OpenMP.

César Pedraza Bonilla

Universidad Nacional de Colombia
Departamento de Ingeniería de Sistemas e Industrial

capedrazab@unal.edu.co

8 de octubre de 2019

Overview

Introducción.

- API para escribir aplicaciones multihilo.
- Conjunto de directivas y librerías para programación multihilo.
- Simplifica la programación de aplicaciones multihilo.

Introducción.

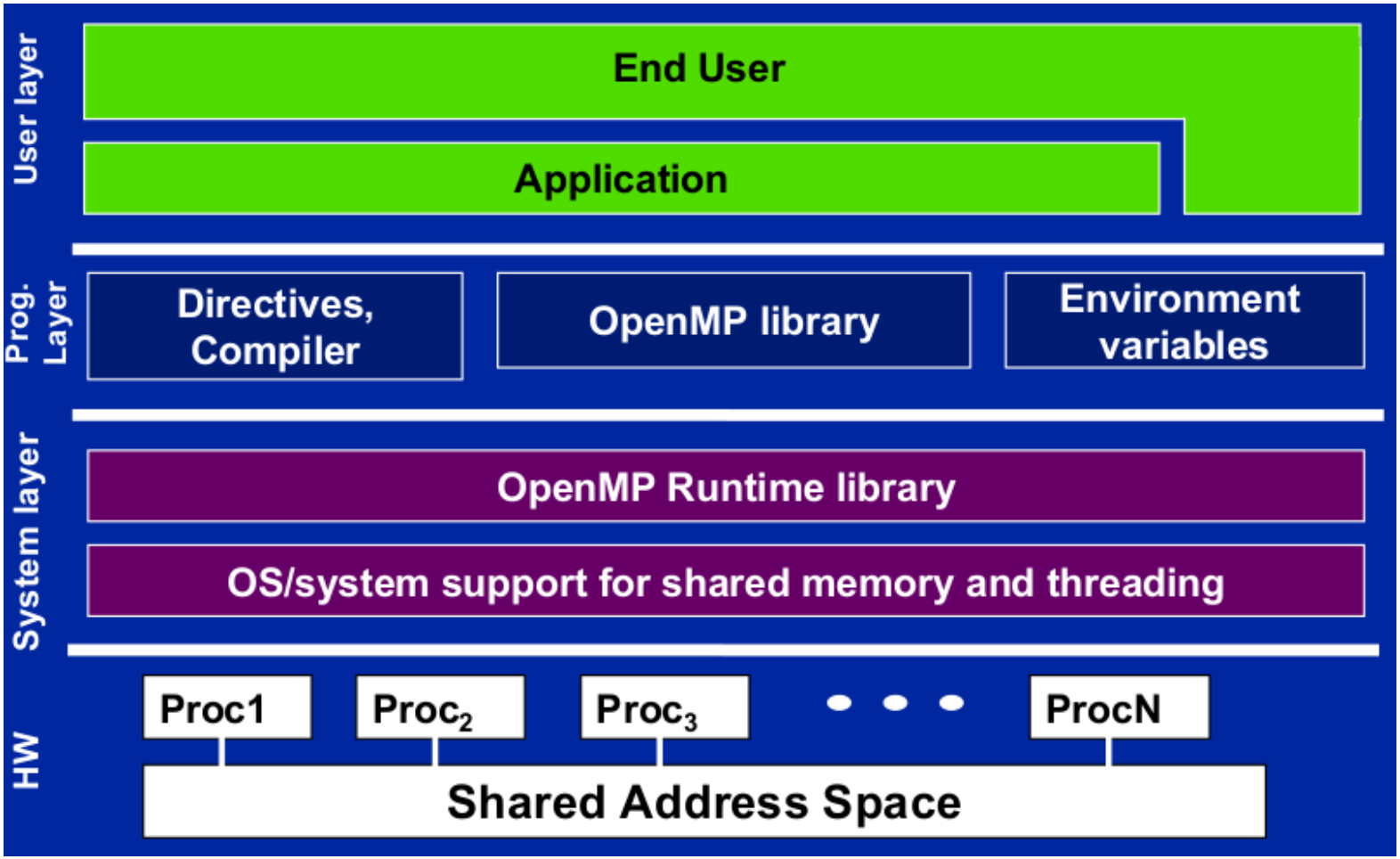


Figura: Arquitectura general de OpenMP. Tim Mattson

Introducción.

- La mayoría de las construcciones de OpenMP son directivas.
- Las construcciones de OpenMP se aplican a bloques de código, un punto de entrada y un punto de salida.
- Por ejemplo:

```
1 #pragma omp parallel num_threads(4)
2
3 #include <omp.h>
```

Introducción.

- OpenMP usa el modelo de hilos.
- Los hilos se comunican a través de variables compartidas.
- Condiciones de carrera: el programador realiza operaciones con resultados de hilos, que no necesariamente se han ejecutado.
- Se usan métodos de sincronismo, pero que son muy costosos computacionalmente. Se debe revisar la forma en que se acceden a las variables para evitar excesos en sincronismos.

Introducción.


```
1  #include "omp.h"
2  void main()
3  {
4    #pragma omp parallel num_threads(4)  //inicio de region paralela
5    {
6      int ID = omp_get_thread_num();  //ID del hilo
7      printf(" hello( %d) ", ID);
8      printf(" world( %d) \n", ID);
9    }    //fin de region paralela
10 }
```

gcc fuente.c -o ejecutable -fopenmp
export OMP_NUM_THREADS=4
./ejecutable

Creación de hilos.

