

### Problema 16 Tema 4

Describe two anomalies that make the parallel execution of the following loop incorrect and explain how to correct them:

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop
for (i=0; i<N; i++) {
    j = f(i); // returns any value between 0 and N-1, different than i
    omp_set_lock(&lck[i]);
    omp_set_lock(&lck[j]);
    int temp = vector[i];
    vector[i] = vector[j];
    vector[j] = temp;
    omp_unset_lock(&lck[j]);
    omp_unset_lock(&lck[i]);
}
```

Hi ha dues anomalies en aquest codi:

1. Cal privatitzar la variable j.
2. Es necessita garantir que l'intercanvi dels elements i i j de vector es faci amb exclusió mútua. Però tal com està especificat en el codi hi ha un **problema de deadlock!** Per exemple, simultàniament una tasca vol intercanviar i=20 amb j=35, i un altra tasca vol intercanviar i=35 amb j=20 (casualitats de la vida!) Com evitar-ho?

```
if (j<i) {omp_set_lock(&lck(j)); omp_set_lock(&lck(i));}
else {omp_set_lock(&lck(i)); omp_set_lock(&lck(j));}

....

omp_unset_lock(&lck(j)); omp_unset_lock(&lck(i));
```

### Problema 19 Tema 4

19. The following piece of code shows the implementation of a barrier synchronisation, for a UMA multiprocessor system with MSI coherency, making use of the atomic instruction t&s:

```
lock:  t&s r2, barr.lock           // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
          barr.flag = 0           // reset flag if first
      mycount = barr.counter++;
      if (mycount == P) {         // last to arrive?
          barr.counter = 0        // reset for next barrier
          barr.flag = 1           // release waiting processors
      } else
          while (barr.flag == 0); // busy wait for release
      barr.lock = 0               // release lock
```

(a) Identify a concurrency problem that exists in this code and propose a solution to solve it.

```

lock:  t&s r2, barr.lock           // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
          barr.flag = 0           // reset flag if first
      mycount = barr.counter++;
      barr.lock = 0;              // release lock
      if (mycount == P) {         // last to arrive?
          barr.counter = 0        // reset for next barrier
          barr.flag = 1           // release waiting processors
      } else
          while (barr.flag == 0); // busy wait for release

```

(b) Based on the solution proposed for the concurrency problem identified in the first question, implement a new version that reduces the synchronization overhead to acquire lock.

Com recordareu de la classe anterior, el t&s introdueix molt tràfic de coherència ja que sobre-escriu de forma innecessària un 1 quan acaba de llegir un 1. Per evitar-ho podem fer un **test-test&set**:

```

lock:  ld r2, barr.lock
      bnez r2, lock
      t&s r2, barr.lock           // acquire lock
      bnez r2, lock
      if (barr.counter == 0)
          barr.flag = 0           // reset flag if first
      mycount = barr.counter++;
      barr.lock = 0;              // release lock
      if (mycount == P) {         // last to arrive?
          barr.counter = 0        // reset for next barrier
          barr.flag = 1           // release waiting processors
      } else
          while (barr.flag == 0); // busy wait for release

```

## Problema 20 Tema 4

Implementació d'un tipus de lock anomenat spin\_lock:

```

void spin_lock (int *lock, int x) {
    int ret;
    do {
        ret = test_and_set (lock );
        if (ret==1)
            pause (x); /* Pause the thread for x time units */
    } while (ret == 1);
}

```

Fixeu-vos que per evitar molt de tràfic de coherència, si detecta que el lock està agafat, el thread s'adorm una estona (x unitats de temps). Així no molesta a altres threads.

Ens demanen escriure dues noves versions, una fent servir test-test&set i un altre fent servir ll-sc amb la mateixa idea de reduir al màxim el tràfic de coherència de memòria. Ens donen les interfícies d'aquestes funcions per poder-les utilitzar en C:

```

int test_and_set (int *lock); // Set *lock to 1 and returns previous value of *lock
int load_linked (int *addr); // returns value of *addr
int store_conditional (int *addr, int value); // store value to addr
                                              // returns 0 if store fails, 1 otherwise

```

Una primera versió fent us de ll-sc seria:

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        value = load_linked (lock);
        ret = store_conditional (lock, 1);
        if (value == 1 || ret == 0)
            pause (x);
    } while (value ==1 || ret == 0);
}
```

On veiem que simplement hem reemplaçat el t&s per la parella ll-sc. Ara haurem de fer pause si o bé el ll retorna un 1 (lock ocupat, value==1) o el sc no s'ha pogut dur a terme (ret==0).

