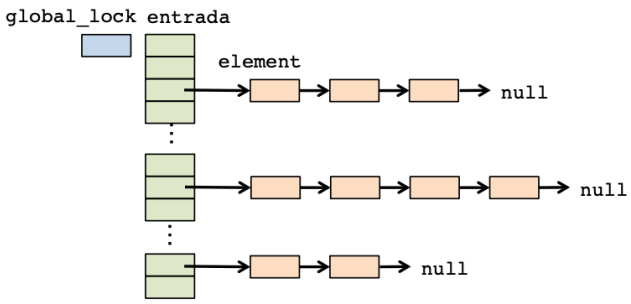
Problema 15

Assume a hash table implemented as a vector of chained lists, such as shown in the next figure and type definition:

```
#define SIZE_TABLE 1048576
typedef struct {
    int data;
    element * next;
} element;

typedef struct {
    omp_lock_t global_lock;
    element *  entrada[SIZE_TABLE];
} HashTable;

HashTable table;
```

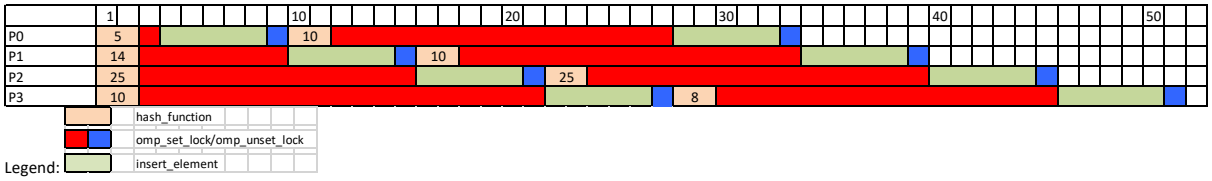


Inside each list, the elements are stored ordered by their `data` field value. Also assume the following code snippet to insert elements of the `ToInsert` vector of size `num_elem` into the mentioned hash table:

```
#define MAX_ELEM 1024
int main() {
    int ToInsert[MAX_ELEM], num_elem, index;
    ...
    omp_init_lock(&table.global_lock);
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop private(index) // default to as many tasks as threads
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock (&table.global_lock);
        insert_elem (ToInsert[i], index);
        omp_unset_lock (&table.global_lock);
    }
    omp_destroy_lock(&table.global_lock);
    ...
}
```

where `hash_function` function returns the entry of the table (between 0 and `SIZE_TABLE`-1 where a specific element has to be inserted and the `insert_elem` function inserts the mentioned element in the corresponding position inside the chained list pointed by the `index` entry of the `HashTable`.

(a) Given the sequence `index`={5,10,14,10,25,25,10,8} returned by `hash_function` for a `ToInsert` vector with `num_elem`=8 elements, draw in a timing diagram the parallel execution with 4 threads, assuming that the `hash_function` function lasts 2 time units, `insert_elem` lasts 5 time units and the "set" and "unset" lock functions last 1 time unit each. The rest of the operations can be considered to use a negligible time.



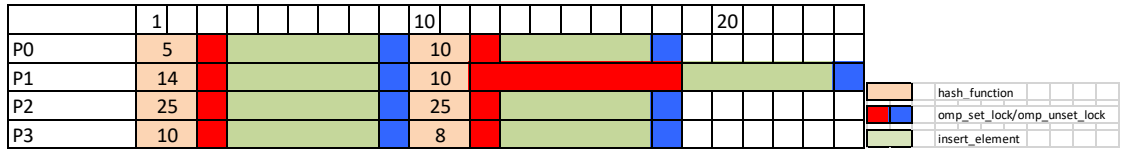
(b) Modify the previous data structure and code to allow parallel insertions in different entries of the `HashTable`.