- Paralelismo estilo *fork()-join()*
- Existe un hilo maestro.
- Ejemplo: crear 4 hilos:

```
1 omp_set_num_threads(4);  #pragma omp parallel num_threads(4)
2
3 int omp_get_num_threads(); //numero de hilos presentes
4 int omp_get_thread_num();  //Thread ID
5 double omp_get_wtime();    //Tiempo en segundos desde un punto
      fijo en el pasado
```

Ejercicio: paralelizar el algoritmo de Leibniz mediante openMP.

Introducción.

- Existe un hilo maestro
- Se generan tantos hilos como se deseen

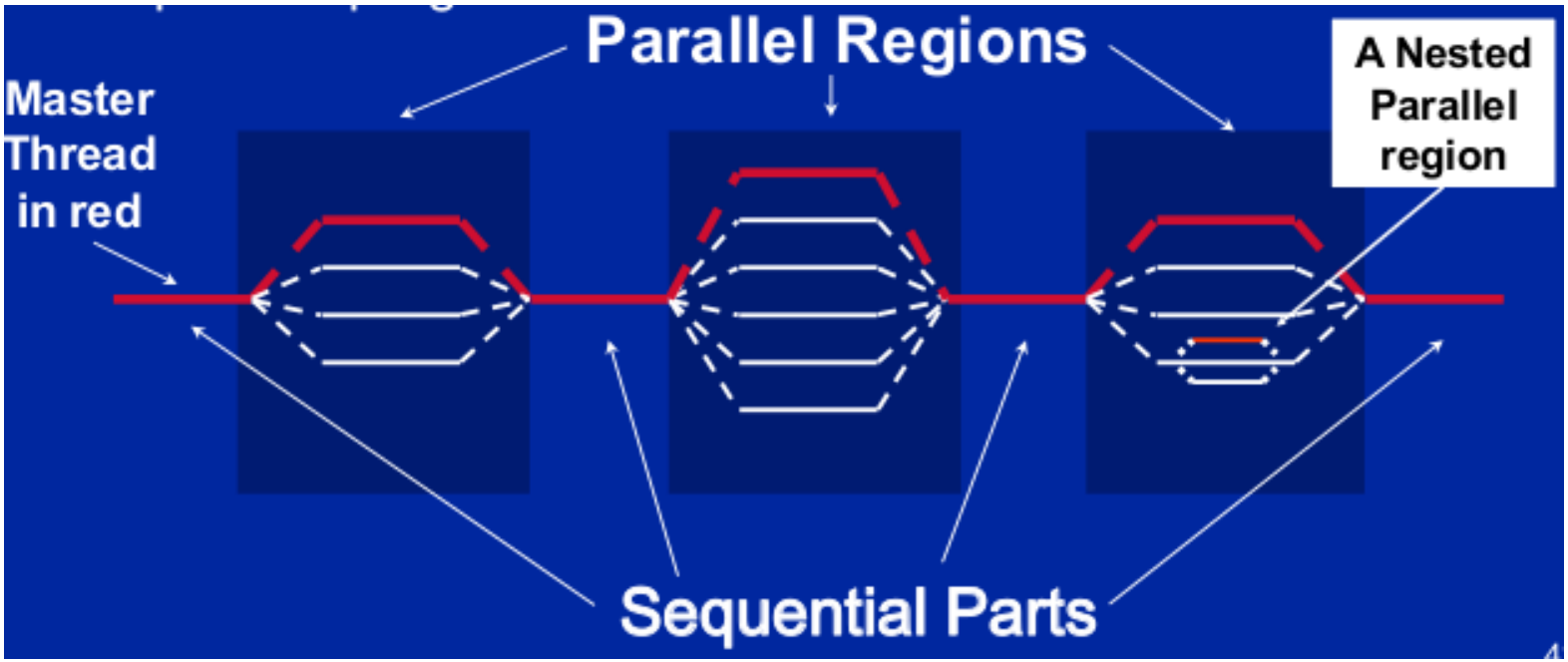


Figura: Arquitectura general de OpenMP. Tim Mattson

Creación de hilos.

Código con OpenMP

```
1  #pragma omp parallel num_threads(4)
2  {
3    foobar ();
4  }
```

Código creado:

```
1  void thunk ()
2  {
3      foobar ();
4  }
5  pthread_t tid[4];
6  for (int i = 1; i < 4; ++i)
7      pthread_create (&tid[i],0,thunk, 0);
8      thunk();
9  for (int i = 1; i < 4; ++i)
10     pthread_join (tid[i]);
```

Creación de hilos.

False sharing.

```
1  #include <omp.h>
2  static long num_steps = 100000;
3  double step;
4  #define NUM_THREADS 2
5  void main (){
6    int i, nthreads; double pi, sum[NUM_THREADS];
7    step = 1.0/(double) num_steps;
8    omp_set_num_threads(NUM_THREADS);
9    #pragma omp parallel
10   {
11     int i, id,nthrds;
12     double x;
13     id = omp_get_thread_num();
14     nthrds = omp_get_num_threads();
15     if (id == 0) nthreads = nthrds;
16     for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
17       x = (i+0.5)*step;
18       ssum[id] += 4.0/(1.0+x*x);
19     }
20   }
21   for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
```

Creación de hilos.

