

Techniques d'optimisation de la parallélisation

Dr. Hugo Taboada

hugo.taboada@cea.fr

- ❖ [Special thanks to Julien Jaeger for a huge part of these slides.](#)
- ❖ <https://www-lihpc.cea.fr/en/team/permanents/taboada.html>

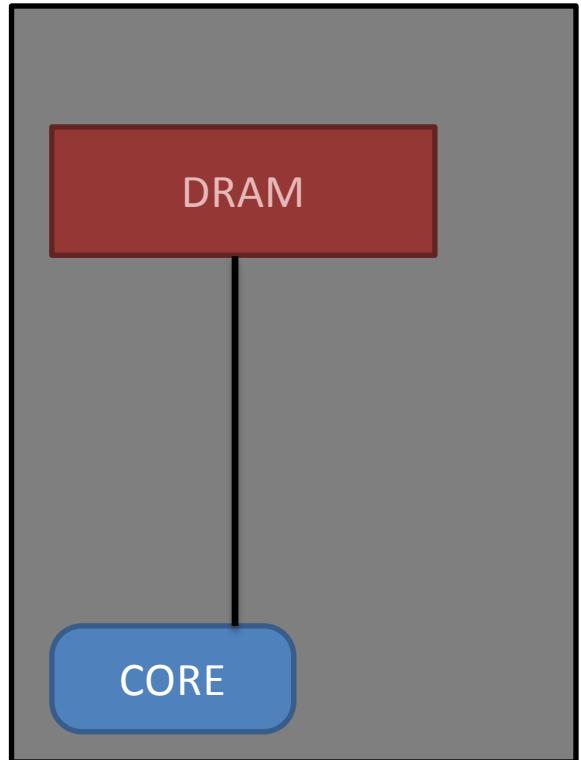


1 ■ Supercomputer architecture

What is a supercomputer ?



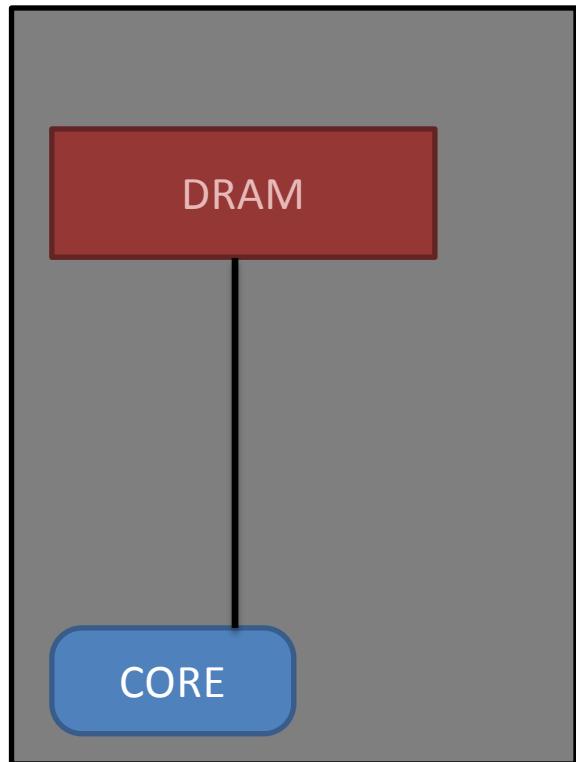
Once upon a time, ...



- ❖ There was an Arithmetic and Logical Unit (ALU) able to realize operations:
 - Arithmetic (add, sub, mul, ...)
 - Logical (AND, OR, ...)
- ❖ ...and a memory to store data



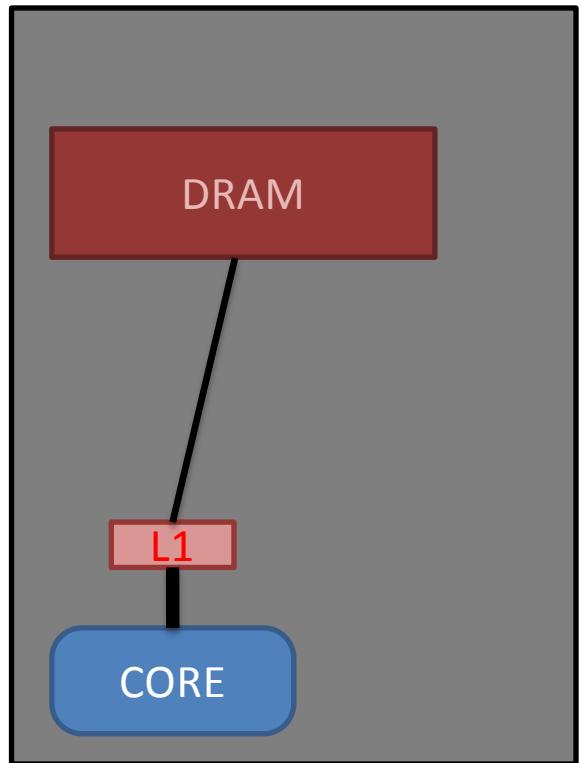
Hitting the memory wall



- ❖ Performance gain was achieved with increase of compute power
 - frequency
- ❖ Problem: compute performances increase faster than memory performances
- ❖ Need to find ways to “feed” the compute unit
 - Otherwise, it is not possible to achieve peak performance



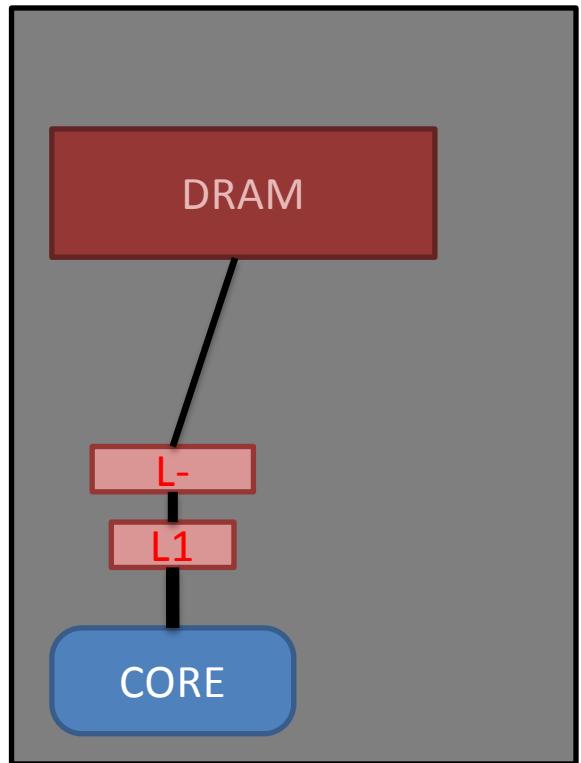
Let there be caches



- ❖ So a data cache was added between compute unit and data memory
- ❖ Higher performance memory closer to the compute unit
 - Better latency
 - Better bandwidth
- ❖ Act like a “buffer” memory close to the compute unit
 - Keep temporarily last used data



HPC saw it was good!

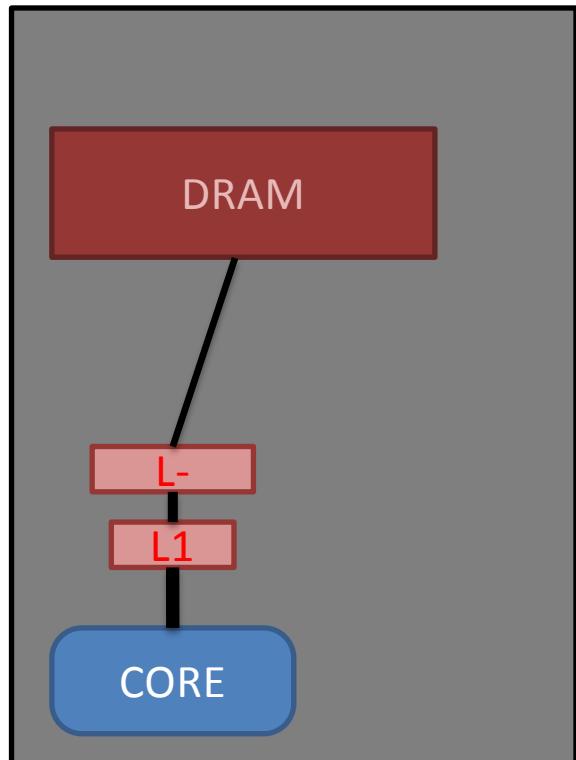


- ❖ Only one level of cache may not be enough
 - Too small to keep all data of interests
 - → hard to play its role and feed the compute unit efficiently
- ❖ Add more cache levels
 - Depending on architecture (often 2 or 3 levels)
 - Further the cache, lower the bandwidth and higher latency and capacity



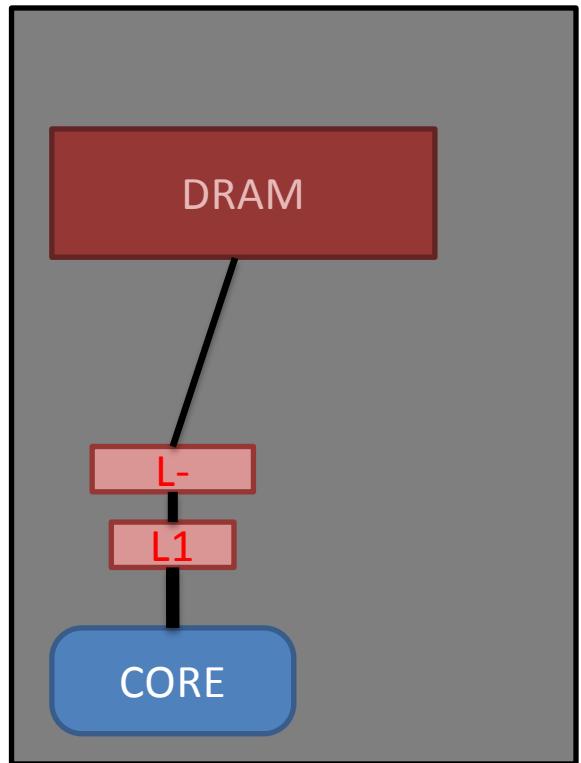
Scotty, we need more (compute) power

- ❖ Each new generation of supercomputers (and processor) generation aims at providing more compute power
- ❖ Main way to reach more compute power: increase the processor frequency





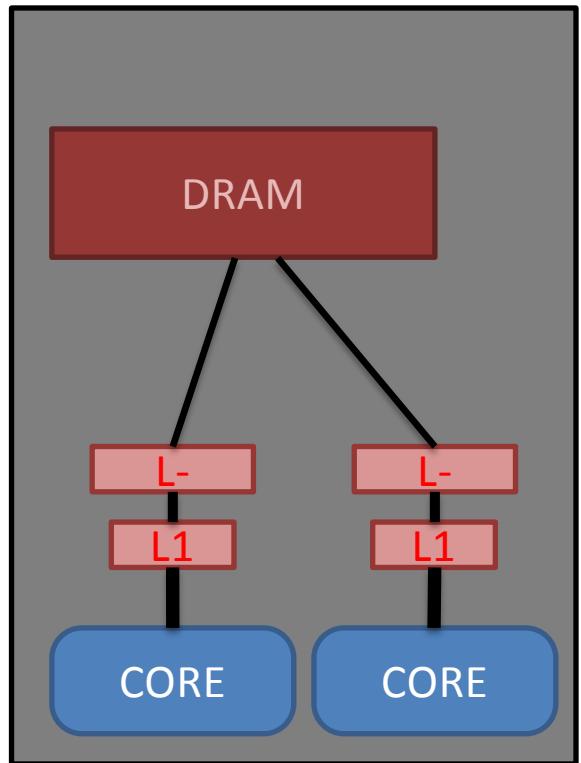
Hitting the heating wall



- ❖ Increase frequency is done with increase of number of electrical transistors
- ❖ Usually roughly keeping the same size of chip
 - Leads to reducing transistor size
- ❖ Density of transistor keeps increasing
 - So does the electrical generated heat
 - Double the density, n^2 increase of heat
 - Too much heat impossible to diffuse



Multiplying cores

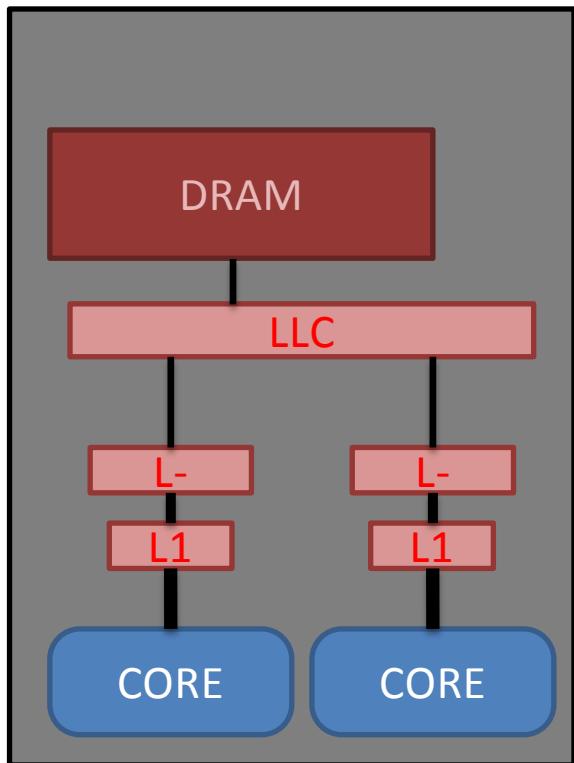


- ❖ So, instead of increasing number of transistor per compute unit...
- ❖ ... let's multiply number of compute units
- ❖ Allows to increase compute power while limiting proximity and density of transistors
 - Double the cores, double the heat



Caches are goooooood!

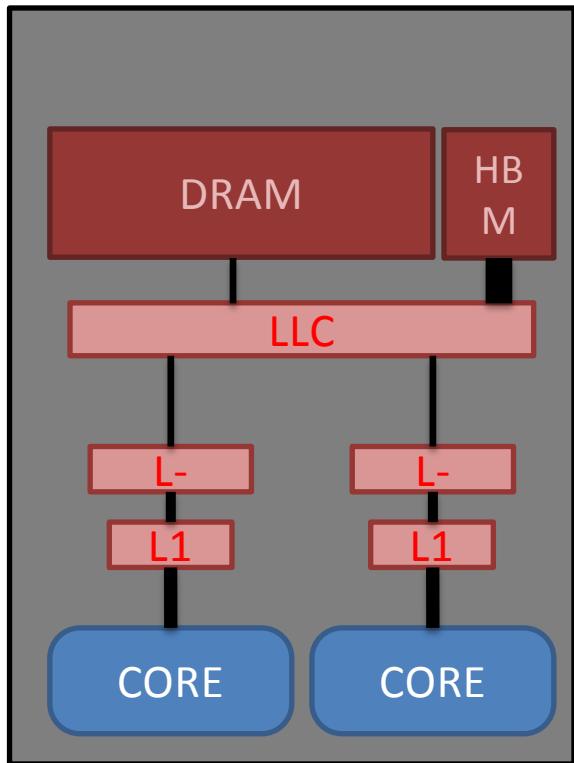
- ❖ Sharing data between cores can happen only through global memory
 - Not very efficient if the cores need to use the same data
 - In this case, caches are not useful
- ❖ Often, the last level cache (LLC) is shared between cores
 - Avoid to go the main memory each time a core need a shared data
 - Also possible to have intermediate shared caches





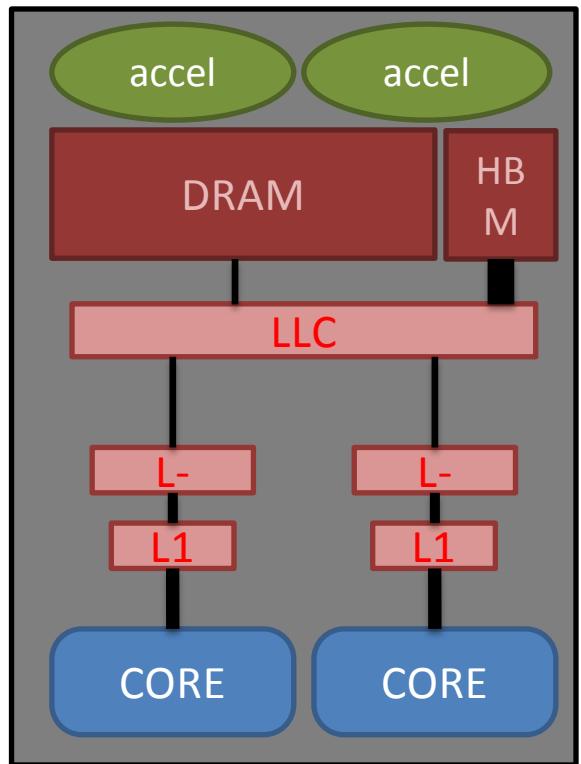
Always need more data

- ❖ Increasing number of cores (and generally compute power) requires always more data to be transferred
- ❖ Even with caches, having only one memory with low bandwidth becomes problematic
- ❖ Adding more global memories, with different characteristics (HBM, NVMe)





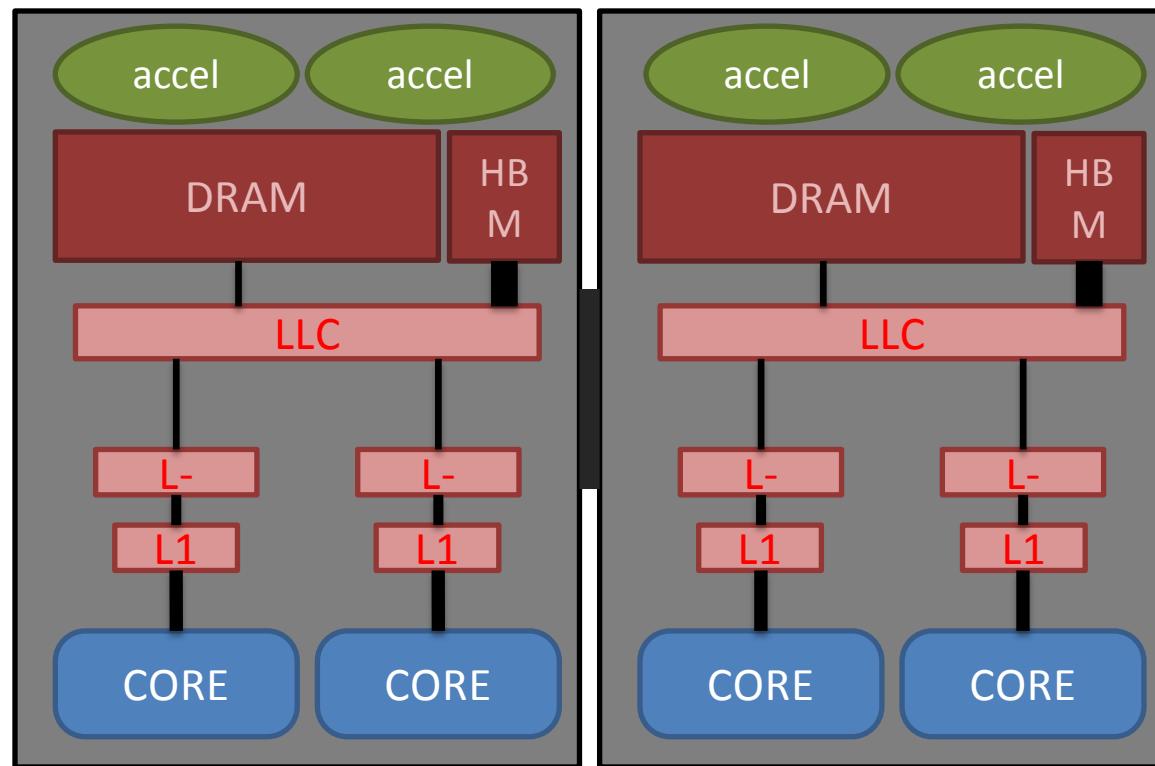
Always need more compute power (with capped power)



- ❖ Cores are built to handle every possible operations
 - Require lots of transistors for all cases
- ❖ Increasing number of cores requires more and more electric power
 - Back to hitting the heating wall
 - Need to provide compute power with less energy
- ❖ Accelerators attached to nodes
 - More cores but specialized -> less power



Building a Supercomputer: Multiplying sockets

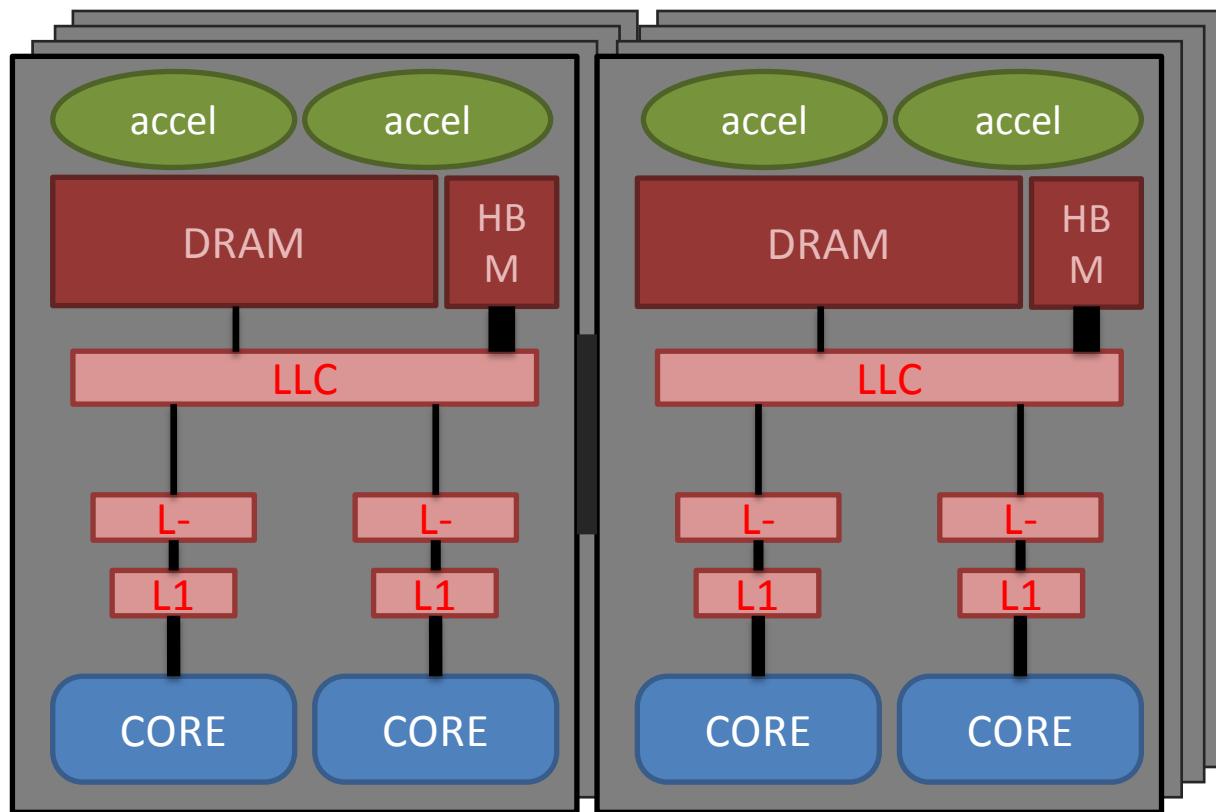


- ❖ Hard to continue extending this number of elements within a limited size
- ❖ Solution: multiply these complex structures and link them together



Building a Supercomputer: Multiplying nodes

- ❖ Same, difficult to continue multiplying sockets
- ❖ Solution: multiply compute nodes, and link them together with a network





Building a Supercomputer: a cabinet



- ❖ Nodes are combined in a rack
- ❖ Racks are combined together inside a cabinet
- ❖ Need network links at every step.



Building a Supercomputer: the supercomputer



- ❖ Finally, cabinets are linked together through network links
 - Network cables
 - Network switches

- ❖ A supercomputer is the whole ensemble of cabinets
 - Compute nodes
 - I/O nodes
 - Login nodes

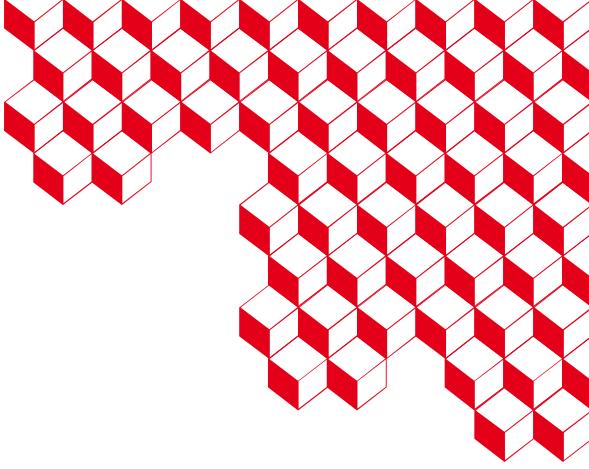


Example: Tera-1000 & Exa-1





2 ■ Memory layout and cache blocking



Memory layout



Memory layout

- ❖ When a program reads data from the main memory (ram), it fetches more than just the required cell
- ❖ Memory transfer unit is the cache line
 - Cache line size depends on the architecture
- ❖ When reading an int (4B) with a cache line of 128B, it fetches 32 ints
 - Contiguous in memory



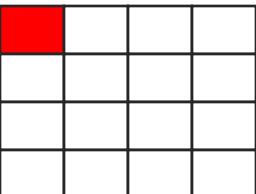
Memory layout

- ❖ To have performance, it is mandatory to follow this hardware feature
 - Reading N ints will cost $N/32$ memory accesses if the accessed cells are contiguous,
 - N memory accesses otherwise
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - C is « row-major »
 - For multi-dimensionnal arrays, data on lines are contiguous
 - $A[8][16]$ for(... $i < 8$...) for(... $j < 16$...)



Memory layout

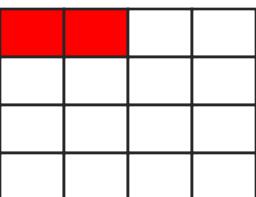
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - C is « row-major »
 - For multi-dimensionnal arrays, data on lines are contiguous
 - A[8][16] for(...i<8...) for(...j<16...)
 - J=0





Memory layout

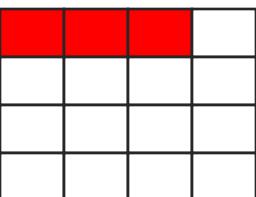
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - C is « row-major »
 - For multi-dimensionnal arrays, data on lines are contiguous
 - A[8][16] for(...i<8...) for(...j<16...)
 - J=1





Memory layout

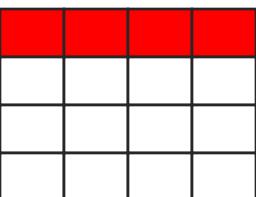
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - C is « row-major »
 - For multi-dimensionnal arrays, data on lines are contiguous
 - A[8][16] for(...i<8...) for(...j<16...)
 - J=2





Memory layout

- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - C is « row-major »
 - For multi-dimensionnal arrays, data on lines are contiguous
 - A[8][16] for(...i<8...) for(...j<16...)
 - J=3





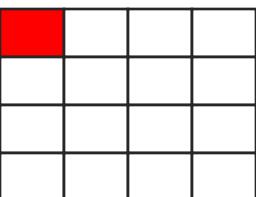
Memory layout

- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - Fortran is « column-major »
 - For multi-dimensionnal arrays, data on columns are contiguous
 - $A[8][16]$ for(... $i < 8$...) for(... $j < 16$...)



Memory layout

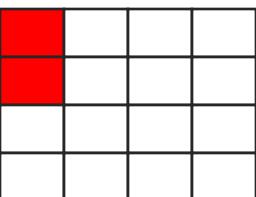
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - Fortran is « column-major »
 - For multi-dimensionnal arrays, data on columns are contiguous
 - $A[8][16]$ for(...i<8...) for(...j<16...)
 - J=0





Memory layout

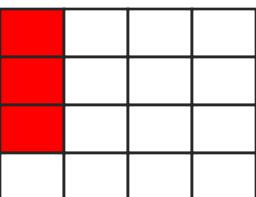
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - Fortran is « column-major »
 - For multi-dimensionnal arrays, data on columns are contiguous
 - $A[8][16]$ for(...i<8...) for(...j<16...)
 - J=1





Memory layout

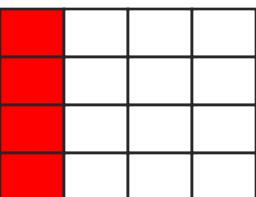
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - Fortran is « column-major »
 - For multi-dimensionnal arrays, data on columns are contiguous
 - $A[8][16]$ for(...i<8...) for(...j<16...)
 - J=2





Memory layout

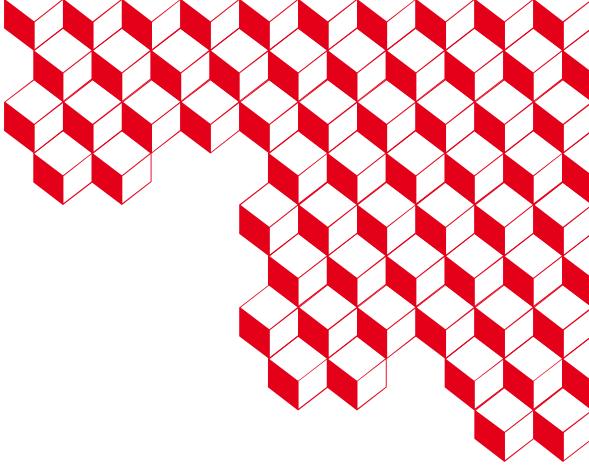
- ❖ To have performance, it is mandatory to follow this hardware feature
- ❖ /!\ Be careful of the memory layout imposed by the programming language
 - Fortran is « column-major »
 - For multi-dimensionnal arrays, data on columns are contiguous
 - $A[8][16]$ for(...i<8...) for(...j<16...)
 - J=3





Data locality

- ❖ Closer the memory to the compute unit, more efficient it is
 - Better (higher) bandwidth
 - Better (lower) latency
- ❖ Fetching data from the main memory is costly
 - Automatically put in cache levels
- ❖ It is then necessary to reuse the data to get high performance



Data Structure and memory consumption



Data structure and memory consumption

- ❖ Let's consider two data structures:

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```

- ❖ Is there a better/worse one between these two structures?



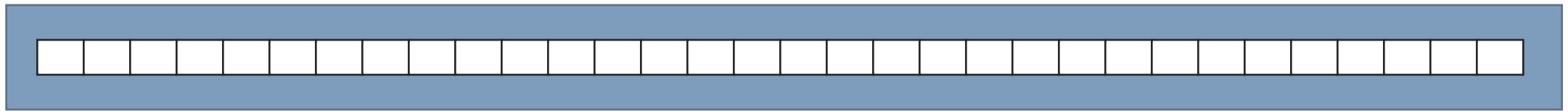
Data structure and memory consumption

- ❖ « int » is a 4 bytes type, and must be aligned on 4 bytes addresses
 - address % 4 = 0
- ❖ « double » is a 8 bytes type, and must be aligned on 8 bytes addresses
 - address % 8 = 0
- ❖ What does it mean for our data structures ?



Data structure and memory consumption

- ❖ Let's consider such memory space:





Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ « int » is a 4 bytes type, and must be aligned on 4 bytes addresses
 - Orange memory cells



Data structure and memory consumption

- ❖ Let's consider such memory space:

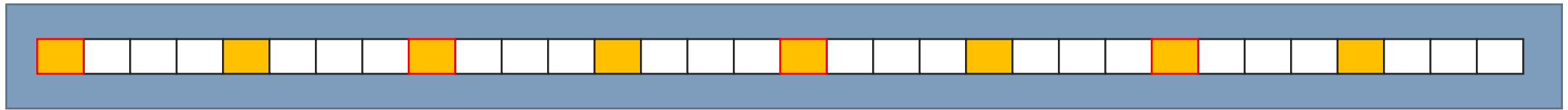


- ❖ « int » is a 4 bytes type, and must be aligned on 4 bytes addresses
 - Orange memory cells
- ❖ « double » is a 8 bytes type, and must be aligned on 8 bytes addresses
 - red borders memory cells



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the first structure

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



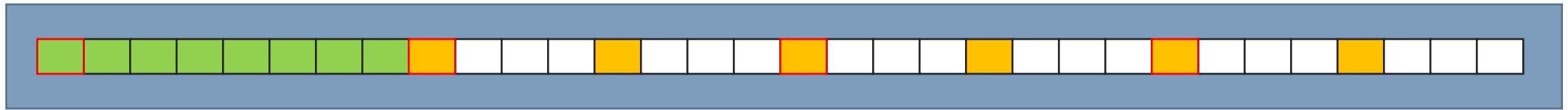
- ❖ With the first structure
- We first need to put an int

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the first structure

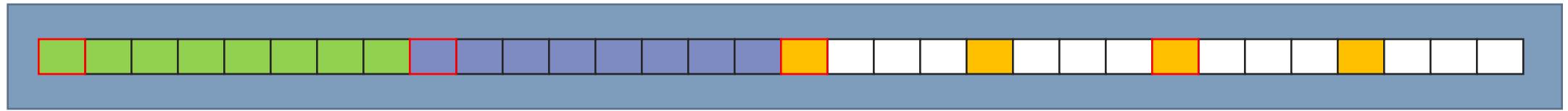
- We first need to put an int
- Then another int
 - Which can be done at the following address

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the first structure

- We first need to put an int
- Then another int
 - Which can be done at the following address
- Then a double
 - Which can be done at the following address

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the first structure

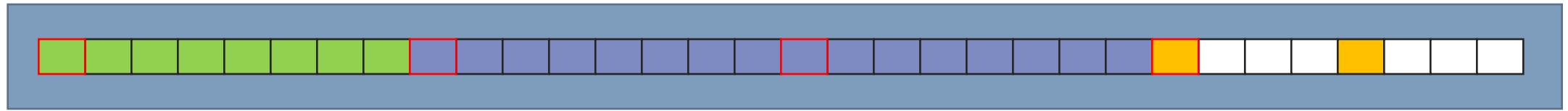
- We first need to put an int
- Then another int
 - Which can be done at the following address
- Then a double
 - Which can be done at the following address
- Then the other double

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the first structure

- We first need to put an int
- Then another int
 - Which can be done at the following address
- Then a double
 - Which can be done at the following address
- Then the other double

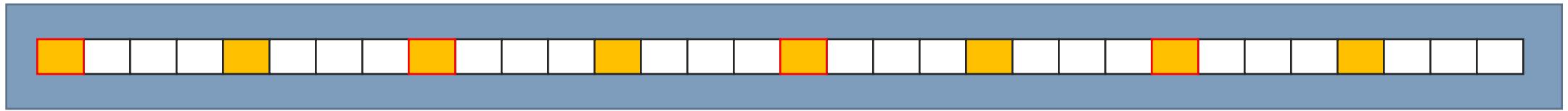
$$\rightarrow 4+4+8+8 = 24 \text{ bytes}$$

```
struct mem_1
{
    int a;
    int c;
    double b;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



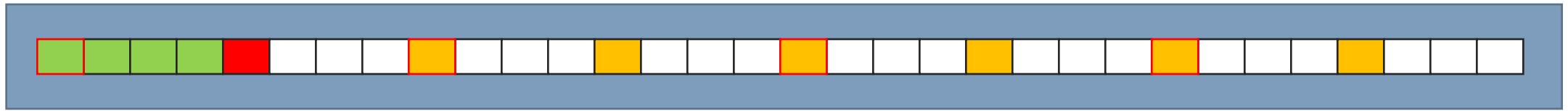
- ❖ With the second structure

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the second structure

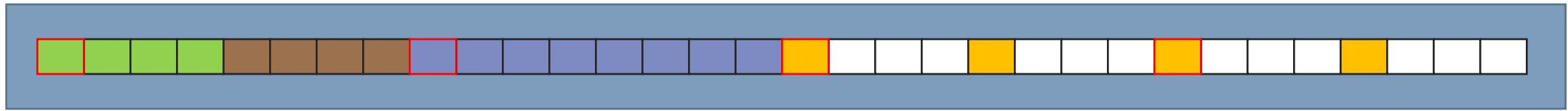
- We first need to put an int
- Then a double
 - But the following address is not valide
 - Address %8 = 4 (and not 0)

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the second structure

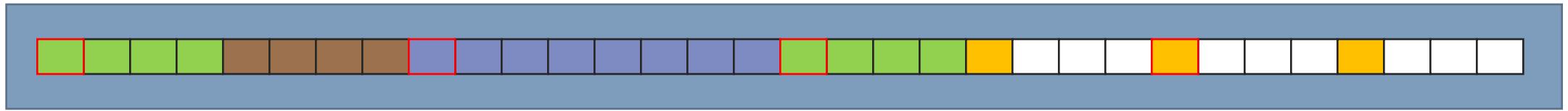
- We first need to put an int
- Then a double
 - But the following address is not valid
 - Address %8 = 4 (and not 0)
 - Need to reach next valid address (with red border)
 - Space in-between is just empty space

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the second structure

- We first need to put an int
- Then a double
 - Need for 4 empty bytes to reach a valid address
- Then the second int
 - Which can be done at the following address

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the second structure

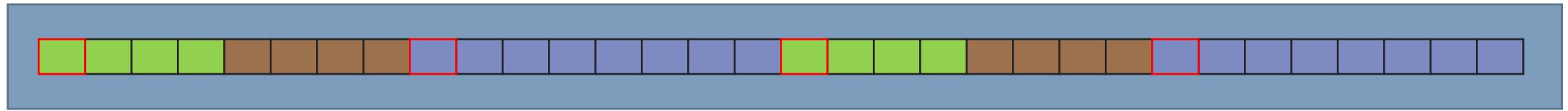
- We first need to put an int
- Then a double
 - Need for 4 empty bytes to reach a valid address
- Then the second int
 - Which can be done at the following address
- Then the second double
 - Same problem as with the first double

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```



Data structure and memory consumption

- ❖ Let's consider such memory space:



- ❖ With the second structure

- We first need to put an int
- Then a double
 - Need for 4 empty bytes to reach a valid address
- Then the second int
 - Which can be done at the following address
- Then the second double
 - Same problem as with the first double

```
struct mem_2
{
    int a;
    double b;
    int c;
    double d;
};
```

$$\rightarrow 4+4+8+4+4+8 = 32 \text{ bytes}$$



Data structure and memory consumption

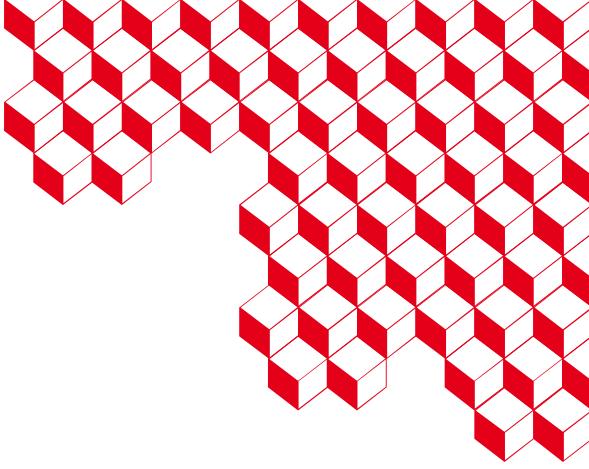
- ❖ Test allocating 1000 times each structure

```
int main()
{
    struct good_mem * good = malloc(sizeof(struct good_mem)*1000);
    struct bad_mem * bad = malloc(sizeof(struct bad_mem)*1000);
```

- ❖ Output:

```
[~ $ ./memory_str.pgr
bad struct array size : 31968 (e.g. 7992 integer) [0x55ba5db37030 -> 0x55ba5db3ed10]
good struct array size : 23976 (e.g. 5994 integer) [0x55ba5db31260 -> 0x55ba5db37008]
=====
ratio = 1.333333
```

- Same results with c++ classes



SOA or AOS Késako?



SOA or AOS (or SOAOS)

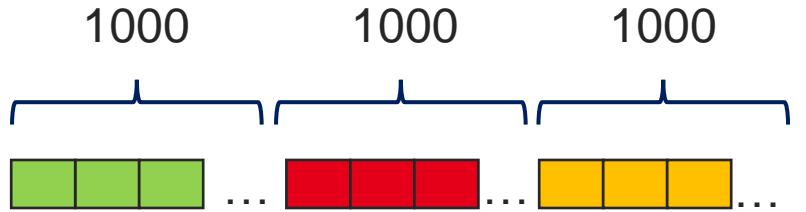
- ❖ SOA: Structure of arrays

```
struct particle
{
    temp [1000];
    press[1000];
    vit [1000];
};
```



SOA or AOS (or SOAOS)

- ❖ SOA: Structure of arrays

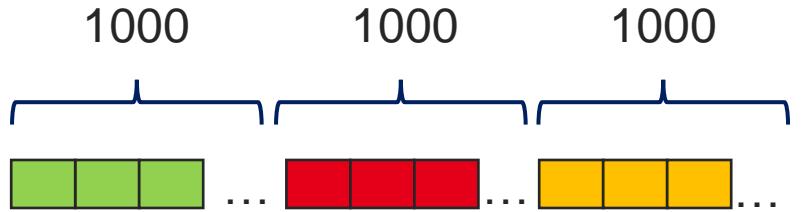


