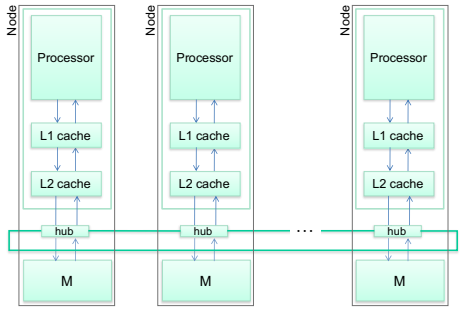
Guió i material per classe no presencial PAR grup 20 Dilluns 12/4/2021, 10:00-12:00

Repàs qüestionari Atenea sobre NUMA coherence



In a NUMA (Non-Uniform Memory Access time) multiprocessor there are two or more identical (NUMA) nodes, each one with a processor and its complete memory hierarchy, including a portion of main memory. The overall memory of the system is physically distributed among all the nodes but logically shared by all of them (i.e., the processor in any node can access to its main memory and also to any memory location in any other node through the interconnection network).

T A given physical memory address can only be stored in the memory of a single node, although multiple copies of the line containing that address may be temporarily stored in the cache memories of other nodes.

F Instructions different from the conventional load and store are required to access to variables stored in other nodes.

- The way data is distributed among the different nodes of a NUMA multiprocessor system ...
  - **F** ... does not have any impact in the performance of the parallel application.
  - **T** ... is determined by the operating system based on a given data allocation policy (for example, first touch).
  - **F** ... dynamically changes with the objective of balancing the number of local accesses that are performed by the processors in the different NUMA nodes.
  - **F** ... is statically determined by the compiler, based on the accesses that are performed by the tasks in the parallel program.

- Assume that the coherence in a NUMA multiprocessor is based on a directory attached to the main memory in each node. The directory structure present in each node provides ...
  - **F** ... coherence information for the memory lines that are stored in the cache memory of the same node.
  - **F** ... information that allows a processor to find the data that is allocated in other nodes.
  - **T** ... information to keep coherent all possible copies in cache of the lines stored in the memory of that node.
  - **F** ... information that allows a processor in that node to find the nearest node where to find a given memory address, in order to minimize the memory access time.

T In a NUMA multiprocessor system, with directory-based coherence protocol, the number of bits in each entry of the directory depends on the number of nodes in the system, with one or several additional bits to keep the state of the associated line.

- The number of entries in the directory of a NUMA node ...
  - **F** ... is the total number of cache lines in the overall NUMA system, helping to identify which caches have a copy of a memory line.
  - **F** ... is determined by the maximum number of copies that are allowed for each line in main memory.
  - **T** ... is the number of lines that are stored in the main memory associated to it.
  - **F** ... depends on the number of NUMA nodes in the system in order to implement the list of nodes with remote copies.

- Assume a NUMA multiprocessor architecture with 1024 nodes, each node with a single processor and 24 GB of main memory, with directory-based MSU coherence protocol; memory lines are 128

bytes wide. In that system, which is the percentage of the whole main memory (**including both data and directory**) that is used by the directory to store all the information related to coherence?

F With the information provided one can not compute the number. You should have provided the size of the cache memory in each node to be able to compute the requested percentage.

F close to 200%

T close to 50%

F close to 100%

Directory entry per memory line: 2 bits status and 1024 bits presence. Total bytes per line? T=128 (data) + (1024+2)/8. Therefore, the percentage is ((1024+2)/8 / T) * 100 = 50,1%.

F False sharing cannot occur across nodes in a NUMA multiprocessor architecture because the directory structure attached to main memory provides information about the location of each variable in the same line.

F In a NUMA multiprocessor architecture, false sharing implies the simultaneous existence of at least two copies of the same cache line in M state in the associated directory entry.

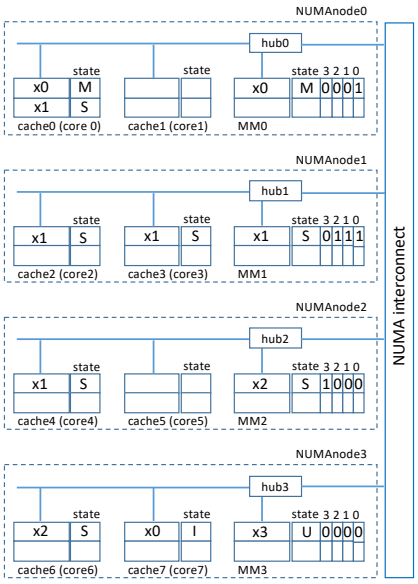
F Two different processors in a cache-coherent multiprocessor architecture (either UMA or NUMA) continuously executing a count++ instruction originate a false sharing situation.