False sharing

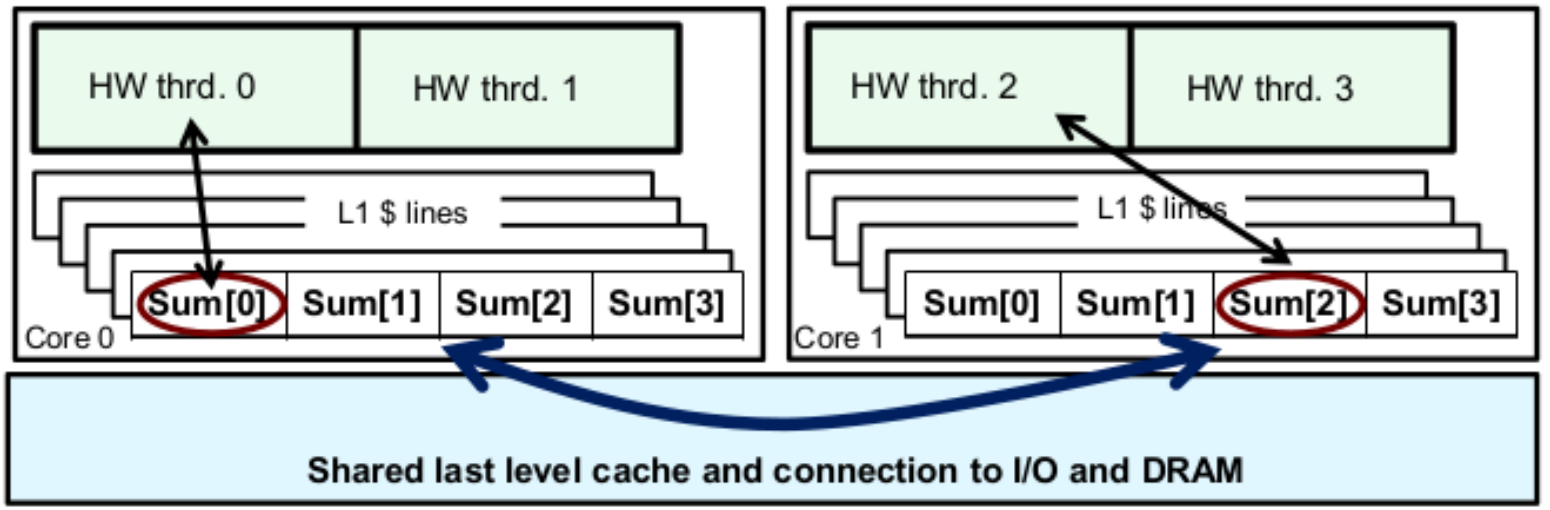


Figura: Arquitectura general de OpenMP. Tim Mattson

Creación de hilos.

False sharing.

- Se observa que se usa un arreglo para acumular los cálculos de cada hilo.
- Los elementos de *sum* son adyacentes en memoria. Puede causar al protocolo de caché que sean leídos más frecuentemente en memoria, dado que son llevados a *cores* distintos. A éste fenómeno se le llama *false sharing*.
- También se observa cuando se copian datos frecuentemente a caché, que sólo van a ser leídos, a causa de posiciones de memoria que van a ser escritas y que son adyacentes.
- Se puede atenuar separando las posiciones de memoria a las que cada hilo accede.

Creación de hilos.

False sharing.

```
1  #include <omp.h>
2  static long num_steps = 100000;
3  #define PAD 8      // assume 64 byte L1 cache line size
4  double step;
5  #define NUM_THREADS 2
6  void main (){
7    int i, nthreads; double pi, sum[NUM_THREADS][PAD];   //separaci'on
8    step = 1.0/(double) num_steps;
9    omp_set_num_threads(NUM_THREADS);
10   #pragma omp parallel
11   { int i, id,nthrds;
12     double x;
13     id = omp_get_thread_num();
14     nthrds = omp_get_num_threads();
15     if (id == 0) nthreads = nthrds;
16     for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
17       x = (i+0.5)*step;
18       ssum[id][0] += 4.0/(1.0+x*x);
19     }
20   }
21   for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Sincronización

Son herramientas o estrategias para imponer un orden y protección en el acceso a datos o recursos. Pej.

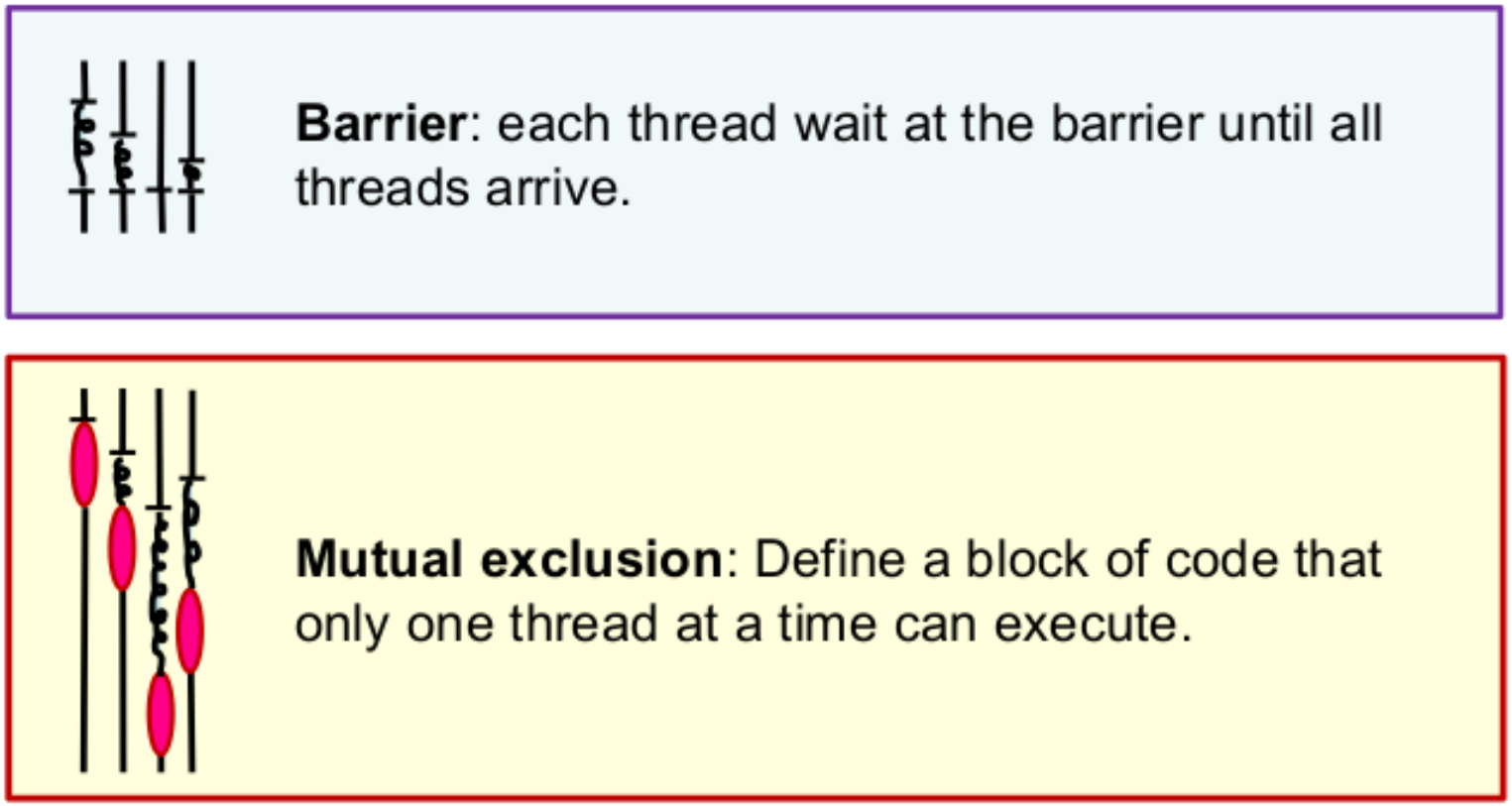


Figura: Arquitectura general de OpenMP. Tim Mattson

Sincronización.

- De alto nivel. Crítica, atómica, barrera, ordenada.
- De bajo nivel. Flush, locks.

Sincronización.

Sincronización crítica.

Hay exclusión mutua: sólo 1 hilo ejecuta al tiempo una sección crítica.

