivas pre-procesador -> pragmas
- Compilador compatible
- Pensado para paralelizar fácilmente programas secuenciales
  - -/+ verdad
- Paralelizar un bloque facilmente con
  - # pragma omp parallel
  - Si no se especifica la cantidad de thread el sistema deberá crear uno por núcleo
- Team threads (los threads que ejecutan un bloque en paralelo)
  - Master + Salves

Resumen !

- Modificar las directivas con cláusulas
  - num_threads
- Rango :
  - omp_get_thread_num
- Cuantos threads
  - omp get_num_threads

Resumen !

- Secciones críticas
  - Atomic
  - Critical
    - Critical con nombre
  - Locks
    - omp_set_lock(&lock);
    - critical section
    - omp_unset_lock(&lock);

Resumen !

- Directivas parallel for
  - loop-carried dependences
  - block partitioning
  - scheduling
    - schedule(type,chunksize)
    - type : static, dynamic, guided, auto, or runtime (OMP_SCHEDULE)
  - Variable scope
    - shared, private
    - default(none) para después explicitamente identificar las variables privadas o compartidas

Resumen !

- Caché coherencia
  - False sharing
- Thread safe