Problema 4 – Tema 3

System with 4 nodes and a total of 8 GB of main memory, line size of 32 bytes and private per-processor caches of 4 MB per processor.

a1) Amount of memory taken by snoopy to keep coherence among caches inside each node:
#bits = #cache lines x 2 bits (MSI) = (4 MB / 32 B) x 2 = (4 x 2^{20} / 2^5) x 2 = 2^{18} bits
in KBytes = 2^{18} bits x (1 B / 2^3 bits) x (1 KB / 2^{10} B) = 2^5 KB = 32 KB
in MBytes = 2^{18} bits x (1 B / 2^3 bits) x (1 MB / 2^{20} B) = 2^{-5} MB = 0,03125 MB

a2) Amount of memory taken by directory to keep coherence among nodes in the whole system:
#bits = #memory lines x #bits per entry = (2 GB / 32 B) x (4 + 2) = (2 x 2^{30} / 2^5) x 6 = 2^{26} x 6 = 3 x 2^{27} bits
in KBytes = 3 x 2^{27} bits x (1 B / 2^3 bits) x (1 KB / 2^{10} B) = 3 x 2^{14} KB = 49152 KB
in MBytes = 3 x 2^{27} bits x (1 B / 2^3 bits) x (1 MB / 2^{20} B) = 3 x 2^4 MB = 48 MB

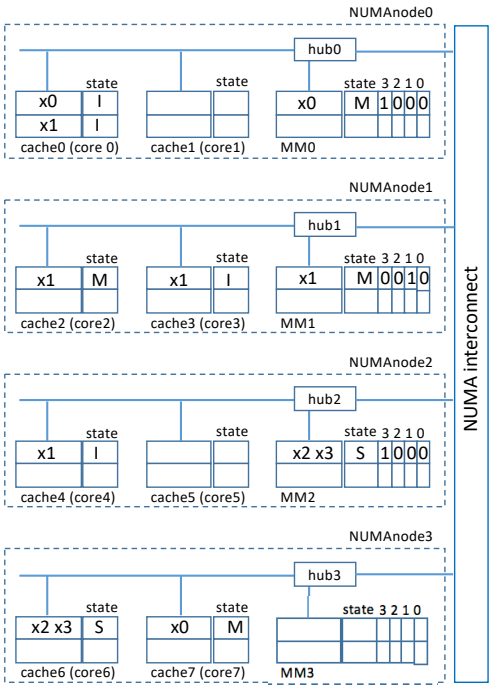


b) Which line is in wrong state, assuming variables x0, x1, x2 and x3 (4 bytes wide each), variables x2 and x3 reside in the same cache line and variables x0 and x1 reside in a different cache line each?

Variables x2 and x3 cannot be mapped in two different NUMA nodes. Each memory line **can only exist in one home node.** Since variable x3 does not exist in NUMAnode3, we just add it in the same line as the one containing x2 in NUMAnode2.

c) core$_7$ writes x0. Indicate final state in memories.

This makes a miss, so NUMAnode$_3$ requests line to NUMAnode$_0$. MM$_0$ needs to provide line but first it needs to get it from private cache in core$_0$. Status in cache$_0$ goes to I. Status in the directory for the line in MM$_0$ doesn't change but sharers bit list yes to reflect location of new dirty line. Status in cache$_7$ goes to M.



BusRdX$_7$ → WrReq$_{3\to0}$ → BusRdX$_{hub0}$ → Flush$_0$ → Dreply$_{0\to3}$

d) core$_2$ writes x1. Indicate final state in memories.

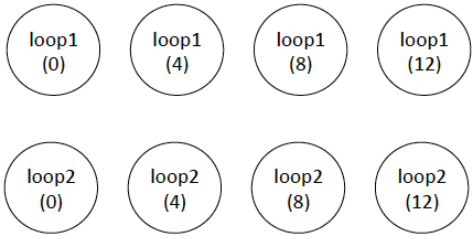
This makes a hit and a BusUpgr$_2$ inside the node, invalidating the copy in private cache of core$_3$. The hub in NUMAnode$_1$ needs to invalidate all other copies in other NUMAnodes (0 and 2). Status changes from S to M, with only NUMAnode1 active in the sharers bit list.

