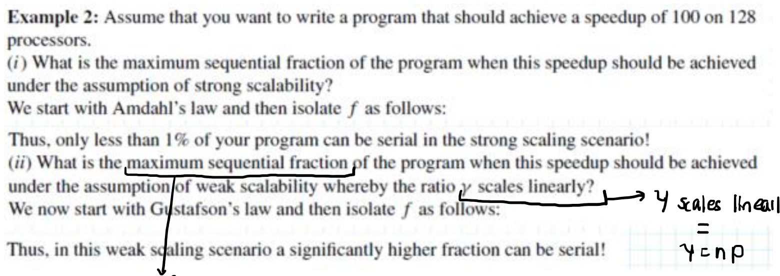
Performance

martes, 22 de marzo de 2022 07:00 a. m.



Example 2: Assume that you want to write a program that should achieve a speedup of 100 on 128 processors.
(*i*) What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of strong scalability?
We start with Amdahl's law and then isolate f as follows:

Thus, only less than 1% of your program can be serial in the strong scaling scenario!
(*ii*) What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of weak scalability whereby the ratio γ scales linearly?
We now start with Gustafson's law and then isolate f as follows:

Thus, in this weak scaling scenario a significantly higher fraction can be serial!

γ scales linearly $=$ $\gamma = np$

¿a?

i) done

ii) Gustafson's Law: Weak scalability, $n = np * n$

$S = [1-a]\,np + a$ with $\gamma = np$, $S = 100$, $np = 128$

Plugging into the Law,

$100 = (1-a)128 + a$
$100 = 128 - 128a + a$
$100 = 128 - 127a$
$127a = 128 - 100$
$127a = 28$
$a = \frac{28}{127} = \boxed{0.22} \rightarrow 22\%$ of Time

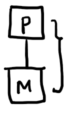
Speedup $\rightarrow$ acceleration

✱ Memory: Cache

Cache: small memory since we want quick access
big memory: bigger distances, electrons take more time to travel = slow

Let's analyze with 1 processor (Von Neumann's Architecture)

Von Neumann arch. { processor connected to Main memory



Processor can process a 100 times one operation than 1 memory access (RAM):

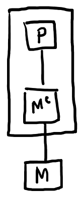
$t_M \approx 100 t_I$ } Problem

t_M → time per mem access
t_I → time per instruction

$=$

Von Neumann's Bottle Neck

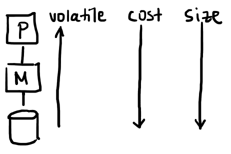
Solution to this: Cache Memory
↳ another memory close to the processor, commonly inside its chip → small memory Fast = More cost than RAM



RAM memory looks for a value the P asks, if it's not on cache, it copies it to cache so that the processor access it from cache.
the advantage is

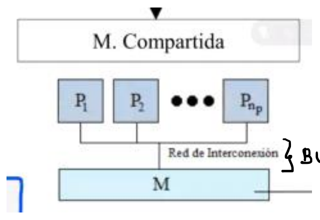
It is based in a principle divided in two
a) Temporal: if x is accessed now it is probable that x is needed later
b) Spatial: if x is accessed now, it is probable that x's surroundings are needed too.
Therefore you copy to cache the space around x (other variables) in case the processor needs it.

If hard drive speed was faster, we wouldn't need RAM Memory:



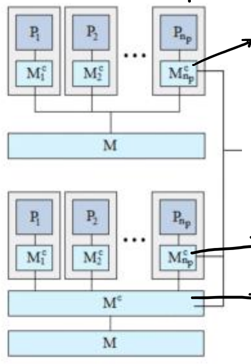
→ RAM Memory is the cache of the hard drive.
→ The source program is distributed into pieces between RAM and cache.

→ For the case of ∞ processors:



} Bus inside computers
↓
Only one thus a problem

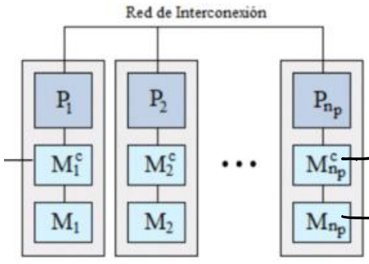
Solution: cache memory, in two ways
Shared Memory



→ one level of cache

the closest to the processor, the quickest the cache is, also the smallest
↑
→ one level of cache memory } we can even have levels of cache here
→ other level of cache memory

Distributed memory

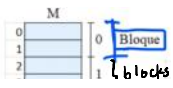


→ one cache memory (may have levels)
→ RAM for each P

How is memory organized?
For 1 processor:

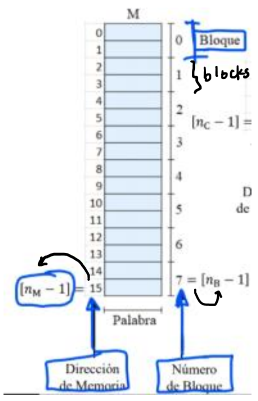
→ Memory usually is organized in words, which is a block with a number that is the address (locality number