```
struct particle
{
    temp [1000];
    press[1000];
    vit [1000];
};
```



SOA or AOS (or SOAOS)

- ❖ SOA: Structure of arrays



```
struct particle
{
    temp [1000];
    press[1000];
    vit [1000];
};
```

- ❖ AOS: Array of structure

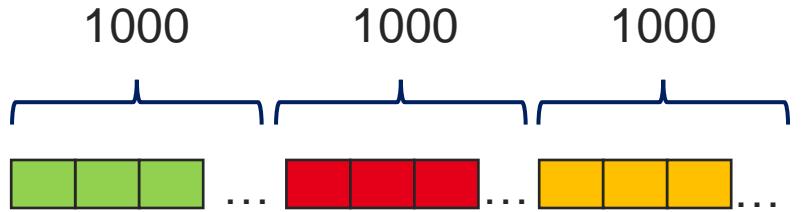
```
struct particle
{
    temp ;
    press;
    vit ;
};

...
Particle[1000];
```



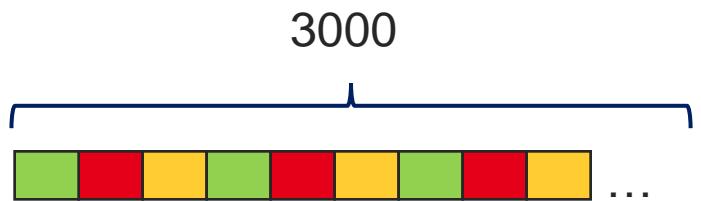
SOA or AOS (or SOAOS)

- ❖ SOA: Structure of arrays



```
struct particle
{
    temp [1000];
    press[1000];
    vit [1000];
};
```

- ❖ AOS: Array of structure



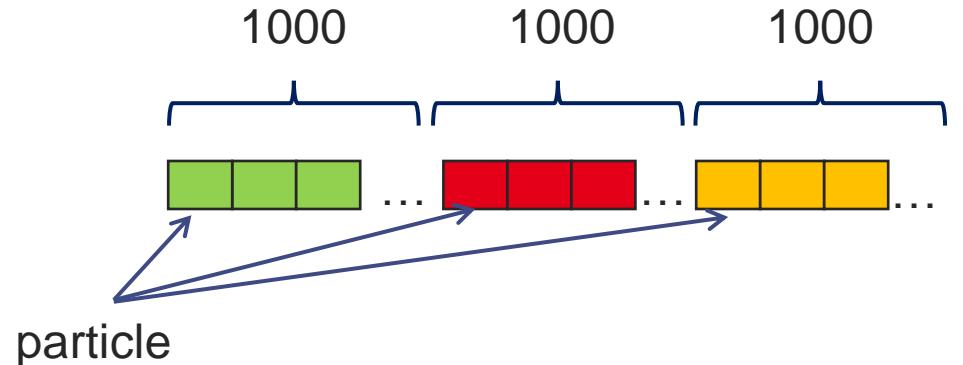
```
struct particle
{
    temp ;
    press;
    vit ;
};

...
Particle[1000];
```

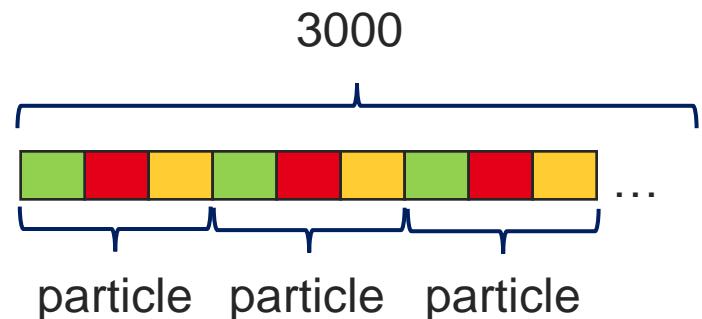


SOA or AOS (or SOAOS)

- ❖ SOA: Structure of arrays



- ❖ AOS: Array of structure