BusUpgr$_2$ → Invalidate$_{1\to0}$ → BusUpgr$_{hub0}$ → Ack$_{0\to1}$
→ Invalidate$_{1\to2}$ → BusUpgr$_{hub2}$ → Ack$_{2\to1}$

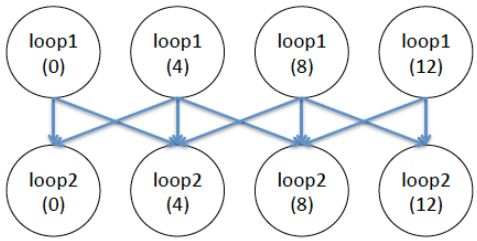
Problema 10 Tema 2

```
#define N 16
#define BS 4
void main() {
char stringMessage[16];
tareador_ON();
for (int ii = 0; ii < N; ii=ii+BS) {
    sprintf(stringMessage,"loop1(%d)",ii);
    tareador_start_task(stringMessage);
    for (int i = ii;  i< ii+BS; i++)                  // BS perfectly divides N
        for (int j = 0; j < N; j++)
            b[i][j] = foo(a[i][j]);
    tareador_end_task(stringMessage);
    }

for (int ii = 0; ii < N; ii=ii+BS) {
    sprintf(stringMessage,"loop2(%d)",ii);
    tareador_start_task(stringMessage);
    for (int i = max(1,ii); i < min(ii+BS, N-1); i++) // min y max to ensure the access
                                                      // within the range of matrix b
        for (int j = 0; j < N; j++)
            c[i][j] = goo(b[i][j], b[i-1][j], b[i+1][j]);
    tareador_end_task(stringMessage);
    }
tareador_OFF();
}
```

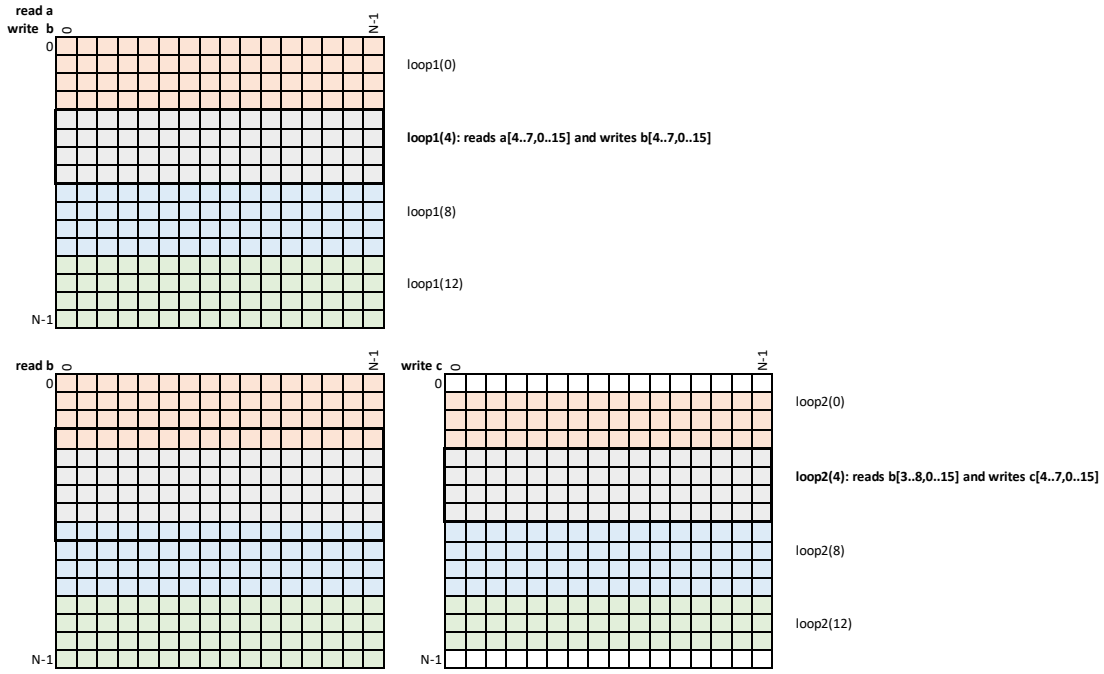
(a) Complete the following task dependence graph (TDG), assuming functions `foo` and `goo` only access to their arguments and do not update any other global variable.