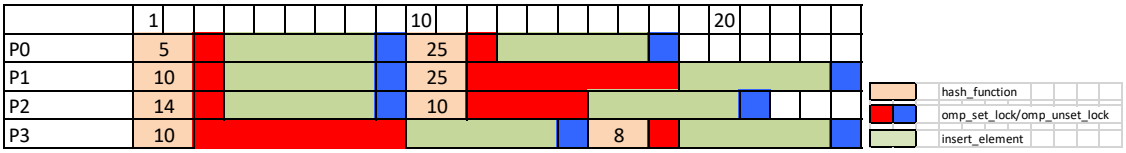
```
typedef struct {
    omp_lock_t lock[SIZE_TABLE];
    element *  table[SIZE_TABLE];
} HashTable;
...
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskloop // default to as many tasks as threads
    for (i=0; i<SIZE_TABLE; i++) omp_init_lock(&table.lock[i]);

    #pragma omp taskloop private(index)
    for (i = 0; i < num_elem; i++) {
        index = hash_function(elems[i], SIZE_TABLE);
        omp_set_lock (&table.lock[index]);
        insert_elem (elems[i], index);
        omp_unset_lock (&table.lock[index]);
    }

    #pragma omp taskloop
    for (i=0; i<SIZE_TABLE; i++) omp_destroy_lock(&table.lock[i]);
}
...
```

(c) For the same sequence of **index** values used previously, draw again the timing diagram of the parallel execution with 4 threads for the implementation proposed in section b). How would that diagram change if the task granularity is changed with **grainsize(1)**? Draw the new timing diagram for each case.

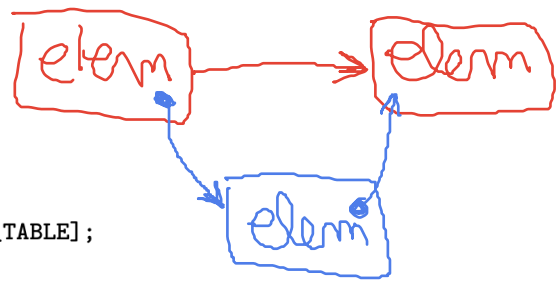




(d) Modify the data structures to allow a higher degree of concurrency in the ordered insertion of elements inside the same chained list (it is NOT necessary neither to implement changes in the **insert_elem** function code nor to draw again the timing diagram).

```
typedef struct {
    int data;
    omp_lock_t lock;
    element * next;
} element;

typedef struct {
    element *  table[SIZE_TABLE];
} HashTable;
```



Ara volem que dos processadors diferents puguin inserir en una mateix index de la hashtable. Per tant el lock ja no pot ser a nivell de index, ara ha de ser a nivell d'element. Per tant, creem un lock per cada element.

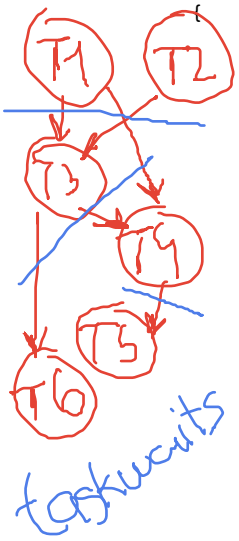
Ara quan volem afegir un element a continuació de l'element actual (un element entre dos elements, com al dibuix), el lock de l'element actual evita que dos threads vulguin inserir després d'aquest mateix element.

La funció saxpy és un bucle que va de 0 a n-1 que en cada iteració agafa un element del vector x i un del vector y, multiplica l'element de x per a i li suma l'element de y i el guarda a y[i].

y -> Lectura + Escriptura
x -> Només lectura

```
void saxpy(int n, float a, float *x, float *y) {
        for (int i = 0; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

Problema 12 Tema 4




```
#pragma omp parallel
#pragma omp single
{
        /* simple initialization just for testing */
        #pragma omp task depend(out:fx)                          // T1
        for (int k = 0; k < N; ++k)
                fx[k] = 2.0f + (float) k;
        #pragma omp task depend(out:fy)                          // T2
        for (int k = 0; k < N; ++k)
                fy[k] = 1.0f + (float) k;

        /* Run SAXPY TWICE */
        #pragma omp task depend(in:fx) depend(inout:fy)          // T3
        saxpy(N, 3.0f, fx, fy);
        #pragma omp task depend(in:fy) depend(inout:fx)          // T4
        saxpy(N, 5.0f, fy, fx);