```
struct particle
{
    temp [1000];
    press[1000];
    vit [1000];
};
```

```
struct particle
{
    temp ;
    press;
    vit ;
};
...
Particle[1000];
```



SOA or AOS (or SOAOS)

- ❖ SOA and AOS are very common way to store data in a simulation



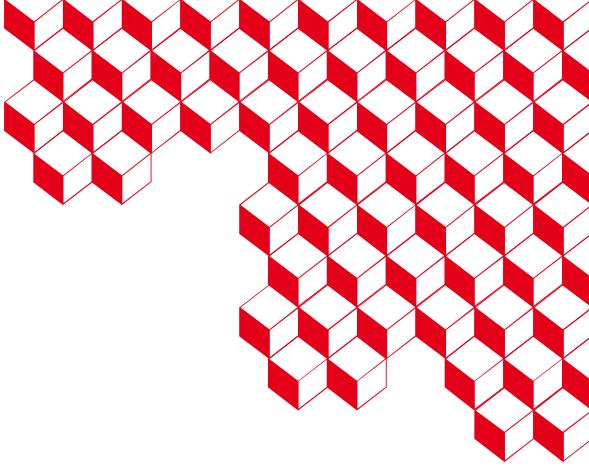
SOA or AOS (or SOAOS)

- ❖ SOA and AOS are very common way to store data in a simulation
- ❖ Which one is better?



SOA or AOS (or SOAOS)

- ❖ SOA and AOS are very common way to store data in a simulation
- ❖ Which one is better?
 - Heavily depends on access pattern
 - SOA : If computation touches only one physical value type
 - i.e., you compute temperatures on their own, then pressure...
 - AOS : if all physical value types are accessed together
 - i.e., you update temp & pressure at each particle



Cache blocking



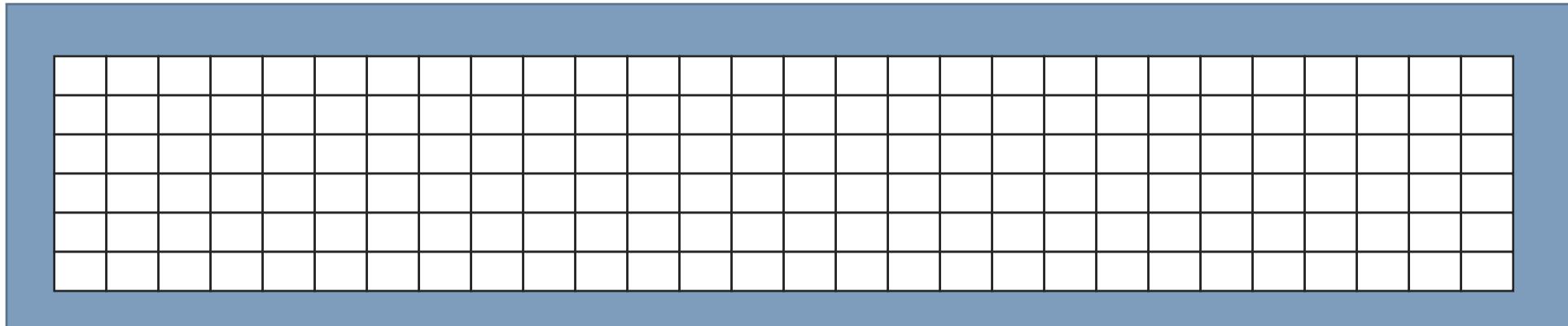
Cache blocking

- ❖ In a multi-dimensional stencil computation, updating a cell requires contributions from its neighbors
- ❖ If the array line is longer than the last level cache, it is not possible to reuse the fetched data from a line to compute the next line
- ❖ It is necessary to change how the array is read, i.e., in which order its cells are accessed, to benefit from the caches.



Blocking example: 2D stencil

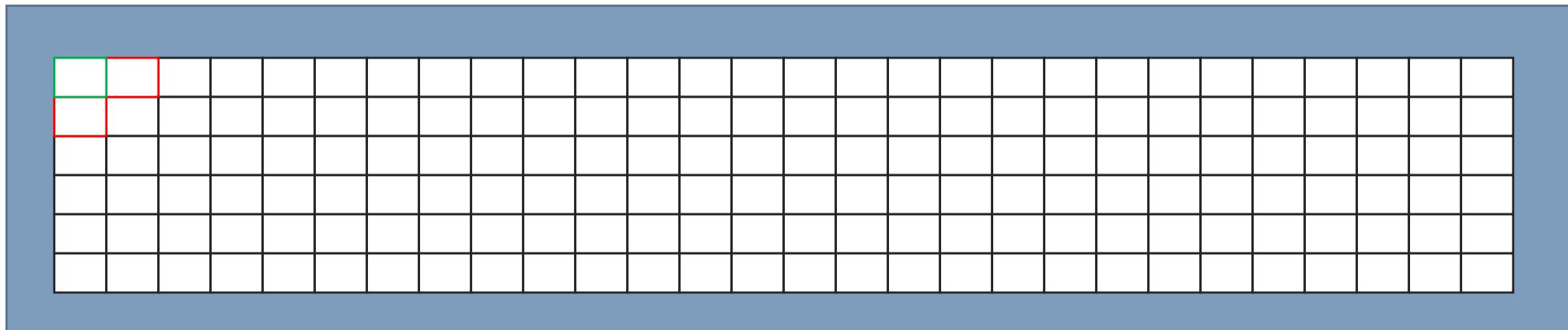
- ❖ Let's consider a 2D jacobi stencil
 - Each cell needs data from its four direct neighbors
 - Up, down, left, right
- ❖ Let's consider a cache line of **four elements**, and a cache able to store **6 cache lines**





Blocking example: 2D stencil

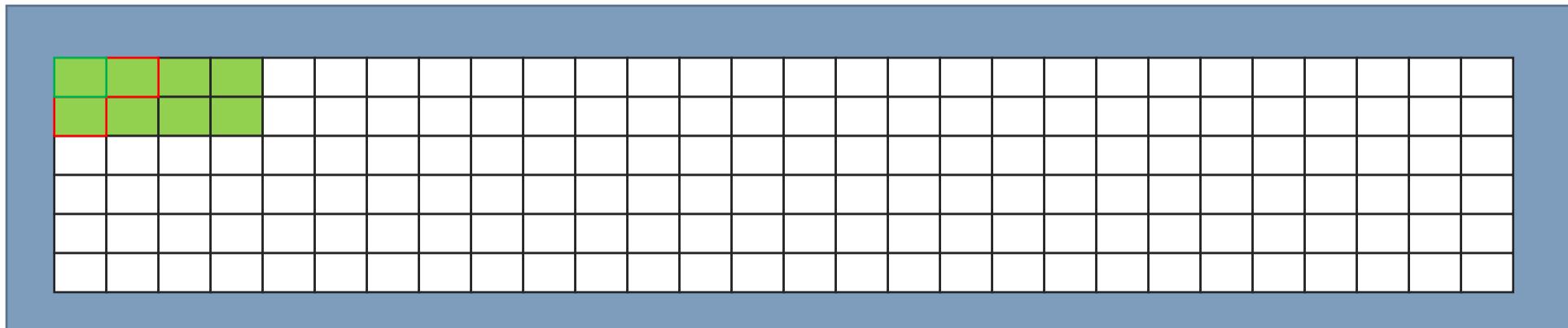
- ❖ First element needs data from right and bottom cells





Blocking example: 2D stencil

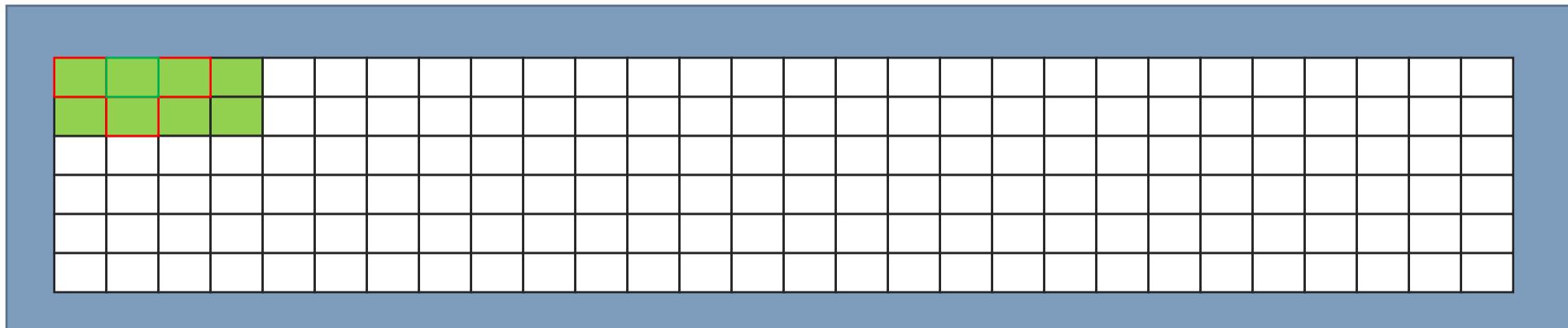
- ❖ First element needs data from right and bottom cells
- ❖ Cache lines having these cells are fetched





Blocking example: 2D stencil

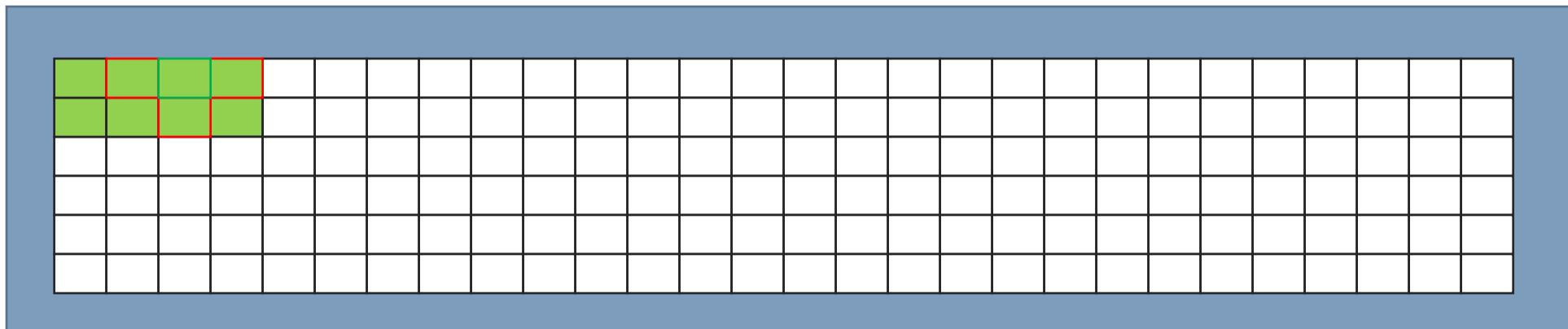
- ❖ Second cell requires data from left, right and bottom
- ❖ Luckily, all data are already in cache!
 - Very fast access to these data





Blocking example: 2D stencil

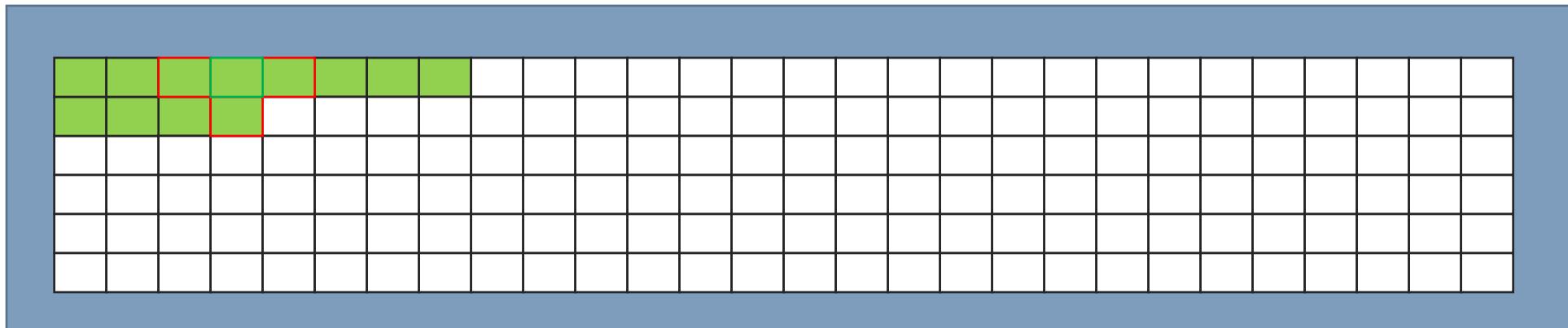
- ❖ Same goes for the third cell





Blocking example: 2D stencil

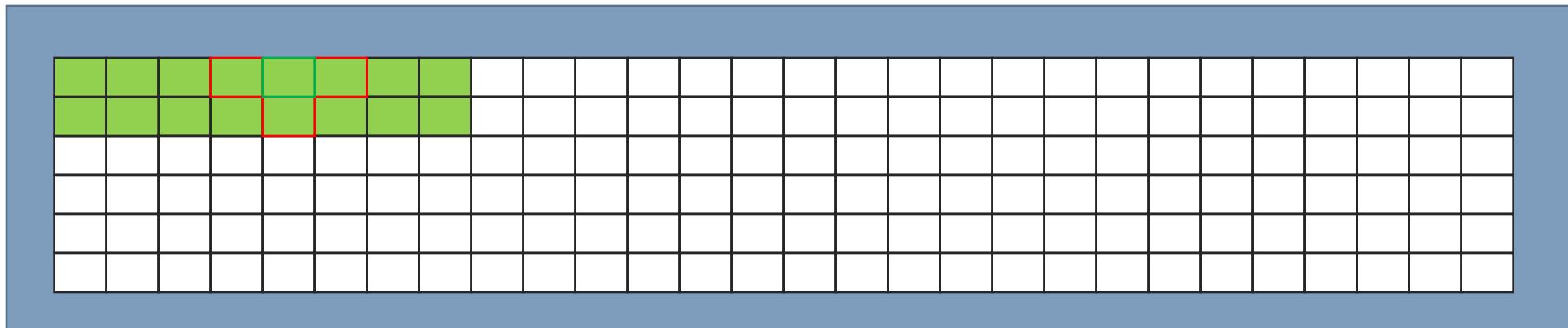
- ❖ For the fourth cell, the data from the right cell is not already in cache
- ❖ The cache line containing the data is fetched from the main memory (costly) and stored in the cache





Blocking example: 2D stencil

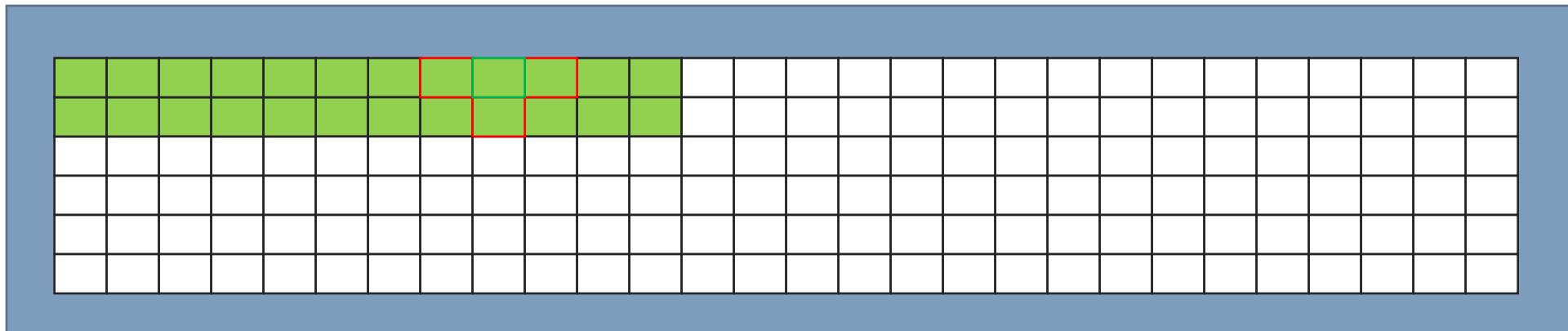
- ❖ For the fourth cell, the data from the right cell is not already in cache
- ❖ The cache line containing the data is fetched from the main memory (costly) and stored in the cache





Blocking example: 2D stencil

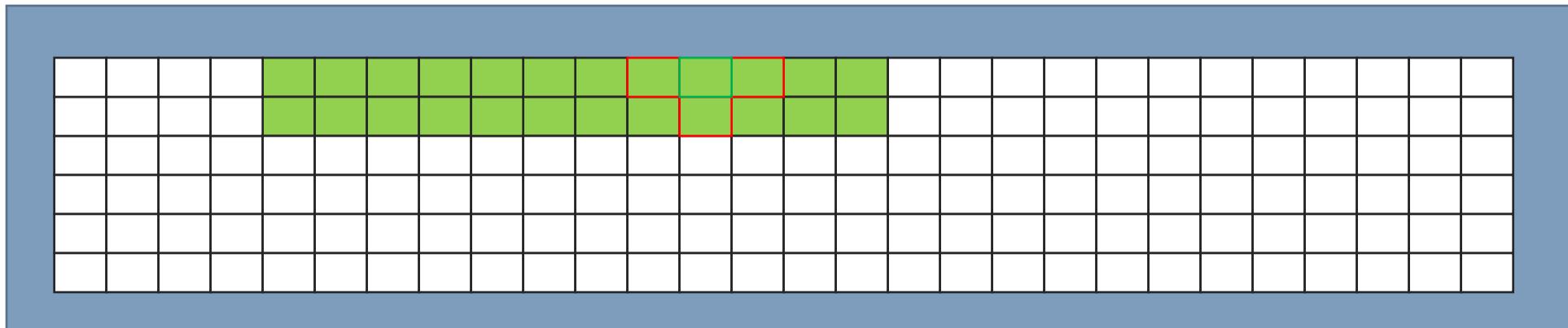
- ❖ And so on...





Blocking example: 2D stencil

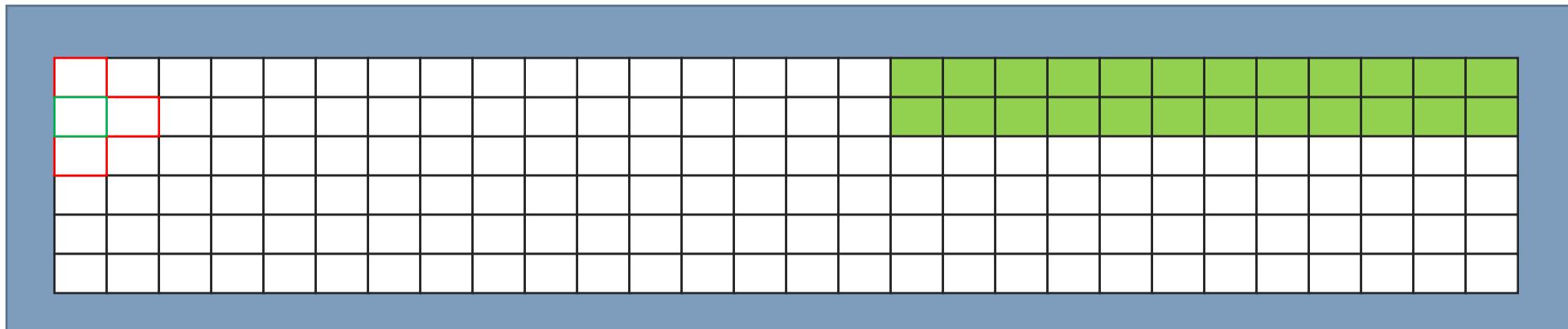
- ❖ ... until the cache is full (four lines)
- ❖ To load new data in the cache, some data need to be evicted
- ❖ Usually, the least recently used line is removed from the cache





Blocking example: 2D stencil

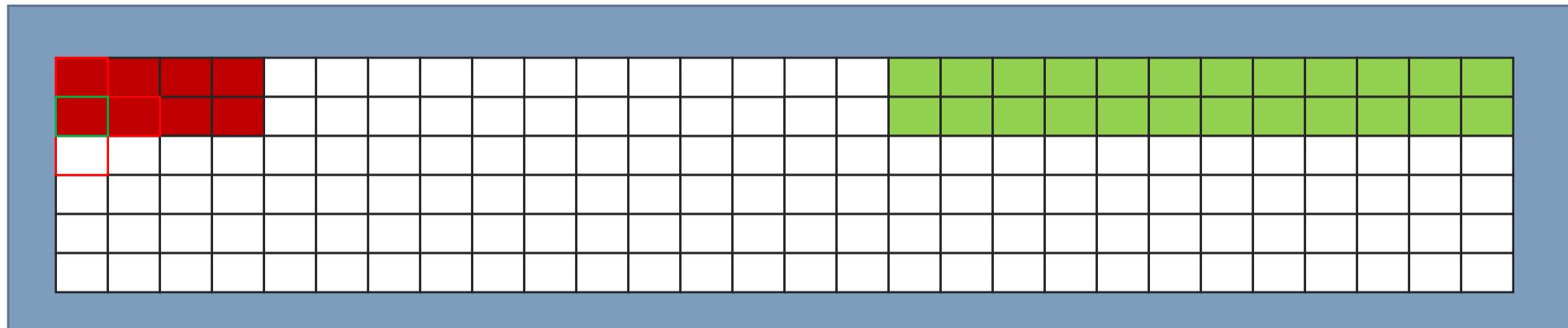
- ❖ At the end of the first line of the array, here are the lines present in the cache
- ❖ Then, when we start the second line, no required elements is in the cache





Blocking example: 2D stencil

- ❖ ... but, on the three needed data cache lines, two of them where stored in the cache at some point.
- ❖ How can we reuse the cache lines already stored?

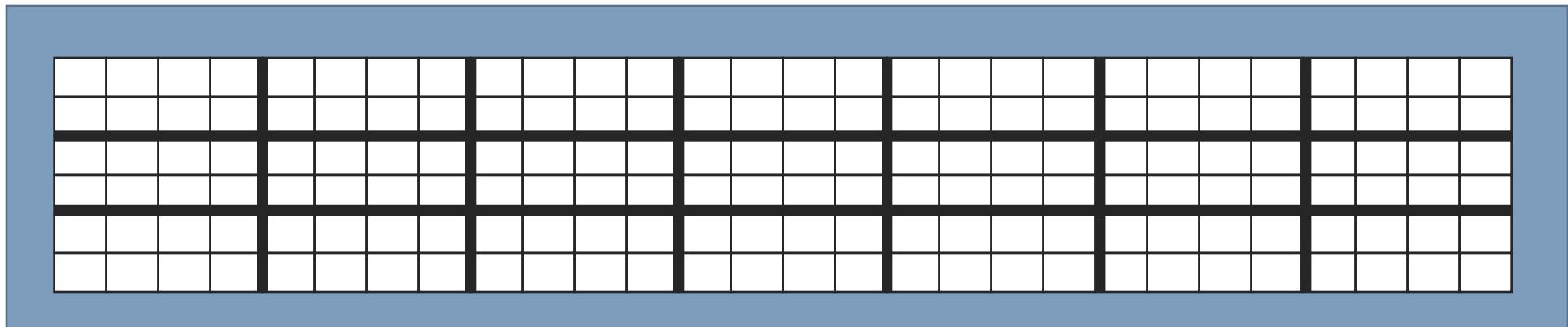




Blocking example: 2D stencil

- ❖ We will now access cells in blocks

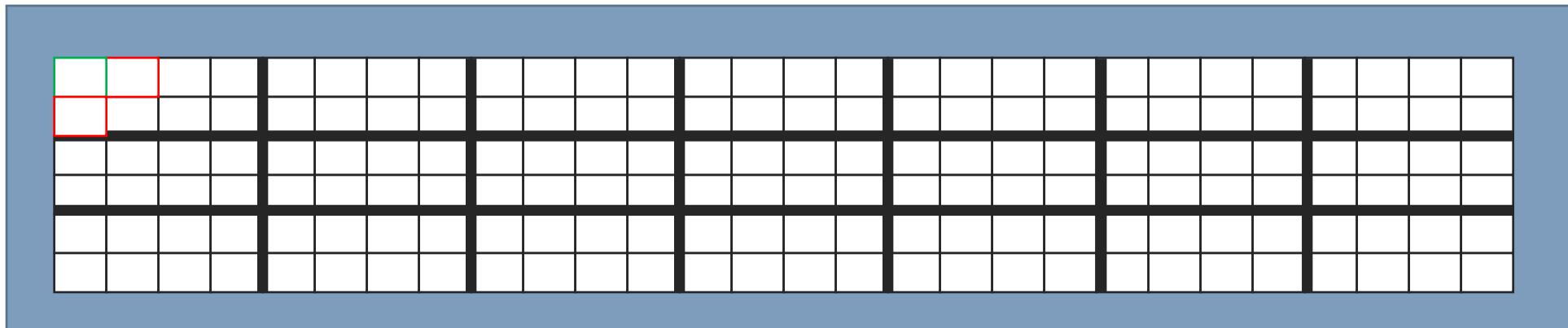
```
for(i=0; i<6; i++)          for(i=0; i<6; i+=2)  
    for(j=0; j<28; j++) =>  for(j=0; j<28; j+=4)  
        A[i][j]                  for(ii=0; ii<2; ii++)  
                                for(jj=0; jj<4; jj++)  
                                    A[i+ii][j+jj]
```





Blocking example: 2D stencil

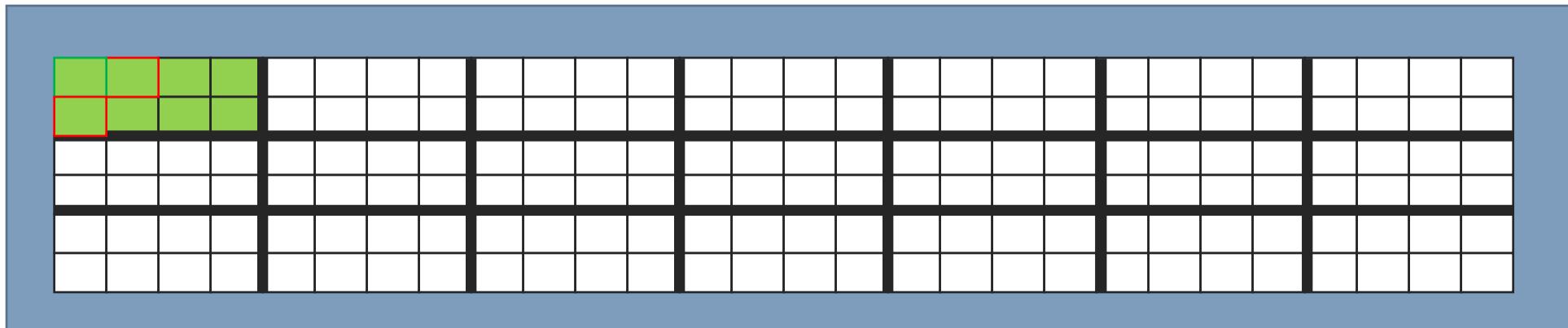
- ❖ As before, the first cell requires data from the right and the bottom cells





Blocking example: 2D stencil

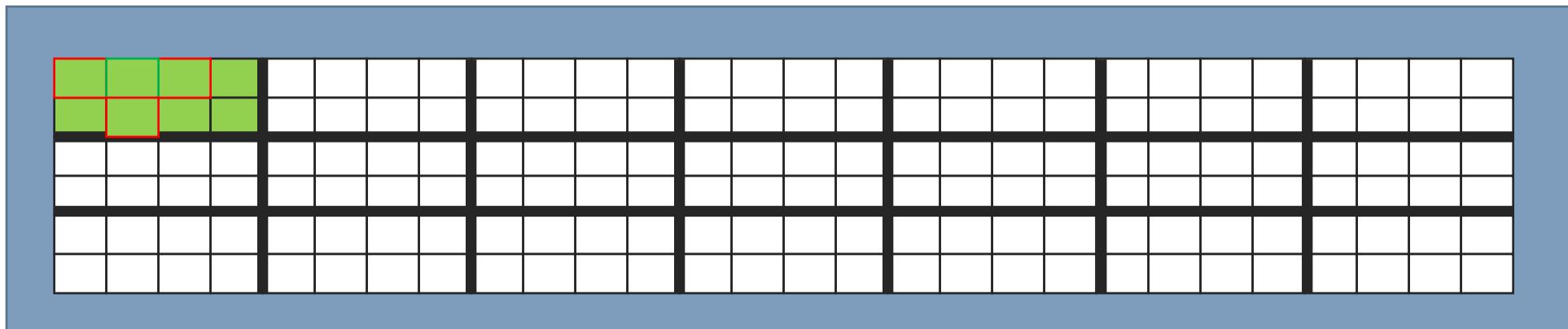
- ❖ As before, the first cell requires data from the right and the bottom cells
- ❖ Cache lines containing these data are fetched and stored in the cache





Blocking example: 2D stencil

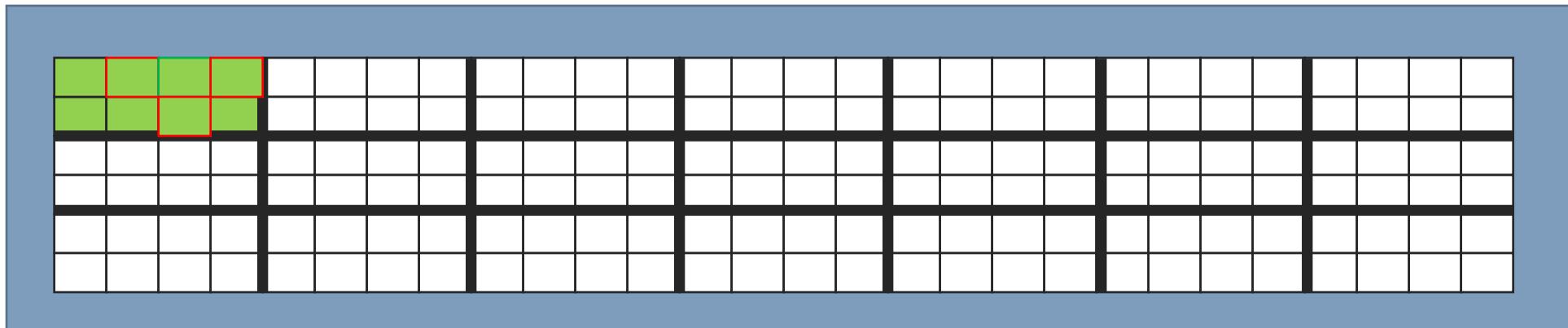
- ❖ Exactly goes as previously until the end of the most internal loop
 - $i=0 ; j=0; ii=0; jj=1$





Blocking example: 2D stencil

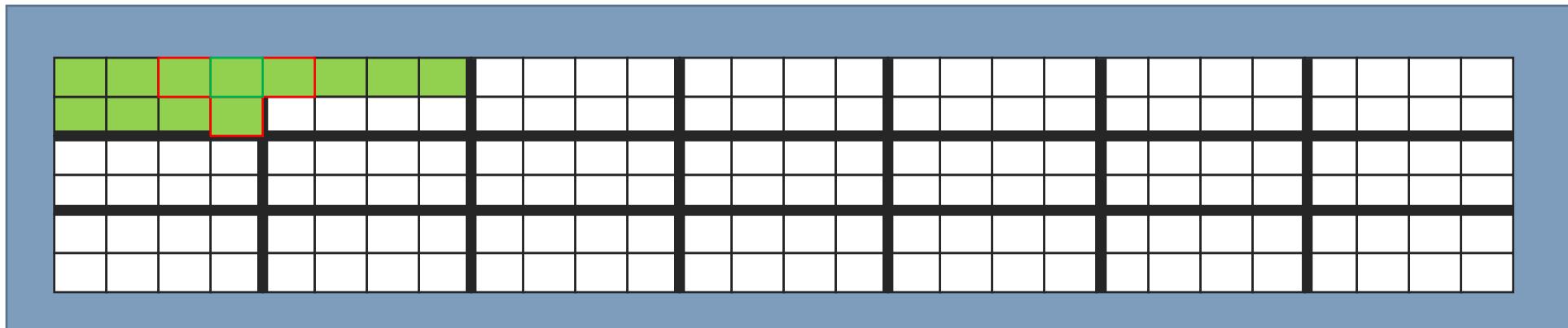
- ❖ Exactly goes as previously until the end of the most internal loop
 - $i=0 ; j=0; ii=0; jj=2$





Blocking example: 2D stencil

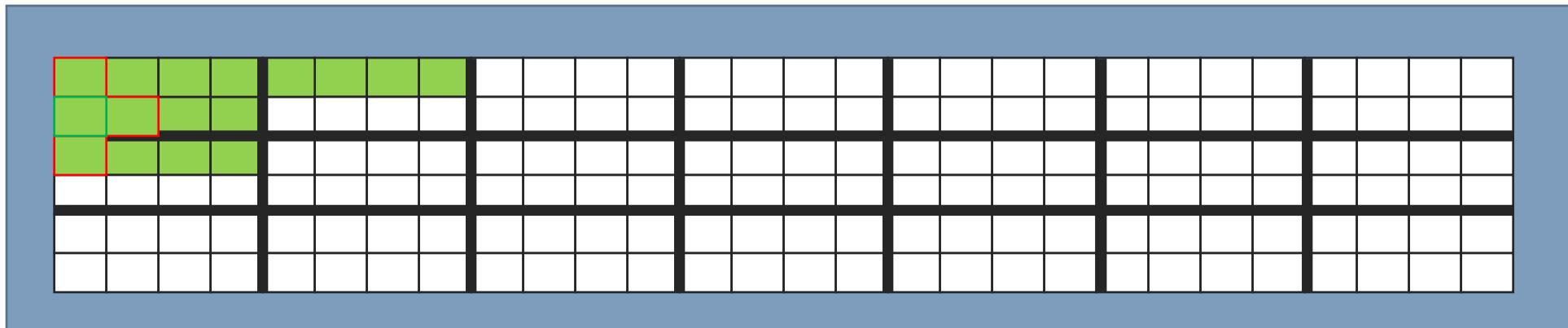
- ❖ Exactly goes as previously until the end of the most internal loop
 - $i=0 ; j=0; ii=0; jj=3$





Blocking example: 2D stencil

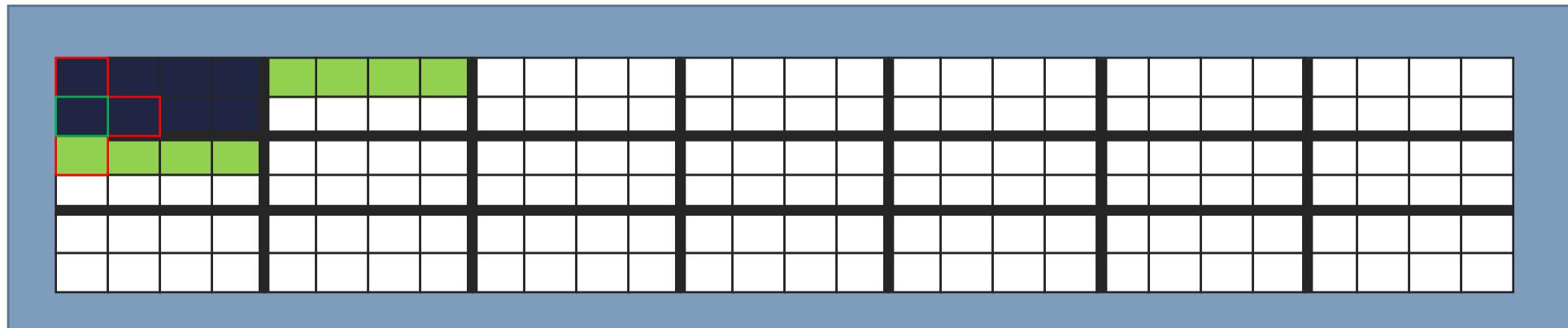
- ❖ The following cell is not the one on the right (same line), but the first cell of the second line
 - $i=0 ; j=0; ii=1; jj=0$





Blocking example: 2D stencil

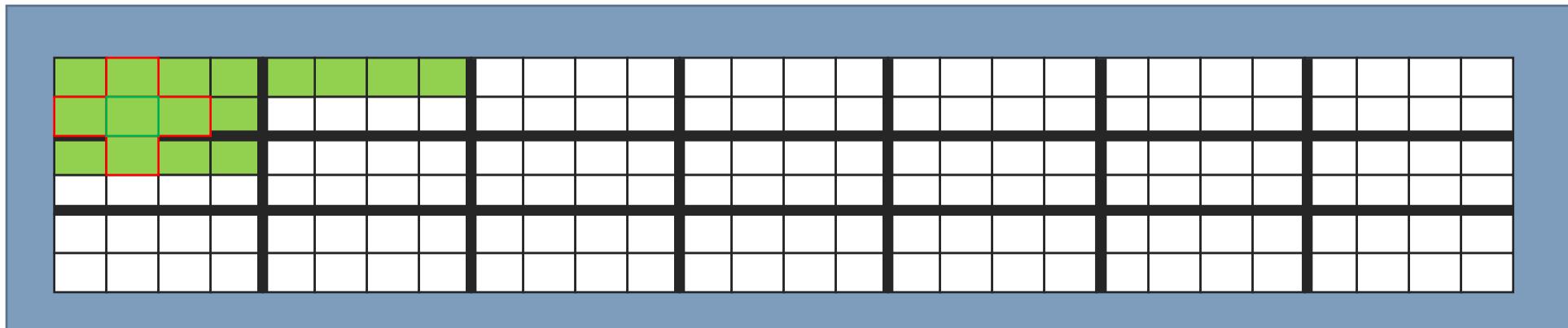
- ❖ Thanks to this new access pattern, the two first cache lines stored in the cache are reused for this new iteration
 - Contrary to the previous access pattern





Blocking example: 2D stencil

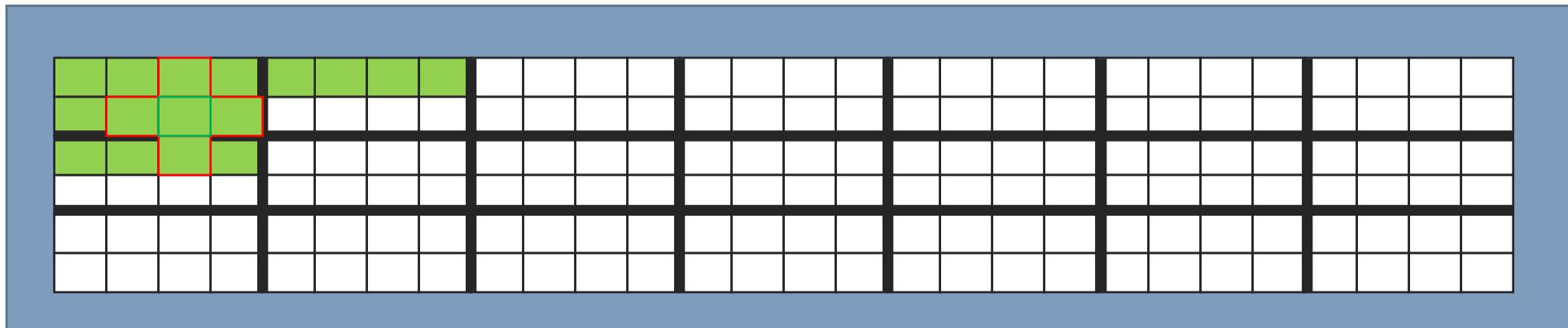
- ❖ The most internal loop continues, updates benefitting from the stored cache lines





Blocking example: 2D stencil

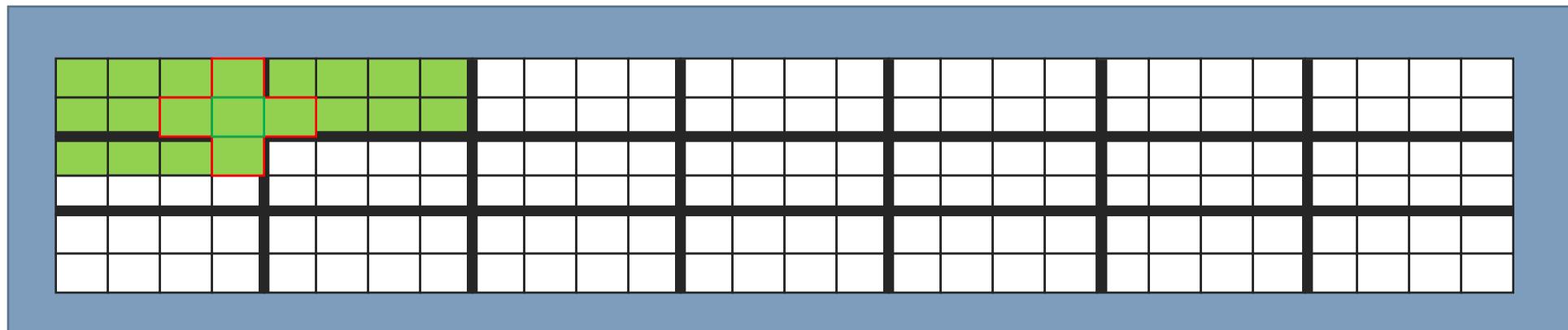
- ❖ The most internal loop continues, updates benefitting from the stored cache lines





Blocking example: 2D stencil

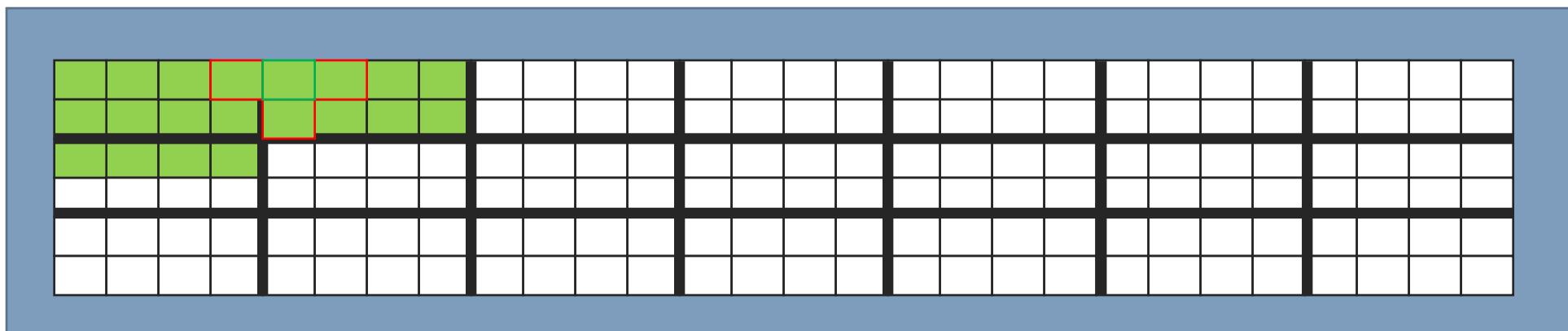
- ❖ As for the first line, when updating the last element in the current cache line, it is necessary to fetch and store the next cache line





Blocking example: 2D stencil

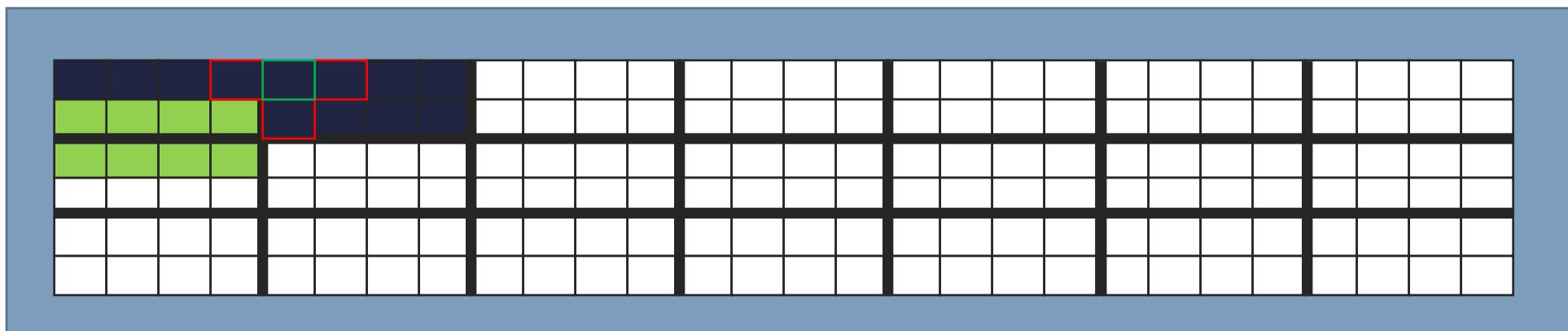
- ❖ Once the first block is done, we begin to update cells from the second block
 - $i=0 ; j=1; ii=0; jj=0$





Blocking example: 2D stencil

- ❖ In our case, once again, every cache lines required for the current update are already stored in the cache





Cache blocking

- ❖ Blocking consists in changing the access pattern of cells in a multi-dimensional array to better benefit from cache lines already stored in caches
 - The previous access pattern example is called Z-pattern or Z-curves
- ❖ Accessing contiguous data is called **spatial locality**
- ❖ Reusing data is called **temporal locality**
 - Blocking aims at improving data locality
 - Easy to do on array, but can be applied to any structure



Cache blocking

- ❖ Blocking values depend on...
 - ... the hardware: **cache size**, cache **line size**
 - ... the software : which data are necessary, how the data are stored (row-major, column-major)

- ❖ The BLAS library is written in Fortran
 - Hence, when invoked from a C program, each input array is transposed before and after invoking the BLAS function
 - Only way to get the improved data locality
 - More efficient than original layout without the transpose



3. NUMA

Intra-nodes



NUMA

- ❖ NUMA means « Non-Uniform memory access »
- ❖ So in a « NUMA » node, access to the memory is not uniform! (duh!)



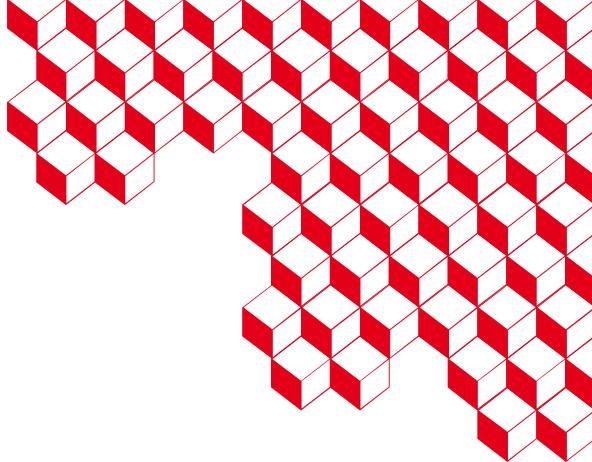
NUMA

- ❖ NUMA means « Non-Uniform memory access »
- ❖ So in a « NUMA » node, access to the memory is not uniform! (duh!)
- ❖ ...



NUMA

- ❖ NUMA means « Non-Uniform memory access »
- ❖ So in a « NUMA » node, access to the memory is not uniform! (duh!)
- ❖ ...
- ❖ OK, but what does it mean?



Cache coherency costs

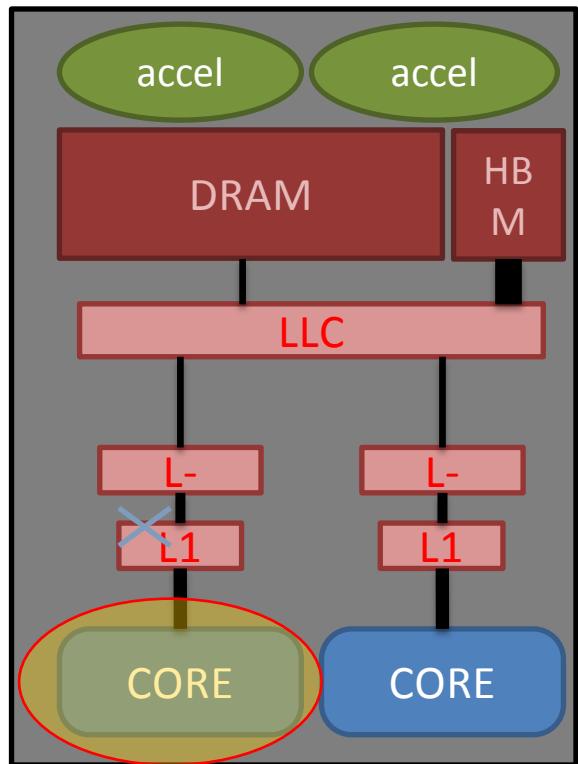
This part is an initiation of cache coherency. Read more about cache coherency to go further.

https://en.wikipedia.org/wiki/MESI_protocol



NUMA access: local L1

- ❖ In a NUMA node, time to access a data depends on where the data is located

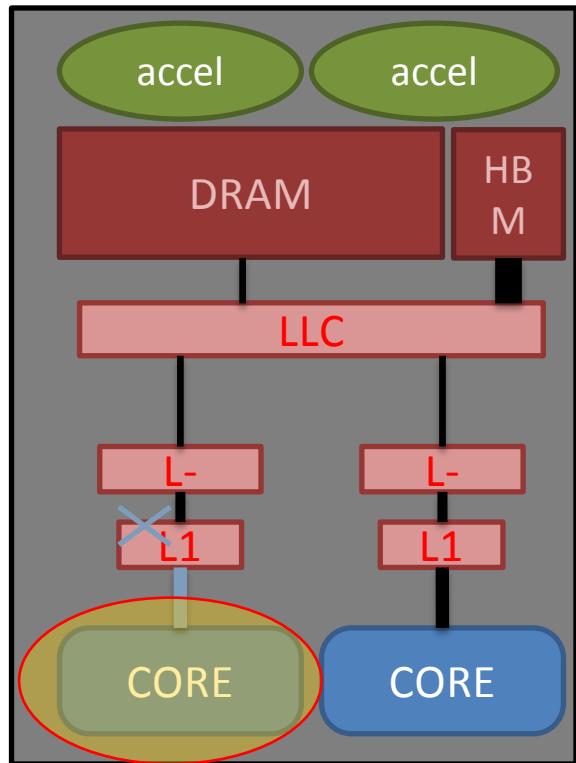


- ❖ Ex: core 0 wants to access a data located in its own L1 cache



NUMA access: local L1

- ❖ In a NUMA node, time to access a data depends on where the data is located

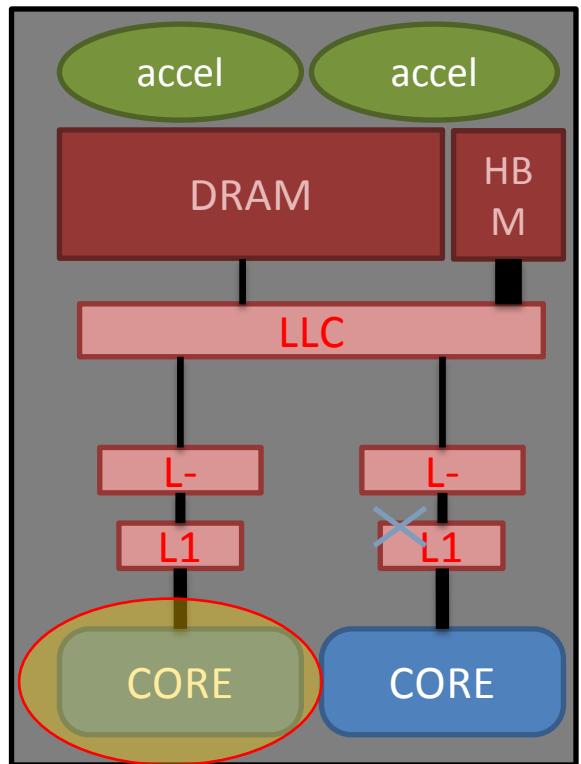


- ❖ Ex: core 0 wants to access a data located in its own L1 cache
 - Need to go accross the link between core 0 and its own L1



NUMA access: distant L1 Cache Coherency cost

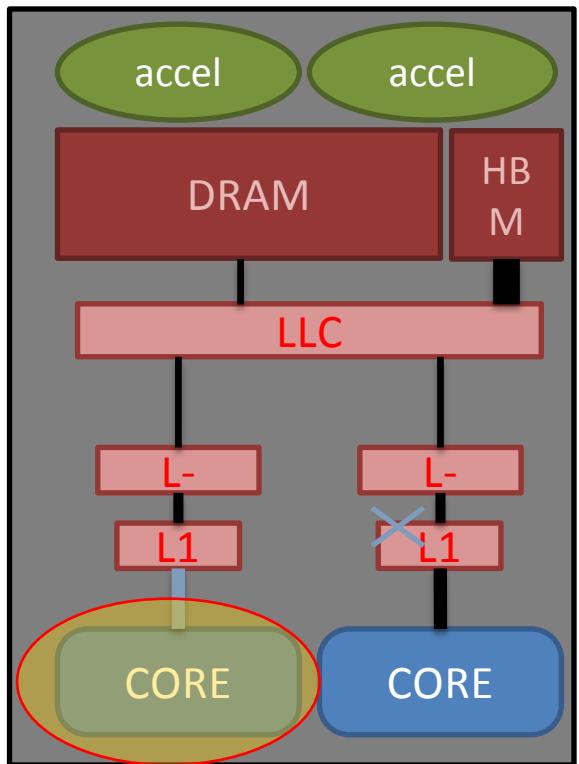
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache





NUMA access: distant L1 Cache Coherency cost

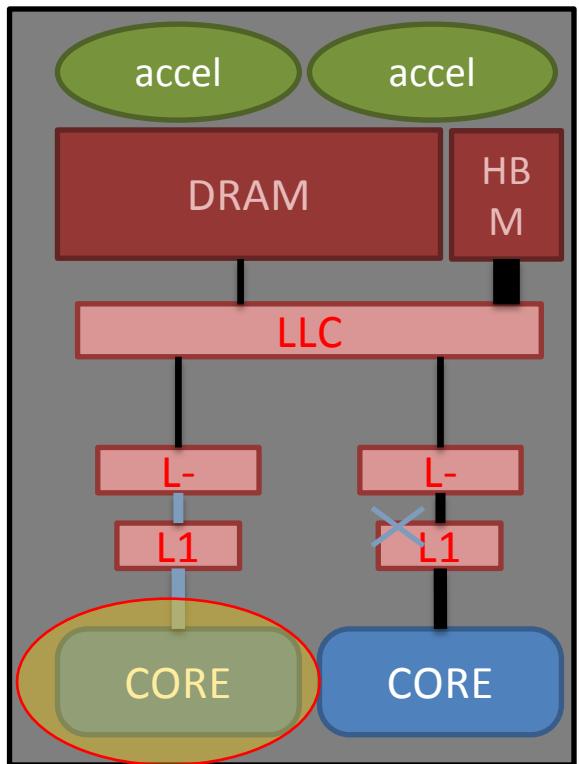
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache
 - Need to go accross the link between core 0 and its own L1





NUMA access: distant L1 Cache Coherency cost

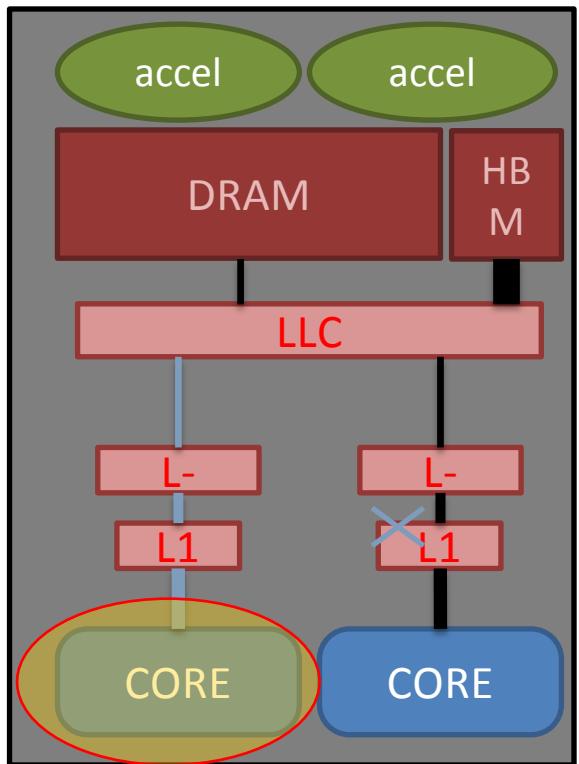
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache
 - Need to go accross the link between core 0 and its own L1
 - Then the link between its L1 and L2





NUMA access: distant L1 Cache Coherency cost

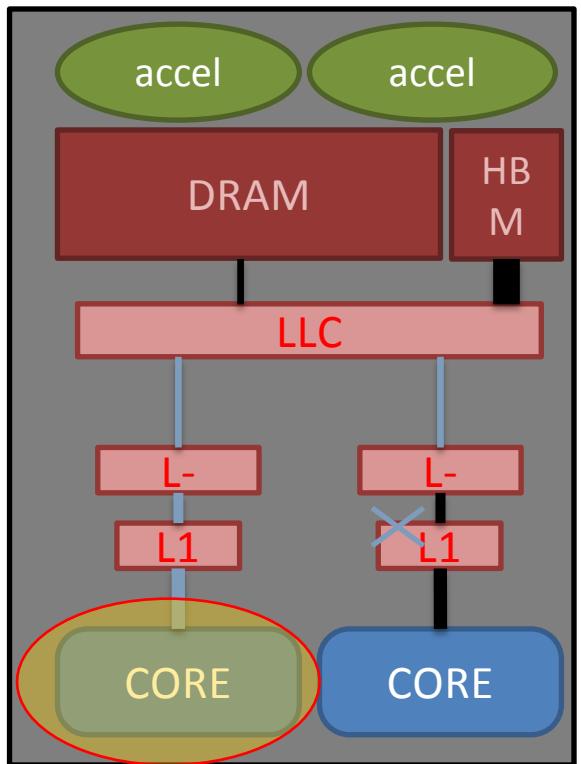
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache
 - Need to go accross the link between core 0 and its own L1
 - Then the link between its L1 and L2
 - Then the link between its L2 and L3





NUMA access: distant L1 Cache Coherency cost

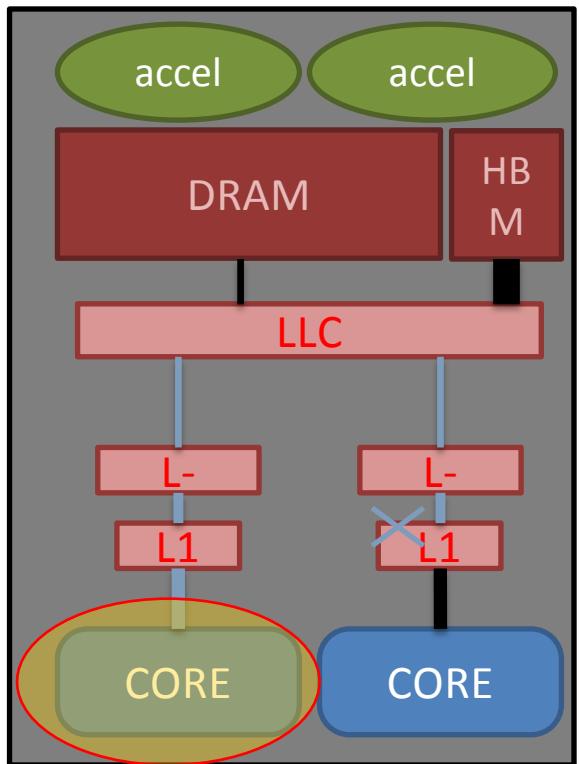
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache
 - Need to go accross the link between core 0 and its own L1
 - Then the link between its L1 and L2
 - Then the link between its L2 and L3
 - Then the link between the L3 and the distant L2





NUMA access: distant L1 Cache Coherency cost

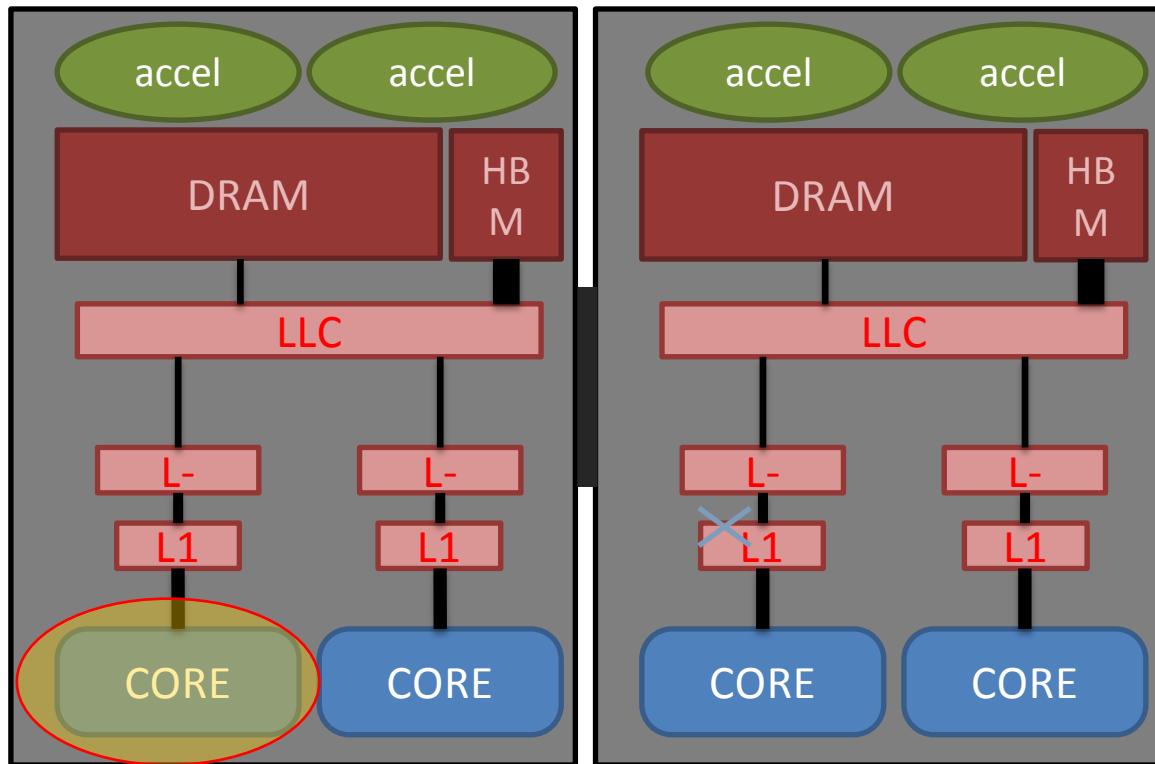
- ❖ Ex: core 0 wants to access a data located in its neighbor's L1 cache
 - Need to go accross the link between core 0 and its own L1
 - Then the link between its L1 and L2
 - Then the link between its L2 and L3
 - Then the link between the L3 and the distant L2
 - Then the link between the L2 and the distant L1





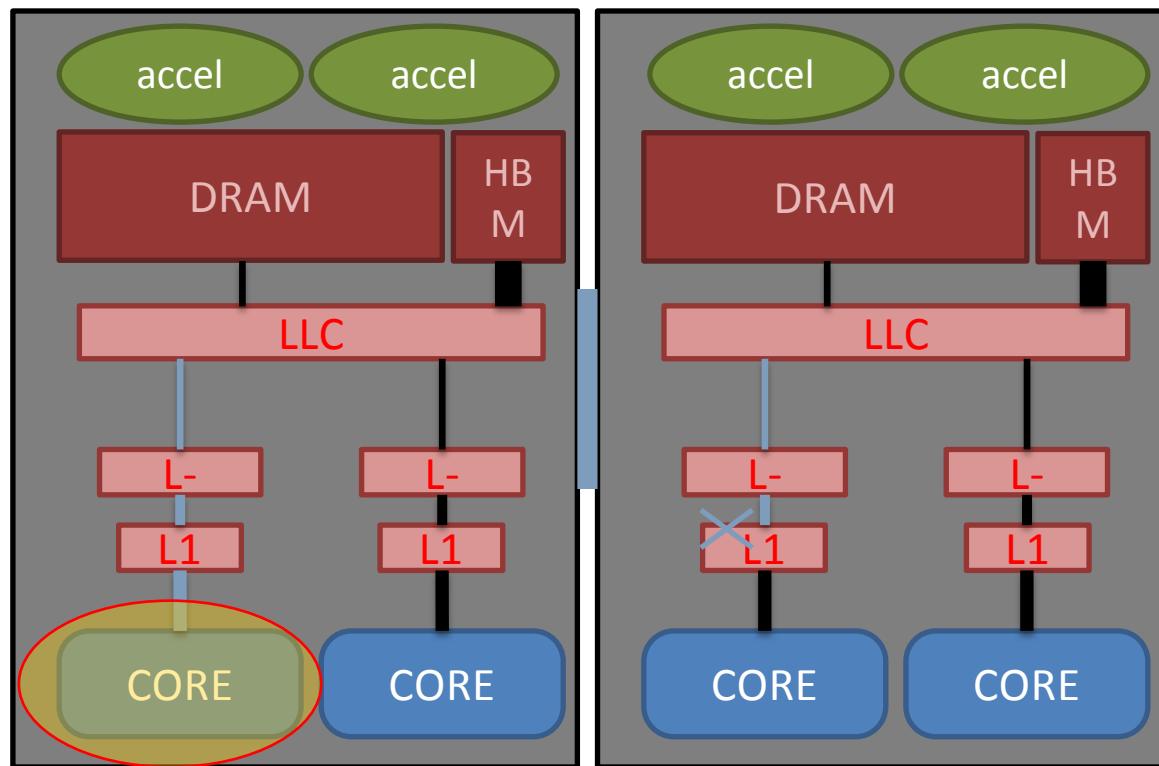
NUMA access: distant L1 Cache Coherency cost

- ❖ Ex: core 0 wants to access a data located in L1 cache on another socket





NUMA access: distant L1 Cache Coherency cost



- ❖ Ex: core 0 wants to access a data located in L1 cache on another socket
 - In addition to the previous links, also needs to cross inter-socket link

Architecture Nehalem

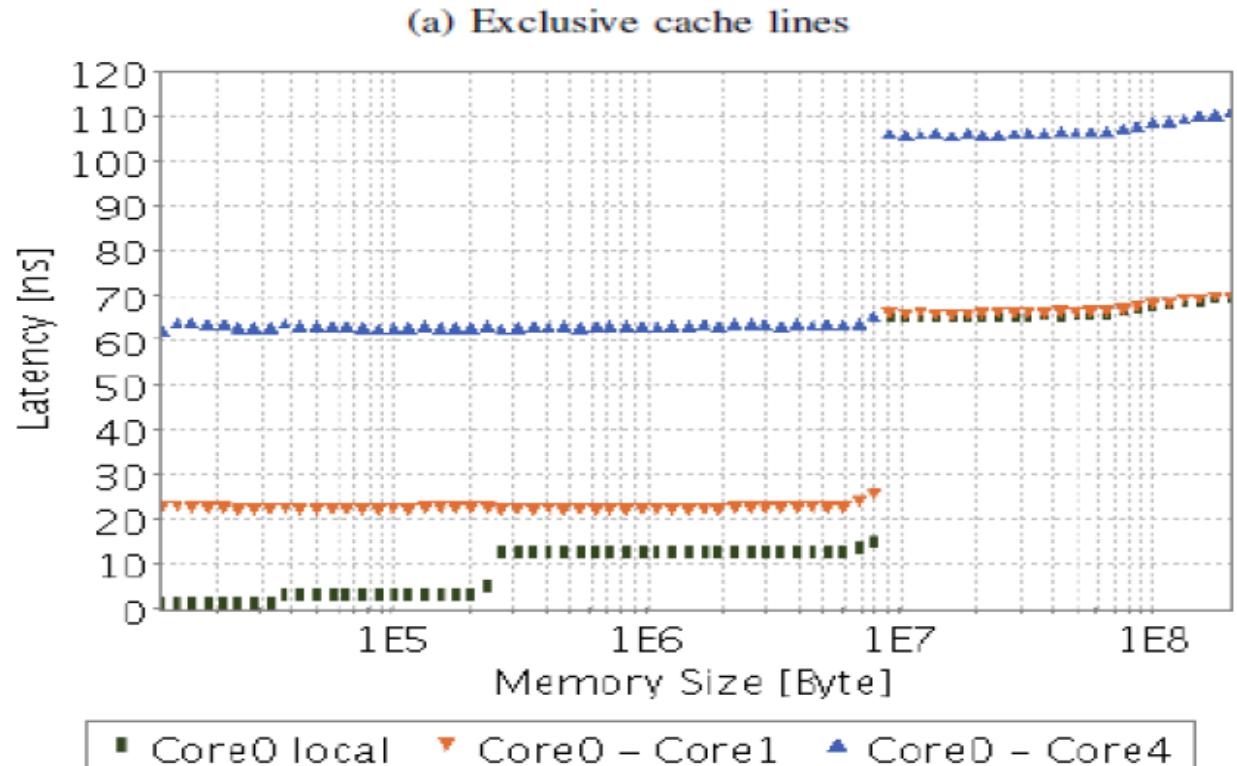


Fig. 6. Read latencies of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the exclusive coherency state[1]

Architecture Nehalem

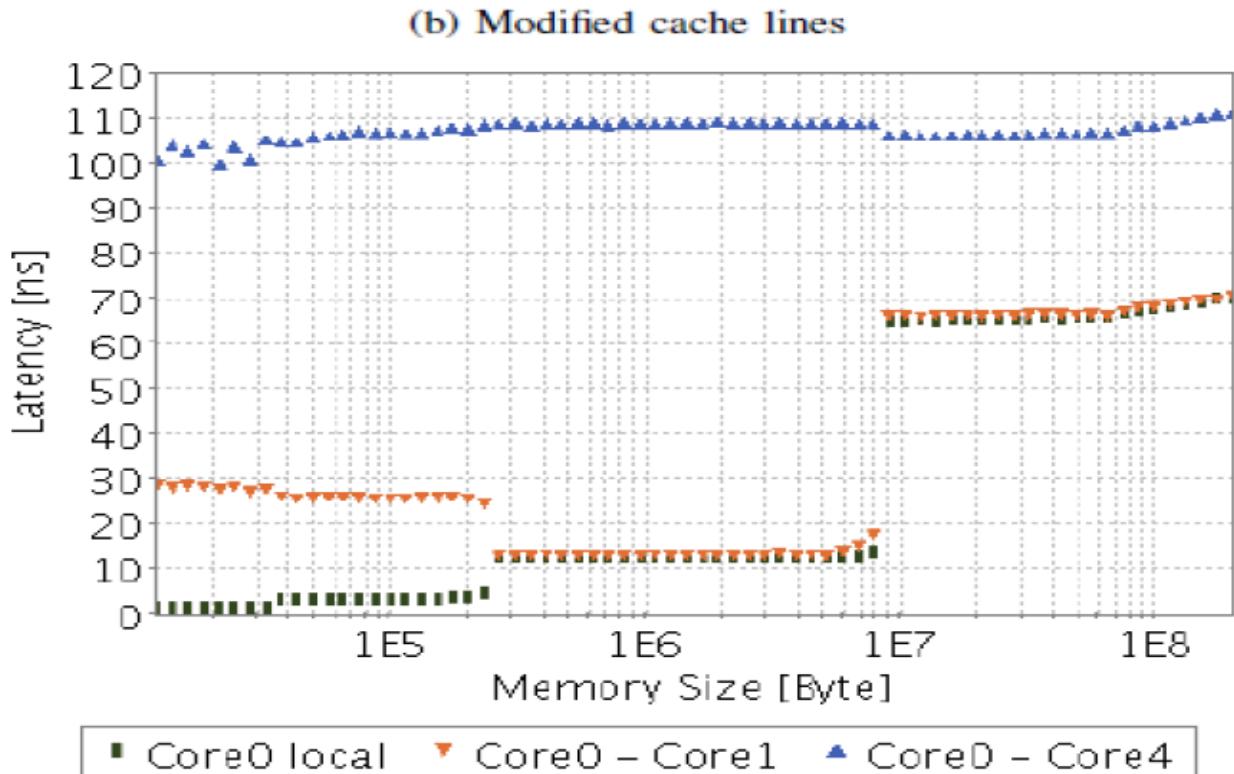


Fig. 7. Read latencies of core 0 accessing cache lines of core 0 (local), core 1 (on die) or core 4 (via QPI) with cache lines initialized to the modified coherency state[1]



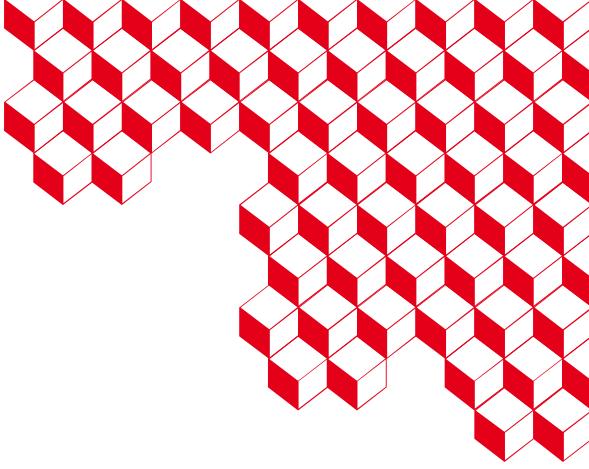
Architecture Nehalem

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	31.1	26.2	45.6	31.1		
Core1	19.3		19.7	9.4	13.2		26.2 10.1
Core4	9.0		9.2		5.6		6.3

Fig. 11. Core 0 Read Bandwidth in Gbps [1]

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	28.8	19.9	45.6	28.8		
Core1	23.4	22.2	17.6	9.4	13.0		19.9 8.4
Core4	9.0			8.3		9.6	5.5

Fig. 12. Core 0 Write Bandwidth in Gbps [1]



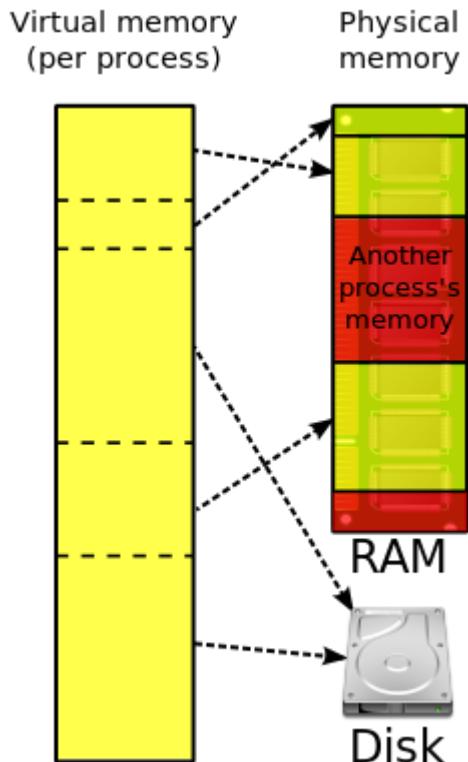
NUMA access on RAM

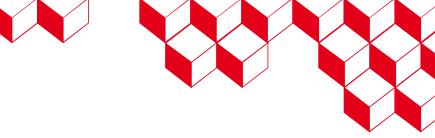
/!\ Be careful this is very important /!



First touch policy on almost all systems

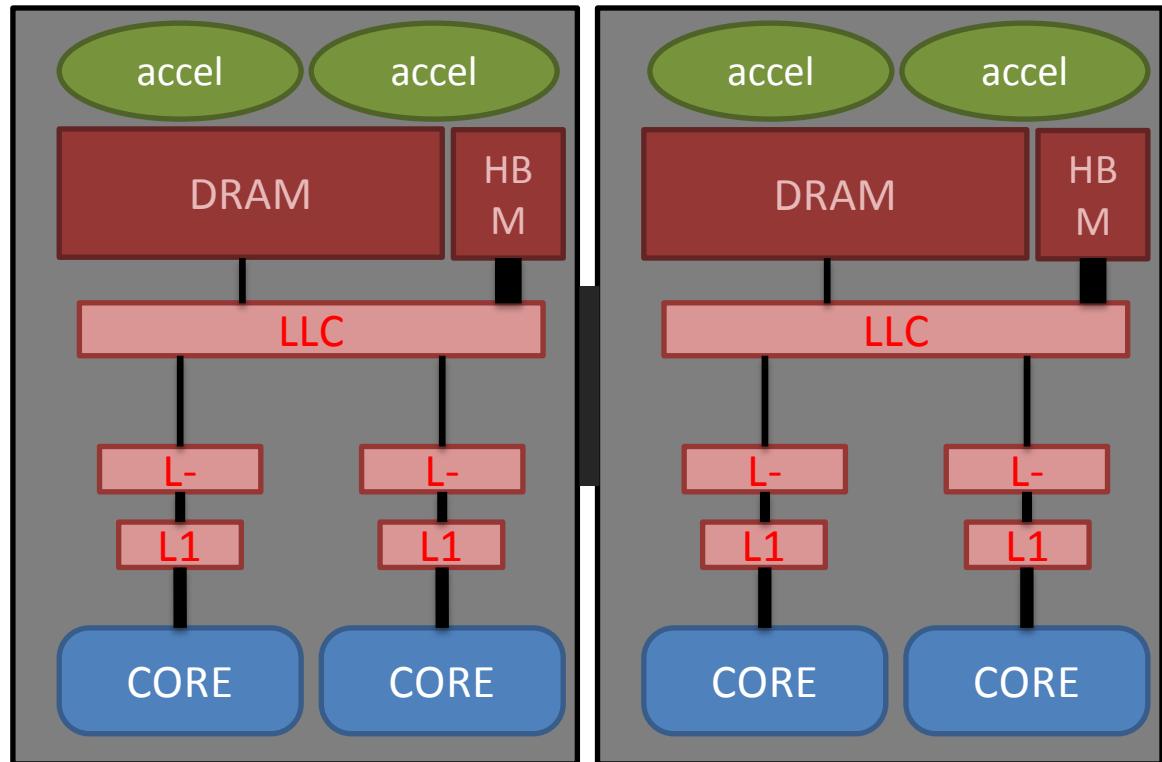
- ❖ Note that the data is not allocated by the malloc function
- ❖ It is allocated the first time the data is touched
 - Hence, if thread 0 calls the malloc and thread 1 is the first touching (e.g., initializing the data), data will be allocated where thread 1 is.
- ❖ Be careful to previously presented problems
 - NUMA effects, false sharing, ...
- ❖ Read pagination to go further !
 - ❖ https://en.wikipedia.org/wiki/Virtual_memory





Access on RAM

- ❖ We want to add 2 vectors



A screenshot of a terminal window titled 'Terminal'. The window contains the following C code:

```
#include <stdlib.h>

extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

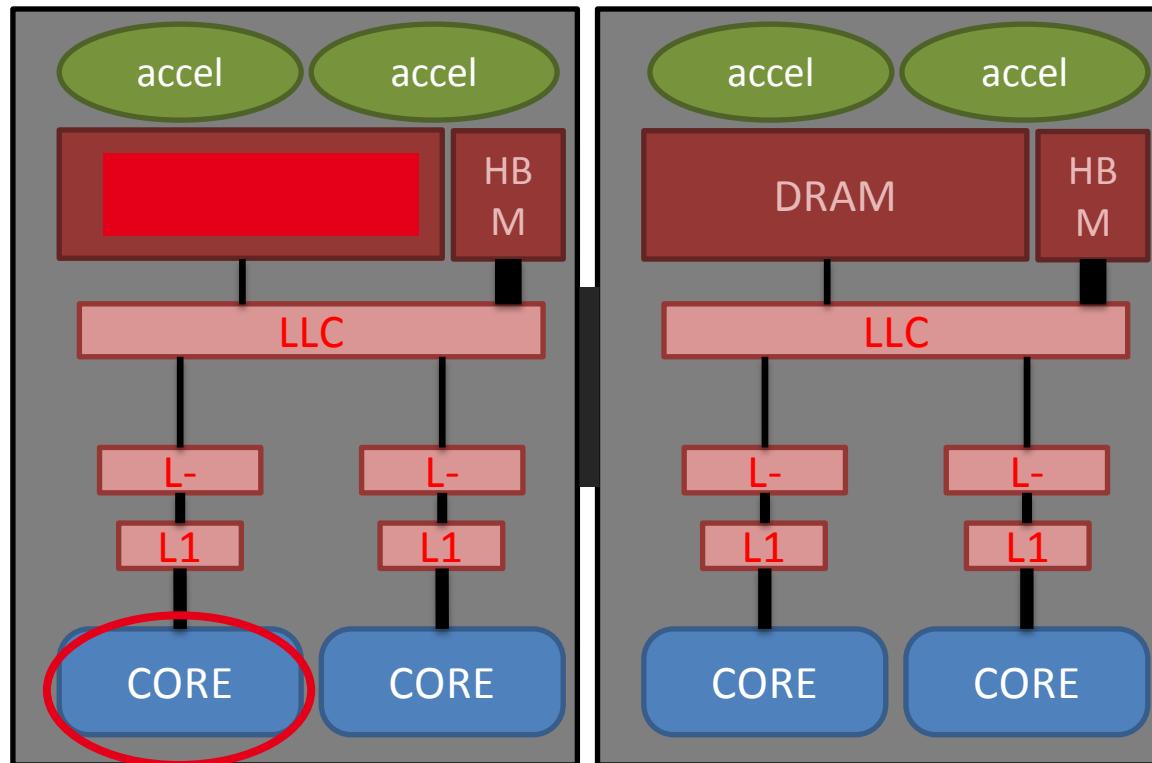
    return 0;
}
```

The terminal window shows the file name 'omp.c' and the command 'sum()' being run. The status bar at the bottom indicates '42%' completion and '14/33' errors.



Access on RAM (sequential)

- ❖ We want to add 2 vectors
 - **init** → initialize data closer to the core



```
#include <stdlib.h>

extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

    return 0;
}
```

omp.c

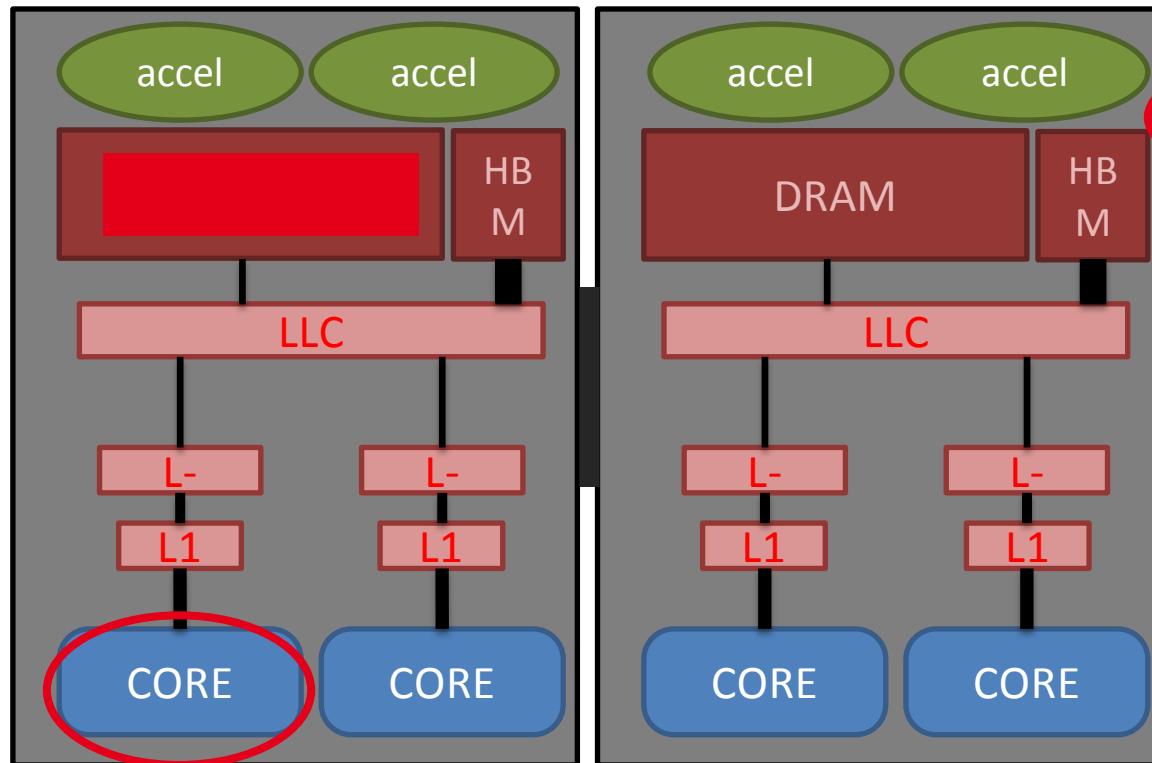
"omp.c" 33L, 538B written

sum() c 42% ln :14/33≡:1



Access on RAM (sequential)

- ❖ We want to add 2 vectors
 - **init** → initialize data closer to the core
 - **sum** → sum the two vectors on the core



A screenshot of a terminal window titled "Terminal". The code in the terminal is:

```
#include <stdlib.h>

extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

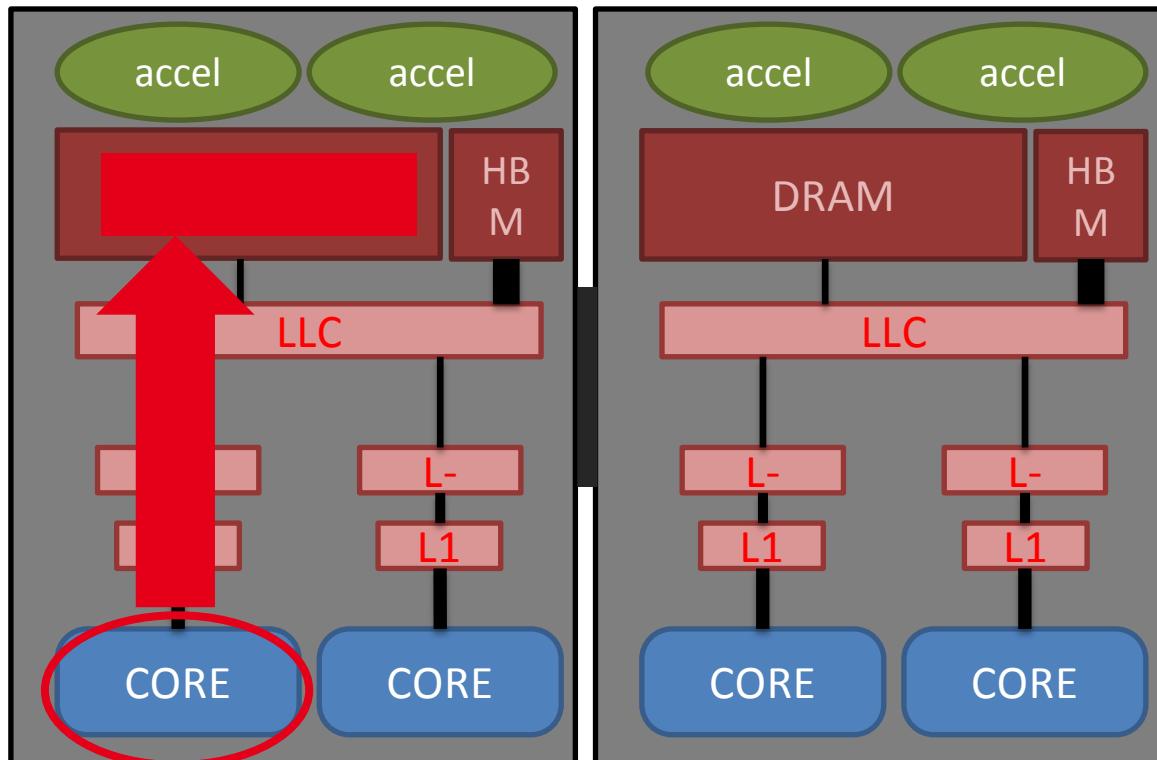
    return 0;
}
```

The terminal shows the command "omp.c" being compiled and run, with the output "sum() c 42% ln :14/33≡%:1" and the message "omp.c 33L, 538B written". A red oval highlights the `sum` function definition in the code.



Access on RAM (sequential)

- ❖ We want to add 2 vectors
 - **init** → initialize data closer to the core
 - **sum** → sum the two vectors on the core
- ❖ The core compute on closest RAM.
→ We are Happy :D



Terminal

```
#include <stdlib.h>

extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

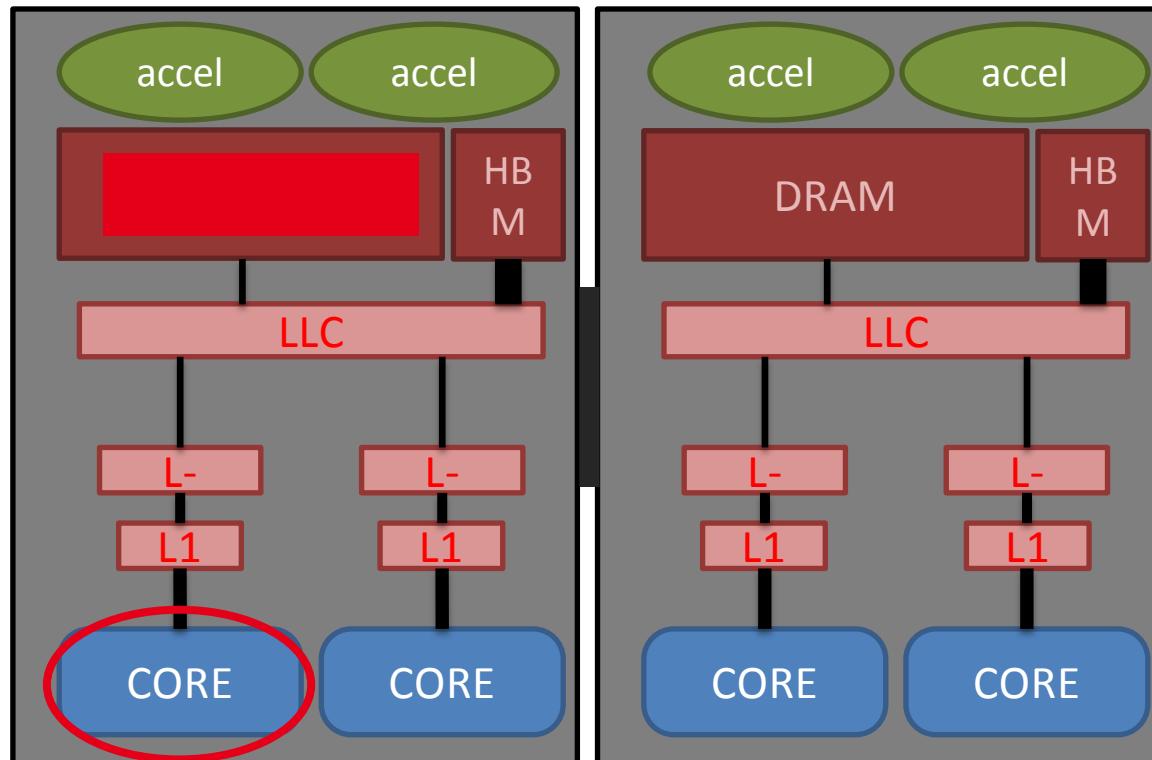
    return 0;
}
```

omp.c 42% ln :14/33≡:1
"omp.c" 33L, 538B written



NUMA access on RAM (parallel compute)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data closer to the core



```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

#pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

    return 0;
}
```

omp.c

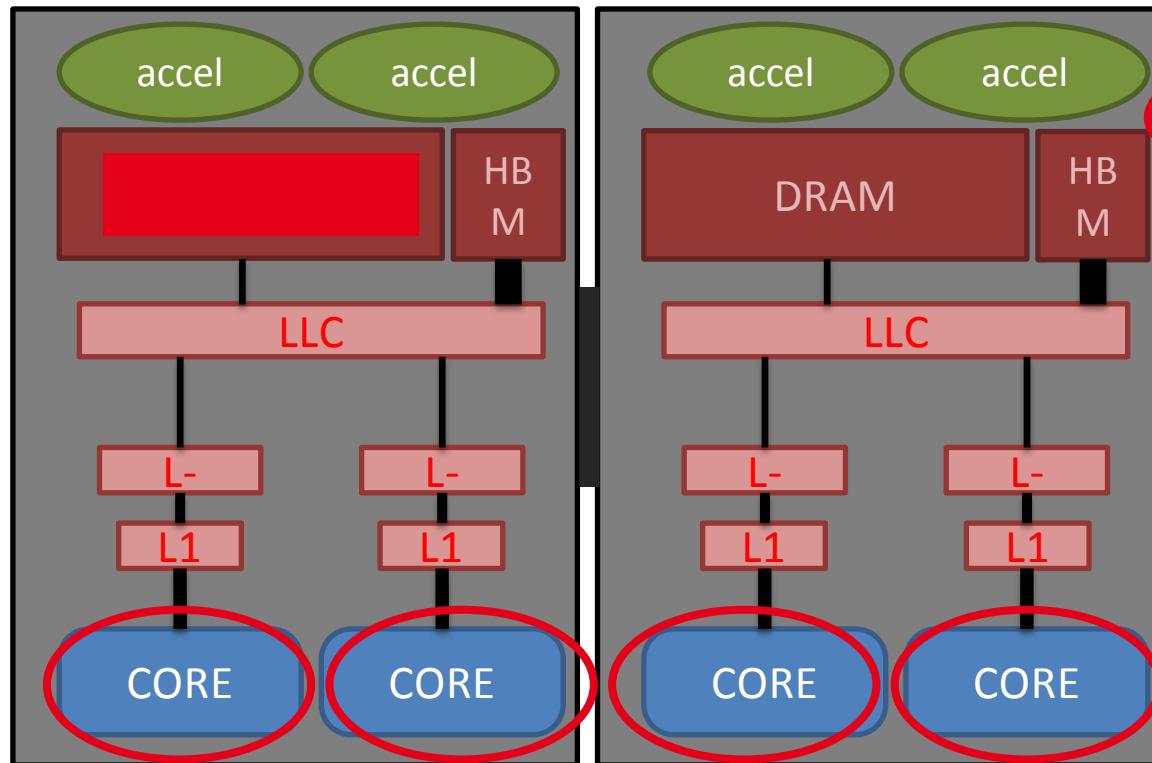
"omp.c" 34L, 581B written

sum() c 41% ln :14/34≡%:1



NUMA access on RAM (parallel compute)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data closer to the core
 - **sum** → sum the two vectors on **all cores**



```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

#pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

    return 0;
}
```

The terminal window displays the following OpenMP code for vector addition:

```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

#pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

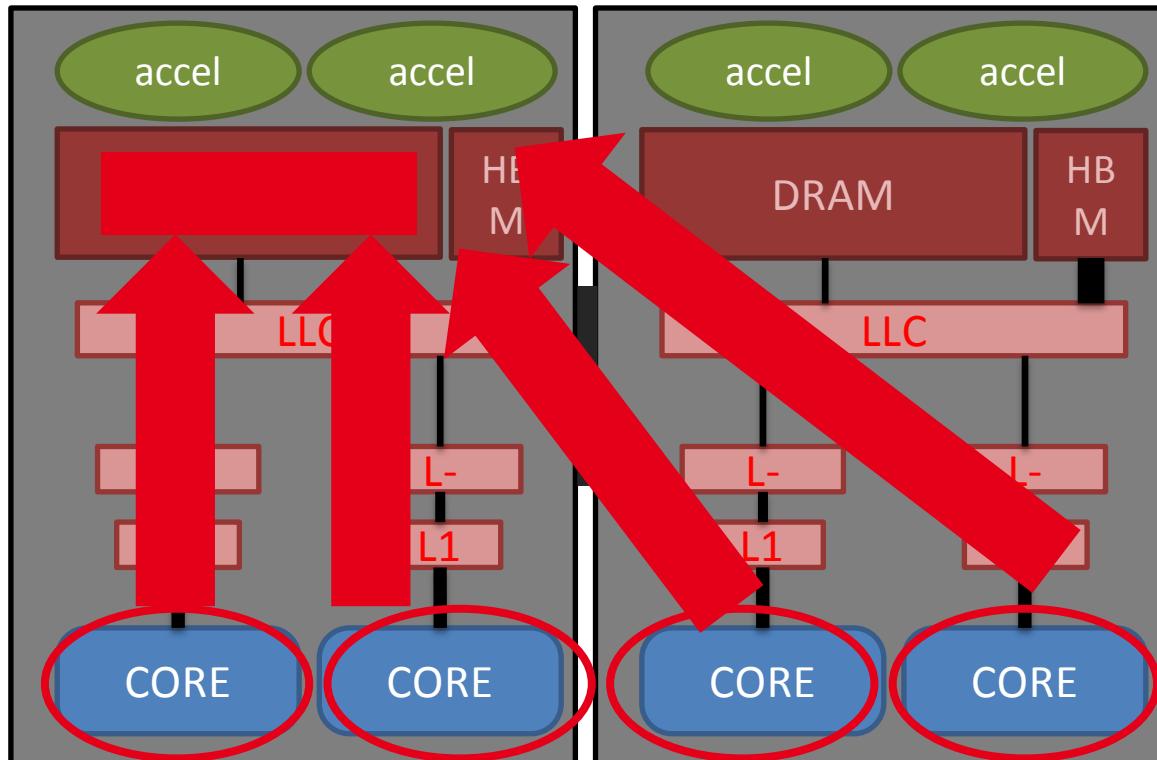
    return 0;
}
```

The code includes functions for initializing vectors and summing them. A red oval highlights the `#pragma omp parallel for` loop in the `sum` function, which indicates that the computation will be distributed across multiple cores.



NUMA access on RAM (parallel compute)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data closer to the core
 - **sum** → sum the two vectors on **all cores**
- ❖ All cores compute on only one NUMA node
→ **We are sad :-(**



Terminal

```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

#pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

    return 0;
}
```

omp.c

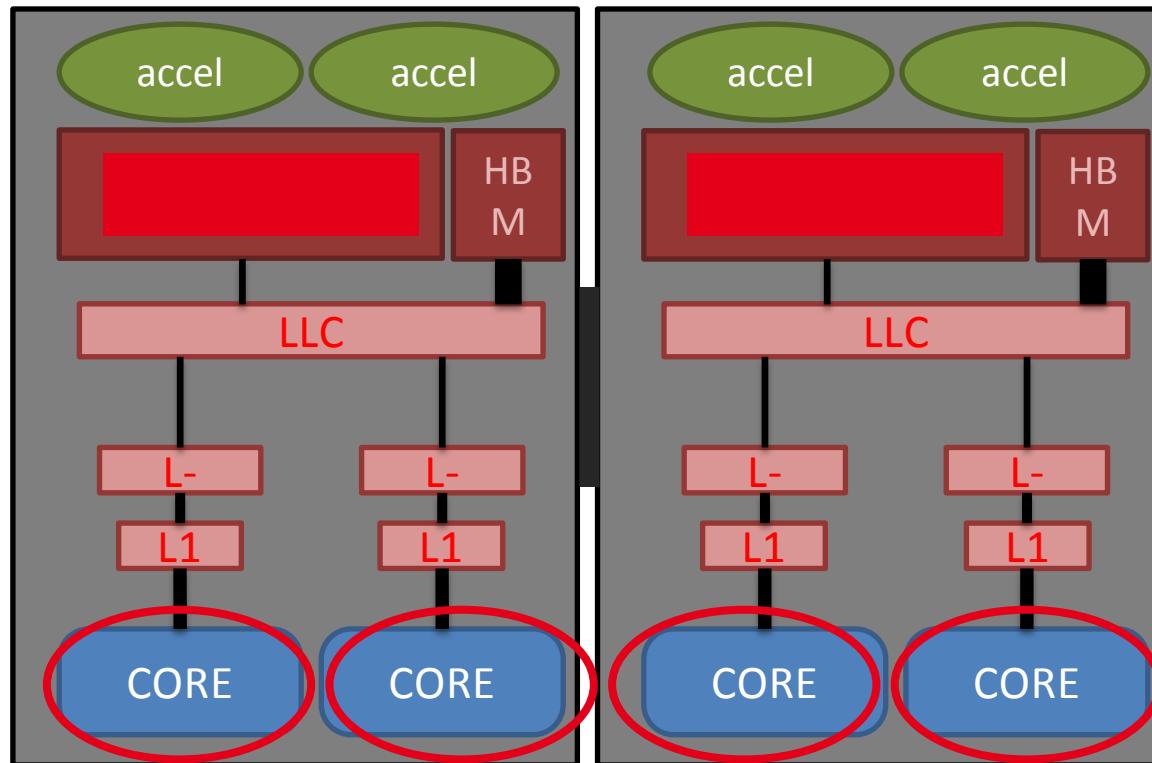
"omp.c" 34L, 581B written

sum() c 41% ln :14/34≡:1



NUMA access on RAM (parallel compute and init)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data in parallel on **all cores**



```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    #pragma omp parallel for
    for(i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    #pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

    sum(c,a,b,size);

    return 0;
}
```

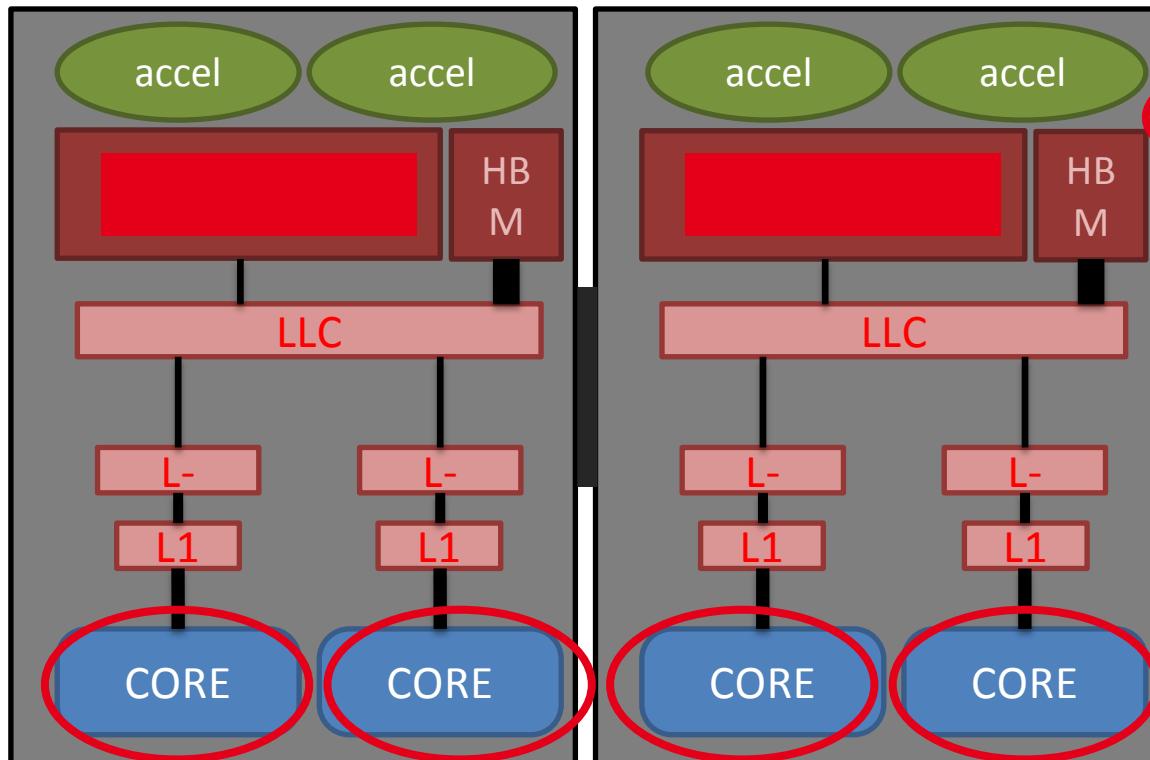
The terminal window shows the following content:

- File: omp.c
- Line: 35L, 608B written
- Status: init() c 22% ln :8/35≡%:3



NUMA access on RAM (parallel compute and init)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data in parallel on **all cores**
 - **sum** → sum the two vectors on **all cores**



```
Terminal
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    #pragma omp parallel for
    for(int i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    #pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

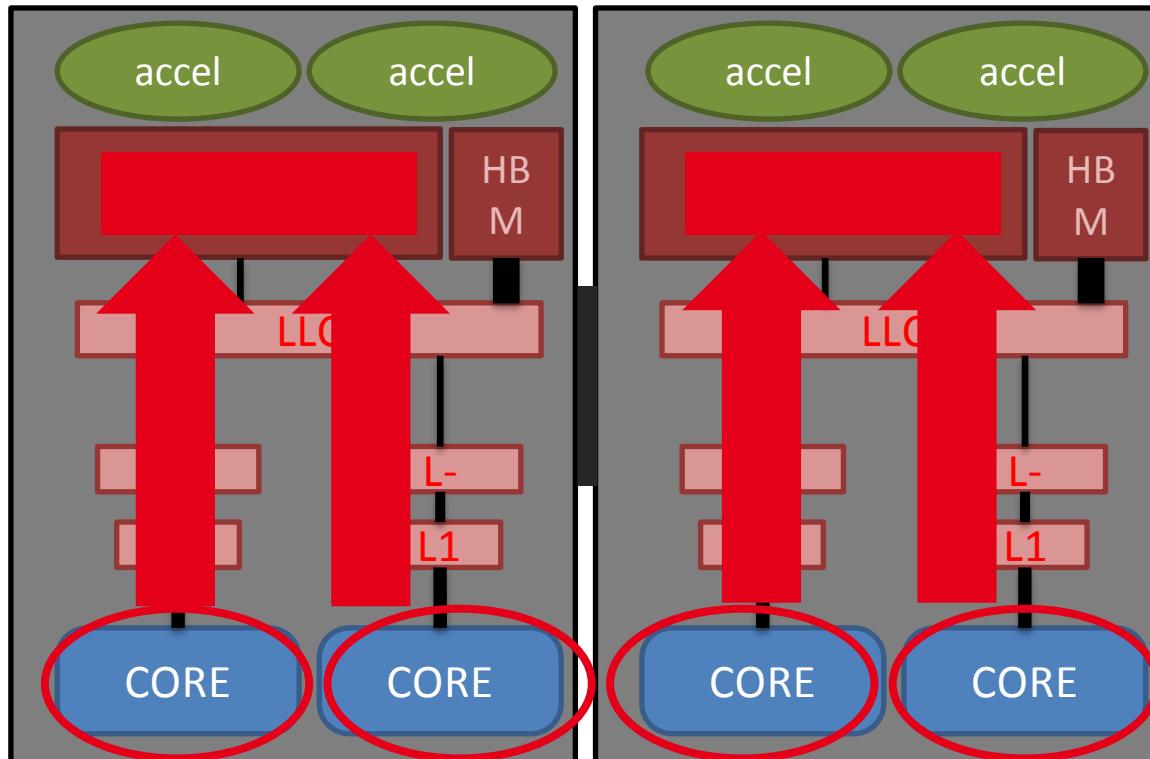
    sum(c,a,b,size);

    return 0;
}
N... omp.c
"omp.c" 35L, 608B written
init() c 22% ln :8/35≡%:3
```



NUMA access on RAM (parallel compute and init)

- ❖ We want to add 2 vectors with OpenMP
 - **init** → initialize data in parallel on **all cores**
 - **sum** → sum the two vectors on **all cores**
- ❖ All cores compute on **all NUMA nodes**
→ We are Happy :D



```
#include <stdlib.h>
#include <omp.h>
extern double somerandomvalue();

void init(double * tab, int size){
    int i;

    #pragma omp parallel for
    for(i=0;i<size;++i){
        tab[i]=somerandomvalue();
    }
}

void sum (double *tab_result, double *tab1, double *tab2, int size){
    int i;

    #pragma omp parallel for
    for(i=0;i<size;++i){
        tab_result[i]=tab1[i]+tab2[i];
    }
}

int main (){
    int size=10000;
    double *a = malloc(size*sizeof(double));
    double *b = malloc(size*sizeof(double));
    double *c = malloc(size*sizeof(double));

    init(a,size);
    init(b,size);

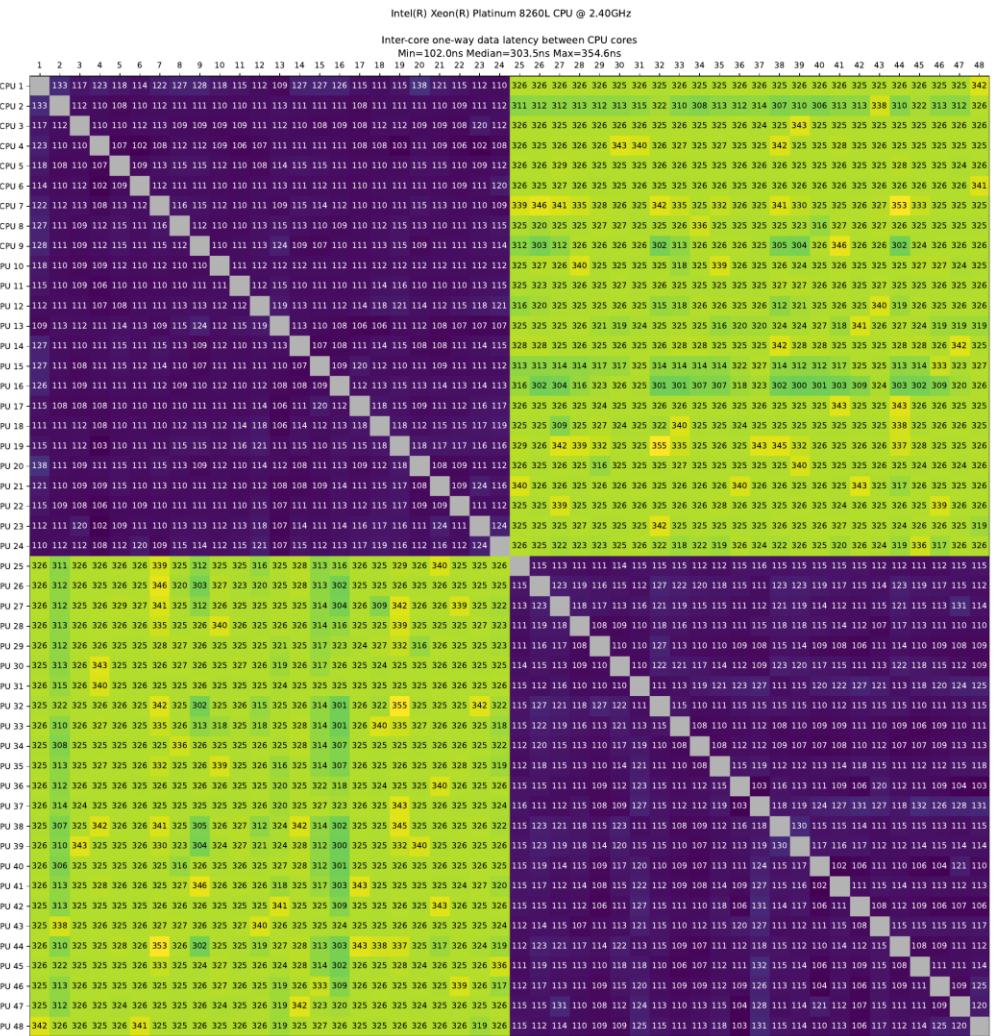
    sum(c,a,b,size);

    return 0;
}
```

More recent architectures : Intel

❖ Cache effect on Intel Platinum 8260L

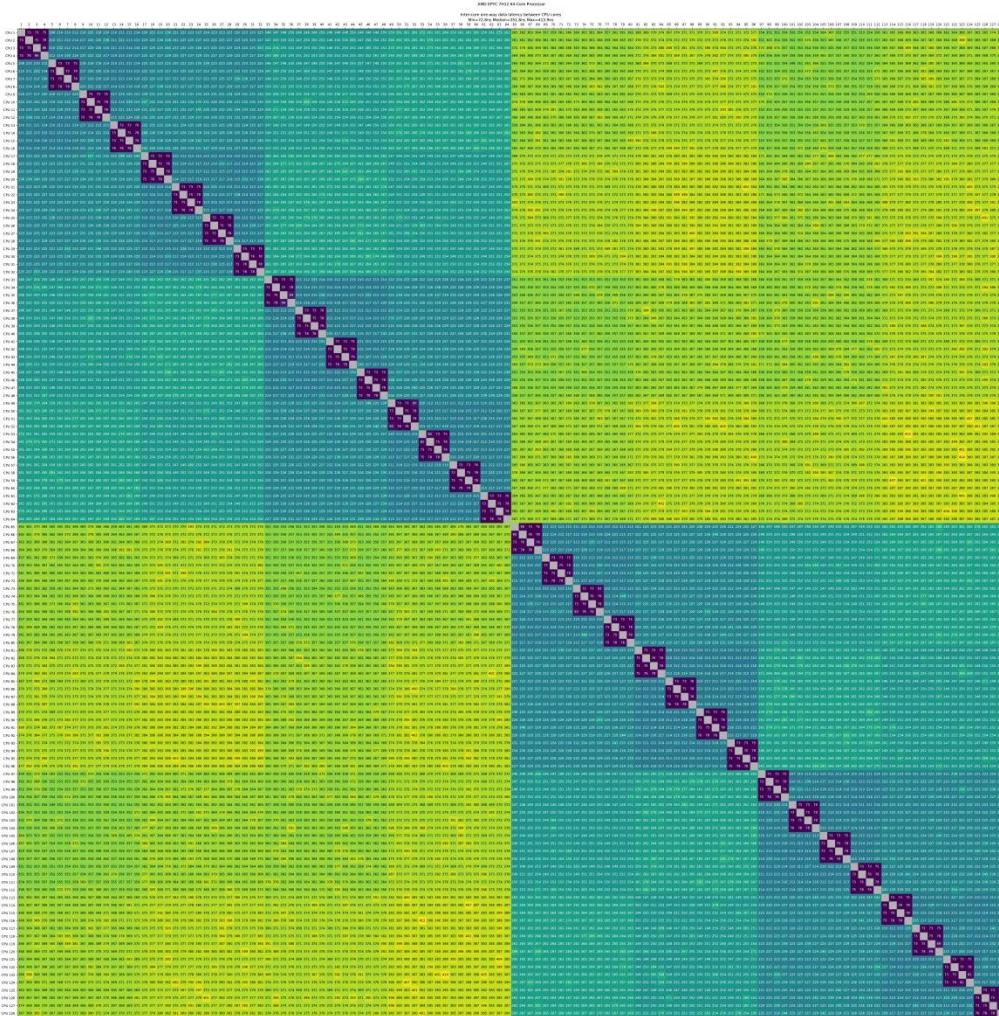
- 24 cores per processor
 - 2 sockets





More recent architectures : AMD

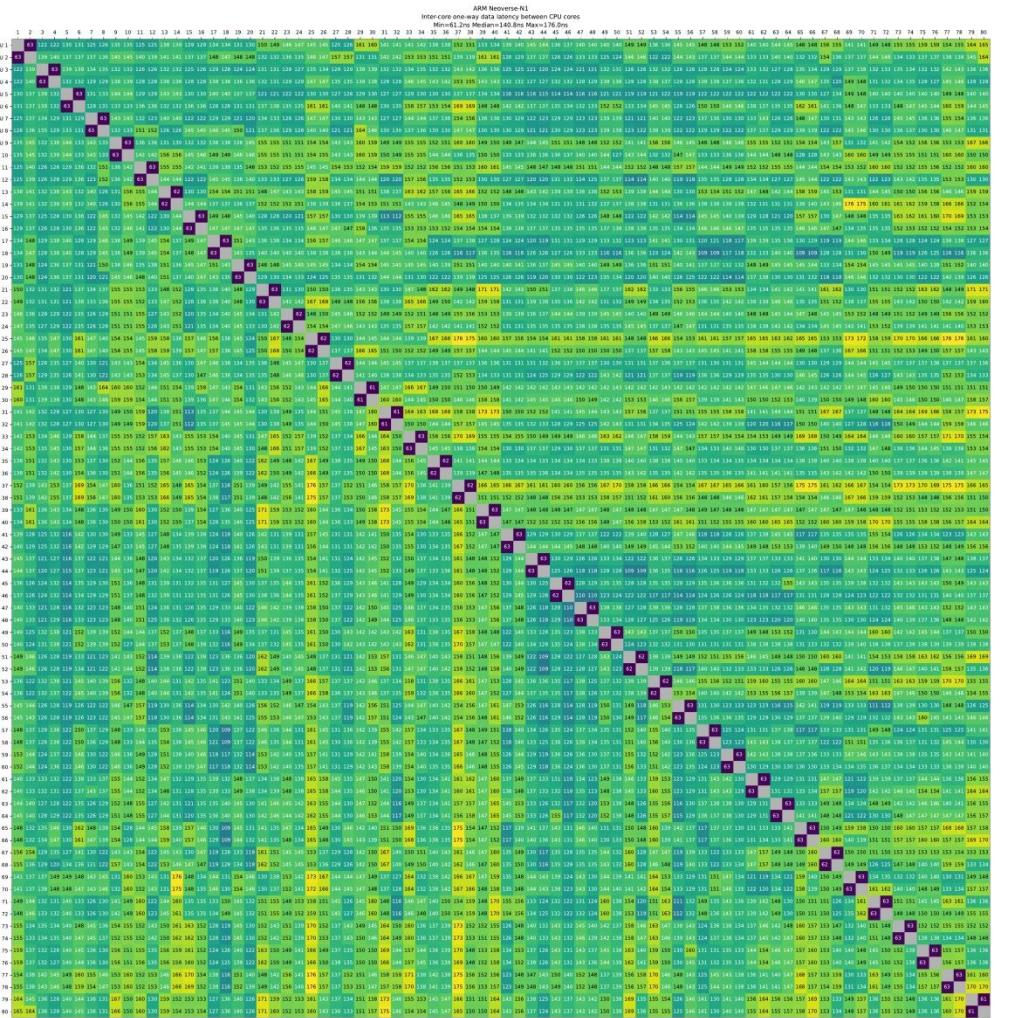
- ❖ Cache effect on AMD EPYC 7H12
 - 64 cores per processor
 - 2 sockets





More recent architectures : ARM

- ❖ Cache effect on ARM Neoverse N1
- 80 cores per processor
- 1 socket





Data locality

- ❖ The closer a memory to the core, the more efficient it is
 - Better (higher) bandwidth
 - Better (smaller) latency
- ❖ Fetching a data in global memory can be very costly
 - Automatically stored in cache levels
- ❖ It is important to reuse data in cache
 - As we have seen previously (cache blocking)



Data locality vs NUMA

- ❖ Having the required data close to the core is necessary to leverage performance
- ❖ However, if the same data are required by several cores, NUMA effects can be costly
- ❖ Important to avoid ping-pongs of data between cores/threads
 - Beware of **false sharing!**
 - False sharing: threads not actually sharing actual data, but sharing caches
 - E.g., thread 0 reads even cells and thread 1 reads odd cells



Data locality vs NUMA

- ❖ Having the required data close to the core is necessary to leverage performance
- ❖ However, if the same data are required by several cores, NUMA effects can be costly
- ❖ Important to avoid ping-pongs of data between cores/threads
 - Beware of **false sharing!**
 - False sharing: threads not actually sharing actual data, but sharing caches
 - E.g., thread 0 reads even cells and thread 1 reads odd cells

■ But wait !! What is false sharing ?!

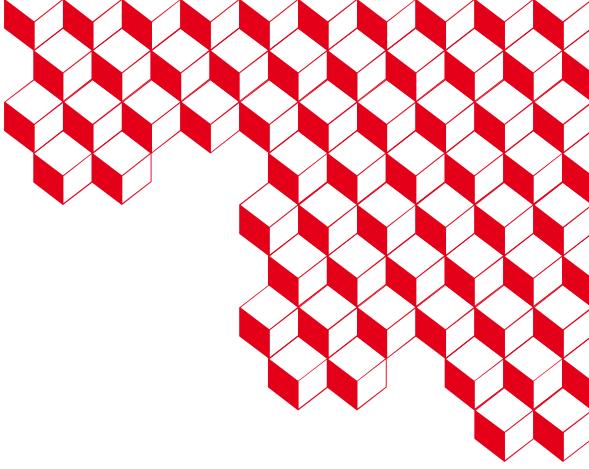


Data locality vs NUMA

- ❖ Having the required data close to the core is necessary to leverage performance
- ❖ However, if the same data are required by several cores, NUMA effects can be costly
- ❖ Important to avoid ping-pongs of data between cores/threads
 - Beware of **false sharing!**
 - False sharing: threads not actually sharing actual data, but sharing caches
 - E.g., thread 0 reads even cells and thread 1 reads odd cells

■ But wait !! What is false sharing ?!

➤ Let speak about concurrency !



Concurrency



Data concurrency

- ❖ Data concurrency is the sharing of data between multiple threads
- ❖ Using shared memory and sharing data is a classical method to reduce memory footprint of the application
- ❖ The goal is to maximize the amount of useful data per cacheline to benefit a maximum from data locality



Data concurrency

- ❖ Problem: if multiple threads access the same memory cell
 - Synchronization issue
 - Cacheline may be going from one cache to another
- ❖ To be efficient, threads must not access the same data at the same time
- ❖ Then, cacheline is in Exclusive mode in L1D and no cacheline movement



False sharing

- ❖ Problem: If multiple threads write/access different memory addresses, but bound to the same cacheline
- ❖ Threads don't access the same data, but cachelines are still shared between threads
 - Same cacheline movements as the previous case
 - Same performance loss
- ❖ This phenomenon is called « False Sharing »



False sharing example

```
#include <chrono>
#include <iostream>
#include <omp.h>
#include <stdlib.h>

double somerandomvalue() { return 42.0; }

void init(double *tab, int size) {
    int i;
#pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        tab[i] = somerandomvalue();
    }
}

void increasing_values_seq(double *tab, int size) {
    int i;
    for (i = 0; i < size; ++i) {
        tab[i] = tab[i] + 1;
    }
}

void increasing_values_omp(double *tab, int size) {
    int i;
#pragma omp parallel for schedule(runtime)
    for (i = 0; i < size; ++i) {
        tab[i] = tab[i] + 1;
    }
}

int main() {
    int size = 1024 * 1024 * 1024;

    // bad c++, only for the example
    double *false_shared_array =
        static_cast<double *>(malloc(size * sizeof(double)));
    double *false_shared_array_ref =
        static_cast<double *>(malloc(size * sizeof(double)));

    init(false_shared_array_ref, size);
    init(false_shared_array, size);

    increasing_values_seq(false_shared_array_ref, size);

    auto start = std::chrono::steady_clock::now();

    increasing_values_omp(false_shared_array, size);

    auto end = std::chrono::steady_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();

    // test if arrays are equals
    bool is_equal = [=](double *tabl, double *tab2) -> bool {
        bool res = true;
        for (int i = 0; i < size; ++i) {
            if (tabl[i] != tab2[i])
                res = false;
        }
        return res;
    }(false_shared_array_ref, false_shared_array);

    std::cout << "duration = " << duration << " is_equal = " << is_equal
        << std::endl;

    free(false_shared_array_ref);
    free(false_shared_array);

    return 0;
}
```

File: false_sharing.cpp [+] Line: 46% [ln : 33/71] File: false_sharing.cpp [+] Line: 100% [ln : 71/71]



False sharing example

- ❖ What happens ?