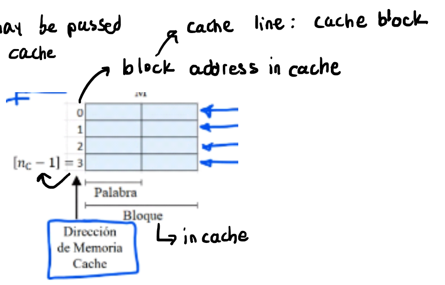
↳ To respect the spatial principle: we copy pieces of ram to the cache, and this means we need to divide RAM into blocks
↓
sets of words } also receive an address



[blocks that may be passed

- cache line: cache block

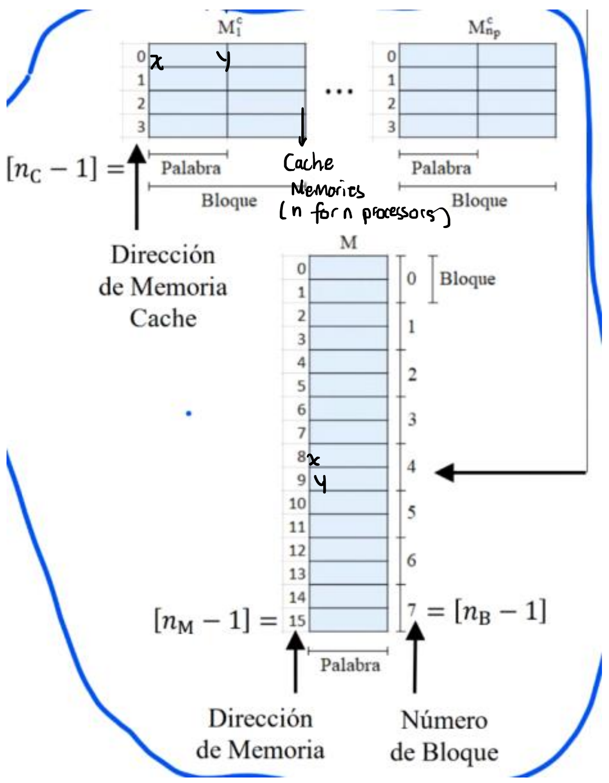




Now the cache memory has also lines, but each line is a block : they are usually called cache lines
↳ in this case has two words space
↳ each RAM block goes to one block / cache line.

If x data value is required by the processor, RAM copies that x word's block to cache.

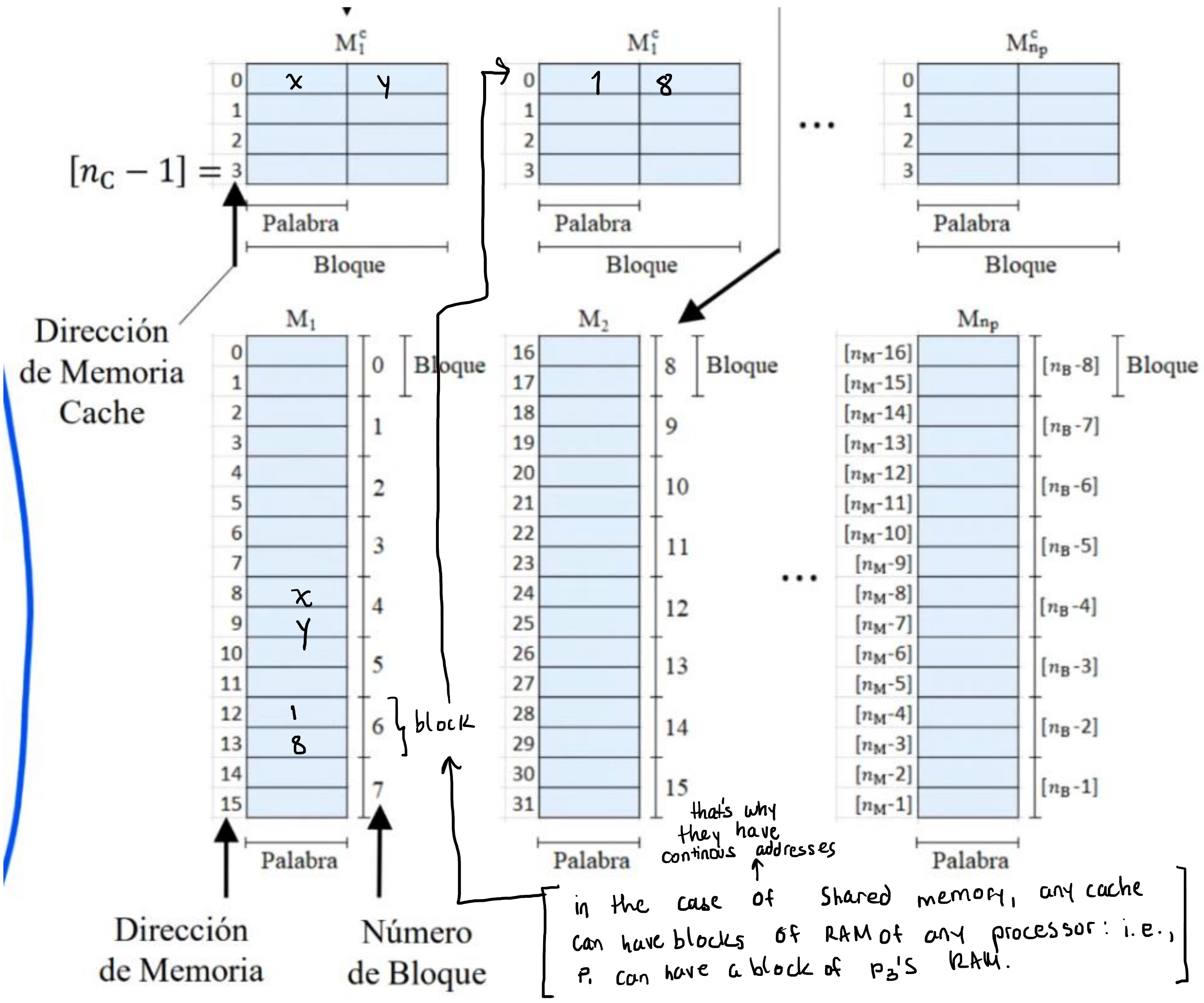
In the case of ∞ processors, i.e., shared memory