Versió fent test-test&set:

```
void spin_lock (int *lock, int x) {
    int ret;
    do {
        /* test */
        while (*lock == 1)
            pause (x);

        /* test-and-set */
        ret = test_and_set (lock );
        if (ret==1)
            pause (x);
    } while (ret == 1);
}
```

Versió load-linked store-conditional (ll-sc):

```
void spin_lock (int *lock, int x) {
    int value, ret;
    do {
        /* test */
        while (load_linked (lock)==1) pause(x);

        /* test-and-set */
        value = load_linked (lock);
        ret = store_conditional (lock, 1);
        if (value == 1 || ret == 0)
            pause (x);
    } while (value ==1 || ret == 0);
}
```

## Tema 5

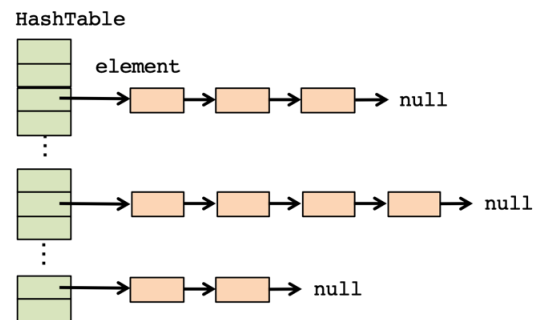
Comencem amb el Tema 5 (Data Decomposition or millor dit, Data-aware Task Decomposition) d'una manera poc usual, que és fent un problema, el **Problema 2** que ens permetrà veure les diferents opcions que tenim de paral·lelitzar tenint en compte la distribució de les dades.

Considerem el programa següent:

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

element * HashTable[SIZE_TABLE];

#define MAX_ELEM 1024
int ToInsert[MAX_ELEM], num_elem, index;
...
for (i = 0; i < num_elem; i++) {
    index = hash_function(ToInsert[i], SIZE_TABLE);
    insert_elem (ToInsert[i], index);
}
...
```



Es tracta d'inserir cadascun dels elements que tenim en el vector ToInsert (num\_elem elements, mida màxima MAX\_ELEM) en la llista corresponent dins de la taula de hash HashTable, de mida SIZE\_TABLE; l'entrada de la taula de hash la determina la funció hash\_function. Per tant tenim dues estructures de dades en el problema: vector

ToInsert[MAX\_ELEM] (només lectura) i taula de hash HashTable[SIZE\_TABLE] (lectura i escriptura).

Ja havíem vist una paral·lelització per aquest programa, simplement indicant que el bucle s'executa amb tasques explícites:

```
omp_lock_t LockTable[SIZE_TABLE]; // One should initialize (with omp_init_lock) and destroy
                                   // (with omp_destroy_lock) each lock in LockTable

#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(GRANULARITY)
for (i = 0; i < num_elem; i++) {
    index = hash_function(ToInsert[i], SIZE_TABLE);
    omp_set_lock(&LockTable[index]);
    insert_elem (ToInsert[i], index);
    omp_unset_lock(&LockTable[index]);
}
```

Com determinariem el valor de la constant GRANULARITY? Si volem aprofitar la localitat espacial en l'accés al vector ToInsert hauria de ser un múltiple del nombre d'elements que hi caben en una línia de cache, p.e. si un enter ocupa 4 bytes i la línia són 64 bytes, doncs el valor mínim per GRANULARITY seria 64/4=8.

Però observeu que, donat el caràcter dinàmic de les tasques, no hi ha manera de garantir quines línies de cache accedeix cada tasca/thread/processor. I si volguéssim assegurar un repartiment d'elements a processadors com es mostra a continuació?

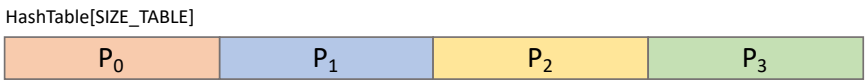


I perquè ho voldríem així? Doncs potser per què abans aquests són els processadors que han calculat aquests elements. Suposso que tenim clar que amb GRANULARITY=MAX\_ELEM/omp\_get\_num\_threads() no serveix, cert? O equivalentment amb NUM\_TASKS(omp\_get\_num\_threads()). El caràcter dinàmic de les tasques no assegura que un processador accedeixi als elements que el pertocuen. Ho sabríeu fer amb tasques implícites?