```
void increasing_values_omp(double *tab, int size) {  
    int i;  
#pragma omp parallel for schedule(runtime)  
    for (i = 0; i < size; ++i) {  
        tab[i] = tab[i] + 1;  
    }  
}
```

```
#!/bin/bash  
  
#equal to schdule static for size = 1024**3  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static,268435456 ./a.out  
echo "schedule static 268435456 : we do not got false sharing :D"  
  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static ./a.out  
echo "schedule static : we do not got false sharing :D"  
  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static,1 ./a.out  
echo "schedule static 1 : we got false sharing :(
```

```
a.out false_sharing.cpp readme.md run static1.txt static268435456.txt  
(23/02/24 23:30:05) /dev/pts/2/0 ~/test/false_sharing  
>> ./run  
duration = 599487262 is_equal = 1  
schedule static 268435456 : we do not got false sharing :D  
duration = 605557178 is_equal = 1  
schedule static : we do not got false sharing :D  
duration = 5079612654 is_equal = 1  
schedule static 1 : we got false sharing :(
```



False sharing example

- ❖ What happens ?
- Speedup = 8.47 !!

```
void increasing_values_omp(double *tab, int size) {  
    int i;  
#pragma omp parallel for schedule(runtime)  
    for (i = 0; i < size; ++i) {  
        tab[i] = tab[i] + 1;  
    }  
}
```

```
#!/bin/bash  
  
#equal to schdule static for size = 1024**3  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static,268435456 ./a.out  
echo "schedule static 268435456 : we do not got false sharing :D"  
  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static ./a.out  
echo "schedule static : we do not got false sharing :D"  
  
OMP_NUM_THREADS=4 OMP_SCHEDULE=static,1 ./a.out  
echo "schedule static 1 : we got false sharing :(
```

```
a.out false_sharing.cpp readme.md run static1.txt static268435456.txt  
(23/02/24 23:30:05) /dev/pts/2/0 ~/test/false_sharing  
>> ./run  
duration = 599487262 is_equal = 1  
schedule static 268435456 : we do not got false sharing :D  
duration = 605557178 is_equal = 1  
schedule static : we do not got false sharing :D  
duration = 5079612654 is_equal = 1  
schedule static 1 : we got false sharing :(
```



False sharing example

- ❖ What happens ?
- Speedup = 8.47 !!

```
void increasing_values_omp(double *tab, int size) {  
    int i;  
#pragma omp parallel for schedule(runtime)  
    for (i = 0; i < size; ++i) {  
        tab[i] = tab[i] + 1;  
    }  
}
```

- OMP_SCHEDULE=static
 - Each thread have their own quarter of the iteration space.
 - Cache lines are not shared among threads
- OMP_SCHEDULE=static,1
 - Each thread share caches lines with other threads
- Example of OMP_NUM_THREADS=4 OMP_SCHEDULE=static,3

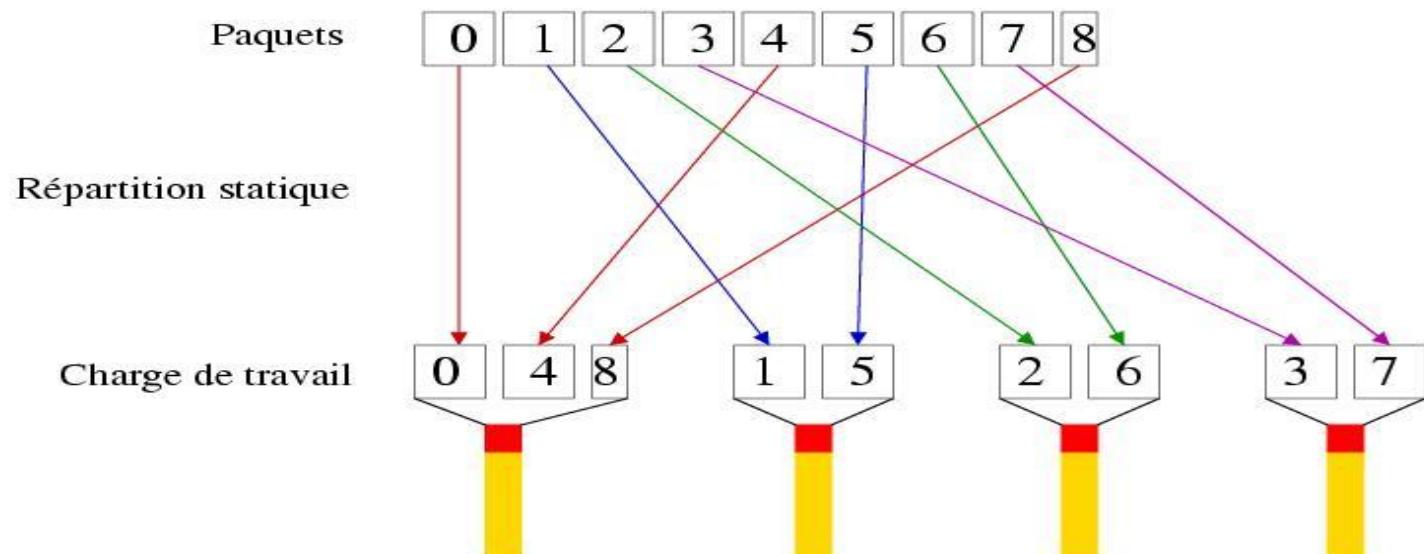
Figure from <https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/>





Static Scheduling

- ❖ Static scheduling splits iteration domain into chunk with equal size (when possible)
 - By default: one chunk of max size
 - When specified, chunk size is constant and chunks are distributed in round robin fashion
 - `OMP_SCHEDULE=static,1`





Concurrency overview

- ❖ At every moment:
 - Only one core can work at a time on the same data
 - Concurrency to acquire the cachelines
 - Adding a thread working on the same data will only increase the data access time

- ❖ This effect can have great impact on HPC applications
 - architectures are multiprocessors
 - High number of cores
 - Concurrency effect increased due to NUMA effect



Concurrency: avoiding False sharing

- ❖ Simplest solution: put every variable on a separate cacheline
- ❖ But it will considerably increase the memory footprint
 - Not realistic on HPC applications
- ❖ Towards true solutions
 - Identify variables used by only one thread
 - Identify variables in read-only mode
 - Frequency of modifications



Concurrency: avoiding False sharing

- ❖ Variables never written can be considered as constant
- ❖ Thus, it is possible for those to be shared
- ❖ To mark them as *const* can help find read-only variables
- ❖ *const* variables are moved to the *readonly* section of the code



Concurrency: avoiding False sharing

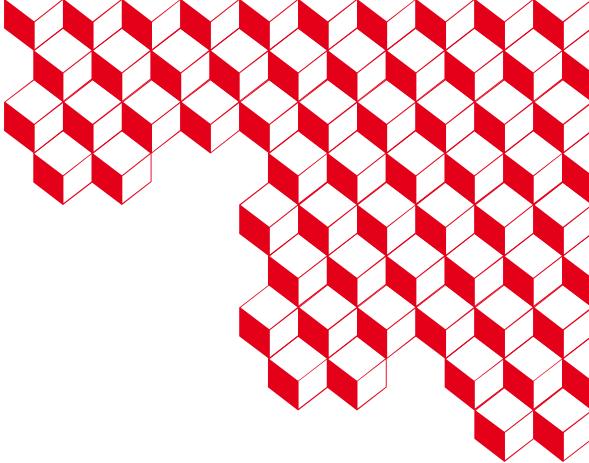
- ❖ A few keywords for global variables:
- ❖ `__thread`: put the variable in a memory section specific to each thread
- ❖ `__attribute__((section(".data.ro")))`: put the variable in the *readonly* section of the code.
 - Approach specific to the GNU linker



Concurrency: avoiding False sharing

- ❖ How to deal variables in structures?
- Group the structure fields depending on their type and access ratio
- Add some *padding* to add memory space between variables

```
int foo = 1;
int baz = 3;
struct {
    struct all {
        int bar;
        int xyzzy;
    };
    char pad[CLSIZE - sizeof(struct all)];
} rwstruct __attribute__((aligned(CLSIZE))) =
{ { .bar = 2, .xyzzy = 4 } };
```



Memory bandwidth



Memory bandwidth

- ❖ Amount of memory bandwidth per socket is limited
- ❖ All cores/threads share the same bandwidth
- The more threads, the less bandwidth per thread
- ❖ Memory bandwidth don't follow core frequency and number for cores
- With the advent of multicores, even more important to handle data locality correctly

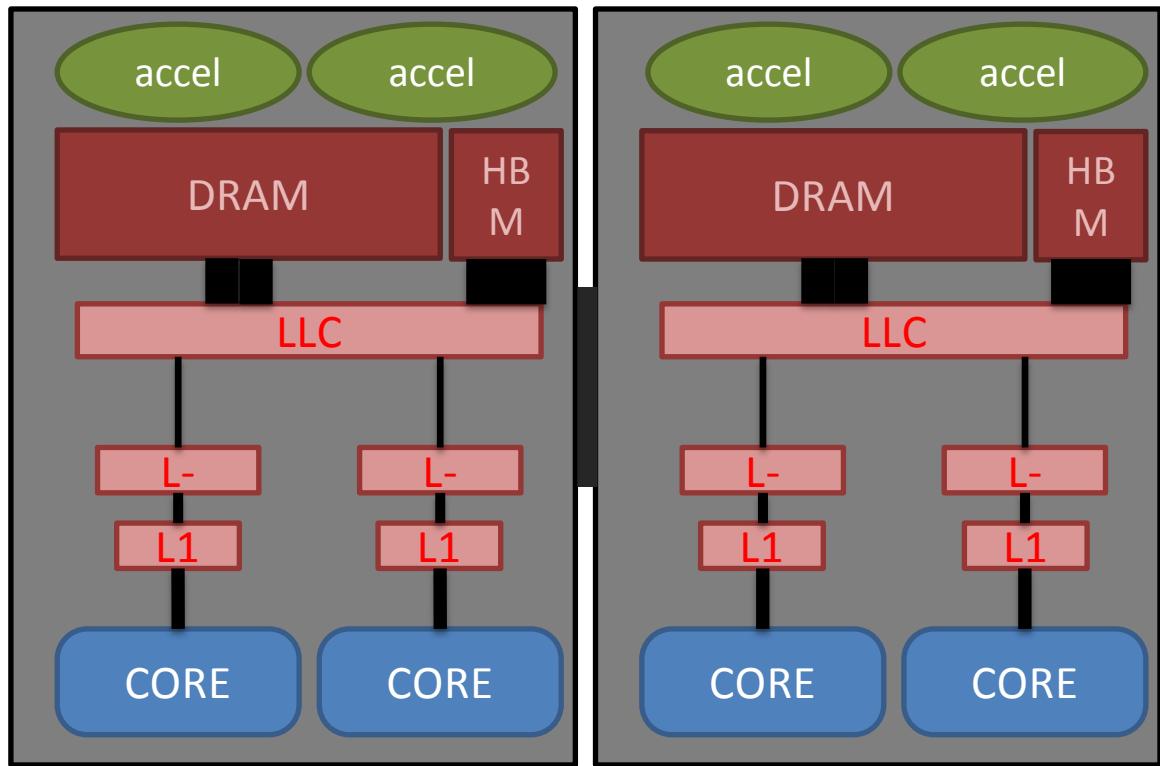


Memory bandwidth

- ❖ NUMA effect also apply on bandwidth
- ❖ The interconnection between sockets is often limited
- ❖ Worst, it is also possible that the memory requests issued from another node have less priority than local memory requests



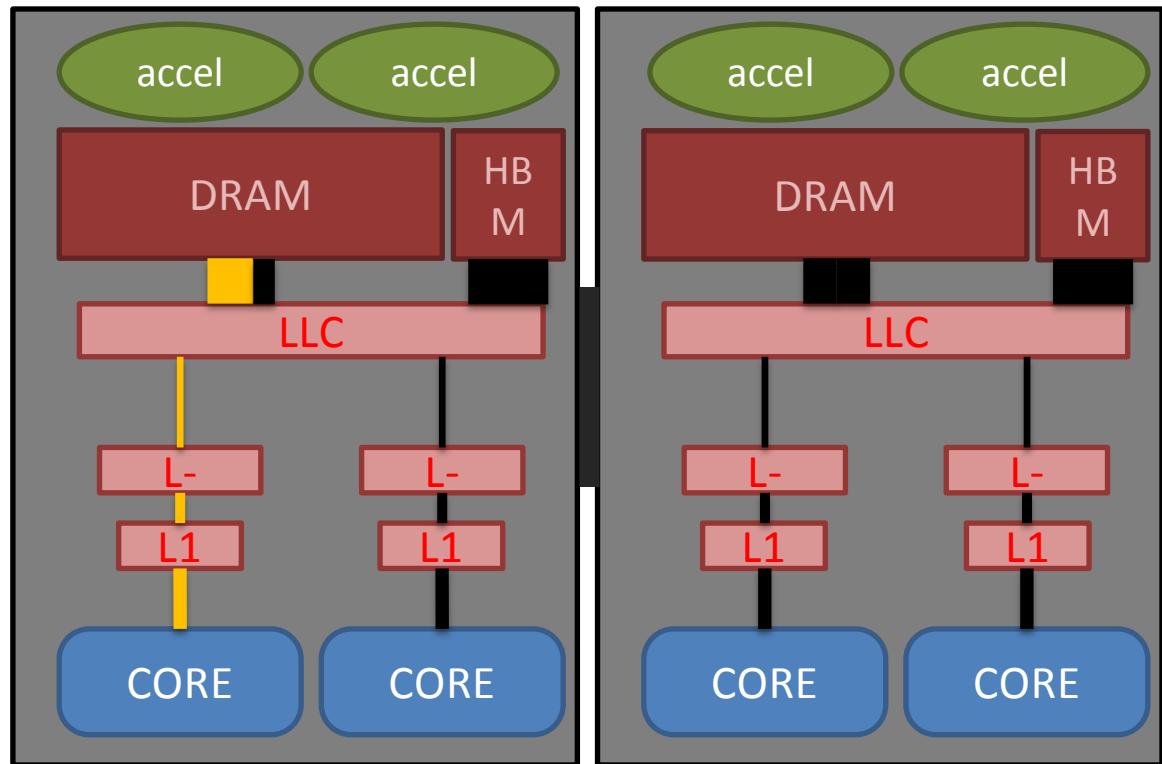
Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?



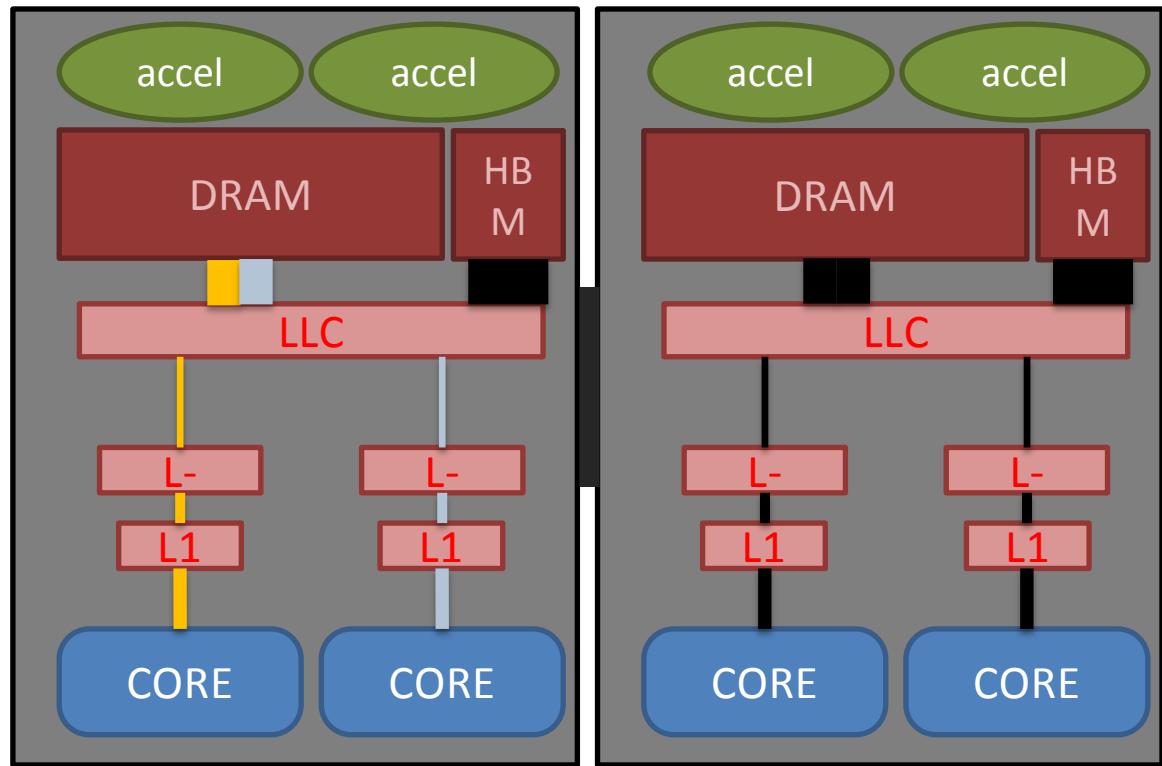
Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?
- ❖ 1) using only one thread may not allow to use the full available bandwidth



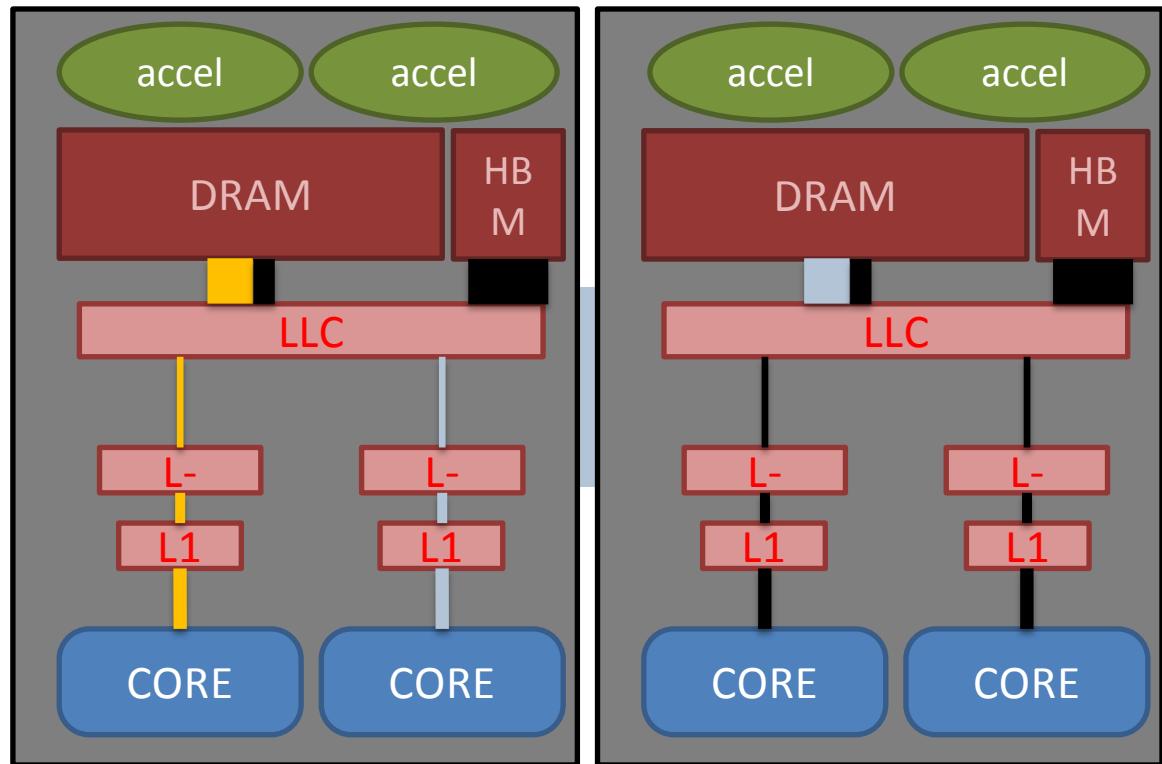
Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?
- ❖ 1) using only one thread may not allow to use the full available bandwidth



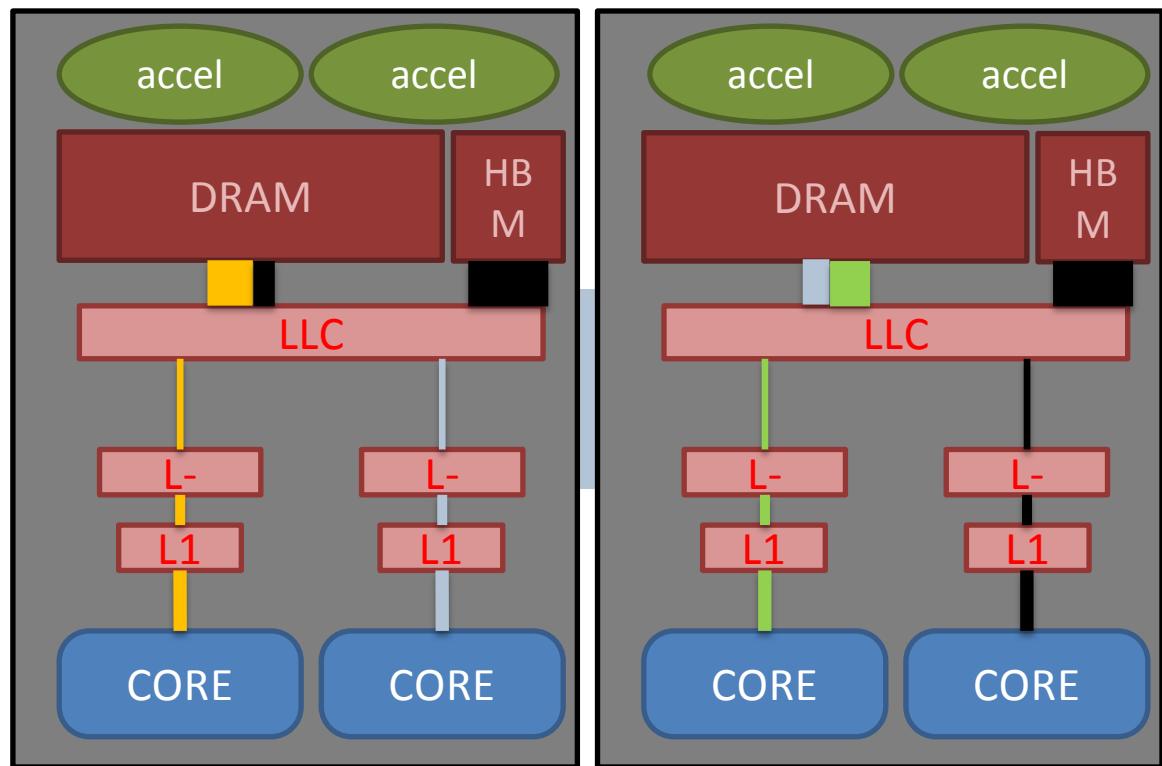
Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?
- ❖ 2) local memory requests may be prioritized



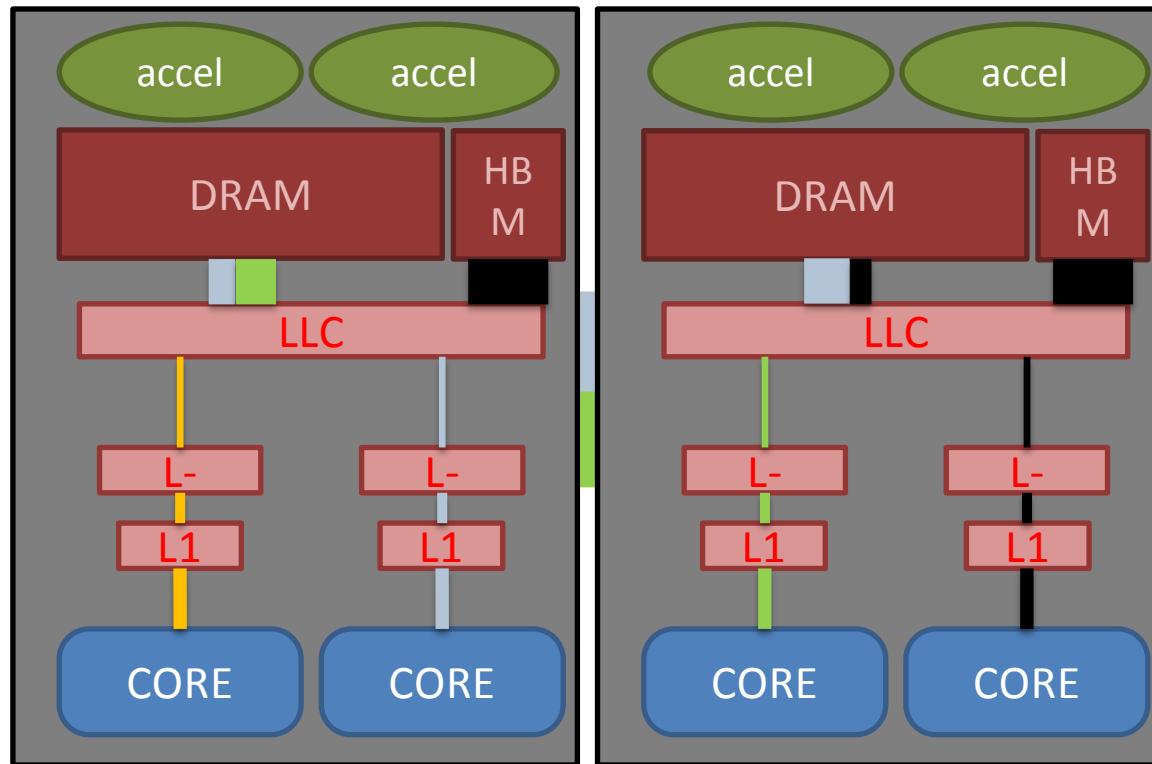
Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?
- ❖ 2) local memory requests may be prioritized



Memory bandwidth



- ❖ Locality brings better latency
- ❖ What about bandwidth ?
- ❖ 3) cross-socket access in both directions at the same time will reduce the bandwidth available for each memory access



Memory bandwidth

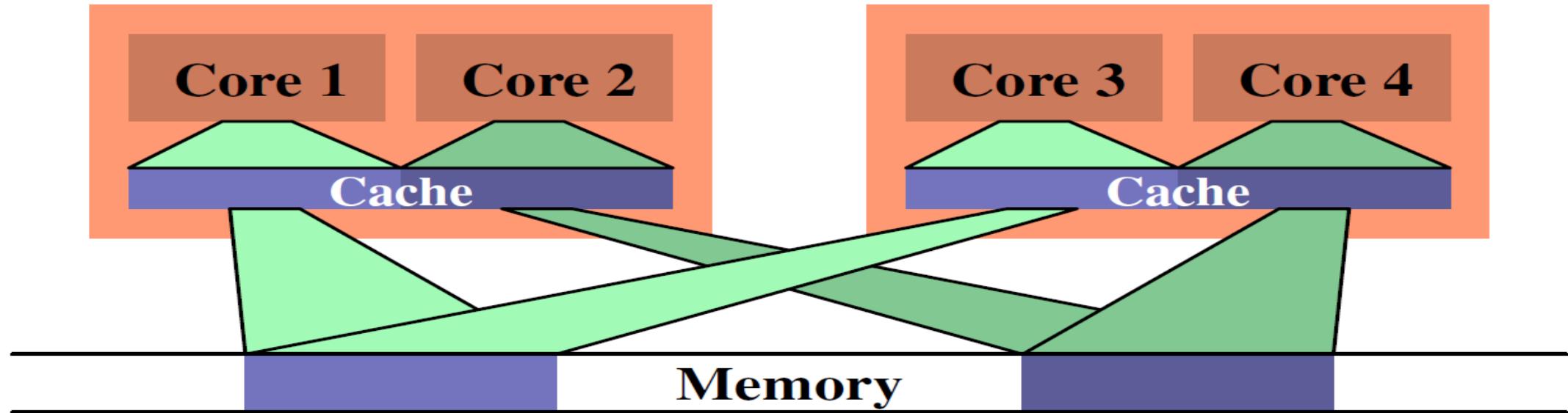


Figure 6.13: Inefficient Scheduling



Memory bandwidth

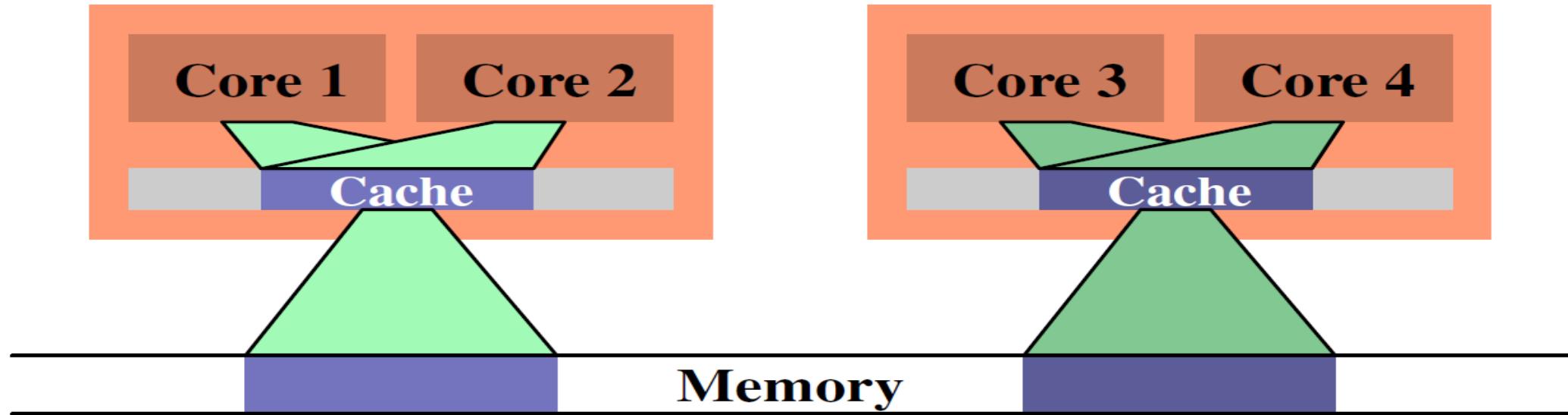


Figure 6.14: Efficient Scheduling



Memory bandwidth example

- ❖ OpenMP code run on a node with 2 sockets Intel Sandy bridge of 8 cores each
 - ❖ 16 cores on the node
- ❖ Addition of 2 arrays in a third one
 - ❖ 1 thread initializes the arrays
 - ❖ Tests with thread 0 and thread 8
 - ❖ The first 8 threads compute the addition in parallel



Memory bandwidth example (8th)

```
#pragma omp parallel
{
    int i;
    int thid = -1;

    thid = omp_get_thread_num();

    if(thid == THREAD_ALLOC)
    {
        printf("thread allocated == %d\n", thid);
        for(i=0; i<GSIZE; i++)
        {
            a[i] = i+1;
            b[i] = GSIZE-i-1;
            r[i] = 0;
        }
    }
    struct timeval tv_start;
    struct timeval tv_stop;

    gettimeofday(&tv_start, NULL);

    #pragma omp parallel
```

```
(c-inti)[jaegerj@inti2015 bandwidth_numa] $ ./bandwidth_test_local.pgr
thread allocated == 0
time for r[i]=a[i]+b[i] with allocation by thread 0 is 28s 961788us
(c-inti)[jaegerj@inti2015 bandwidth_numa] $ ./bandwidth_test_distant.pgr
thread allocated == 8
time for r[i]=a[i]+b[i] with allocation by thread 8 is 36s 261318us
```

```
#pragma omp parallel
{
    int i;
    int j;
    int thid = -1;
    thid = omp_get_thread_num();

    if(thid < (thnb/2))
    {
        int loop_b = (GSIZE / (thnb/2)) * thid;
        int loop_e = (GSIZE / (thnb/2)) * (thid+1);

        for(j=0; j< REPET; j++)
        {
            for(i=loop_b; i<loop_e; i++)
            {
                r[i] = a[i] + b[i];
            }
        }
    }
}
```

- ❖ 1 socket access either local or distant



Memory bandwidth example (64th milan)

```
#pragma omp parallel
{
    int i;
    int thid = -1;

    thid = omp_get_thread_num();

    if(thid == THREAD_ALLOC)
    {
        printf("thread allocated == %d\n", thid);
        for(i=0; i<GSIZE; i++)
        {
            a[i] = i+1;
            b[i] = GSIZE-i-1;
            r[i] = 0;
        }
    }

    struct timeval tv_start;
    struct timeval tv_stop;

    gettimeofday(&tv_start, NULL);

#pragma omp parallel
```

```
#pragma omp parallel
{
    int i;
    int j;
    int thid = -1;
    thid = omp_get_thread_num();

    if(thid < (thnb/2))
    {

        int loop_b = (GSIZE / (thnb/2)) * thid;
        int loop_e = (GSIZE / (thnb/2)) * (thid+1);

        for(j=0; j< REPET; j++)
        {
            for(i=loop_b; i<loop_e; i++)
            {
                r[i] = a[i] + b[i];
            }
        }
    }
}
```

```
[jaegerj@inti6200 bandwidth numa] $ ./bandwidth_test_local_milan_128th.pgr
thread allocated == 0
time for r[i]=a[i]+b[i] with allocation by thread 0 is: 24s 210515us
[jaegerj@inti6200 bandwidth numa] $ ./bandwidth_test_distant_milan_128th.pgr
thread allocated == 64
time for r[i]=a[i]+b[i] with allocation by thread 64 is 33s 766941us
```

- ❖ 1 socket access either local or distant



Memory bandwidth example (64th milan)

```
#pragma omp parallel
{
    int i;
    int thid = -1;

    thid = omp_get_thread_num();

    if(thid == THREAD_ALLOC)
    {
        printf("thread allocated  == %d\n", thid);
        for(i=0; i<GSIZE; i++)
        {
            a[i] = i+1;
            b[i] = GSIZE-i-1;
            r[i] = 0;
        }
    }

    struct timeval tv_start;
    struct timeval tv_stop;

    gettimeofday(&tv_start, NULL);

#pragma omp parallel
```

```
#pragma omp parallel
{
    int i;
    int j;
    int thid = -1;
    thid = omp_get_thread_num();

    if(thid < (thnb/2))
    {

        int loop_b = (GSIZE / (thnb/2)) * thid;
        int loop_e = (GSIZE / (thnb/2)) * (thid+1);

        for(j=0; j< REPET; j++)
        {
            for(i=loop_b; i<loop_e; i++)
            {
                r[i] = a[i] + b[i];
            }
        }
    }
}
```

```
[jaegerj@inti6200 bandwidth_numa] $ ./bandwidth_test_all_local_and_distant_mila
thread allocated first arrays== 0
thread allocated second arrays == 64
time for r[i]=a[i]+b[i] with all allocations local is: 24s 122235us
time for r[i]=a[i]+b[i] with all allocations distant (cross) is: 38s 115657us
```

- ❖ Each thread of both sockets access either local or distant memory
- ❖ Local = same performance



Memory bandwidth example (64th milan)

```
#pragma omp parallel
{
    int i;
    int thid = -1;

    thid = omp_get_thread_num();

    if(thid == THREAD_ALLOC)
    {
        printf("thread allocated  == %d\n", thid);
        for(i=0; i<GSIZE; i++)
        {
            a[i] = i+1;
            b[i] = GSIZE-i-1;
            r[i] = 0;
        }
    }

    struct timeval tv_start;
    struct timeval tv_stop;

    gettimeofday(&tv_start, NULL);

#pragma omp parallel
```

```
#pragma omp parallel
{
    int i;
    int j;
    int thid = -1;
    thid = omp_get_thread_num();

    if(thid < (thnb/2))
    {

        int loop_b = (GSIZE / (thnb/2)) * thid;
        int loop_e = (GSIZE / (thnb/2)) * (thid+1);

        for(j=0; j< REPET; j++)
        {
            for(i=loop_b; i<loop_e; i++)
            {
                r[i] = a[i] + b[i];
            }
        }
    }
}
```