```
1  float res;
2  #pragma omp parallel
3  {
4    float B; int i, id, nthrds;
5    id = omp_get_thread_num();
6    nthrds = omp_get_num_threads();
7    for(i = id; i < niters; i + nthrds){
8      B = big_job(i);
9    #pragma omp critical
10     consume (B, res);     // just 1 thread executes consume at a time
11 }
12 }
```

Ejercicio: pi, realizar la reducción en un hilo con sincronización

Sincronización.

Cálculo de pi con sincronización crítica para la reducción:

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{             double  pi;          step = 1.0/(double) num_steps;
              omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
               int i, id,nthrds;    double x, sum;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
               id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
               for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                          x = (i+0.5)*step;
                          sum += 4.0/(1.0+x*x);
               }
            #pragma omp critical
                    pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

Sincronización.

Sincronización Atómica.

Permite hacer exclusión mutua pero sólo aplica para leer y actualizar una variable. La operación atómica debe ser del tipo:
x binop= expr, x++, ++x, x—, –x
Pej, se proteje la actualización de la variable X.

```
1 #pragma omp parallel
2 {
3   double tmp, B;
4   B = DOIT();
5   tmp = big_ugly(B);
6  #pragma omp atomic
7   tmp = big_ugly(B);
8   X += tmp;
9 }
```

Sincronización.

Cálculo de pi con sincronización atómica para la reducción:

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           double  pi;          step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
             int i, id,nthrds;    double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
            id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
            }
             sum = sum*step;
          #pragma atomic
                  pi += sum ;
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi so updates don’t conflict

Sincronización.

Sincronización por barrera.

Cada hilo espera hasta que los todos hayan llegado hasta un punto determinado del programa.

```
1 #pragma omp parallel
2 {
3     int id=omp_get_thread_num();
4     A[id] = big_calc1(id);
5 #pragma omp barrier
6     B[id] = big_calc2(id, A);
7 }
```

Anidamiento paralelo - Worksharing

- Una construcción paralela crea un SPMD (Single Program, Multiple Data).
- Es posible partir un anidamiento para que sea ejecutado en múltiples hilos.
- La forma en que se separan las cargas de un programa en hilos:
  - Loop construct
  - Sections/section constructs
  - Single construct
  - Task construct

Anidamiento paralelo - Worksharing

Loop construct

```
1 #pragma omp parallel
2   {
3     #pragma omp for
4     for (I=0;I<N;I++){
5       NEAT_STUFF(I);
6     }
7 }
```

La variable *I* se crea de forma privada para cada hilo.

Anidamiento paralelo - Worksharing


```
1  //secuencial
2  for(i=0;i<N;i++) {
3      a[i] = a[i] + b[i];
4  }
5
6
7  //OpenMP parallel region
8  #pragma omp parallel
9  {
10   int id, i, Nthrds, istart, iend;
11   id = omp_get_thread_num();
12   Nthrds = omp_get_num_threads();
13   istart = id * N / Nthrds;
14   iend = (id+1) * N / Nthrds;
15   if (id == Nthrds-1)iend = N;
16   for(i = istart; i < iend; i++) {
17       a[i] = a[i] + b[i];
18   }
19 }
```

Anidamiento paralelo - Worksharing


```
1  //OpenMP parallel region and a worksharing for construct
2  #pragma omp parallel
3  #pragma omp for
4  for(i=0;i<N;i++) {
5    a[i] = a[i] + b[i];
6  }
7
```

Anidamiento paralelo - Worksharing

La cláusula *schedule*.

- *schedule(static [,chunk]).* Se asignan bloques de tamaño *chunk* a cada hilo.
- *schedule(dynamic[,chunk]).* Cada hilo ejecuta *chunk* iteraciones de una cola
- *schedule(guided[,chunk]).* Cada hilo ejecuta bloques de hilos de forma dinámica. El tamaño del bloque es grande al inicio y luego disminuye.
- *schedule(runtime).* El tamaño del bloque es tomado de la variable *OMP_SCHEDULE*
- *schedule(auto).* Escoje la máquina.

omp_schedule.c

Anidamiento paralelo - Worksharing

La directiva *omp parallel for* for incluye el *omp parallel* de forma implícita