Solución:



Observeu que l'obtenim analitzant els accessos a les matrius a, b i c que fan les en el primer i el segon bucle, tal com es mostra a continuació:



(b) Write the expression that determines the execution time T_4, clearly indicating the contribution of the computation time $T_{4(comp)}$ and data sharing overhead $T_{4(mov)}$, for the two following assignments of tasks to processors:

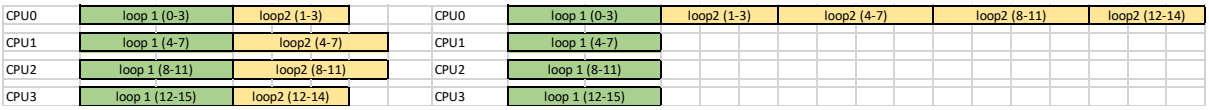
Task	Assignment 1	Assignment 2
loop1(0)	0	0
loop1(4)	1	1
loop1(8)	2	2
loop1(12)	3	3
loop2(0)	0	0
loop2(4)	1	0
loop2(8)	2	0
loop2(12)	3	0

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrices `a`, `b` and `c` are initially distributed by rows (N/BS consecutive rows per processor); 3) once the second loop is finished, you don't need the return matrices to their original distribution; 4) data sharing model with $t_{comm} = t_s + m \times t_w$, being t_s y t_w the start–up time and transfer time of one element, respectively; and 5) the execution time for a single iteration of the innermost loop body takes t_c.

Solución:

	Assignment 1	Assignment 2
$T_{4(comp)}$	$2 \times 64 \times t_c$	$(64 + 224) \times t_c$
$T_{4(mov)}$	$2 \times (t_s + 16 \times t_w)$	$3 \times (t_s + 64 \times t_w)$

Per la part de càlcul els diagrames temporals serien:



Per la part de moviment de dades: amb el Assignment 2 P0 necessita accedir a les parts de la matriu b que estan en els processadors P1, P2 i P3. En canvi, amb l'Assignment 1 cada processador només necessita accedir a l'última fila del processador anterior (excepte P0) i la primera fila del processador següent (excepte P3). Observeu que l'escriptura de C la fa P0 sobre una còpia local de la matriu, no cal retornar-la als processadors P1, P2 i P3, tal com diu l'enunciat.

.

Problema 6 – Tema 3

Given the following code excerpt including OpenMP directives:

```
int vector[N];
typedef struct {
    int even = 0;
    int odd = 0;
    int dummy[PAD];
} count[NUM_THREADS];
...
#pragma omp parallel
{
   int id = omp_get_thread_num();
   int nt = omp_get_num_threads();
   for (int ii=id*CHUNK; ii < N; ii += nt*CHUNK)
       for (int i=ii; i < min(N, ii+CHUNK); i++)
           if (vector[i]%2) contar[id].odd++;
           else contar[id].even++;
}
```

count

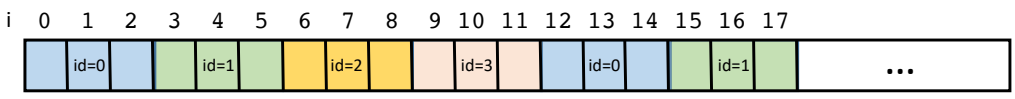
in which each implicit task executes groups of consecutive `CHUNK` iterations in a round robin (cyclic) way. Assuming that each integer (`int`) variable occupies 4 bytes, a cache line occupies 32 bytes, and that the initial address of `vector` and `count` are aligned with the start of a cache line, we ask:

(a) Compute the minimum value for constant `PAD` in the previous program in order to avoid *"false sharing"* during the execution of the parallel loop.

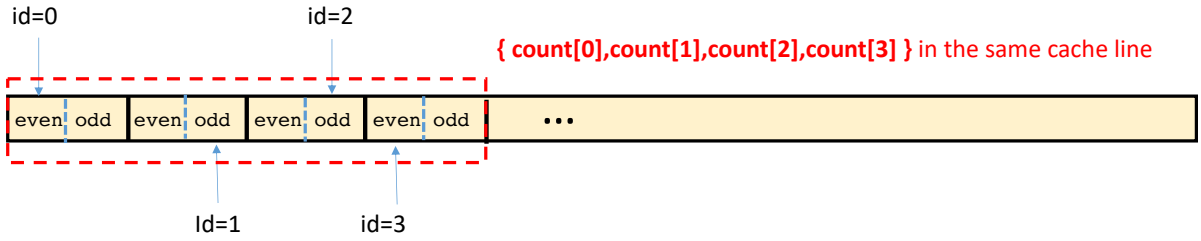
(b) Compute the minimum value for constant `CHUNK` in order to improve spatial locality, within each thread, when reading the elements of `vector`.