```
[jaegerj@inti6200 bandwidth_numa] $ ./bandwidth_test_all_local_and_distant_mila
thread allocated first arrays== 0
thread allocated second arrays == 64
time for r[i]=a[i]+b[i] with all allocations local is: 24s 122225us
time for r[i]=a[i]+b[i] with all allocations distant (cross) is: 38s 115657us
```

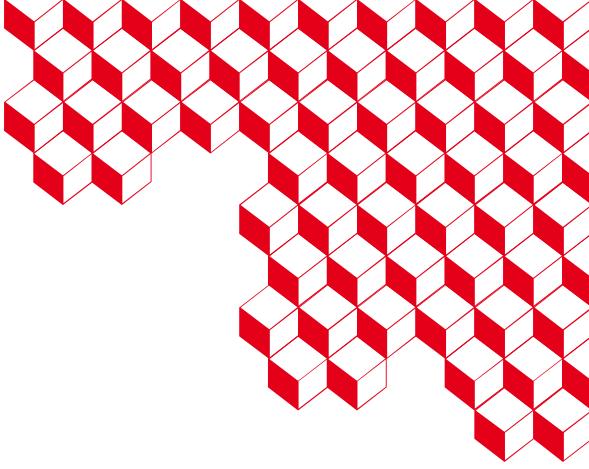
- ❖ Each thread of both sockets access either local or distant memory
- ❖ Distant = 5s slower



Conclusion on memory bandwidth

- ❖ It is necessary to fix the affinity between data and threads
- ❖ The system scheduler may move threads, but not the associated data

- ❖ To avoid this problem:
 - ❖ The user needs to know the data locality
 - ❖ The user must bind the threads to the cores



NUIOA, What is that ?



NUMA and IO

- ❖ NUMA can impact all IOs (NUIOAs)
 - Communications through the network
 - GPGPU
 - Disk access
 - ...
- ❖ Phenomenon often ignored
- ❖ But effects more and more visibles



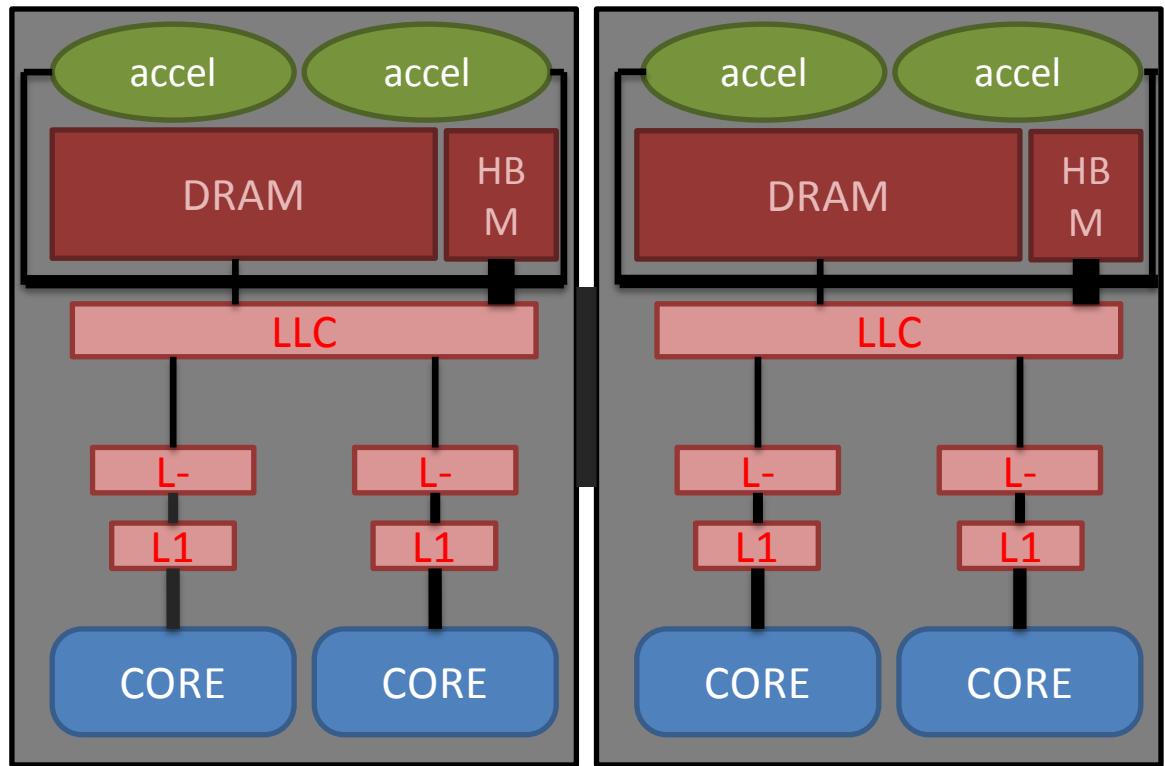
NUMA and IO

- ❖ NUMA can impact all IOs (NUIOAs)
 - Communications through the network
 - GPGPU
 - Disk access
 - ...
- ❖ Phenomenon often ignored
- ❖ But effects more and more visibles



NUMA and IO

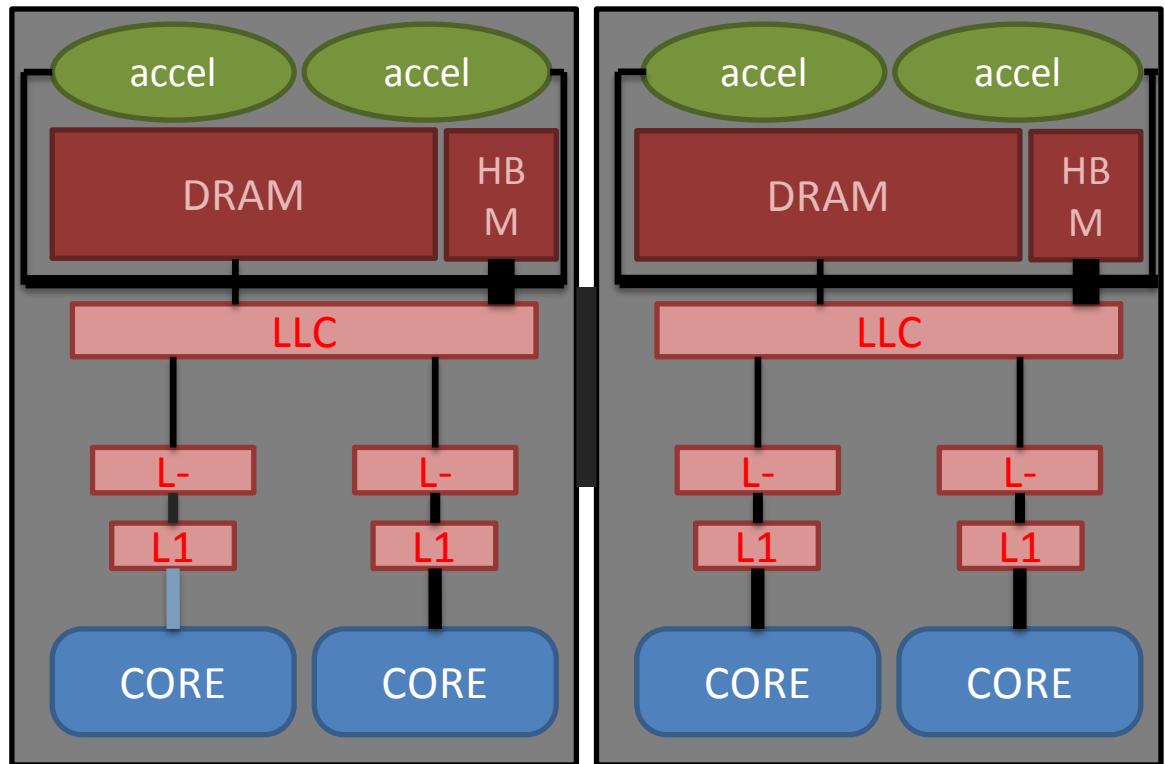
- ❖ Core0 accessing accelerator 0





NUMA and IO

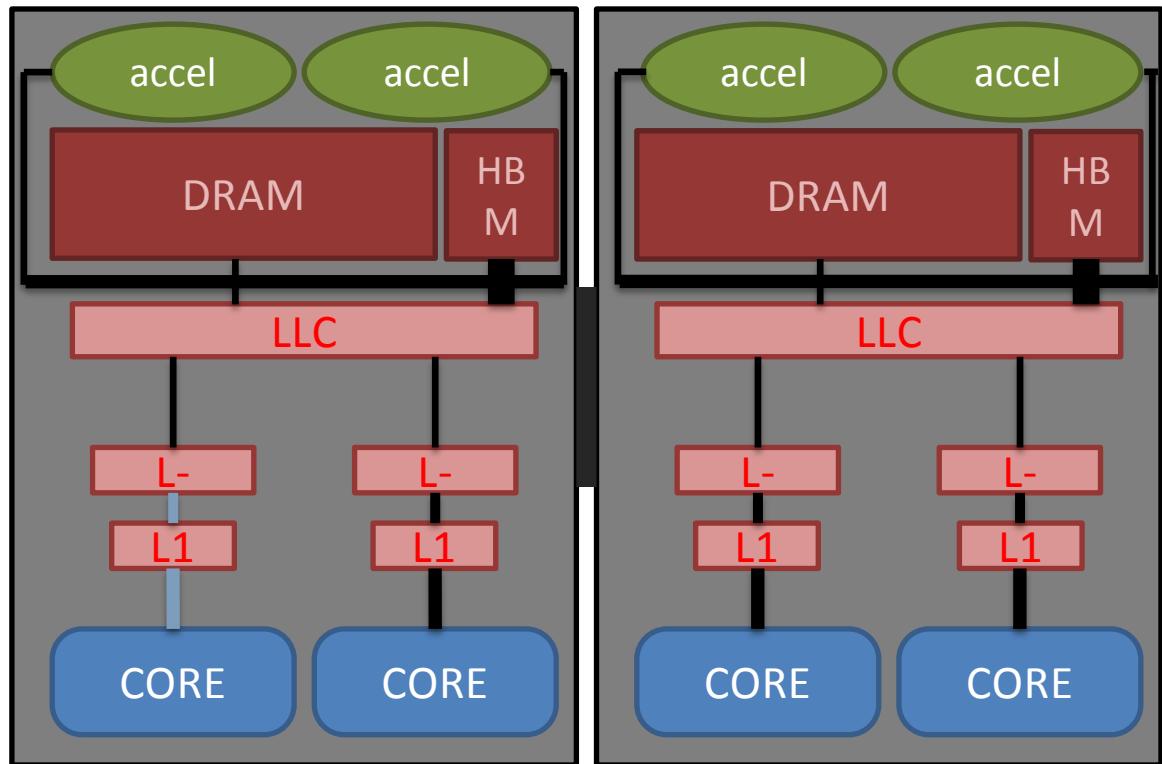
- ❖ Core0 accessing accelerator 0





NUMA and IO

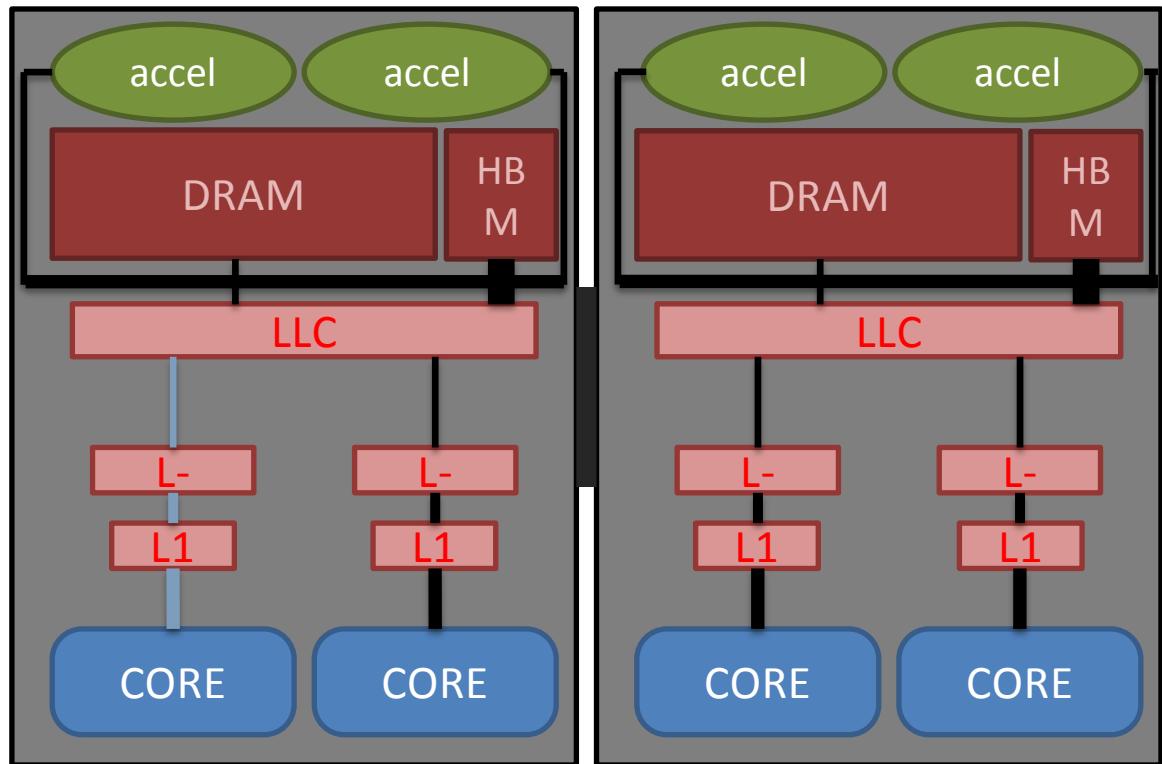
- ❖ Core0 accessing accelerator 0





NUMA and IO

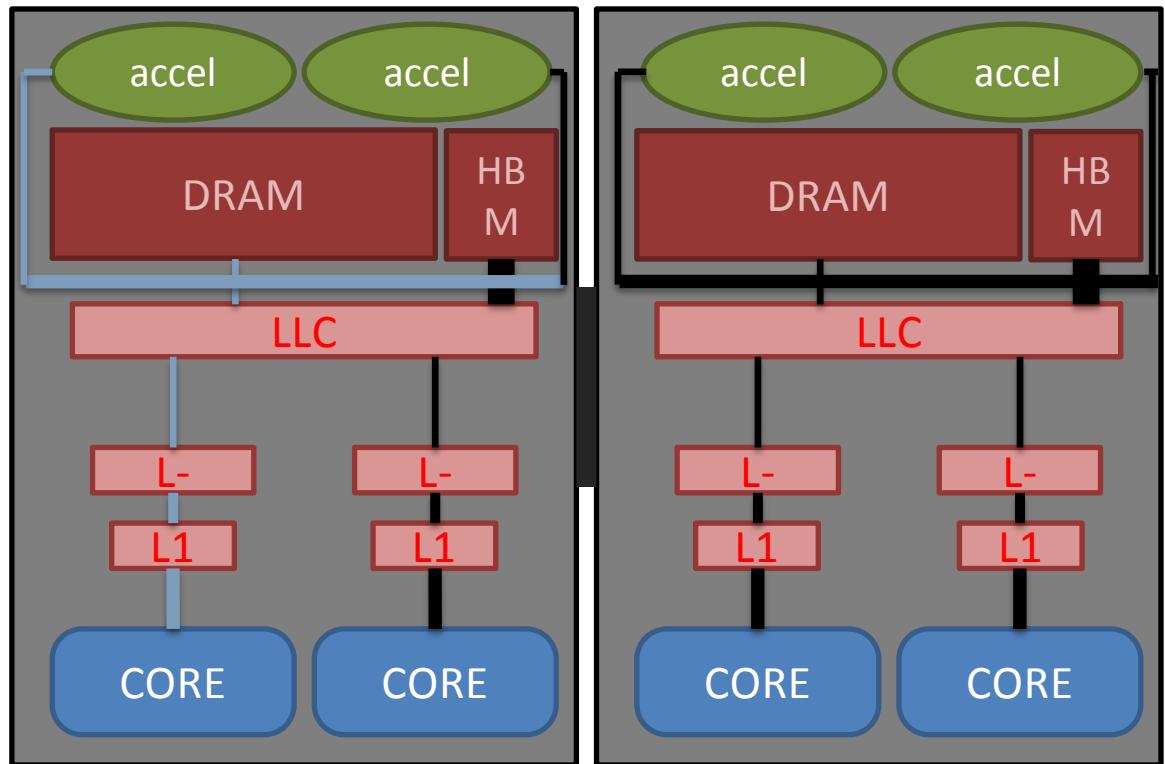
- ❖ Core0 accessing accelerator 0





NUMA and IO

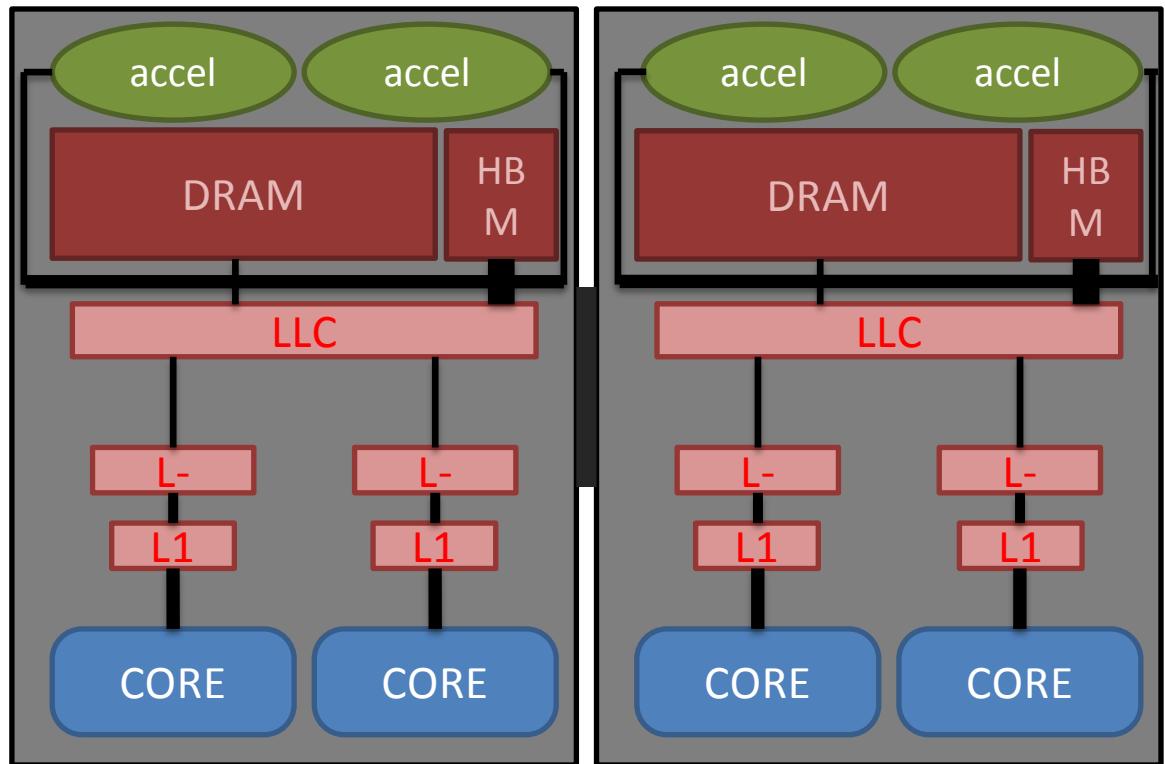
- ❖ Core0 accessing accelerator 0





NUMA and IO

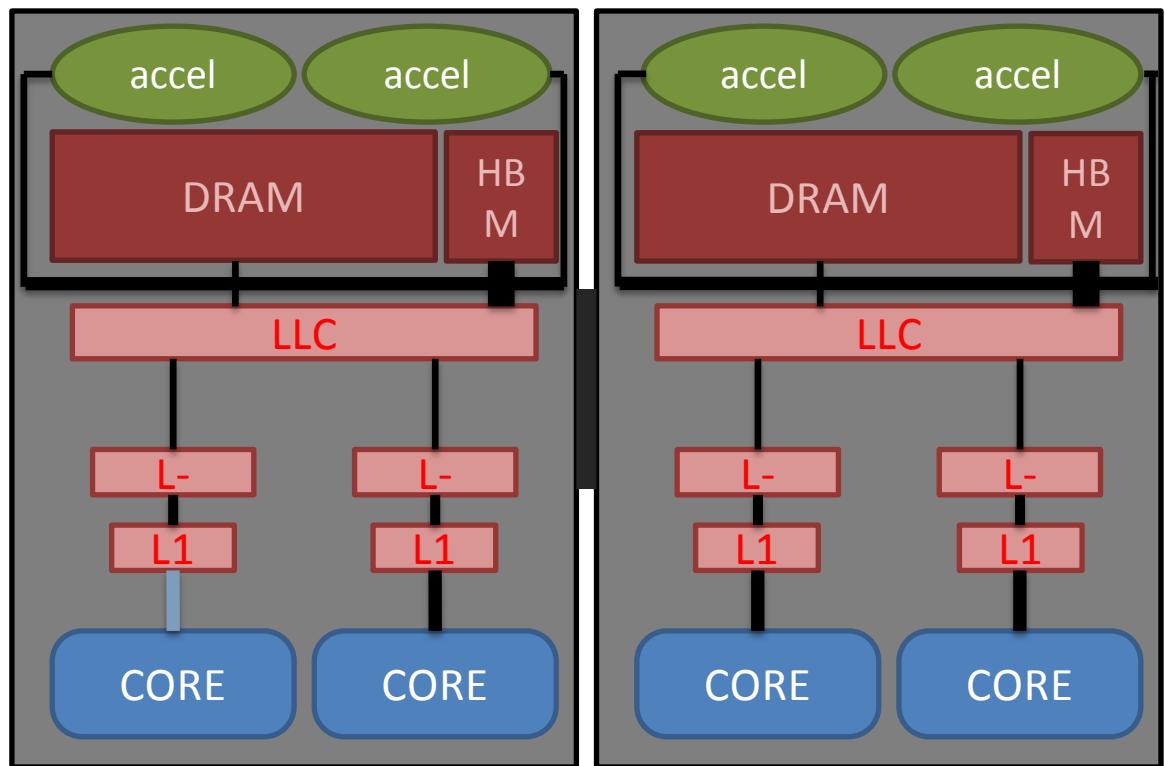
- ❖ Core0 accessing accelerator 2





NUMA and IO

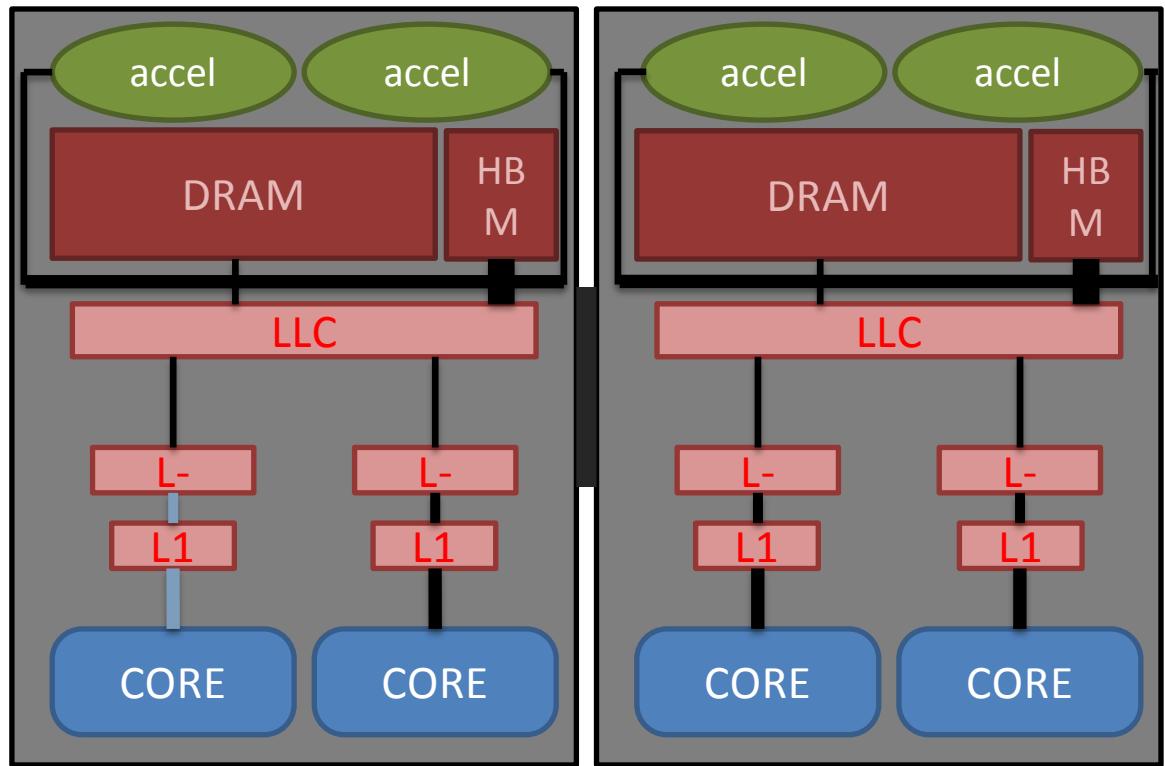
- ❖ Core0 accessing accelerator 2





NUMA and IO

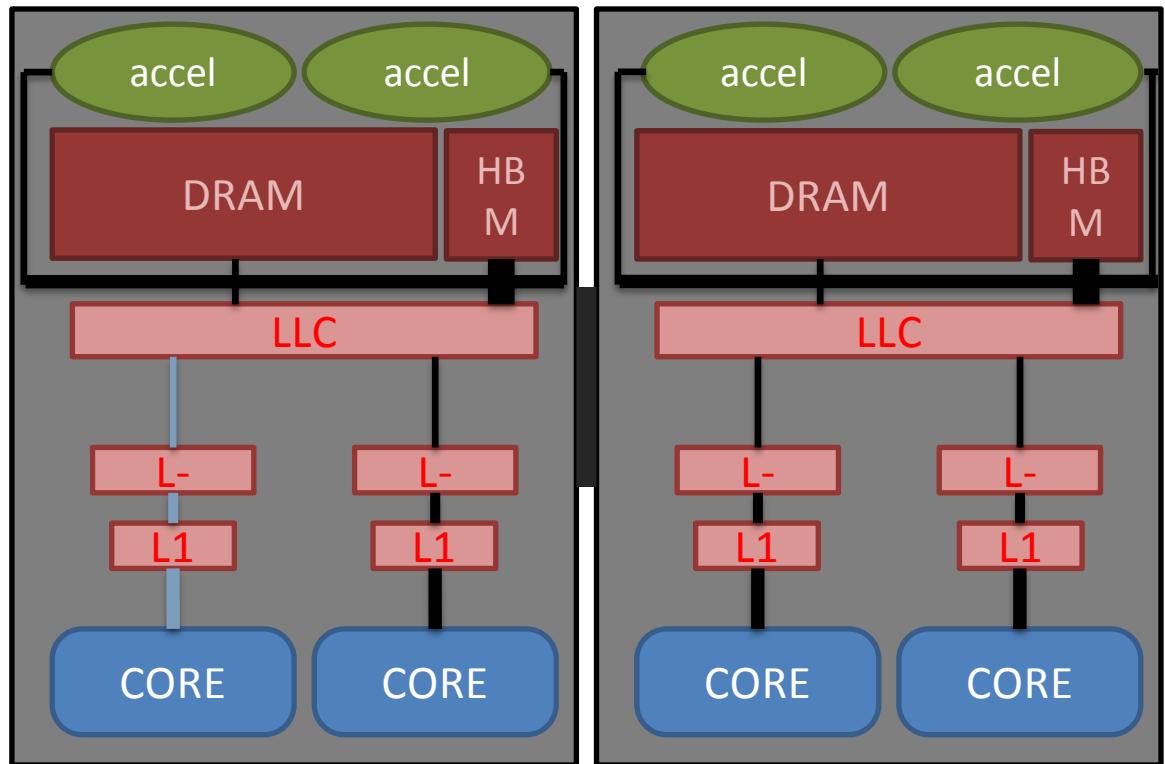
- ❖ Core0 accessing accelerator 2





NUMA and IO

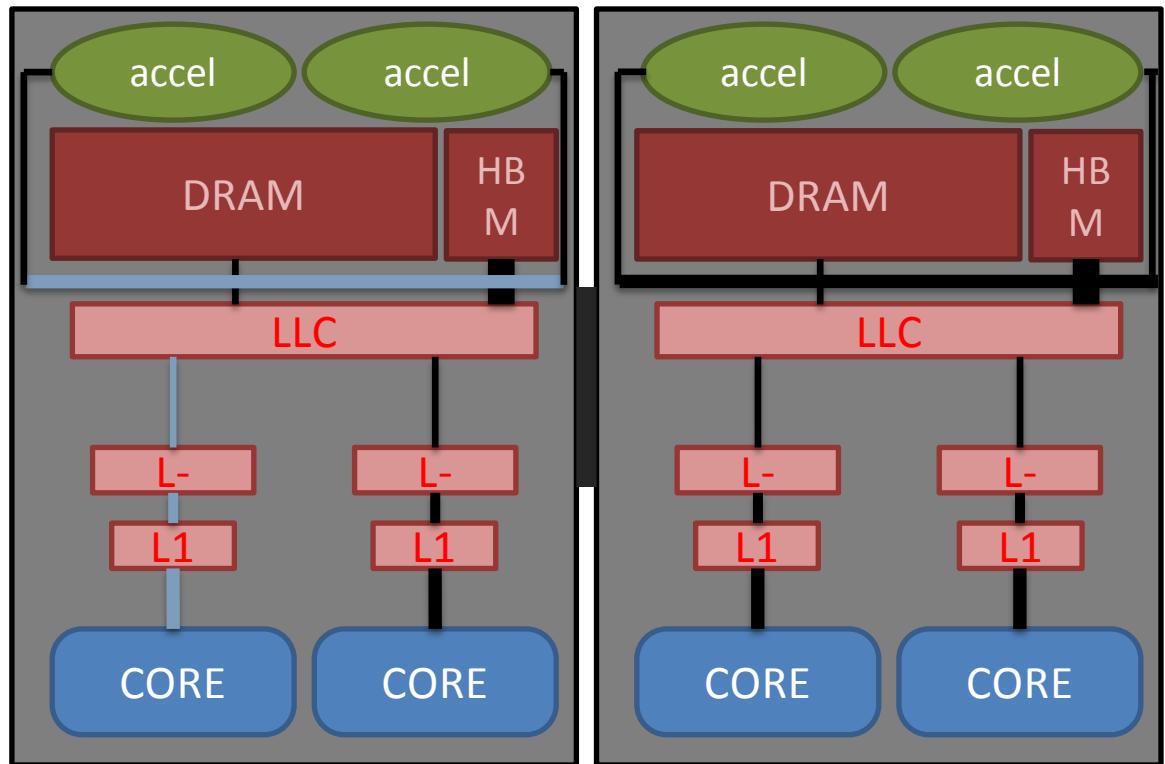
- ❖ Core0 accessing accelerator 2





NUMA and IO

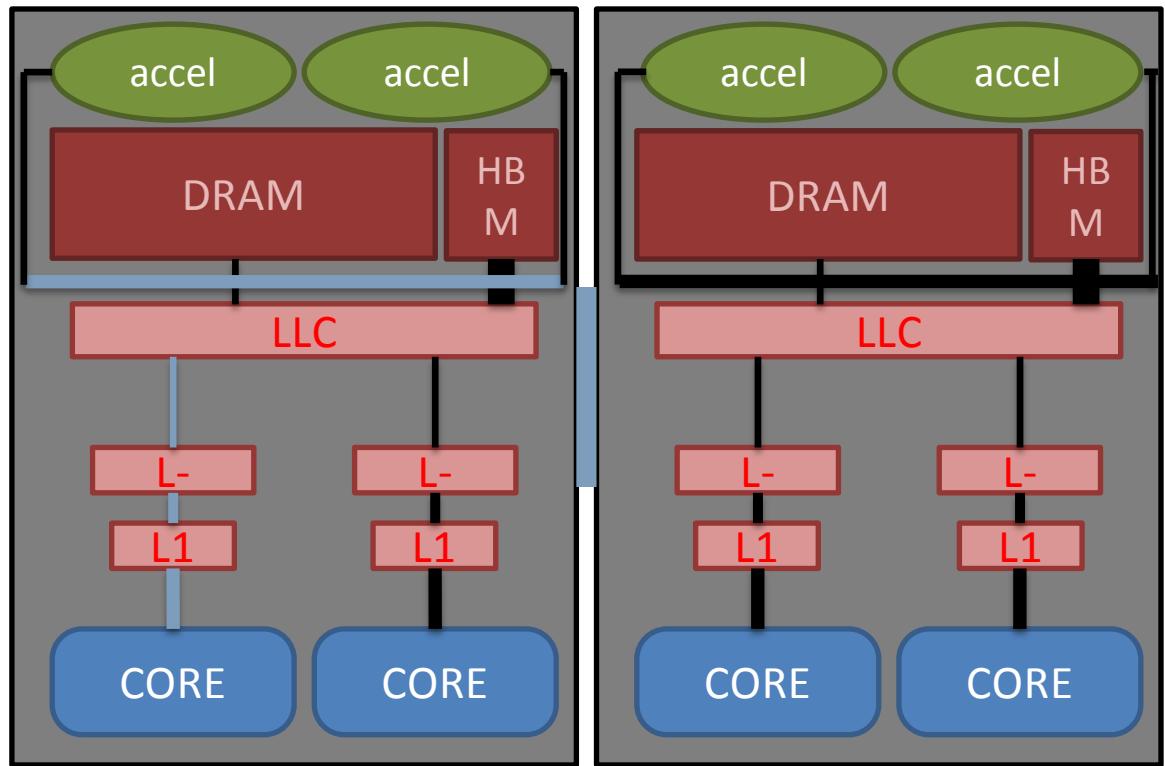
- ❖ Core0 accessing accelerator 2





NUMA and IO

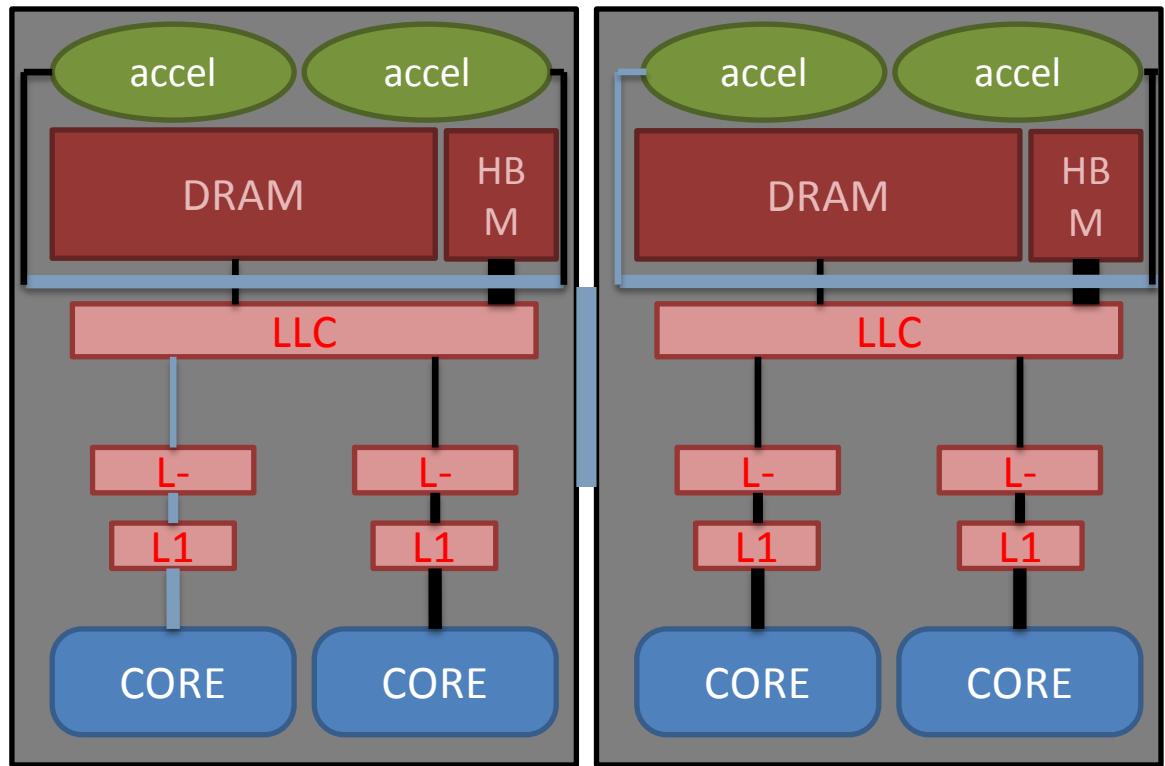
- ❖ Core0 accessing accelerator 2





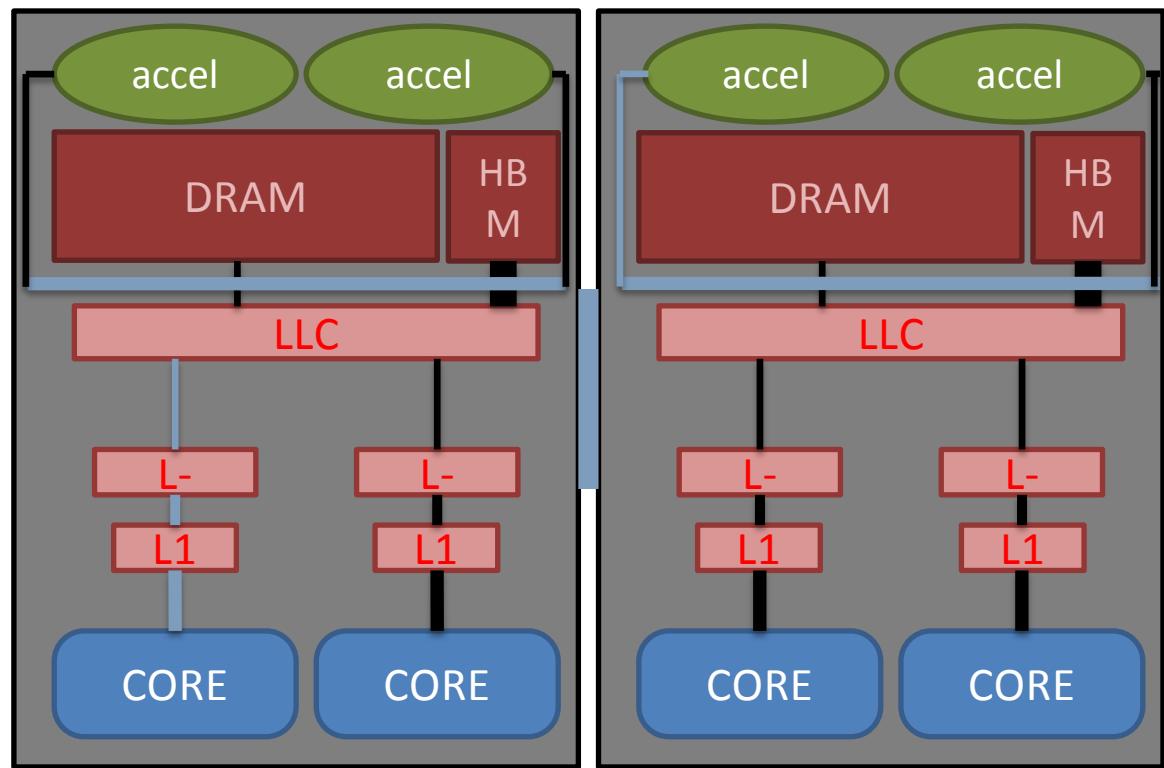
NUMA and IO

- ❖ Core0 accessing accelerator 2





NUMA and IO



- ❖ Core0 accessing accelerator 2
- ❖ Affect all kind of devices attached to the socket
 - ❖ Including network card



NUMA and IO

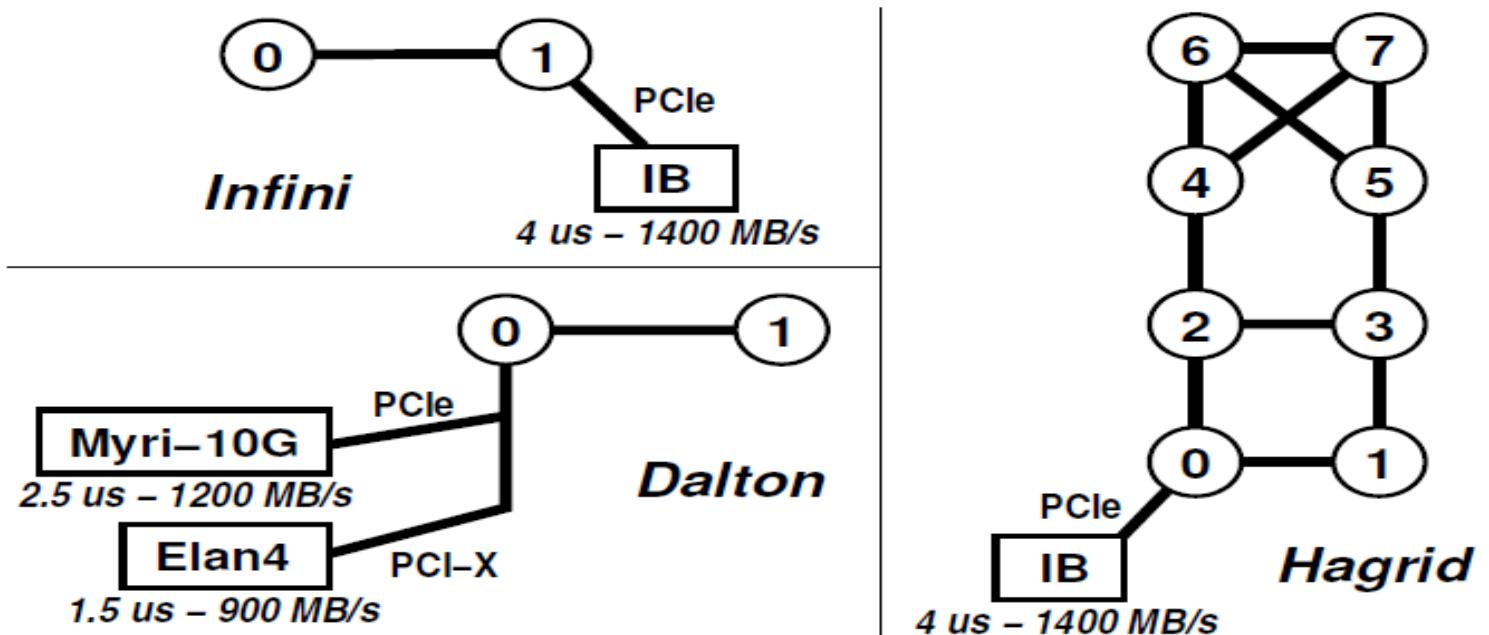


Figure 2. Topology of the experimentation platforms based on 2-socket and 8-socket dual-core OPTERON.



NUMA and IO

	Local Node	Distant Node	Overhead
MYRI-10G	1739	1794	55
ELAN4	1610	1670	60
INFINIBAND	464	559	95

Table 1. Impact of NUMA placement on small request round-trip latency (in nanoseconds) with various high-speed interconnect interfaces on 2-socket machines.



NUMA and IO

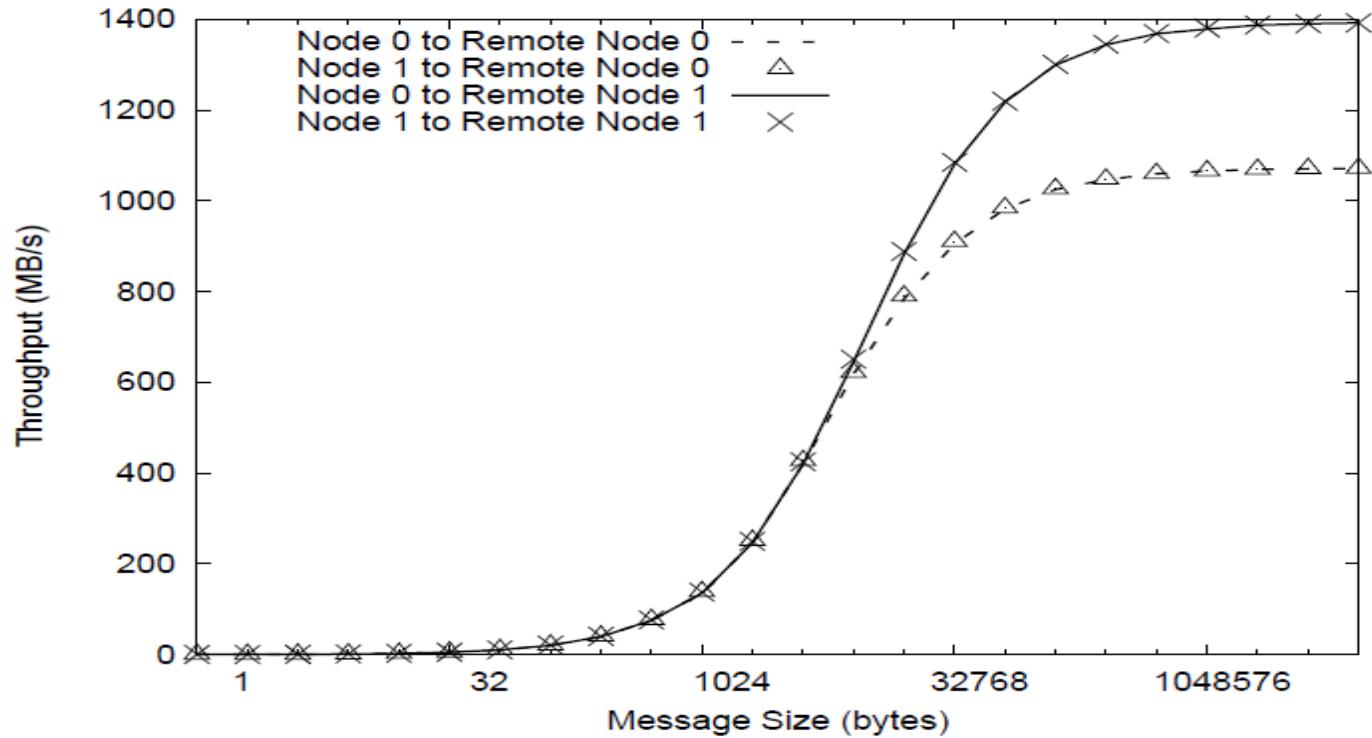
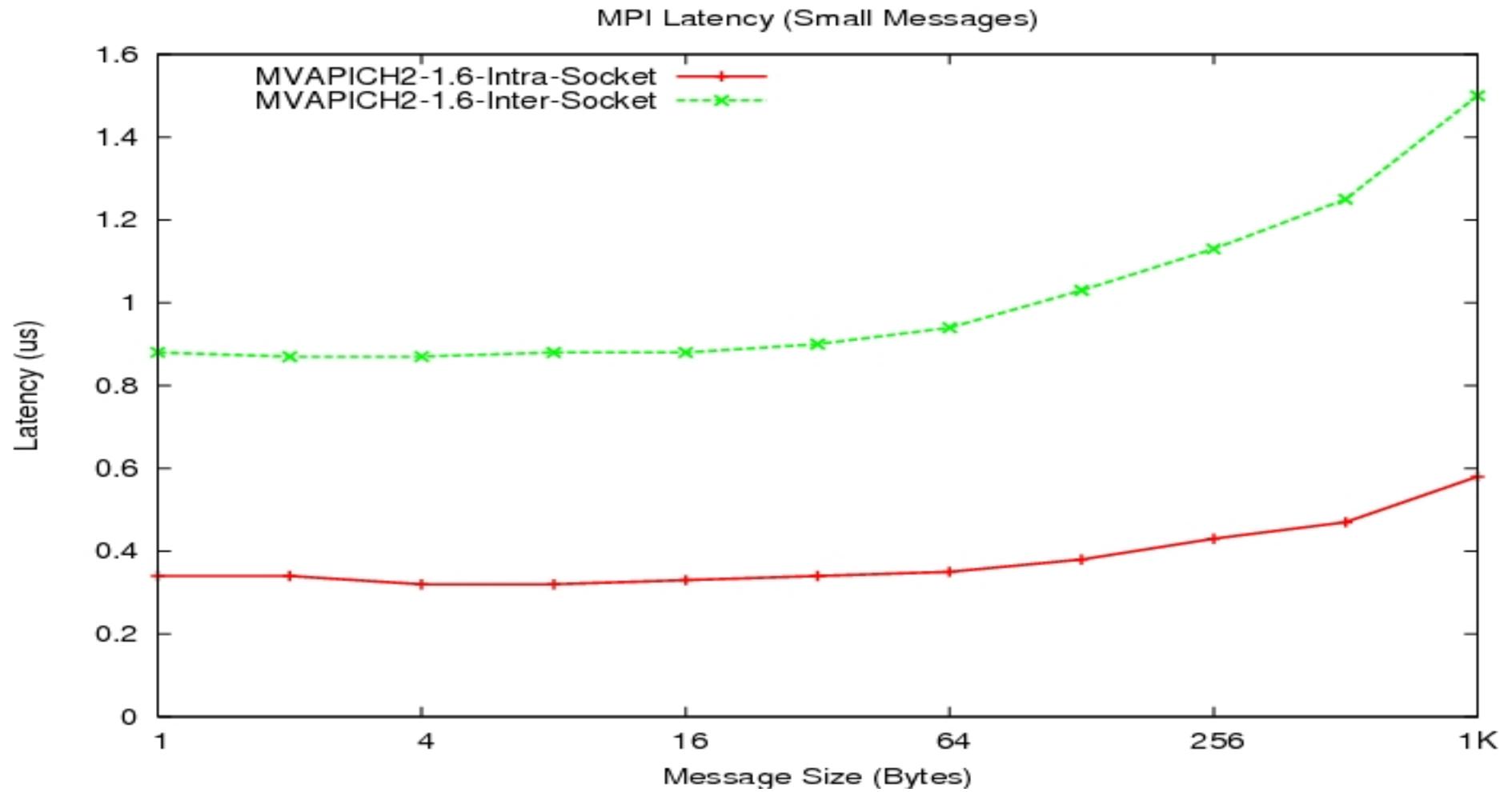


Figure 4. Performance of RDMA Write with OPENIB VERBS on 2-socket machines depending on the placement on both machines.



Latency evaluation



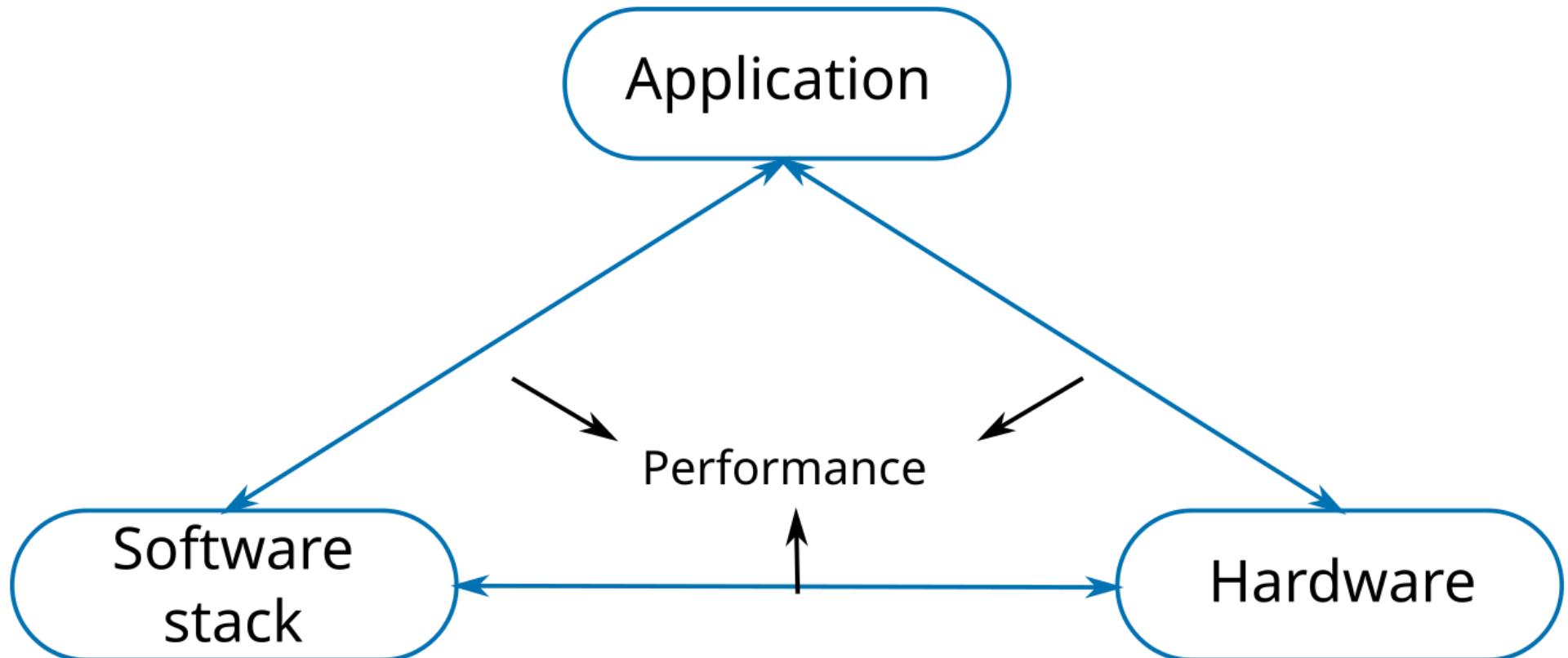


4. The Roofline model



Performance of application

Performance comes from the complex interactions between these three aspects



As such, performance optimization requires to understand these interactions

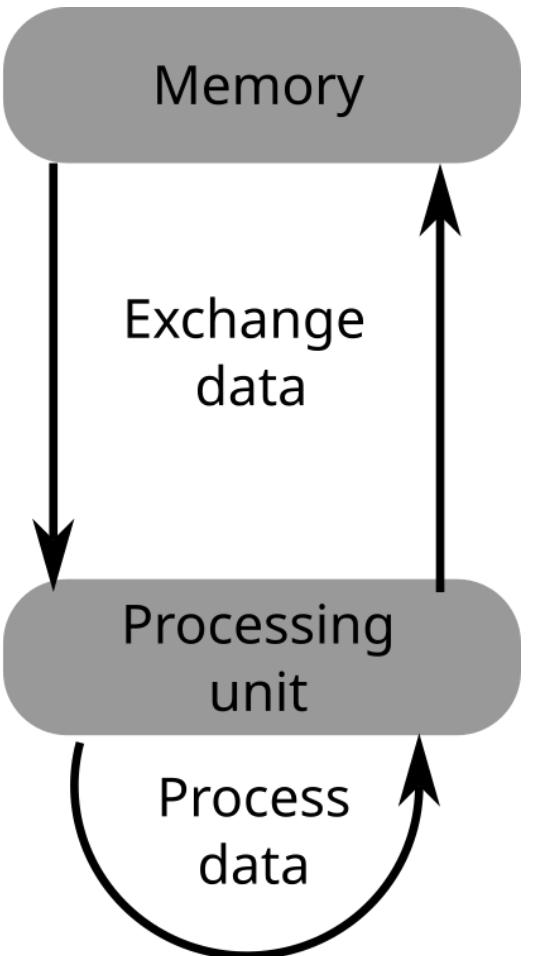


Performance analysis

- ❖ Performance analysis tools and models are used to help understand these interactions and direct optimization efforts
 - ❖ One such model is the **Roofline Model**, theorized by S. Williams et al. in [1].
 - ❖ Simple but versatile and visual model
 - ❖ Used in many analysis suites such as Intel Advisor or NSight Compute
-
- ❖ [1] : Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: an insightful visual performance model for multicore architectures*. Commun. ACM 52, 4 (April 2009), 65–76. <https://doi.org/10.1145/1498765.1498785>

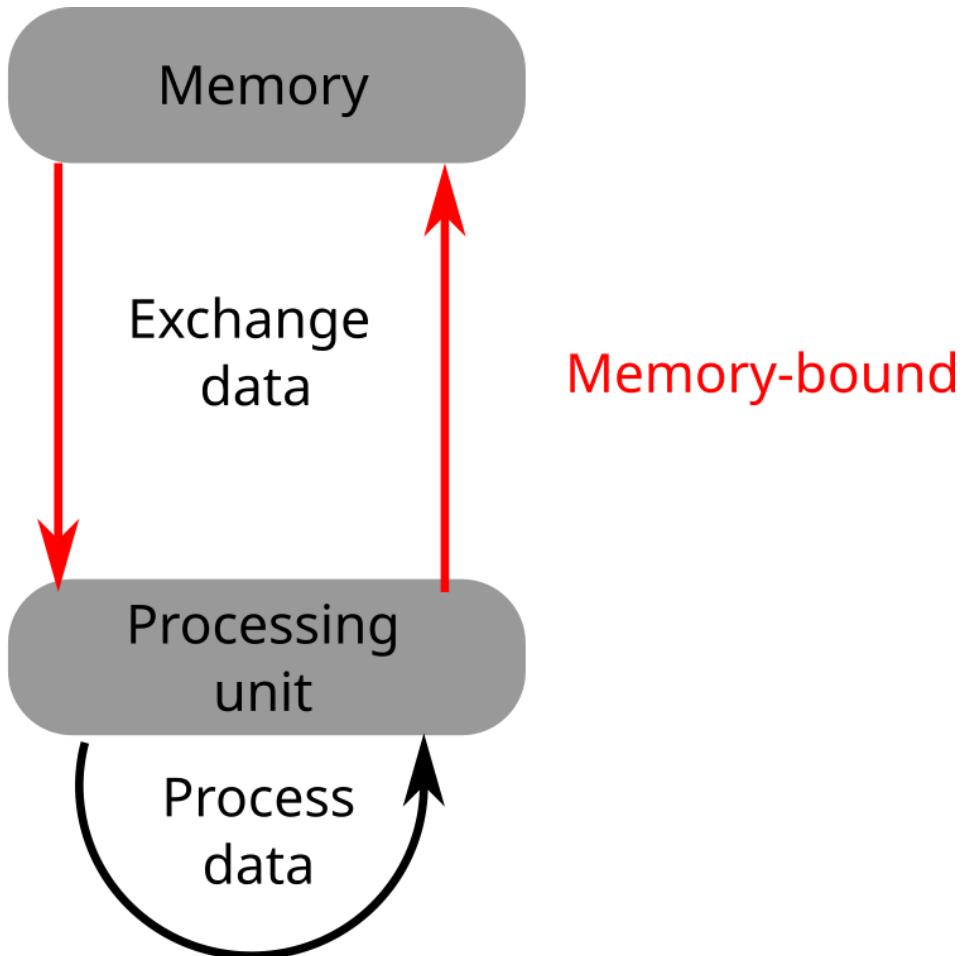


The Roofline Model



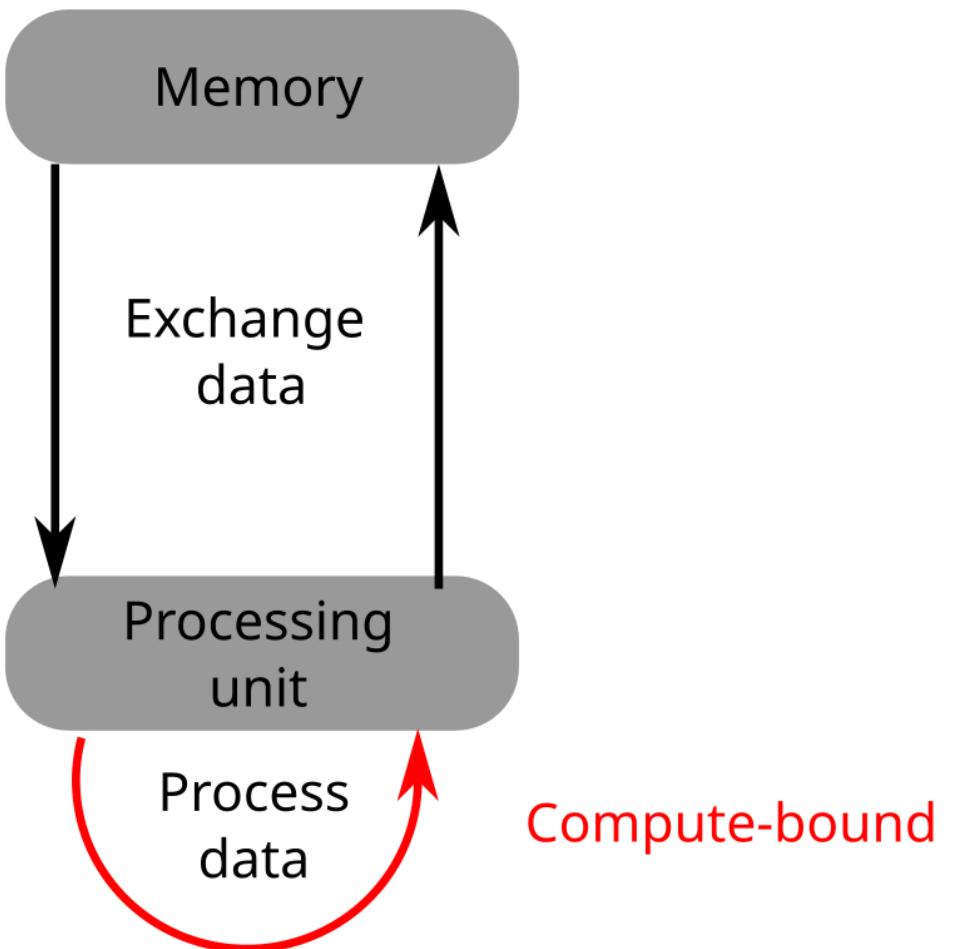


The Roofline Model





The Roofline Model





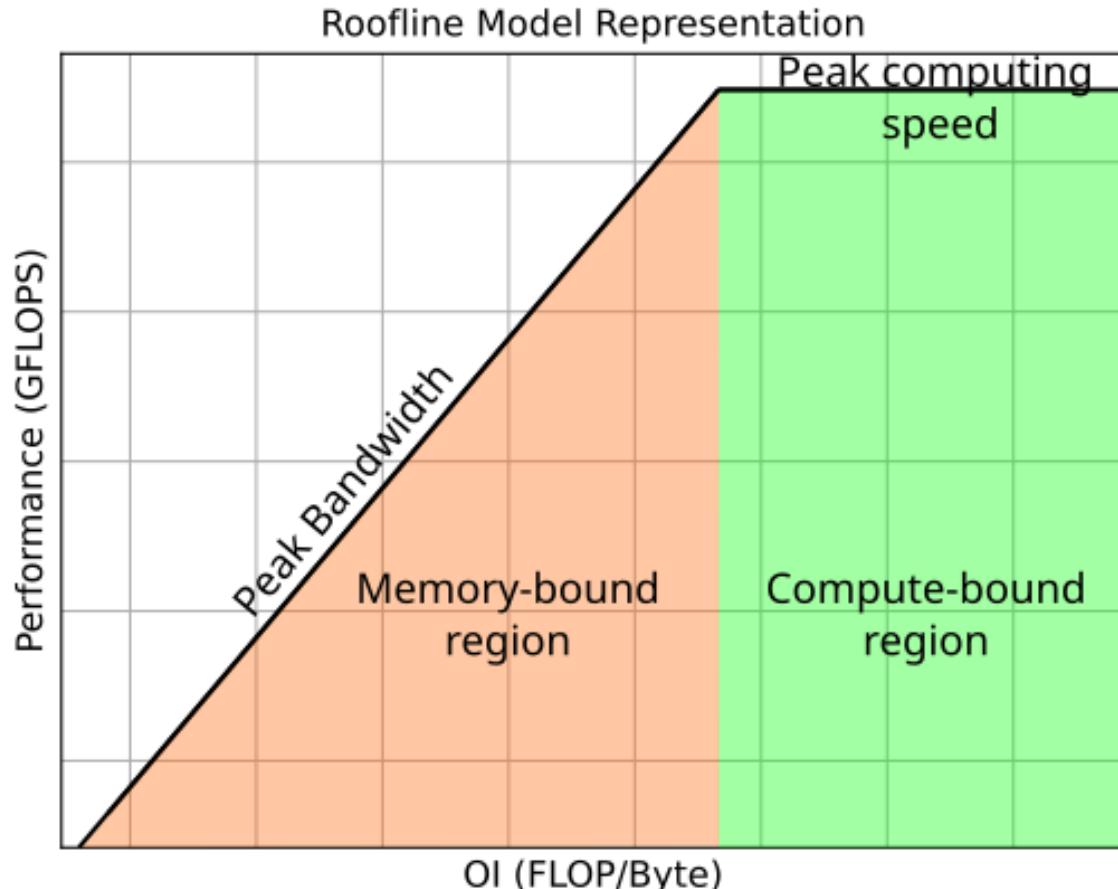
The Roofline Model

- ❖ Computation can only be done if:
 - Operands are available
 - Result can be written
- ❖ Memory bandwidth limits overall performances
- ❖ Operational Intensity is defined as:

OI → Number of operation / Memory needed to these operations



The Roofline Model



$$\text{roofline(OI)} = \min(\text{Peak Bandwidth} \times \text{OI}, \text{Peak computing speed})$$

- ❖ According to its Operational Intensity (Ratio of operations per bytes moved), one program is either **memory-bound** or **compute-bound**.



The Roofline Model

- ❖ Pour aller plus loin, je vous conseille l'excellent cours de Thierry Dumont:
- ❖ http://lyoncalcul.univ-lyon1.fr/ed/DOCS_2015-2016/intensite.pdf



5. Tools



Linux perf tool

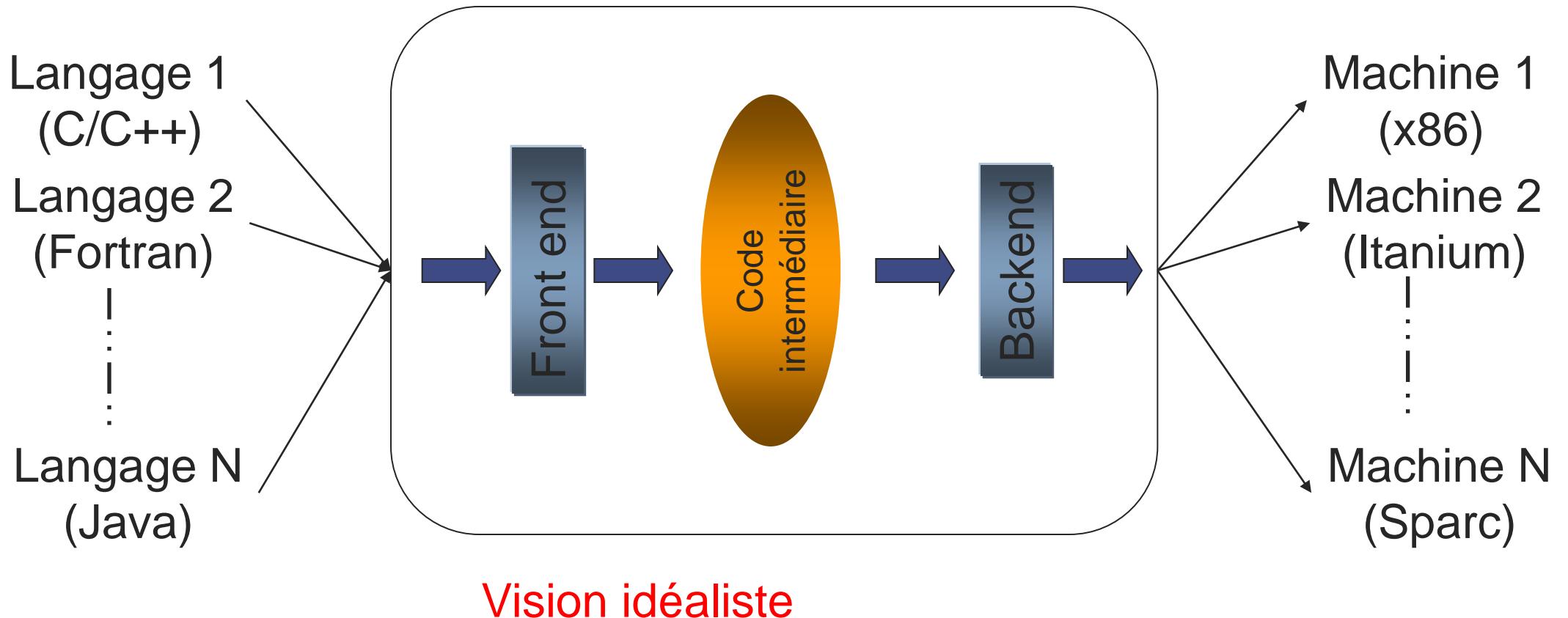
- ❖ Linux perf + KDAB/hotspot
- ❖ Gprof
- ❖ Tau
- ❖ Scalasca and scorep
 - ❖ <https://www.youtube.com/watch?v=3HtR89L2u9Q>
- ❖ Maqao
- ❖ ILP <https://www.uops.info/>
- ❖ godbolt

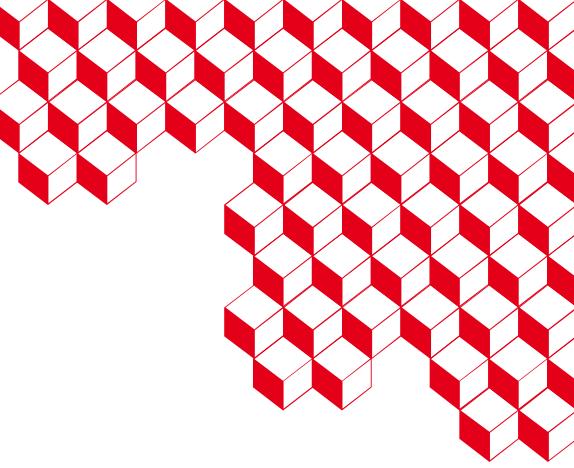


6. Compilers



Compilateur standard

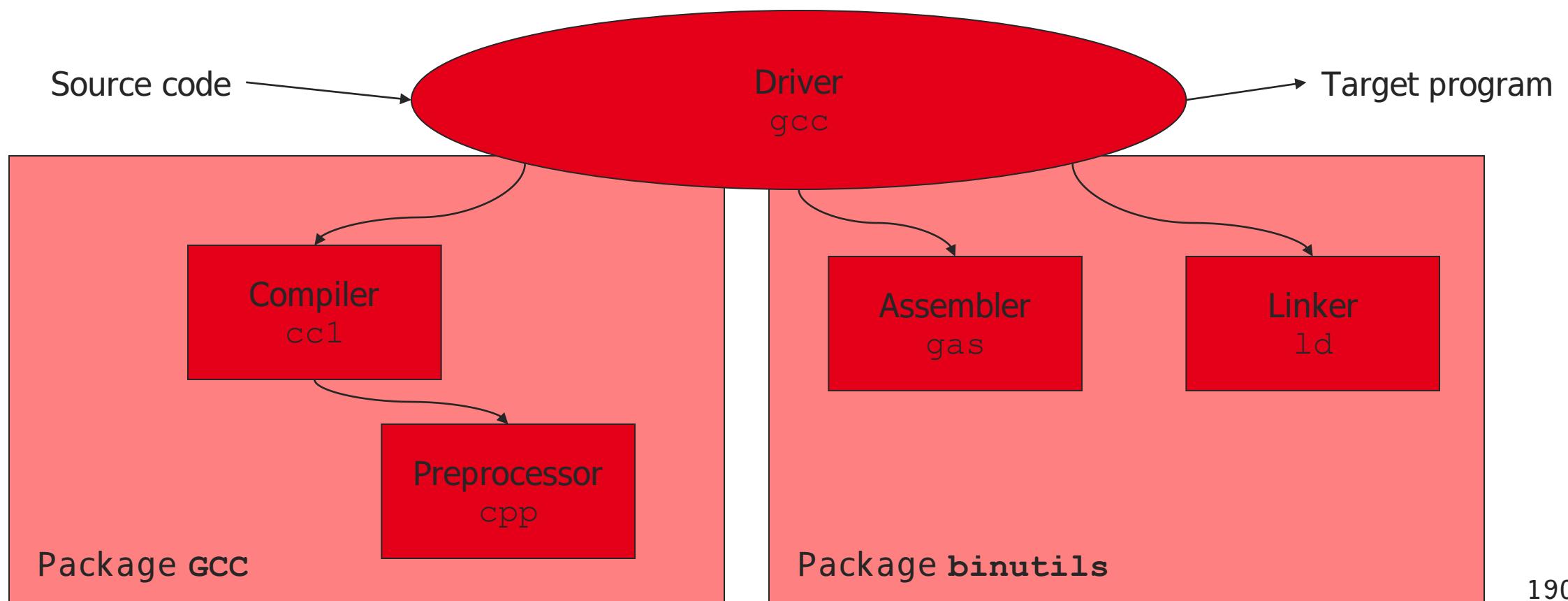




GCC

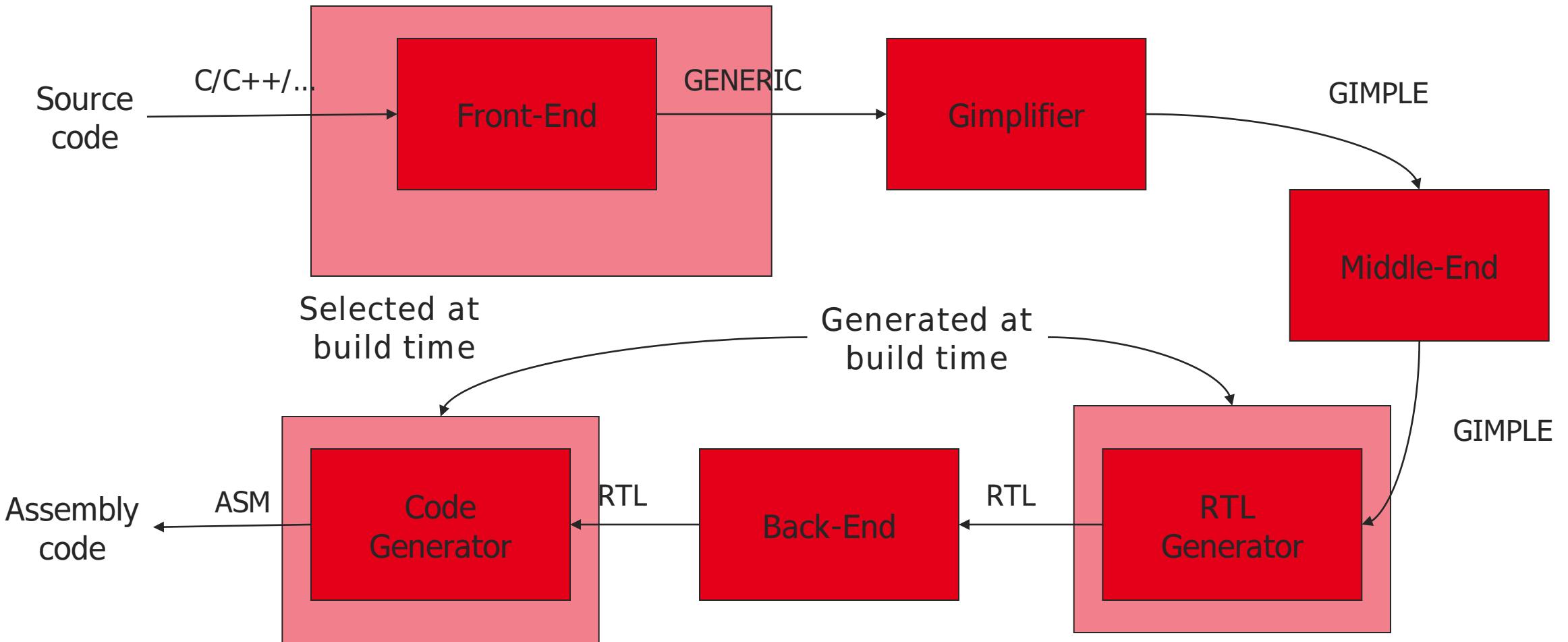


Architecture de GCC





GCC Architecture





Préprocessor

- ❖ CPP : Gestion des directives de précompilation
- ❖ Syntaxe des directives
 - #keyword
- ❖ Exemple de directives
 - #ifdef
 - #include
 - #warning
 - #error
- ❖ Explosion de la taille du code après *preprocessing*
- ❖ Attention #pragma n'est pas traité par le préprocesseur



Front-end

- ❖ Lecture du fichier source en entrée
 - C, C++, Fortran, Java, C#, ...
- ❖ Vérification de la validité du code
 - Analyse lexicale
 - Analyse syntaxique
 - Analyse sémantique
 - Cf. CPA cours 1
- ❖ Chaque front-end est dans un répertoire différent :
 - C, ObjectiveC → \${SOURCE}/gcc/c/, \${SOURCE}/gcc/c-family/
 - C++ → \${SOURCE}/gcc/cp/, \${SOURCE}/gcc/c-family/
 - Fortran → \${SOURCE}/gcc/fortran/
- ❖ En sortie, le code est représenté en GENERIC
 - Sauf pour C/C++ qui génère directement du GIMPLE



GENERIC

- ❖ Représentation intermédiaire sous forme d'arbre
- ❖ Indépendant du langage source
- ❖ Processus de création d'une représentation GENERIC
 - Génération de l'arbre de syntaxe abstraite par le *parser*
 - Le parser peut garder cette représentation
 - Suppression des constructions spécifiques au langage
 - Emission de l'arbre GENERIC à la fin de la phase de *parsing*
- ❖ Tous les noeuds sont définis dans \$ (SOURCE) /gcc/tree.def
 - Notion de *tree codes*



Middle-end

- ❖ Optimisation haut niveau
 - Indépendante de l'architecture
- ❖ Granularités
 - Optimisation par fonction
 - Optimisation par boucle
 - Optimisation inter-procédurale
- ❖ Ordre des transformations géré par le *pass manager* de GCC
- ❖ Travail sur une représentation intermédiaire nommée GIMPLE
 - En conjonction avec d'autre RI (par exemple CFG)
 - Détails dans le prochain cours



GIMPLE

- ❖ Représentation intermédiaire de haut niveau
 - Introduite dans GCC 4.4
 - Basée sur une représentation avec un arbre
 - Nœud avec une sémantique
- ❖ Sous-ensemble simplifié de GENERIC
 - Représentation 3-adiresses
 - Aplatissement du flot de contrôle
 - Simplifications et nettoyage (la grammaire est restreinte)
 - Transformation de GENERIC vers GIMPLE
 - `gimplify_function_tree()` dans le fichier `gimplify.c`
- ❖ Deux niveaux de GIMPLE
 - *High GIMPLE*
 - *Low GIMPLE*



GIMPLE – Exemple en C

- ❖ Exemple simple
 - Langage C
 - Une seule fonction main

- ❖ Compilation avec sortie des fichiers intermédiaires :
 - gcc -fdump-tree-all test.c
 - Génération de la représentation GIMPLE entre les transformations