        #pragma omp task depend(in:fx)                           // T5
        for (int k = 0; k < N; ++k) {
                fprintf(fpx, "%f", fx[k]);
        }
        #pragma omp task depend(in:fy)                           // T6
        for (int k = 0; k < N; ++k) {
                fprintf(fpy, "%f", fy[k]);
        }
}
```

Definim una tasca per cada bucle. Sabem que T1 i T2 poden anar totalment en paralel ja que T1 actualitza fx i T2 la fy. Per tant no tenen dependències.

T3 té com a dependències T1 i T2 ja que necessita fx i fy per a fer el càlcul de saxpy. T3 llavors ens genera fy.

T4 té com a dependència T3 (ja que és qui ha generat la última còpia de fy. I també necessita fx que ha generat T1. Genera fx.

T5 depen de T4 per a fer l'escriptura de fx.

T6 escriu fy i per tant depen de T3 que és l'últim que escriu fy.

La versió amb `taskwait` o `taskgroup`, si la demanessin requeriria pensar abans quines dependències hi ha entre les tasques per assegurar l'ordre correcte. Pinteu primer el graf de tasques (TDG) que sortiria.

```
#pragma omp parallel
#pragma omp single
{
/* simple initialization just for testing */
#pragma omp task                                  // T1 and …
for (int k = 0; k < N; ++k)
        fx[k] = 2.0f + (float) k;
#pragma omp task                                  // … T2 can run in parallel …
for (int k = 0; k < N; ++k)
        fy[k] = 1.0f + (float) k;
#pragma omp taskwait                              // … but need to wait for them

/* Run SAXPY TWICE */
// #pragma omp task                               // T3 runs alone, no task …
saxpy(N, 3.0f, fx, fy);
// #pragma omp taskwait                           // … and no taskwait!

#pragma omp task                                  // T4-T5 fused into a single task
{
saxpy(N, 5.0f, fy, fx);

for (int k = 0; k < N; ++k) {
        fprintf(fpx, "%f", fx[k]);
}

#pragma omp task                                  // … in parallel with T6
for (int k = 0; k < N; ++k) {
        fprintf(fpy, "%f", fy[k]);
}
}
```

Generem una tasca perquè la faci algú altre i m'espero a que acabi. Tant per tant la fa el mateix processador. Per això no fa falta crear una tasca aqui.

Un cop acabem la invocació de saxpy, continuem amb la generació de tasques.

Com que T5 depèn de T4, és una tonteria generar dues tasques i posar un taskwait. Ho podem fer de forma sequencial dins una mateixa tasca que engloba tot el processament sequencial. T4 i T5 s'ajunten en una sola tasca que podrà anar amb paralel amb T6.

Barrera implícita del single.

No és qüestió de més o menys paral·lelisme, és qüestió de simplificar l'escriptura del programa paral·lel (programming productivity) i evitar errors.

Slide 59 (diferents opcions de combinacions de clàusules depend)

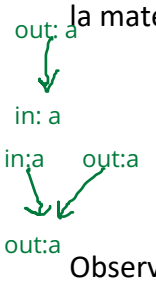
Què impliquen les diferents direccionalitats en la clàusula `depend`? Doncs una `in` sobre una variable `var` fa que la tasca s'esperi a qualsevol tasca creada prèviament que tingui la mateixa variable `var` com a `out` (individual o combinada en forma de `inout`):

```
// variables a i b son shared
#pragma omp task depend(out:a)
a = foo1();
#pragma omp task depend(in:a) depend(out:b)
b = foo2(a);
#pragma omp task depend(in:b)
foo3(b);
```

És la típica dependència productor->consumidor. Una petita modificació del exemple:

```
// variable a es shared
#pragma omp task depend(out:a)
a = foo1();
#pragma omp task depend(inout:a)
a += foo2(a);
#pragma omp task depend(in:a)
foo3(a);
```

I una `out` sobre una variable `var` que provoca? Doncs a què qualsevol tasca prèvia que tingui la mateixa variable `var` com a `in` o com a `out` acabi. Perquè? En el sequencial, el valor que em queda es el valor de a de T3. Si contemplem la situació 1) ens queda que es queda el valor de T1 i no ho volem.