Abans de res observem com es reparteixen les iteracions de i entre els threads:

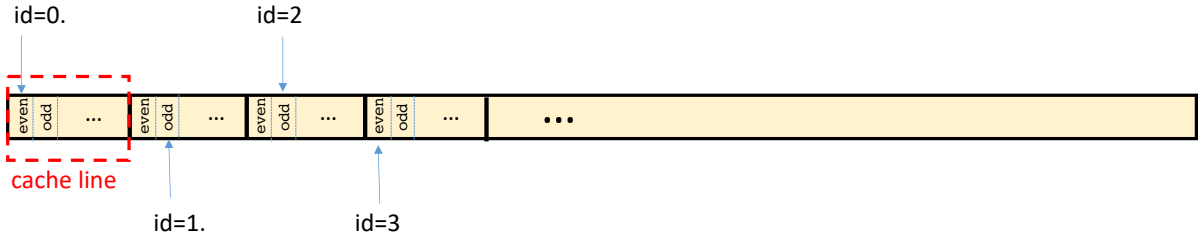
For CHUNK=3



a) Mirem com estan emmagatzemats els elements de `count` en memòria. Primer considerem que `PAD=0`. Cada enter son 4 bytes, per tant cada element de contar ocupa 8 bytes. Si cada línia de cache son 32 bytes tenim que hi caben 4 elements consecutius del vector `count`.

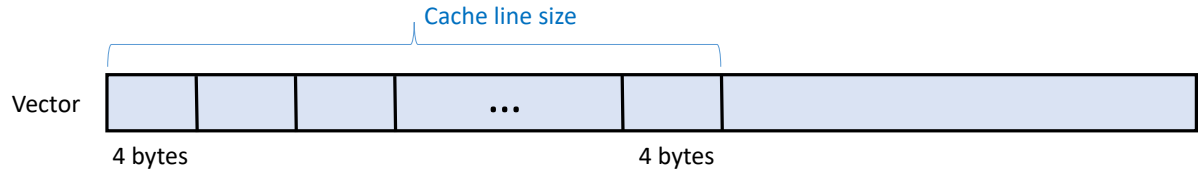


El problema apareix per que son 4 comptadors accedits per 4 threads consecutius, cada thread llegint i escrivint els seus comptadors de even i odd → **false sharing**. Quin valor de PAD ens evitaria l'aparició de false sharing? Doncs aquell valor que faci que cada element de contar ocupi una línia sencera de cache.



Per tant PAD = (cache_line_size – sizeof(even) – sizeof(odd)) / sizeof(int) = (32-8)/8=6.

b) Quin valor mínim hauria de tenir CHUNK per tal de que s'exploti la localitat espacial al accedir-hi. Hem de veure quants elements consecutius caben en una linia de cache:



Per tant, CHUNK = cache_line_size / sizeof(int) = 32/4=8. D'aquesta manera, per cada miss que fa un processador al accedir al primer element de la línia farà 7 hits pels següents. Valors múltiples de 8 també tindrien el mateix efecte, però ens demanen el valor mínim.

Quizz 2 Atenea NUMA

Given a NUMA multiprocessor system with 4 nodes, each node with one processor, 64 KB of private cache and 8 GB of main memory. Coherence among nodes is kept with a directory-based MSU protocol, using the transactions explained in class.

Assume that the home node for variable *var* is NUMAnode0 and that at a given moment there exist clean copies of that variable in the cache memories in NUMAnode1 and NUMAnode2, but no copies in NUMAnode0 and NUMAnode3. If the processor in NUMAnode1 wants to update (write to) variable *var*, which of the following coherence actions do NOT occur?

Trieu-ne una o més:

- ☐ The processor in the local node (NUMAnode1) issues PrWr.
- ☐ The local node sends a UpgrReq command to the home node (NUMAnode0).
- ☐ Since there is another copy of the line in the cache of a remote node (NUMAnode2), the home node sends an Invalidate command to it.
- ☒ Since there is another copy of the line in the cache of a remote node (NUMAnode2), the local node sends an Invalidate command to it.