```
int main() {  
    int x = 10 ;  
    if (x) {  
        int y = 5 ;  
        x = x*y+15 ;  
    }  
}
```



GIMPLE – Exemple en C

Fichier test.c:

```
int main() {  
    int x = 10 ;  
    if (x) {  
        int y = 5 ;  
        x = x*y+15 ;  
    }  
}
```

Fichier test.c.004t.gimple:

```
main() {  
    int D.2720;  
    int x;  
    x = 10 ;  
    if (x!=0) goto <D.2718>;  
    else goto <D.2719>;  
<D.2718>:  
{  
    int y;  
    y=5;  
    D.2720 = x*y;  
    x = D.2720+15  
}  
<D.2719>:  
}
```

- ❖ Déclaration de temporaires
 - D.2720
- ❖ Simplification pour le code 3 adresses
 - D.2720 = x*y
- ❖ Flot de contrôle avec goto



GIMPLE – Exemple en C

❖ Génération du code GIMPLE

■ `gcc -fdump-tree-all-raw test.c`

Fichier `test.c.004t.gimple`:

```
main() {  
    int D.2720;  
  
    int x;  
  
    x = 10 ;  
  
    if (x!=0) goto <D.2718>;  
  
    else goto <D.2719>;  
  
<D.2718>:  
{  
    int y;  
    y=5;  
    D.2720 = x*y;  
    x = D.2720+15  
}  
  
<D.2719>:  
}
```

Fichier `test.c.004t.gimple`:

```
main()  
gimple_bind <  
    int D.2720;  
    int x;  
    gimple_assign<integer_cst,x,10,NULL>  
    gimple_cond <ne_expr, x, 0, <D.2718>, <D.2719> >  
    gimple_label <<D.2718>>  
    gimple_bind <  
        int y;  
        gimple_assign<integer_cst, y, 5, NULL>  
        gimple_assign<mult_expr, D.2720, x, y>  
        gimple_assign<plus_expr,x, D.2720,15>  
    >  
    gimple_label<<D.2719>>  
    >
```

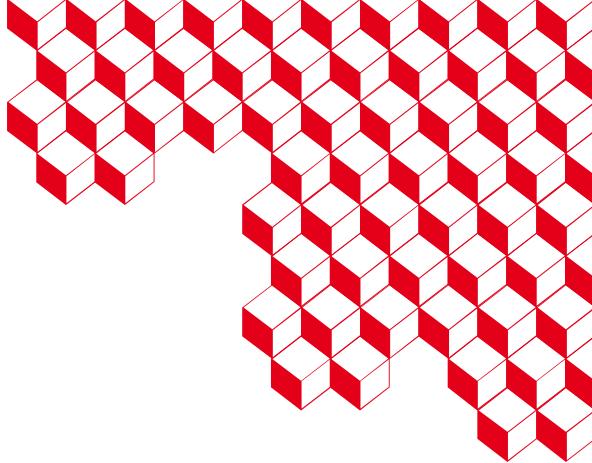


GIMPLE – Exemple en C

Fichier test.c.004t.gimple:

```
main() {  
    int D.2720;  
  
    int x;  
    x = 10 ;  
    if (x!=0) goto <D.2718>;  
    else goto <D.2719>;  
  
<D.2718>:  
{  
    int y;  
    y=5;  
    D.2720 = x*y;  
    x = D.2720+15  
}  
  
<D.2719>:  
}
```

```
Fichier test.c.011t.cfg  
  
main() {  
    int y;  
    int x;  
    int D.2720;  
  
<bb2>:  
    x=10;  
    if (x!=0) goto <bb 3>;  
    else goto <bb 4>;  
  
<bb 3>:  
    y=5;  
    D.2720 = x*y;  
    x=D.2720+15;  
  