```
#pragma omp parallel private(index, i)
{
    // cada tasca implicita calcula el rang d'elements de ToInsert que li toquen
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (MAX_ELEM/numprocs);
    int upper = lower + (MAX_ELEM/numprocs);
    for (i = lower; i < min(upper, num_elems); i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        omp_set_lock(&LockTable[index]);
        insert_elem (ToInsert[i], index);
        omp_unset_lock(&LockTable[index]);
    }
}
```

Això seria una paral·lelització anomenada **INPUT BLOCK DATA DECOMPOSITION**, ja que el vector que només és d'entrada (de lectura) és el que determina el repartiment de feina (iteracions). Hem d'assegurar que cada processador només accedeixi a aquells elements que li pertocuen, això és el que anomenarem la **OWNER COMPUTES RULE**, és a dir el propietari és el que fa la inserció a la HashTable.

Quant a l'accés a la HashTable ... quins dos problemes tenim? El primer que necessitem protegir amb un lock el seu accés, amb l'overhead que això implica. I el segon, és que no tenim cap mena de localitat en l'accés a HashTable. Perquè? Doncs per què qualsevol thread pot voler inserir en qualsevol entrada. Podríem fer una estratègia de paral·lelització que assegurés un repartiment de les entrades de la HashTable entre tasques/threads/processadors? Per exemple:



És el que anomenaríem una **OUTPUT BLOCK DATA DECOMPOSITION**, ja que la HashTable és una estructura de dades de sortida (resultat del bucle paral·lel). Quin codi paral·lel hauríem d'escriure per aconseguir-ho?

```
#pragma omp parallel private(index, i)
{
    // cada tasca implícita calcula el rang d'elements de HashTable que li toquen
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int lower = myid * (SIZE_TABLE/numprocs);
    int upper = lower + (SIZE_TABLE/numprocs);
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        if ((index >= lower) && (index < upper))
            insert_elem (ToInsert[i], HashTable[index].first);
    }
}
```

I és més, NO necessitem cap mena de sincronització: cada tasca implícita només accedeix a aquells elements que te en la seva memòria, i ningú més hi accedeix! Però ja sabem que res és gratuït en aquesta vida ... aquí estem pagant el cost del fet que cada tasca implícita ha de recórrer TOT el vector ToInsert per detectar quins elements s'han d'inserir en les entrades de la HashTable li pertocuen. És el mateix concepte d'abans, la **Owner Computes Rule** per la que les tasques executades en un processador es responsabilitzen d'actualitzar, en aquest cas, les entrades de la taula de hash de les quals el processador en és el propietari.

Summary:

Podem tenir les següents distribucions possibles (només mostrem BLOCK):

	ToInsert BLOCK distributed	ToInsert in single node
HashTable BLOCK distributed	<div><div>ToInsert[MAX_ELEM]</div><div><div>P<sub>0</sub></div><div>P<sub>1</sub></div><div>P<sub>2</sub></div><div>P<sub>3</sub></div></div><div>HashTable[SIZE_TABLE]</div><div><div>P<sub>0</sub></div><div>P<sub>1</sub></div><div>P<sub>2</sub></div><div>P<sub>3</sub></div></div></div>	<div><div>ToInsert[MAX_ELEM]</div><div><div>P<sub>0</sub></div></div><div>HashTable[SIZE_TABLE]</div><div><div>P<sub>0</sub></div><div>P<sub>1</sub></div><div>P<sub>2</sub></div><div>P<sub>3</sub></div></div></div>
HashTable in single node	<div><div>ToInsert[MAX_ELEM]</div><div><div>P<sub>0</sub></div><div>P<sub>1</sub></div><div>P<sub>2</sub></div><div>P<sub>3</sub></div></div><div>HashTable[SIZE_TABLE]</div><div><div>P<sub>0</sub></div></div></div>	<div><div>ToInsert[MAX_ELEM]</div><div><div>P<sub>0</sub></div></div><div>HashTable[SIZE_TABLE]</div><div><div>P<sub>0</sub></div></div></div>

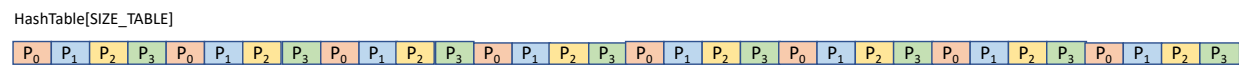
Si fem la paral·lelització del bucle i (INPUT BLOCK DATA DECOMPOSITION) llavors tindríem:

	ToInsert BLOCK distributed	ToInsert in single node
HashTable BLOCK distributed	Accés local a ToInsert (només lectura). Accés remot per actualitzar HashTable, amb sincronització.	Accés remot a ToInsert. Congestió en l'accés a ToInsert. Accés remot per llegir/escriure HashTable, amb sincronització.
HashTable in single node	Accés local a ToInsert (només lectura). Accés remot per llegir/escriure HashTable, amb sincronització. Congestió en l'accés a HashTable.	Accés remot a ToInsert. Congestió en l'accés a ToInsert. Accés remot per llegir/escriure HashTable, amb sincronització. Congestió en l'accés a HashTable.

I si en canvi fem la paral·lelització alternativa (OUTPUT BLOCK DATA DECOMPOSITION) llavors tindríem:

	ToInsert BLOCK distributed	ToInsert in single node
HashTable BLOCK distributed	Accés remot a ToInsert (només lectura). Accés local per actualitzar HashTable, sense sincronització.	Accés remot a ToInsert. Congestió en l'accés a ToInsert. Accés local per llegir/escriure HashTable, sense sincronització.
HashTable in single node	Accés remot a ToInsert (només lectura). Accés remot per llegir/escriure HashTable, sense sincronització. Congestió en l'accés a HashTable.	Accés remot a ToInsert. Congestió en l'accés a ToInsert. Accés remot per llegir/escriure HashTable, sense sincronització. Congestió en l'accés a HashTable.

I si hagués resultat que per algun motiu relacionat amb load balancing la HashTable s'haguera distribuït de forma *cyclic*? És a dir una **OUTPUT CYCLIC DATA DECOMPOSITION**?

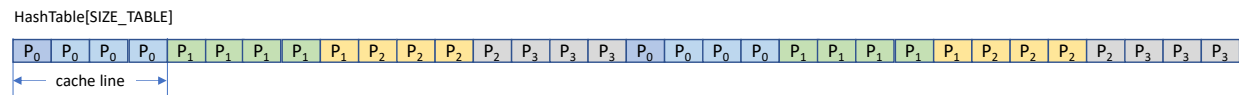


El codi per aconseguir-ho seria com es mostra a continuació:

```
#pragma omp parallel private(index, i)
{
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();

    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        if ((index % numprocs) == myid)
            insert_elem (ToInsert[i], HashTable[index].first);
    }
}
```

O també podríem fer una **OUTPUT BLOCK-CYCLIC DATA DECOMPOSITION**? Si, així cada tasca implícita s'encarregaria de totes les entrades consecutives de la taula de hash que caben en una línia de cache:



El codi per aconseguir-ho seria com es mostra a continuació:

```
#pragma omp parallel private(index, i)
{
    int myid = omp_get_thread_num();
    int numprocs = omp_get_num_threads();

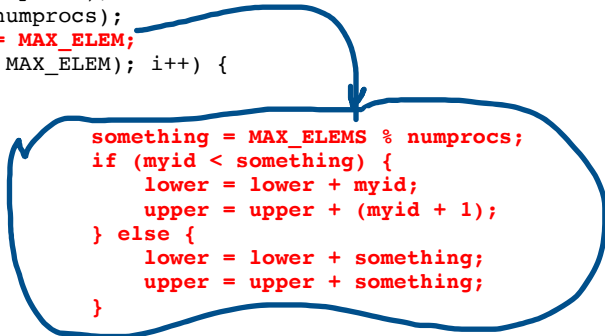
    for (i = 0; i < MAX_ELEM; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        if ((index/BS) % numprocs == myid)
            insert_elem (ToInsert[i], HashTable[index].first);
    }
}
```

on BS és el nombre d'elements consecutius per processador (de fet els que comparteixen una mateixa línia de cache).

**Homework 1: com seria el codi si volem fer INPUT CYCLIC DATA DECOMPOSITION o INPUT BLOCK-CYCLIC DATA DECOMPOSITION?**

**Homework 2: quina diferència hi ha entre els dos codis següents en quant al repartiment d'iteracions en una distribució BLOCK, quan MAX\_ELEM no és múltiple de P?**

```
int myid = omp_get_thread_num();
int numprocs = omp_get_num_threads();
int lower = myid * (MAX_ELEM/numprocs);
int upper = lower + (MAX_ELEM/numprocs);
if (myid == numprocs-1) upper = MAX_ELEM;
for (i = lower; i < min(upper, MAX_ELEM); i++) {
    ...
}
```



```
something = MAX_ELEMS % numprocs;
if (myid < something) {
    lower = lower + myid;
    upper = upper + (myid + 1);
} else {
    lower = lower + something;
    upper = upper + something;
}
```

Quina és quina?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Quin es el màxim desbalanceig de càrrega en cada cas?

Problemes a fer: Problema 3, 5 i 8 del Tema 5