```
// variable a es shared
#pragma omp task depend(out:a)
a = foo1();
#pragma omp task depend(in:a)
foo2(a);
#pragma omp task depend(out:a)
a = foo3();
```

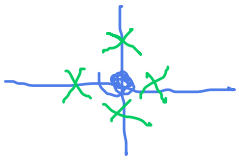
Si posem noms de variable diferent es soluciona!

Posar dos outs sobre una mateixa variable SI ens crea una relació d'ordre: El segon out no es podrà executar fins que no acabi el primer out. Amb això pot passar que T2 agafi un valor de a erroni (el generat per T3) en cas que s'executi posteriorment. Per tant T3 no es pot executar tampoc fins que T2 hagi acabat.

Observeu que si `foo3` no espera i s'executa en paral·lel amb `foo1`a podem tenir dues situacions: 1) que `foo3` acabi abans que `foo1`: en aquest cas `foo2` agafa el valor correcte però al final de tot el valor d'`a` que queda es el de `foo1` i no el de `foo3`. O 2) que `foo3` acabi més tard que `foo1`: en aquest cas `foo2` (que no començarà fins que `foo1` acabi) podrà no agafar el resultat que ha generat la `foo1`, sinó el que ha generat `foo3`. És una dependència deguda al re-ús de la variable `a`.

Slide 60

Exemple de dependències similars a les de Gaus-Seidel que hem treballat al llarg del curs. De fet, si fos el Gauss-Seidel hauríem s'escriure com es mostra a continuació, amb tots els accessos indicats amb la direccionalitat corresponent:



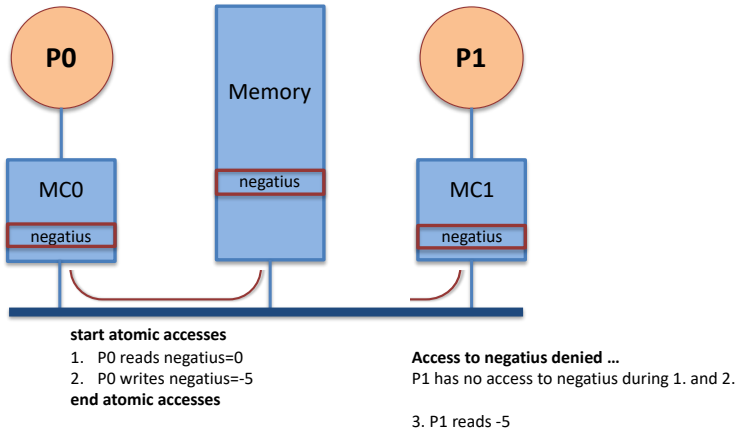
```
#pragma omp parallel private(i, j)
#pragma omp single
{
    for (i=1; i<n i++) {
        for (j=1; j<n;j++) {
            #pragma omp task
                        depend(in : block[i-1][j], block[i][j-1])
                        depend(in : block[i+1][j], block[i][j+1])
                        depend(inout: block[i][j])
            foo(i,j);
        }
    }
}
```

Sincronització

Avui anem a veure quin suport ens necessita donar l'arquitectura per tal de suportar les instruccions de sincronització que hem vist en OpenMP. Mirem el codi següent que acumula els elements negatius en un vector ...

```
int negatius=0;
…
#pragma omp taskloop
for (i=0; i<N; i++) {
    if (a[i]<0) {
        #pragma omp atomic
        negatius += a[i];
    }
}
```

La suma està protegida amb un `#pragma omp atomic` per tal de garantir que la lectura i escriptura es facin sense que cap altre thread hi accedeixi. En el cas de arquitectures UMA, el bus i el protocol de coherència que vàrem estudiar en el Tema 3 ens permeten això, fixeu-vos en el diagrama següent, pel cas d'un bus:



Mentre el processador P_0 està accedint a `negatius` dins de l'atòmic, el bus i el protocol impedeixen que cap altre processador (P_1 en el nostre cas) pugui accedir a la línia que conté la variable `negatius`. La manera més simple d'entendre-ho és pensar que el bus queda bloquejat durant la durada de la lectura i escriptura atòmiques. Altres implementacions més eficients no bloquegen el bus: la cache de P_0 respon a qualsevol petició de la línia amb una nova comanda del bus, "Retry" que provoca que P_1 torni a repetir l'accés.

Slides 65 i 66

En el cas d'accés a una regió `#pragma omp crítical` o protegida amb una variable tipus omp_lock_t, la situació és similar