```
1 double res[MAX]; int i;
2 #pragma omp parallel
3 {
4     #pragma omp for
5     for (i=0;i< MAX; i++) {
6         res[i] = huge();
7     }
8 }
```

```
1 double res[MAX]; int i;
2 #pragma omp parallel for
3 for (i=0;i< MAX; i++) {
4     res[i] = huge();
5 }
```

Los dos casos son idénticos.

Anidamiento paralelo - Worksharing

Iteraciones anidadas

```
1 #pragma omp parallel for collapse(2)
2   for (int i=0; i<N; i++) {
3     for (int j=0; j<M; j++) {
4       .....
5     }
6 }
```

Se especifica el número de iteraciones a ser paralelizadas (2)
Muy útil cuando N es igual al número de hilos.

Anidamiento paralelo - Worksharing

Reducción.

- La reducción es la combinación de valores en una variable de acumulación.
- La cláusula para hacer reducción es reduction (op : list).
- Dentro de la itereción:
  - Se crea una copia local de cada variable y es inicializada.
  - La variable local es la que se actualiza.
  - Las copias locales se reducen a un solo valor y almacenado en la variable original.

Anidamiento paralelo - Worksharing

Reducción.

Ejemplo secuencial:

```
1 double ave=0.0, A[MAX]; int i;
2 for (i=0;i< MAX; i++) {
3     ave + = A[i];
4 }
5 ave = ave/MAX;
```

Con OpenMP:

```
1 double ave=0.0, A[MAX]; int i;
2 #pragma omp parallel for reduction (+:ave)
3 for (i=0;i< MAX; i++) {
4     ave + = A[i];
5 }
6 ave = ave/MAX;
```

Anidamiento paralelo - Worksharing.

Operandos para reducción y valores iniciales

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Figura: Arquitectura general de OpenMP. Tim Mattson

Sincronización con barrera.

Cada hilo espera hasta que todos hayan llegado hasta ese punto.

```
1  #pragma omp parallel shared (A, B, C) private(id)
2  {
3    id=omp_get_thread_num();
4    A[id] = big_calc1(id);
5    #pragma omp barrier
6    #pragma omp for
7      for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
8    #pragma omp for nowait
9    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
10     A[id] = big_calc4(id);
11 }
```

Sentencia master

Especifica un bloque de código que sólo va a ser ejecutado por el hilo maestro.

```
1
2 #pragma omp parallel
3 {
4   do_many_things();
5   #pragma omp master
6   { exchange_boundaries(); }
7   #pragma omp barrier
8   do_many_other_things();
9 }
```

Sentencia single

Especifica un bloque de código que se desea ejecutar por un solo hilo. No necesariamente el maestro. Implícitamente lleva una barrera al final. Se puede omitir con *nowait*

```
1 #pragma omp parallel
2 {
3   do_many_things();
4   #pragma omp single
5     { exchange_boundaries(); }
6     do_many_other_things();
7 }
```

Sentencia sections

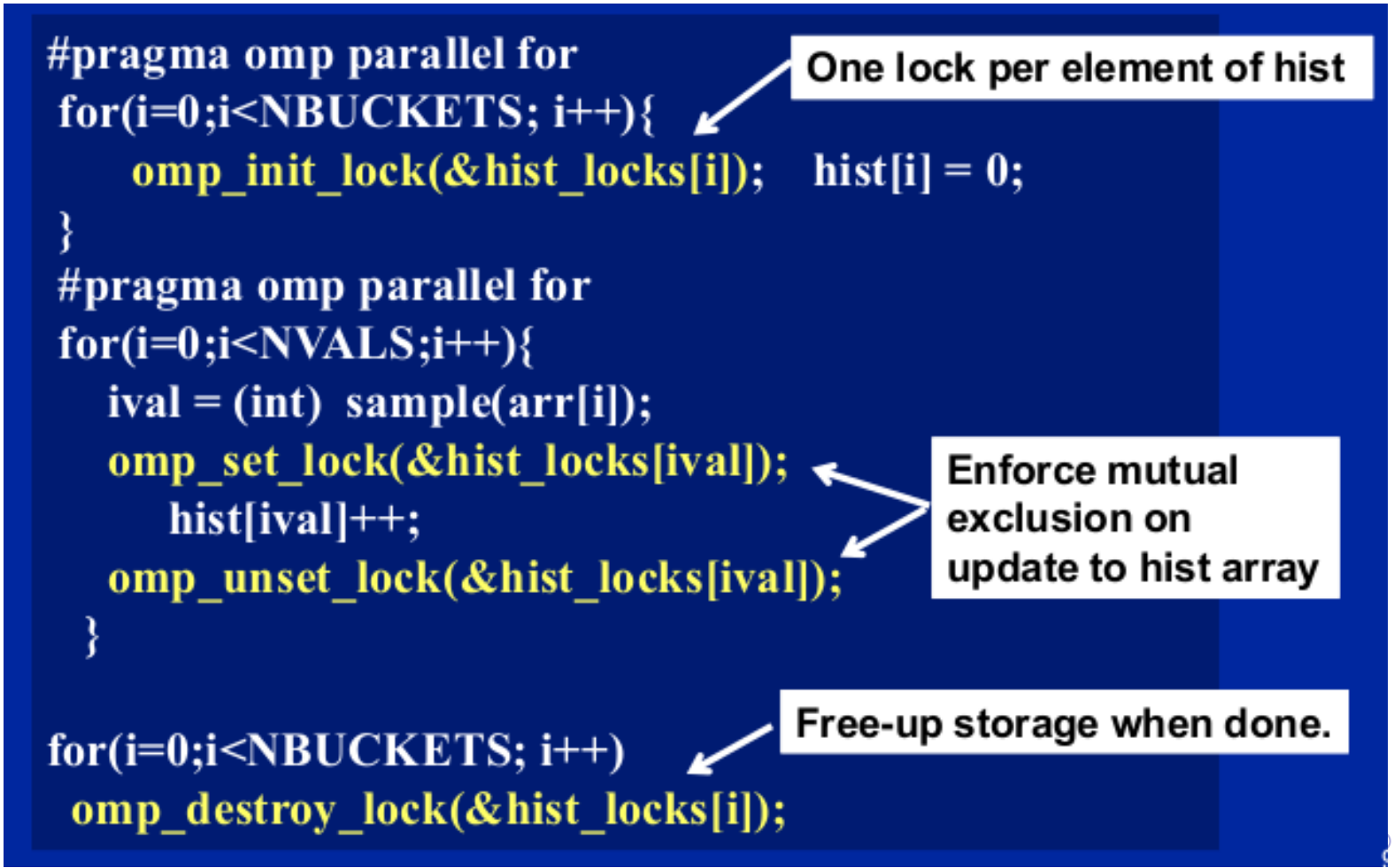
Permite asignar una estructura diferente de código a cada hilo.