En aquest cas, important recordar que, en el protocol explicat, el node local no envia mai cap comanda de coherència als nodes remots; és el home node qui es comunica amb ells ja que és el qui te la llista de nodes remots sharers.

After the previous access, the processor in NUMAnode3 reads the same variable *var*. Which of the following coherence actions do NOT occur?

Trieu-ne una o més:

- [ ] The processor in the local node (NUMAnode3) issues PrRd.
- [x] The local node issues a Fetch command to the home node (NUMAnode0) to obtain the line containing *var*.
- [ ] The home node then issues a Fetch command to the remote NUMAnode1 in order to get the most up-to-date copy of the requested memory line.
- [ ] The remote node issues a Dreply command with the only valid copy of the line in the whole system.

En aquest cas, important recordar que les comandes que s'envien entre node local i home poden ser RdReq, WrReq i UpgrReq, a les que el home respon amb Dreply en els primers dos casos i Ack en el tercer. Entre home i remotes les comandes poden ser Fetch i Invalidate, a les que el node remot respon amb Dreply i Ack, respectivament.

Variacions al Problema 6 – Tema 3

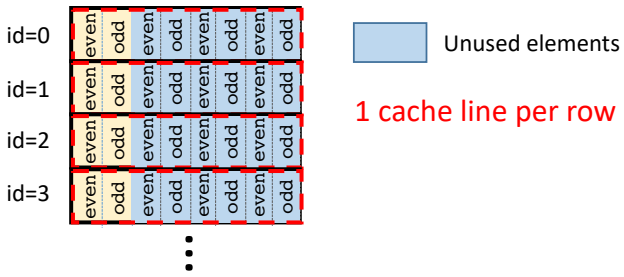
Donat els dos codis alternatius al codi original del Problema 6:

```
int vector[N];
typedef struct {
    int even = 0;
    int odd = 0;
} count[NUM_THREADS][PAD];

...

#pragma omp parallel
{
int id = omp_get_thread_num();
int nt = omp_get_num_threads();
for (int ii=id*CHUNK; ii < N; ii += nt*CHUNK)
    for (int i=ii; i < min(N, ii+CHUNK); i++)
        if (vector[i]%2) count[id][0].odd++;
        else count[id][0].even++;
}
```

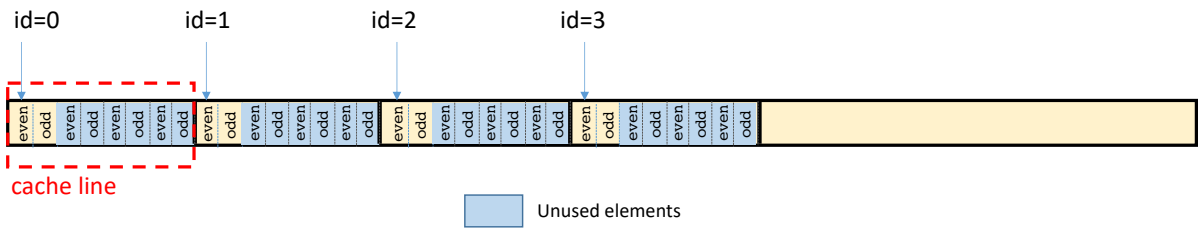
```
int vector[N];
typedef struct {
    int even = 0;
    int odd = 0;
} count[NUM_THREADS*PAD];

...

#pragma omp parallel
{
int id = omp_get_thread_num();
int nt = omp_get_num_threads();
for (int ii=id*CHUNK; ii < N; ii += nt*CHUNK)
    for (int i=ii; i < min(N, ii+CHUNK); i++)
        if (vector[i]%2) count[id*PAD].odd++;
        else count[id*PAD].even++;
}
```

Trobeu el valor de PAD en cada cas que assegura que no hi haurà *false sharing* en el accés a `count`. Observeu com canvia en cada cas la manera d'accedir a `count`.

En el primer cas (esquerra), PAD=4, aiuxí cada fila de la matriu ocupa una línia de cache:



Pel segon cas (dreta), PAD=4, de manera que en cada línia de cache hi ha 1 element usat i 3 de no utilitzats.



En tots el casos, en una línia tenim 8 enters (32 bytes, 4 bytes per int).