```
int negatius=0;
…
#pragma omp taskloop
for (i=0; i<N; i++) {
    if (a[i]<0) {
        #pragma omp critical
        negatius += a[i];
    }
}
```

```
int negatius=0;
omp_lock_t flag;
omp_init_lock(&flag);
…
#pragma omp taskloop
for (i=0; i<N; i++) {
    if (a[i]<0) {
        omp_set_lock(&flag);
        negatius += a[i];
        omp_unset_lock(&flag);
}}
```

Mirem el codi que hi ha en la slide 66 que fa una possible implementació dels locks. La variable `flag` és del tipus `omp_lock_t` i està inicialitzada a 0. Per comprovar si es pot accedir a la regió crítica el que cal fer és llegir `flag`, comparar si és 0 i si no tornar-ho a intentar; quan valgui 0, llavors la ficarem a 1 per indicar que la regió crítica està ocupada. Quan el thread acabi d'executar la regió crítica tornarà a ficar `flag` a 0, per tal que si algun altre thread està esperant pugui entrar. Però que passaria si hi ha un dos threads simultàniament intentant entrar: tots dos podrien veure que `flag` val zero i entrar a la vegada → error! Per evitar-ho hauríem d'assegurar que el load i el store son atòmics, és a dir, que entre el load i el store ningun altre thread pot accedir a la variable.

Slide 67

Per aconseguir això el llenguatge màquina inclou instruccions especials que indiquen al snoopy que ha de comportar-se així:

- test-and-set (`t&s`): reads value in location and sets to 1

Si a `flag` ja hi havia un 1 (està ocupat), llavors tornarà a escriure un 1 (innecessari) i tornarà a provar sort. Si hi havia un 0, llavors escriurà atòmicament un 1 i entrarà a la regió crítica.

Slide 68

- atomic exchange (`xchg`): interchange of a value in a register with a value in memory

La slide mostra la implementació del lock amb aquesta instrucció `xchg`.

- fetch-and-op (e.g. `f&add`): read value in location and replace with result after simple arithmetic operation (usually add, increment, sub or decrement).

Aquesta última és precisament la que ens permetrà implementar el atòmic del exemple anterior:

```
...
lea r1, a[i]           // calcula l'adreça efectiva de a[i]
mov r2, [r1]           // accedeix a l'adreça calculada
f&add negatius, r2     // instrucció atòmica sobre variable negatius
...
```

Si un altre processador vol acumular sobre negatius, la instrucció `f&add` farà que el snoopy de la seva memòria cache el faci esperar.

Slides 69

Mireu-vos aquesta slide, en la que es mostra el "enorme" tràfic de coherència que provoquen les instruccions de sincronització. I tot provocat pel fet que mentre un thread s'espera esta contínuament "sobreescrivint" un 1 a la variable flag, provocat que el protocol d'invalidació faci la seva funció.

Slide 70

Per reduir el tràfic de coherència podríem haver canviat el codi original:

```
set_lock: t&s r2, flag
          bnez r2, set_lock // already locked?
```

Per aquest que primer només llegeix i, un cop veu que està lliure (i per tant té possibilitats d'agafar el lock), fa el `t&s`:

```
set_lock: ld r2, flag       // test with regular load
                            // lock is cached meanwhile it is not updated
          bnez r2, set_lock // test if the lock is free
          t&s r2, flag
          bnez r2, set_lock // test and acquire lock if STILL free
```

Això se li diu test-test&set, ja que primer fa un test normal (lectura, load) i si veu que el lock està agafat (és a dir, torna un 1) no cal que faci l'escriptura per tornar a escriure un 1. Torneu a pensar si això té alguna importància en el tràfic pel bus en el protocol MSI (**slide 71**).

Slide 72

El ll marca la posició de memòria, i al fer l'store es mira si entre el ll i l'store conditional algú altre l'ha llegit. Si hi ha algú altre els esborro de la llista; si un altre thread vol fer un sc mirarà si està encara a la llista i si no hi és retornarà un 0.

En sistemes NUMA no és possible, o si voleu no té sentit, bloquejar tota la xarxa d'interconnexió que connecta a tots els nodes del sistema. De nou la funcionalitat recau en el protocol de coherència, el directori i en un parell de noves instruccions:

- Load-linked (ll): retorna el valor actual que hi ha a memòria. i marca que algú l'ha llegit
- Store-conditional (sc): escriu a memòria si d'ençà a que ha fet el ll cap altre processador ha modificat la mateixa variable. En aquest cas la instrucció sc retorna 1; en cas contrari retorna 0.

No entrem en el detall de com s'implementa això en el directori, els pocs que feu l'especialitat i feu MP ja ho estudiareu.

Però sí que podem veure som s'implementarien els locks i els atòmics amb les instruccions de ll-sc:

```
set_lock:   ll r2, flag       // first test with load linked
            mov r1, #1
            sc r1, flag       // try to store 1
            beqz r1, set_lock // repeat if someone else did it before me
            bnez r2, set_lock // test if the lock was free

            ...

unset_lock: st flag, #0 // free the lock
```

Slide 73

De fet fixeu-vos que això es pot millorar, ja que no cal escriure amb el sc si el lock ja està agafat:

```
set_lock:   ll r2, flag       // first test with load linked
            bnez r2, set_lock // test if the lock is free
            mov r1, #1
            sc r1, flag       // try to store 1
            beqz r1, set_lock // repeat if someone else did it before me
```

Penseu si això té alguna importància o no de cara al tràfic de coherència de memòria per la xarxa NUMA.

Fem el ll de la variable flag i saltem, mentre la variable valgui 1. Els threads que volen entra al lock s'afegeixen a la llista de threads que han llegit la variable flag (els que han fet el ll).

Llavors en el moment en que la variable flag valgui zero, intentaràn posar-lo a 1 fent un store-conditional. Si soc el primer que fa l'store-conditional veuré que estic a la llista de threads que han llegit i escriuré un 1. Si soc un dels threads que va després, que es troben que algú ha escrit abans, la condició no es cumplirà i tornaré a saltar a set_lock.

Slide 74

Ens podem trobar amb la necessitat d'implementar altres objectes de sincronització més enllà dels bàsics que hem presentat. Qualsevol objecte haurà de fer servir però aquests per a garantir sincronitzacions i consistència de memòria.

Problemes pel proper dilluns: 20, 19, 16 + quizzes Atenea

Problemes resolts, per mirar i preguntar dubtes

Problema 8 Tema 4

```
#define N 131072
long count_key(long Nlen, long *a, long key) {
    long count = 0;
    for (int i=0; i<Nlen; i++)
        if(a[i]==key) count++;
    return count;
}

int main() {
   long a[N], key = 42, nkey=0;
   for (long i=0;i<N;i++) a[i] = random()%N;
      a[N%43]=key; a[N%73]=key; a[N%3]=key;
   nkey = count_key(N, a, key);     // count key sequentially
   nkey = count_iter(N, a, key);    // count key in a using an iterative decomposition
   nkey = count_recur(N, a, key);   // count key in a with divide and conquer
}
```

Versió iterativa, on es generen tantes tasques com threads en la regió paral·lela

```
long count_iter(long Nlen, long *a, long key) {
  long count = 0;

  #pragma omp taskloop reduction(+:count) num_tasks(omp_get_num_threads())
  for (int i=0; i<Nlen; i++)
      if(a[i]==key) count++;

  return count;
}

int main() {
  …
  #pragma omp parallel
  #pragma omp single
  nkey = count_iter(N, a, key);  // count key: iterative decomposition
  …
}
```

Una versió amb tasques explicites (no `taskloop`) podria ser la següent, tot i que amb overhead addicional; només per mostrar l'ús de reduccions en tasques:

```
#pragma omp taskgroup task_reduction(+:count)
for (int i=0; i<Nlen; i++)
    #pragma omp task in_reduction(+:count)
    if (a[i]==key) count++;
```

Al final del `taskgroup` la el valor de la variable `count` contindrà les contribucions de totes les tasques generades en el bucle. Seria equivalent a un `taskloop` amb `granularity(1)`.

Versió recursiva divide-and-conquer sense incloure cut-off