```
1  #pragma omp parallel
2  {
3      #pragma omp sections
4      {
5          #pragma omp section
6            X_calculation();
7          #pragma omp section
8            y_calculation();
9          #pragma omp section
10           z_calculation();
11     }
12 }
```

Por defecto se inserta una barrera al final de la directiva *sections*.

Locks

Es posible proteger una variable al interior de un *loop* para evitar una condición de carrera.



Programación con OpenMP.

Rutinas para uso en tiempo de ejecución.

- Modificar o verificar el número de hilos:

```
1 omp_set_num_threads(), omp_get_num_threads(),omp_get_thread_num(), omp_get_max_threads()
```

- ¿Estamos en una región paralela?

```
1 omp_in_parallel()
```

- ¿Desea que el sistema cambie en número de hilos de forma dinámica de una región paralela a otra?

```
1 omp_set_dynamic, omp_get_dynamic();
```

- ¿Cuántos procesadores tenemos?

```
1 omp_num_procs()
```

Programación con OpenMP.

Rutinas para uso en tiempo de ejecución. Ejemplo:

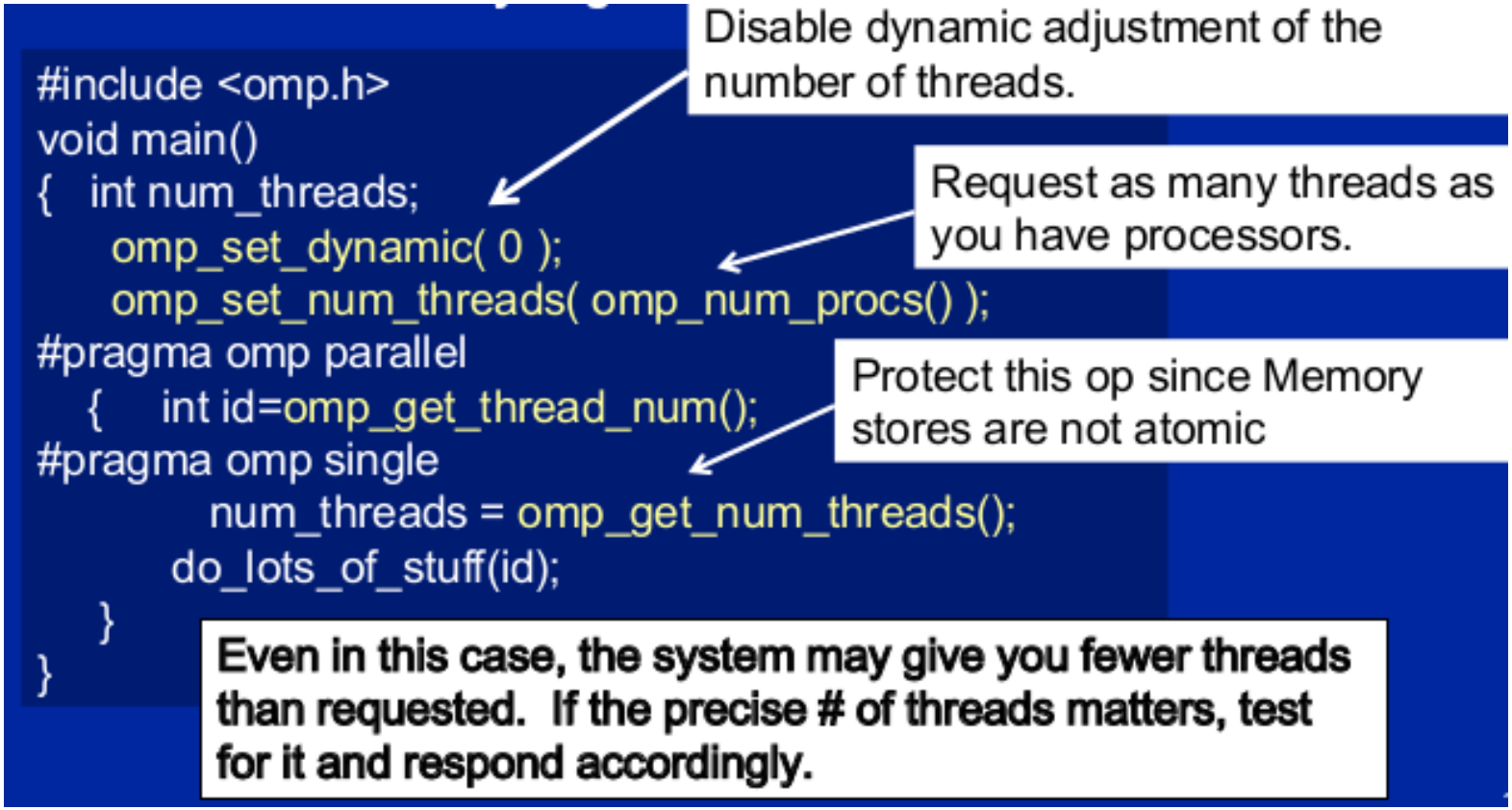


Figura: Arquitectura general de OpenMP. Tim Mattson

Data sharing

- Todas las variables declaradas como globales son compartidas.
- Las variables declaradas en funciones que son llamadas desde los hilos son privadas.