Same idea: if x is required, we copy Block 4 into cache (x and y)

the bigger the block, the more probable it is to find something in the cache

In distributed Memory:



M_1^c
M_1^c
M_{np}^c
x
y
1
8
$[n_C - 1] =$
Palabra
Bloque
Dirección de Memoria Cache
M_1
M_2
M_{np}
Bloque
block
Dirección de Memoria
Número de Bloque
$[n_M\text{-}16]$
$[n_B\text{-}8]$
$[n_B\text{-}1]$
that's why they have continous addresses
in the case of Shared memory, any cache can have blocks of RAM of any processor: i.e., P_1 can have a block of P_3's RAM.

→ We have one RAM and one cache per processor.
→ If P_1 wants address 8 data, say x. It is in block 4, so it is copied to cache (the block) into its cache, the processor's cache (P_1's cache, in this case)

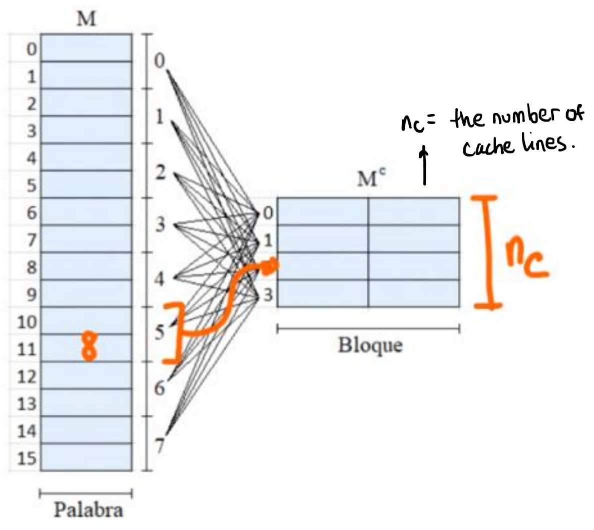
→ What happens if the cache gets full? We delete the cache block that has the most time inside the cache. In this way, we are allowing each $\underbrace{\text{cache block}}_{\text{cache line}}$ to be there.

→ The idea of a distributed memory system the idea is to make memory look like one: that's why they have continous memory addresses, and that's why we can copy to any processor's cache.

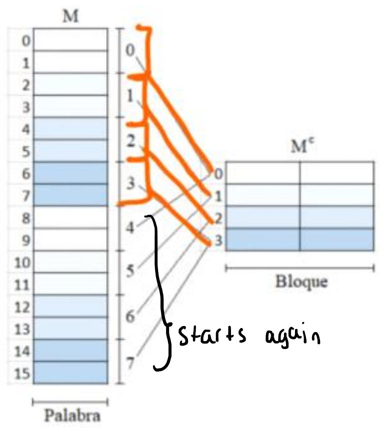
<u>Memory</u> <u>mappings</u>

Mapping: defines in which cache address will a Ram address be copied in

Above, we used the Completely Associative Mapping, where $n = n_C$
The other extreme is: Associative of 1 via



In the direct case and its intermediate case: the first RAM block always goes to the first cache line, and so on. When the cache is full the following RAM address start again in cache line ∅.



As we go further, the electronics and cost increase towards Completely Associative