```
long count_recur(long Nlen, long *a, long key) {
  long count1=0, count2=0, Nl2;
  if (Nlen == 1) {
     if (*(a) == key) count1=1;
  }
  else {
     Nl2 = Nlen/2;
     #pragma omp task shared(count1)
     count1 = count_recur(Nl2, a, key);
     #pragma omp task shared(count2)
     count2 = count_recur(Nlen-Nl2, a+Nl2, key);
     #pragma omp taskwait
  }
  return (count1+count2);
}

int main() {
  …
  #pragma omp parallel
  #pragma omp single
  nkey = count_recur(N, a, key);  // count key: recursive decomposition
  …
}
```

Versió incloent cut-off per nivell

```
long count_recur(long Nlen, long *a, long key, int level) {
  long count1=0, count2=0, Nl2;
  if (Nlen == 1) {
     if (*(a) == key) count1=1;
  }
  else if (!omp_in_final()){
     Nl2 = Nlen/2;
     #pragma omp task shared(count1) final(level >= MAX_LEVEL)
     count1 = count_recur(Nl2, a, key, level+1);
     #pragma omp task shared(count2) final(level >= MAX_LEVEL)
     count2 = count_recur(Nlen-Nl2, a+Nl2, key, level+1);
     #pragma omp taskwait
  } else {
     count1 = count_recur(Nl2, a, key, level+1);
     count2 = count_recur(Nlen-Nl2, a+Nl2, key, level+1);
  }
  return (count1+count2);
}

int main() {
  …
  #pragma omp parallel
  #pragma omp single
  nkey = count_recur(N, a, key, 0);  // count key: recursive decomposition
  …
}
```

Problema 11 Tema 4

Farem una solució basada en task dependences fent servir #pragma omp task:

```
float sample[INPUT_SIZE+TAP1];
float coeff1[TAP1], coeff2[TAP2];
float data_out[INPUT_SIZE], final[INPUT_SIZE];

void main() {
float sum;
...
#pragma omp parallel
#pragma omp single
for (int i=0; i<INPUT_SIZE; i++) {
    #pragma omp task depend(out:data_out[i]) private(sum) firstprivate(i)
    {
        // Producer
        sum = 0.0;
        for (int j=0; j<TAP1; j++)
            sum += sample[i+j] * coeff1[j];
        data_out[i] = sum;
    }

    #pragma omp task depend(in:data_out[i]) firstprivate(i)
    {
        // Consumer
        for (int j=0; j<TAP2; j++)
            final[i] += correction(data_out[i], coeff2[j]);
    }
}
}
```

Hi ha dependències, dins de cada iteració del bucle `i`, entre la tasca `producer` i tasca `consumer`. La dependència queda definida per la compartició de l'element del vector `data_out[i]`; en el producer és `out`, en el consumer és in. Totes les iteracions de i podrien anar en paral·lel, això requereix privatització de `sum` i fer firstprivate de `i` (implícit en aquest cas). La sincronització implícita al final del single/parallel ja assegurarà qualsevol dependència amb tasques posteriors.

Problema 10 Tema 4

The following sequential code in C finds all positions in vector `DBin` in which a set of keys (contained in vector `keys`) appear. Positions where keys appear are stored in a new vector `DBout` (the order in `DBout` of the positions found is irrelevant).

```
#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger
                 than the number of keys

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

     getkeys (keys, nkeys);              // get keys
     Init_Dbin (DBin, DBsize);           // initialize DBin
     clear_Dbout (DBout, nkeys, DBsize); // initialize DBout
     clear_counter (counter, nkeys);     // initialize counter

     for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
           if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```

DBin		keys	counter	DBout					
D		A	2	4	7				
G		B	0						
E		D	1	0					
O		X	0						
A		E	1	2					
G		G	3	1	5	8			
T		C	0						
A		O	1	3					
G		T	1	6					
…		R	0						
		M	0						
		…	…	…	…	…	…	…	…

(a) Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop `i`, making use of the `taskloop` directive, in which you minimise the serialisation introduced by the synchronisation that you may introduce. **Note:** you are not allowed to change the structure of the two loops.