```
1 double A[10];
2 int main() {
3   int index[10];
4   #pragma omp parallel
5       work(index);
6   printf(" %d\n",  index[0]);;
7 }
```

```
1 extern double A[10];
2 void work(int *index) {
3   double temp[10];
4   static int count;
5 }
```

Data sharing

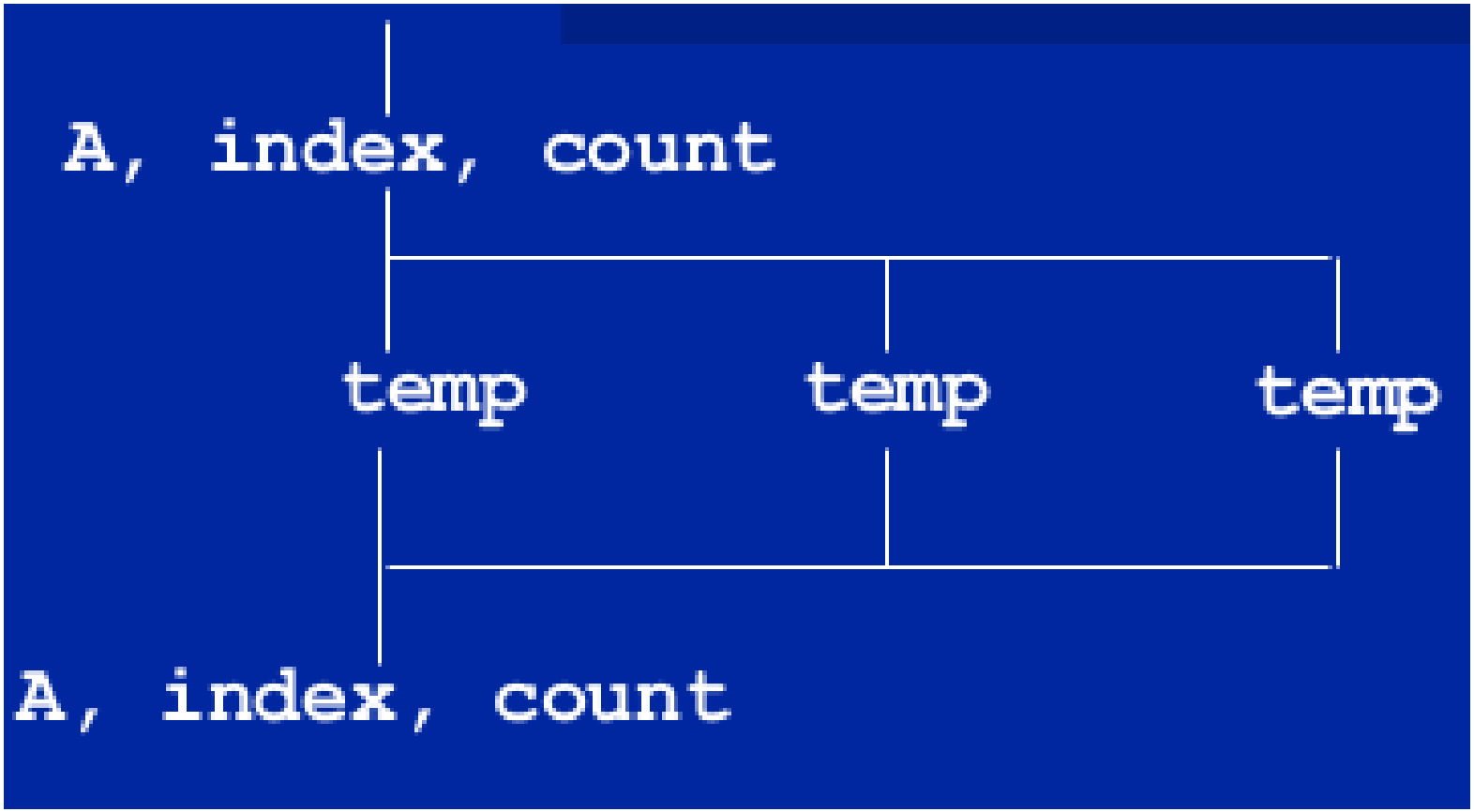


Figura: Arquitectura general de OpenMP. Tim Mattson

Data sharing

- Es posible cambiar los atributos de almacenamiento
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE)
- El valor final de un *parallel loop* a la variable compartida fuera del *loop*.
  - LASTPRIVATE

Data sharing

Cláusula *private*

```
1 void wrong() {
2     int tmp = 0;
3     #pragma omp parallel for private(tmp)
4         for (int j = 0; j < 1000; ++j)
5         tmp += j;
6     printf(" %d\n",  tmp);  //Aqui tmp = 0
7 }
```

Data sharing

Cláusula *firstprivate*

Las variables son inicializadas desde una variable compartida

```
1 incr = 0;
2   #pragma omp parallel for firstprivate(incr)
3     for (i = 0; i <= MAX; i++) {
4         if ((i%2)==0) incr++;   //cada hilo tiene una copia de incr con valor
          inicial de 0
5         A[i] = incr;
6     }
```

Data sharing

Cláusula *lastprivate*

Las variables actualizan el valor de una variable compartida tomando el valor de la última iteración

```
1  void sq2(int n, double *lastterm)
2  {
3      double x; int i;
4      #pragma omp parallel for lastprivate(x)
5          for (i = 0; i < n; i++){
6              x = a[i] * a[i] + b[i] * b[i];
7              b[i] = sqrt(x);
8          }
9      *lastterm = x;
10 }
```

Al final x tiene el valor de la ultima iteracion

Tasks

- Son unidades independientes de trabajo. Pueden ser anidados.
- Se componen de: código, variables de entorno y variables de control interno.
- Cada hilo realiza el trabajo de una tarea.
- El planificador determina cuando se ejecuta una tarea. Puede ser ejecutada inmediatamente o puede ser aplazada.

```
1  int fib ( int n )
2  {
3      int x,y;
4      if ( n < 2 ) return n;
5      #pragma omp task
6          x = fib(n-1);
7      #pragma omp task
8          y = fib(n-2);
9      #pragma omp taskwait
10     return x+y  //Hay un error en la reduccion...
11 }
```

Tasks Solución:

```
1  int fib ( int n )
2  {
3      int x,y;
4      if ( n < 2 ) return n;
5      #pragma omp task shared (x)
6          x = fib(n-1);
7      #pragma omp task shared(y)
8          y = fib(n-2);
9      #pragma omp taskwait
10         return x+y;
11 }
```

Las tareas permiten crear hilos en tiempo de ejecución. Las tareas permiten crear hilos a partir de otros.

Tasks

Ejecución de las tareas:

```
#pragma omp parallel
{
  #pragma omp single
  {  //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task
      process(p);
    p = p->next;  //block 3
    }
  }
}
```

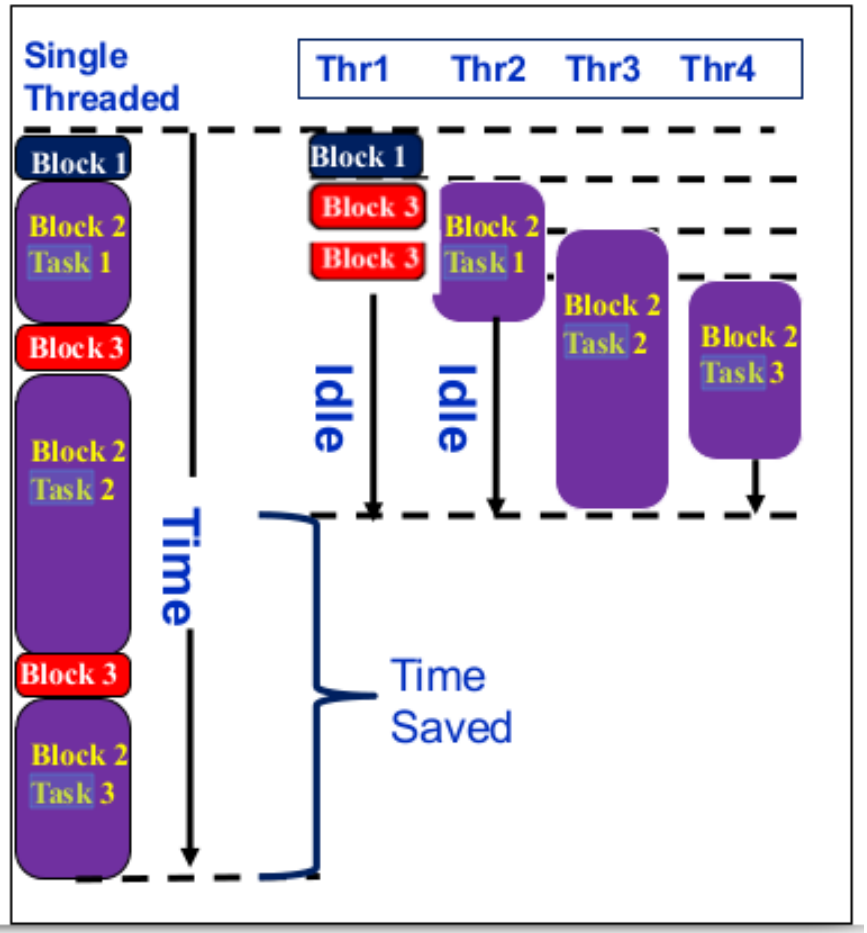


Figura: Arquitectura general de OpenMP. Tim Mattson

Tasks

```
1  #include <omp.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5  #define ITER 1e09
6
7  int xfunc(){
8    double x;
9    int i;
10   for(i = 0; i < ITER; i++){
11     x = x + sin(i);
12   }
13 }
14
15 int yfunc(int a){
16   double x;
17   int i;
18   for(i = 0; i < ITER; i++){
19     x = x + cos(i);
20   }
21 }
```

Tasks

```
1  int main(){
2    printf("\nejemplo de uso de tasks");
3  #pragma omp parallel
4  {
5    #pragma omp single
6    {
7      #pragma omp task
8      xfunc();  //executed by 1 thread
9
10     #pragma omp task
11     yfunc(1);  //executed by 1 thread
12
13     #pragma omp task
14     xfunc(1);  //executed by 1 thread
15
16     #pragma omp task
17     yfunc(1);  //executed by 1 thread
18   }
19 }
20 return 0;
21 }
```