Solution: We could simply use `critical` to update `DBout` and `counter`, as follows:

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop private(k)
for (i = 0; i < DBsize; i++)
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                #pragma omp critical
                DBout[k][counter[k]++] = i;
            }
```

However, a solution that minimises the serialisation introduced by that `critical` region should make use of looks, as many as possible keys, as follows:

```
omp_lock_t lock[nkeys];
for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
...
#pragma omp parallel
#pragma omp single
#pragma omp taskloop private(k) num_tasks(4)
for (i = 0; i < DBsize; i++)
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                omp_set_lock(&lock[k]);
                DBout[k][counter[k]++] = i;
                omp_unset_lock(&lock[k]);
            }
...
for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);
```

(b) Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop `k`, again making use of the `taskloop` directive, in which you maximise the parallelism that can be exploited. **Notes:** 1) `taskloop` has an implicit `taskgroup` synchronisation that you can omit with the `nogroup` clause; 2) observe that the number of keys is not large when compared to the possible number of processors to use; and 3) you are not allowed to change the structure of the two loops.

Solution: The first naive solution below is not appropriate:

```
#pragma omp parallel private(i)
#pragma omp single
for (i = 0; i < DBsize; i++)
    #pragma omp taskloop num_tasks(4)
    for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) {
                DBout[k][counter[k]++] = i;
            }
```

In this solution we don't need to introduce any kind of synchronisation between tasks since, for each iteration `i`, each task updates different set of rows of `DBout` (the implicit `taskgroup` at the end of `taskloop` ensures this). However the correct solution should consider that the number of keys is not large compared to the number of threads executing the parallel region, so we create a small number of tasks in each `taskloop` construct, insufficient to feed all threads.

For this reason, we simply remove the implicit task barrier at the end of the `taskloop` so that multiple `taskloop` constructs for different iterations of the `i` loop can be active, thus generating enough tasks to feed all the threads available. However, now multiple tasks may update the same set of rows of `DBout`, forcing us to protect the update of that variable, as before, with locks to minimise serialisation.

```
    omp_lock_t lock[nkeys];
    for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
    ...
    #pragma omp parallel private(i)
    #pragma omp single
    for (i = 0; i < DBsize; i++)
        #pragma omp taskloop num_tasks(4) nogroup
        for (k = 0; k < nkeys; k++)
                if (DBin[i] == keys[k]) {
                    omp_set_lock(&lock[k]);
                    DBout[k][counter[k]++] = i;
                    omp_unset_lock(&lock[k]);
                }
    ...
    for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);
```

(c) Finally, write a third *OpenMP* parallelisation that implements a `task`–based **recursive divide–and–conquer decomposition strategy**, with the following requirements: 1) the recursion splits the input vector `DBin` in two almost identical halves, with a base case that corresponds to checking a single element of `DBin`; 2) uses a **cut–off strategy based on the size of the input vector**, so that tasks are only generated while that size is larger than `CUT_SIZE`; 3) only uses *OpenMP* pragmas and clauses for the implementation of the cut–off strategy; and 4) you have to use the synchronisation mechanism, if needed, that maximises the parallelism in the program.

Solution:

```
#include <omp.h>

#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

#define CUT_SIZE ... /* Set the desired value here */

omp_lock_t lock[nkeys];

void find_keys( double *DBin, double *keys, double** DBout,
                unsigned int *counter, unsigned int ini, unsigned int end)
{
  unsigned int k;

  if ((end-ini) == 0) { /* Base case */
    for (k = 0; k < nkeys; k++)
      if (DBin[ini] == keys[k]) {
          omp_set_lock(&lock[k]);
          DBout[k][counter[k]++] = ini;
          omp_unset_lock(&lock[k]);
      }
  } else {
    unsigned int half = (end-ini+1) / 2;
    #pragma omp task final(half <= CUT_SIZE) mergeable
    find_keys(DBin, keys, DBout, counter, ini, ini+half-1);
    #pragma omp task final(half <= CUT_SIZE) mergeable
    find_keys(DBin, keys, DBout, counter, ini+half, end);
  }
}
```

```
int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int k, counter[nkeys];

    for (k = 0; k < nkeys; k++) omp_init_lock(&lock[k]);
    ...

    getkeys(keys, nkeys);              // get keys
    init_DBin(DBin, DBsize);           // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys);     // initialize counter to zero

    #pragma omp parallel
    #pragma omp single
    find_keys(DBin, keys, DBout, counter, 0, DBsize-1);
    ...
    for (k = 0; k < nkeys; k++) omp_destroy_lock(&lock[k]);

    // Use results
}
```