<bb 4>:  
    return ;  
}
```



Other compilers?

TBD

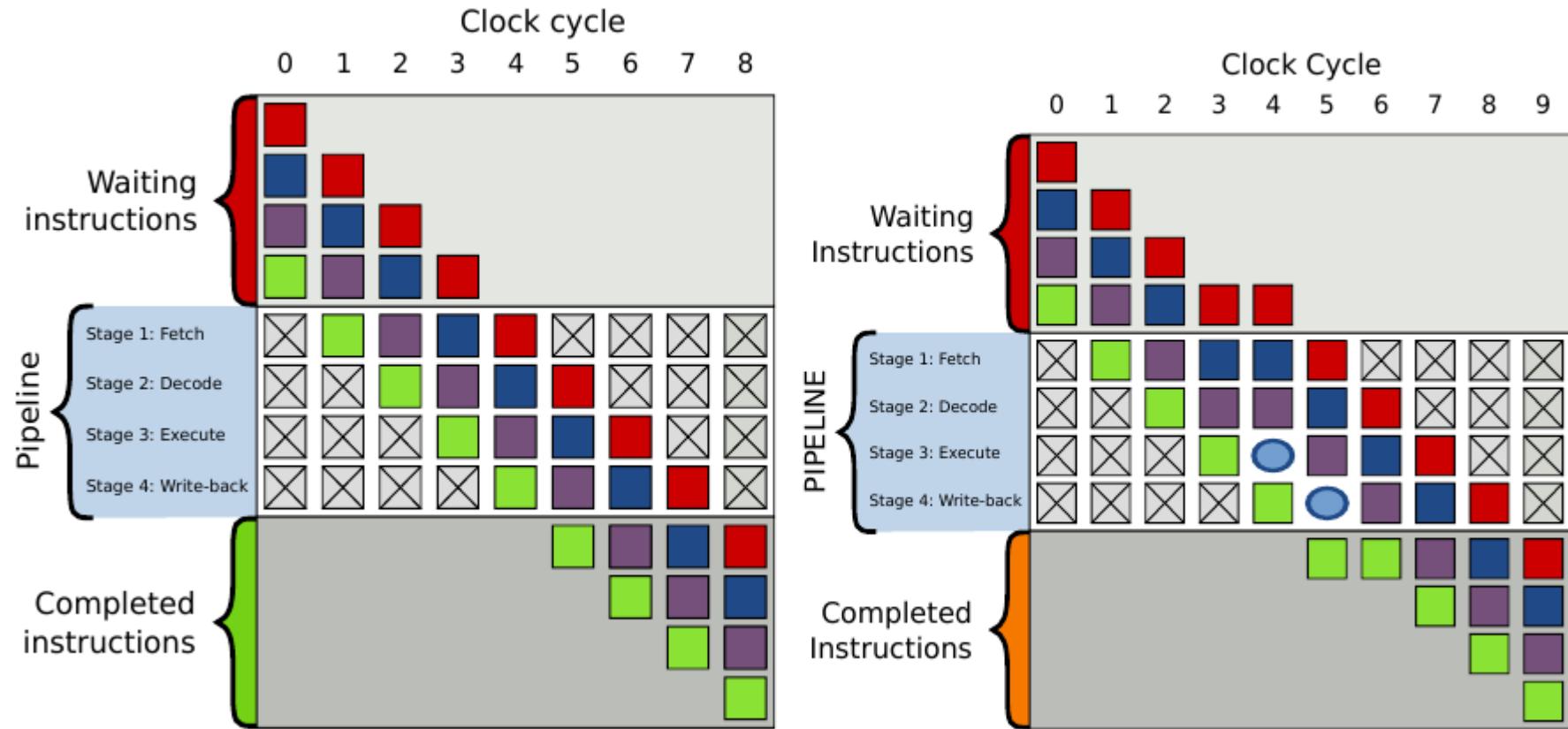


7 ■ ILP

Instruction Level Parallelism



Pipeline





8. ■ Vectorization

FLOPS

FLOPs count of different floating-point instructions on a machine with vector size of 256 bits on double of 64 bits

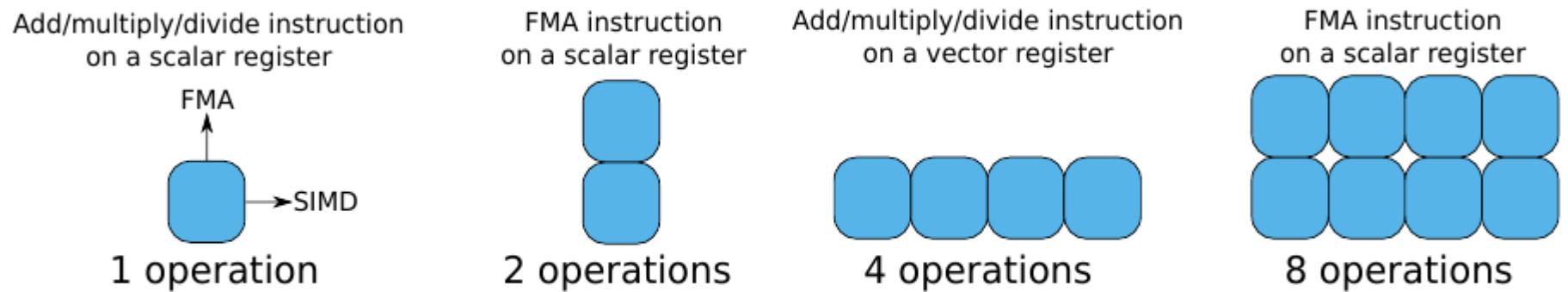


Figure 2.4: Number of FLOPs issued with different type of floating-point instructions



9. MPI



MPI Optimisation

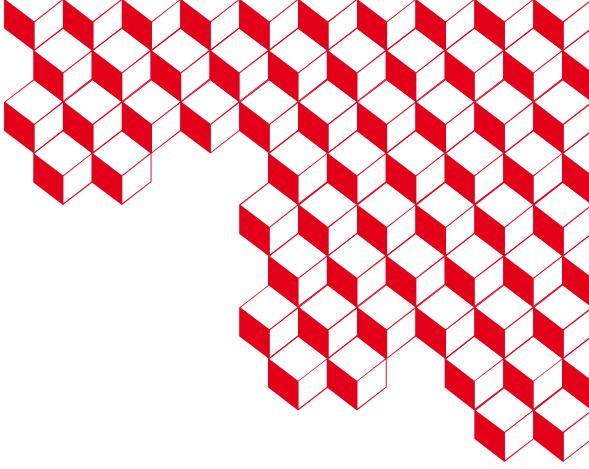
- ❖ Writing and optimizing a parallel program is not trivial
- ❖ A few keys to avoid crippling issues for your parallel performance
 - **Think parallel:**
 - use a parallel efficient algorithm instead of trying to parallelize a sequential algorithm
 - **Follow Amdahl:**
 - Reduce the amount of sequential parts in you programs
 - Avoid synchronizations in your programs
- ❖ And know what you are doing
 - Following the next courses will help you know most traps of parallelism



MPI Optimisation

- ❖ On the user side
 - Choose a scalable algorithm
 - How to decompose data between MPI processes

- ❖ On the MPI implementation side
 - Collective comm. Algorithms
 - Diving inside MPI implementations
 - Overheads to avoid
 - The tricks and options to know



MPI optim.: user side



MPI app. performance

- ❖ Performance of an application using MPI usually strongly depends on the MPI performance
- ❖ MPI performance is driven by the quality of the implementation...
 - *Next part of this lecture*
- ❖ ... and how MPI primitives are called in the application



User needs knowledge

- ❖ To build a performant MPI application, the programmer needs knowledge
 - On the MPI implementation used
 - Knowing strong/weak points of the implementation
 - Knowing specific options to enhance performance
 - On parallel algorithmic
 - Building an efficient parallel algorithm
 - Building a scalable efficient algorithm



Know what you do

- ❖ A small difference in the code can have huge performance influence
- ❖ It is important to know the parallel algorithm/pattern of your communications
- ❖ A good scalability is a trade-off between number for communications and concurrency of these communications



Know what you will do

- ❖ The number of communications and the communication pattern may also depend on the code design
- ❖ It is important to know how to distribute the data among your MPI processes
 - Also depends on what will be sent
 - And to which rank



Domain decomposition

- ❖ When running an MPI application, the computation is distributed among all the MPI processes
 - Hence, the same for the data needed by these computations
- ❖ In domain decomposition, a mesh is distributed among MPI processes
- ❖ Communications occur on the borders of these partitions
 - Halo cells or ghost cells



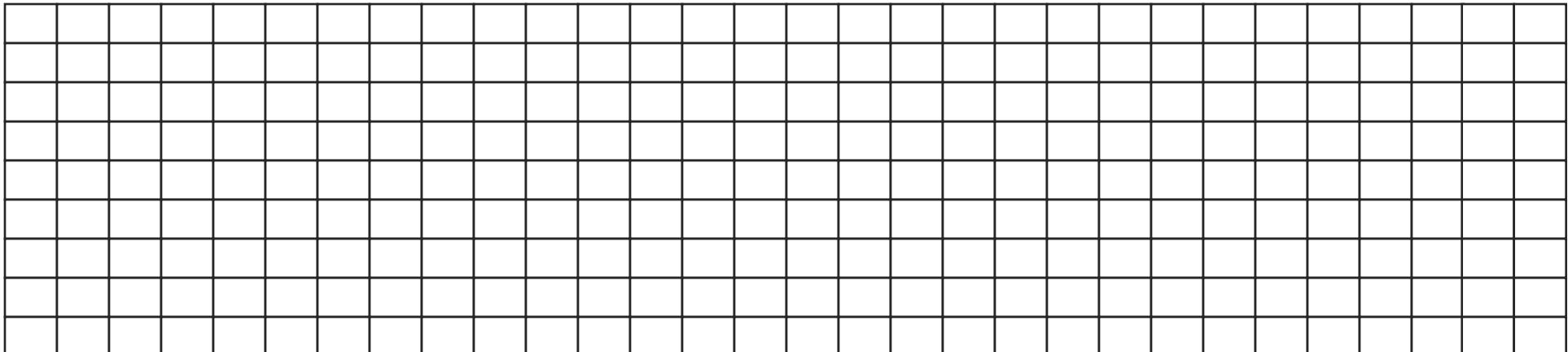
Domain decomposition

- ❖ Let's consider a very simple mesh: a matrix
- ❖ Computations: Stencil and Matrix multiplications
- ❖ The matrices are decomposed in several parts
 - Each sub-computation is done on different MPI processes
- ❖ How to decompose our matrices?



DD example: 2D matrix stencil

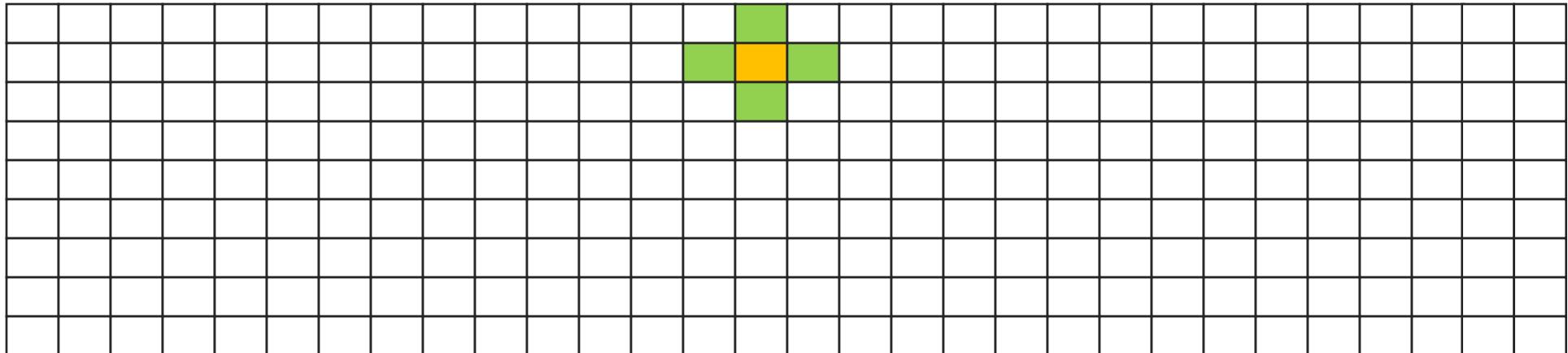
- ❖ a 30x9 integer Matrix
- ❖ 3 MPI processes
- ❖ Jacobi 4-point stencil
 - A cell computation requires values from the 4 direct neighbours





DD example: 2D matrix stencil

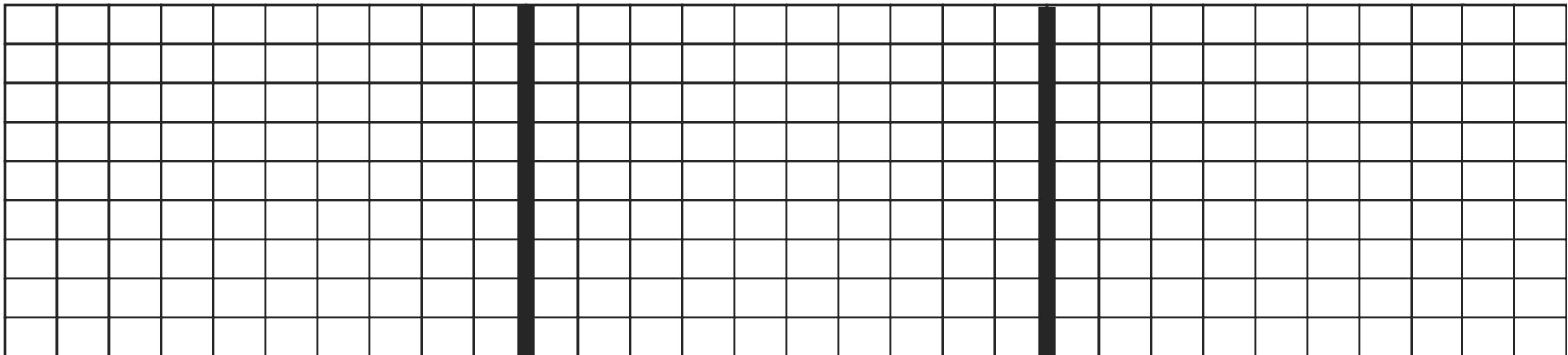
- ❖ a 30x9 integer Matrix
- ❖ 3 MPI processes
- ❖ Jacobi 4-point stencil
 - A cell computation requires values from the 4 direct neighbours





DD example: 2D matrix stencil

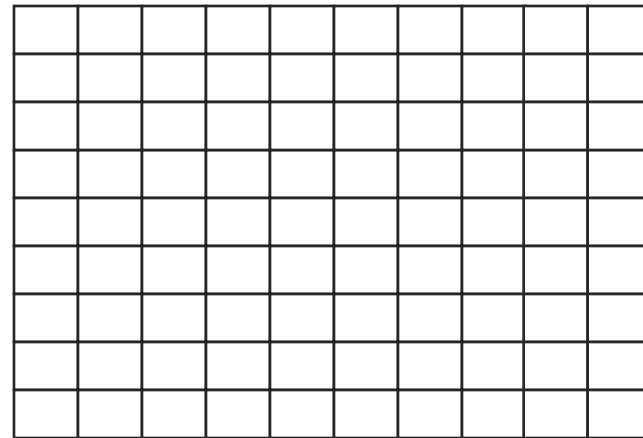
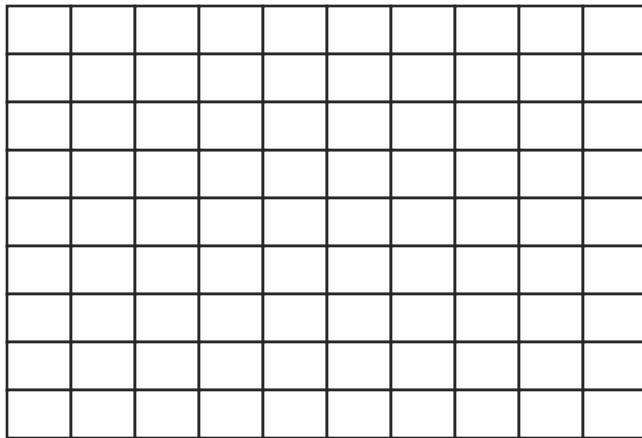
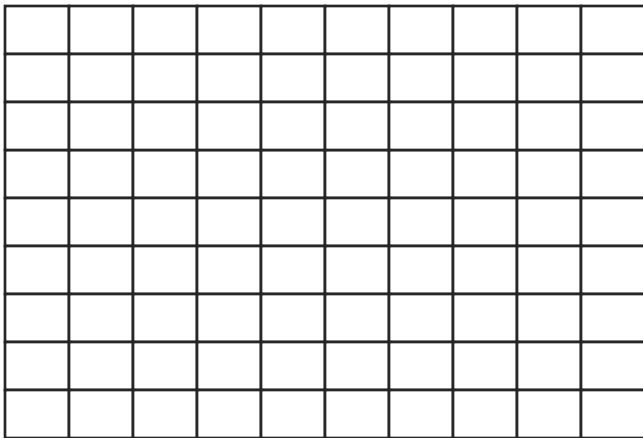
- ❖ a 30x9 integer Matrix
- ❖ 3 MPI processes
- ❖ Jacobi 4-point stencil
 - A cell computation requires values from the 4 direct neighbours
- ❖ Vertical decomposition





DD example: 2D matrix stencil

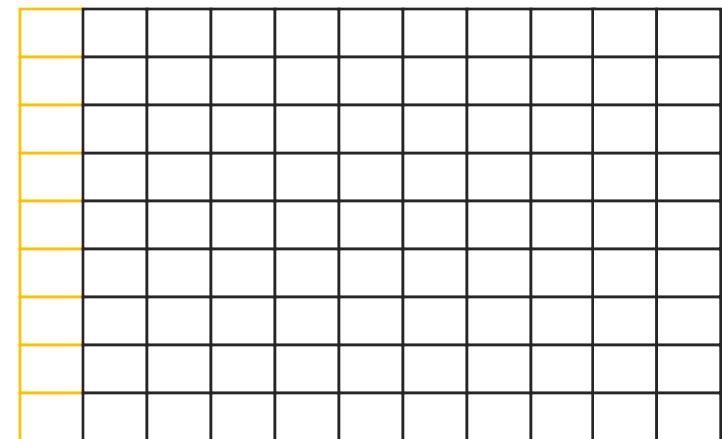
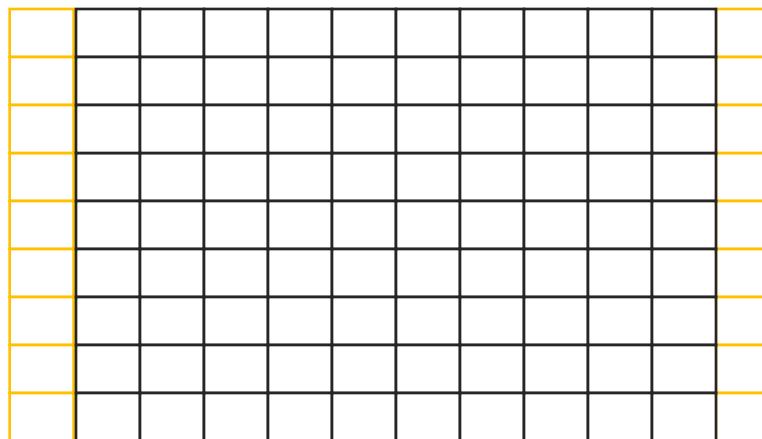
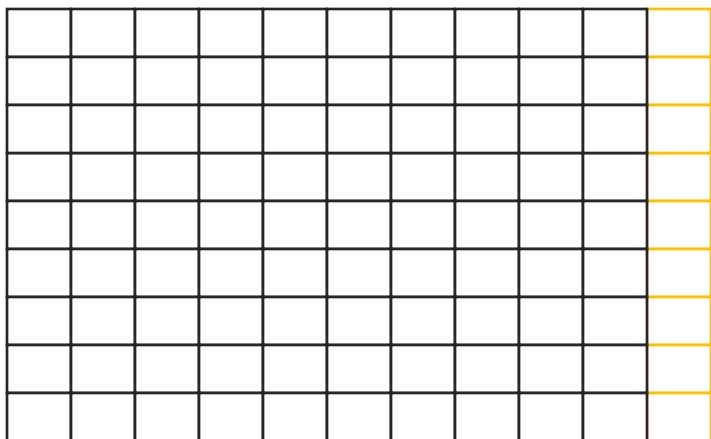
- ❖ Vertical decomposition





DD example: 2D matrix stencil

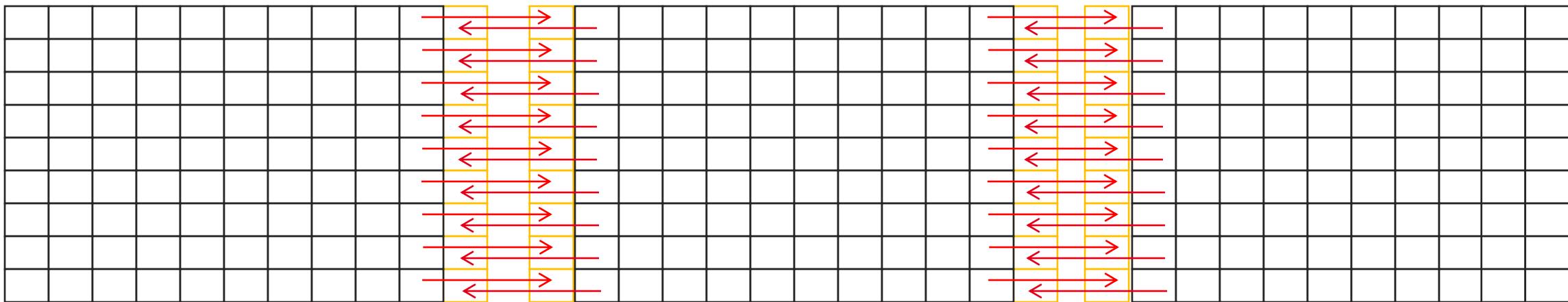
- ❖ Vertical decomposition





DD example: 2D matrix stencil

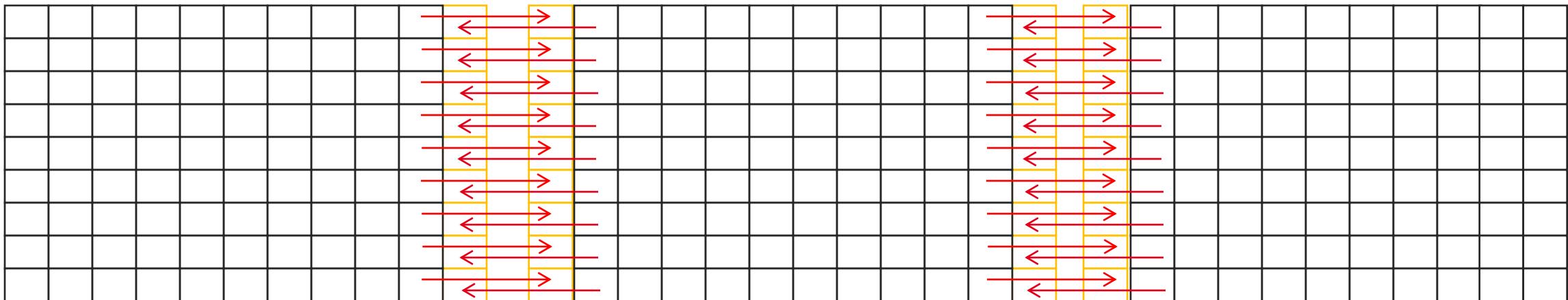
- ❖ Vertical decomposition
- ❖ Ghost cells exchange





DD example: 2D matrix stencil

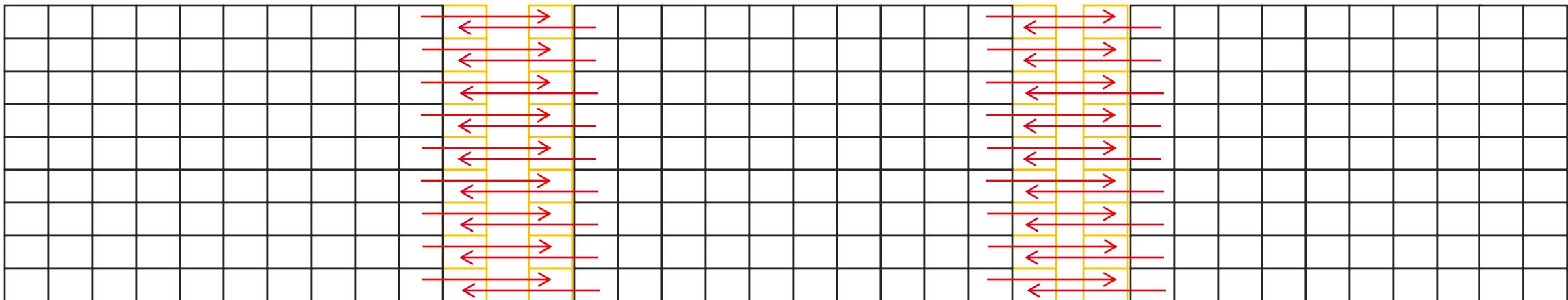
- ❖ Vertical decomposition
- ❖ Ghost cells exchange:
 - `MPI_Send(&(a[0][1], 1, MPI_INT, ...)` `MPI_Send(&(a[1][1], 1, MPI_INT, ...)`
 - `MPI_Send(&(a[2][1], 1, MPI_INT, ...)` `MPI_Send(&(a[3][1], 1, MPI_INT, ...)`
 - ...
 - `MPI_Send(&(a[0][(L/3)], 1, MPI_INT, ...)` `MPI_Send(&(a[1][(L/3)], 1, MPI_INT, ...)`





DD example: 2D matrix stencil

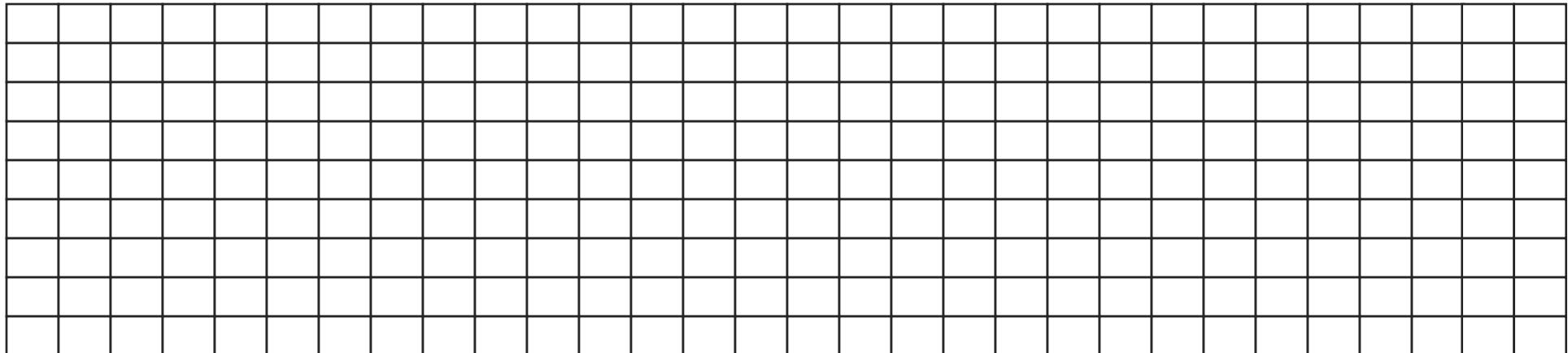
- ❖ Vertical decomposition
- ❖ Ghost cells exchange: **18 sends**
 - `MPI_Send(&(a[0][1], 1, MPI_INT, ...)` `MPI_Send(&(a[1][1], 1, MPI_INT, ...)`
 - `MPI_Send(&(a[2][1], 1, MPI_INT, ...)` `MPI_Send(&(a[3][1], 1, MPI_INT, ...)`
 - ...
 - `MPI_Send(&(a[0][(L/3)], 1, MPI_INT, ...)` `MPI_Send(&(a[1][(L/3)], 1, MPI_INT, ...)`





DD example: 2D matrix stencil

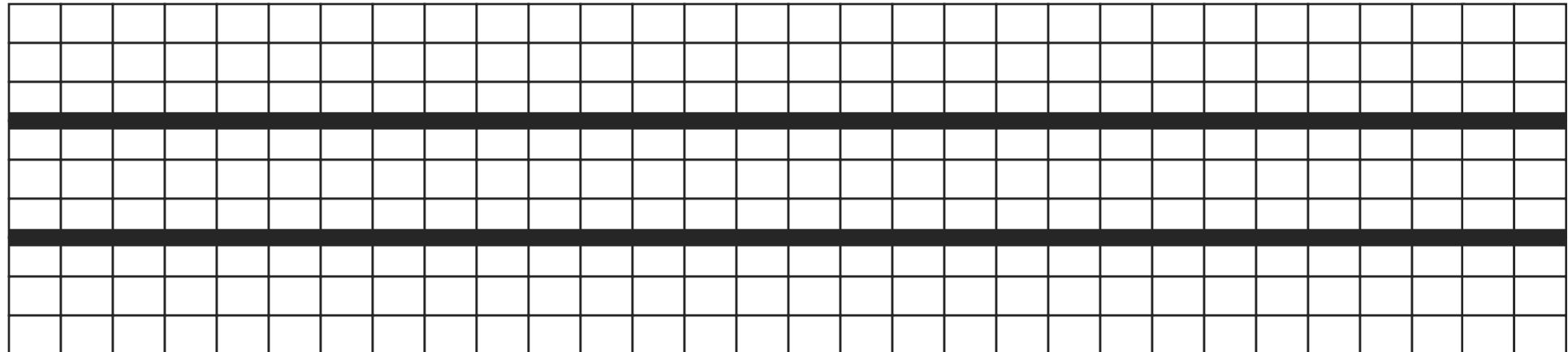
- ❖ a 30x9 integer Matrix
- ❖ 3 MPI processes
- ❖ Jacobi 4-point stencil





DD example: 2D matrix stencil

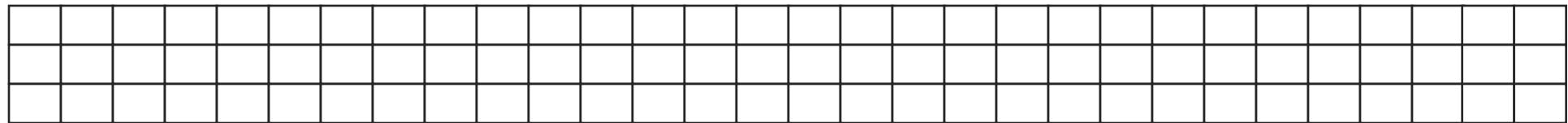
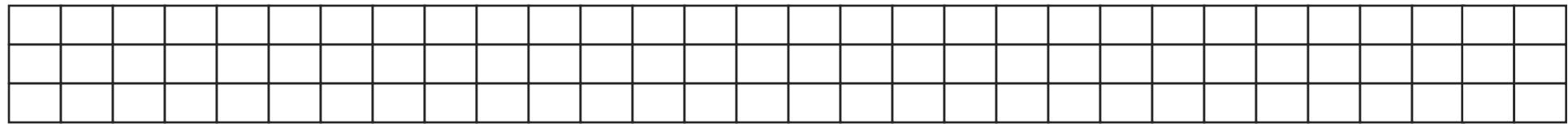
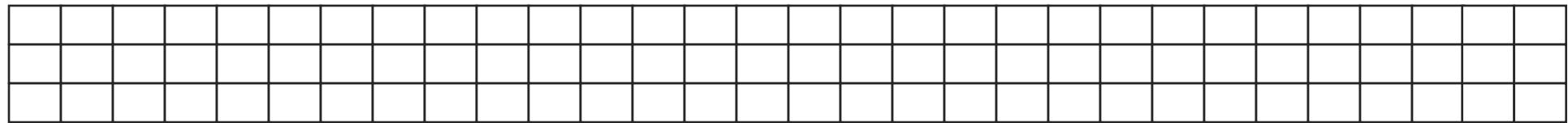
- ❖ a 30x9 integer Matrix
- ❖ 3 MPI processes
- ❖ Jacobi 4-point stencil
- ❖ Horizontal decomposition





DD example: 2D matrix stencil

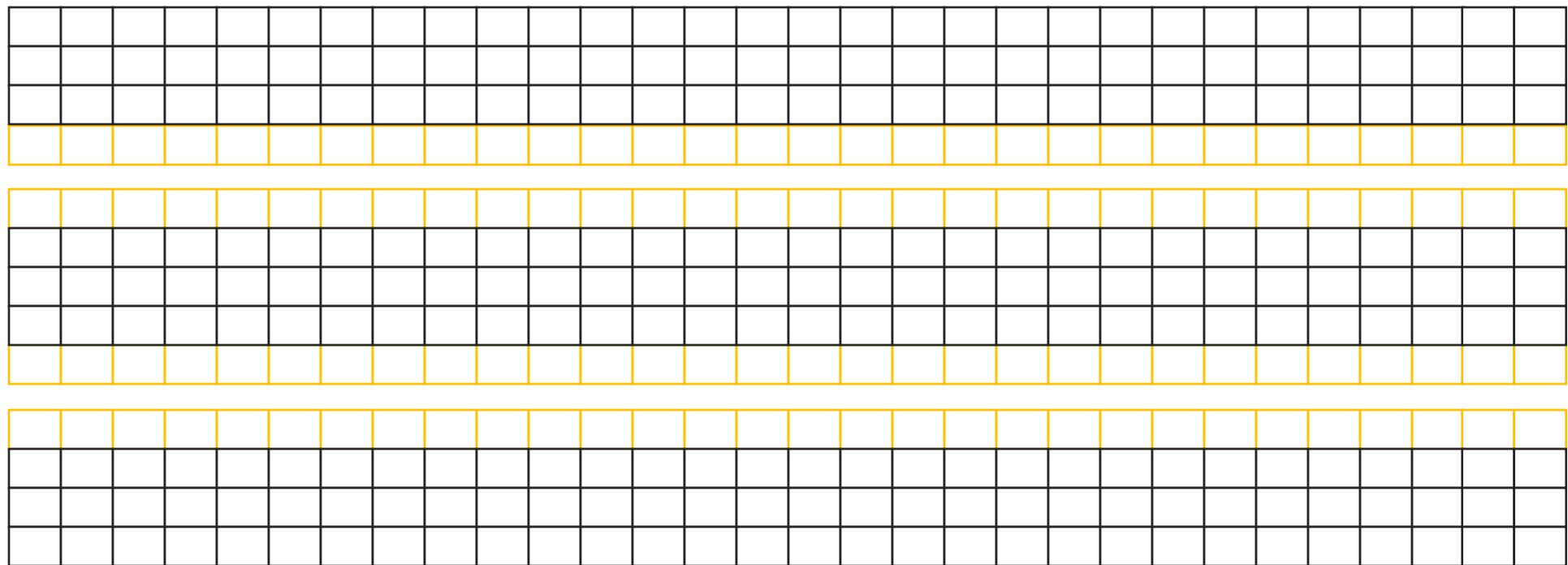
- ❖ Horizontal decomposition





DD example: 2D matrix stencil

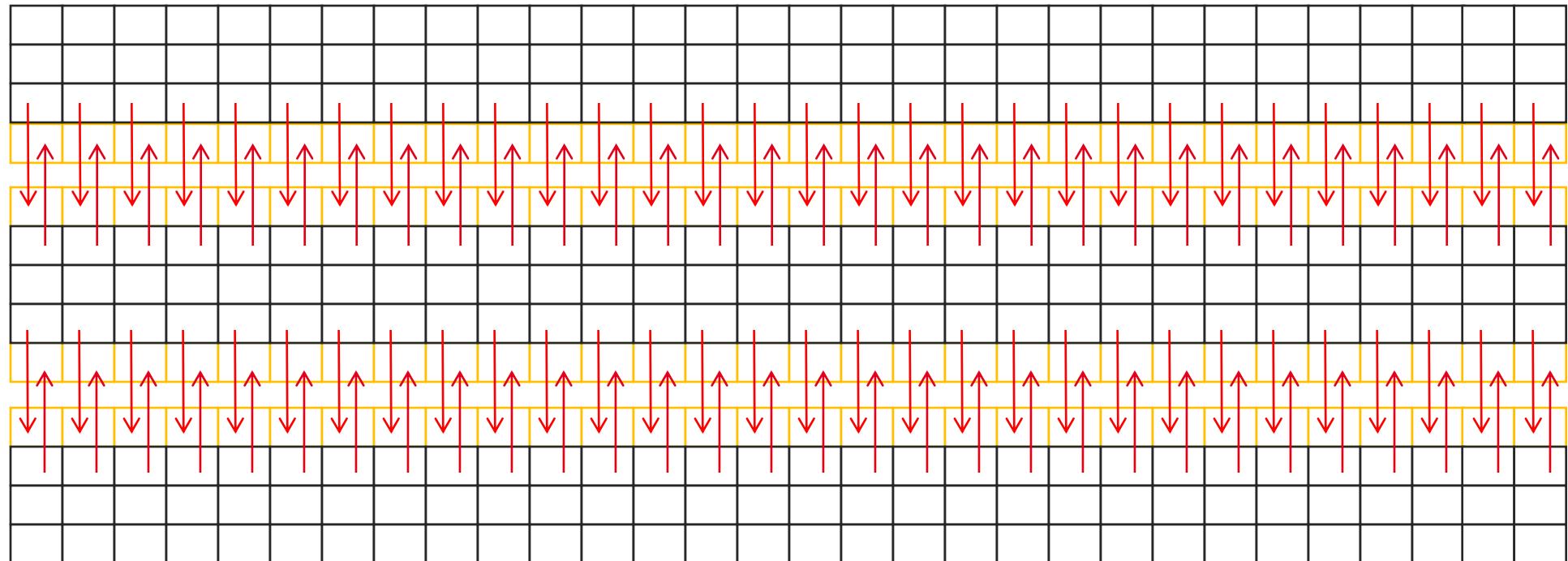
- ❖ Horizontal decomposition





DD example: 2D matrix stencil

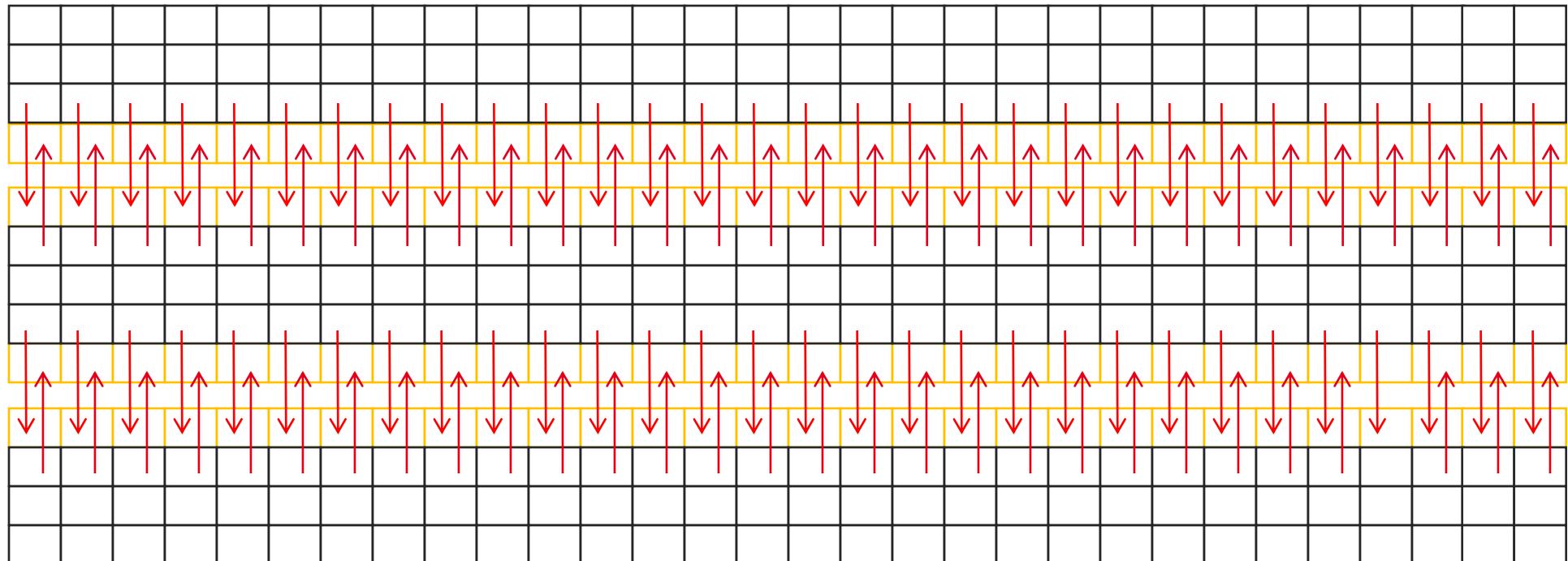
- ❖ Horizontal decomposition
- ❖ Ghost cells exchange





DD example: 2D matrix stencil

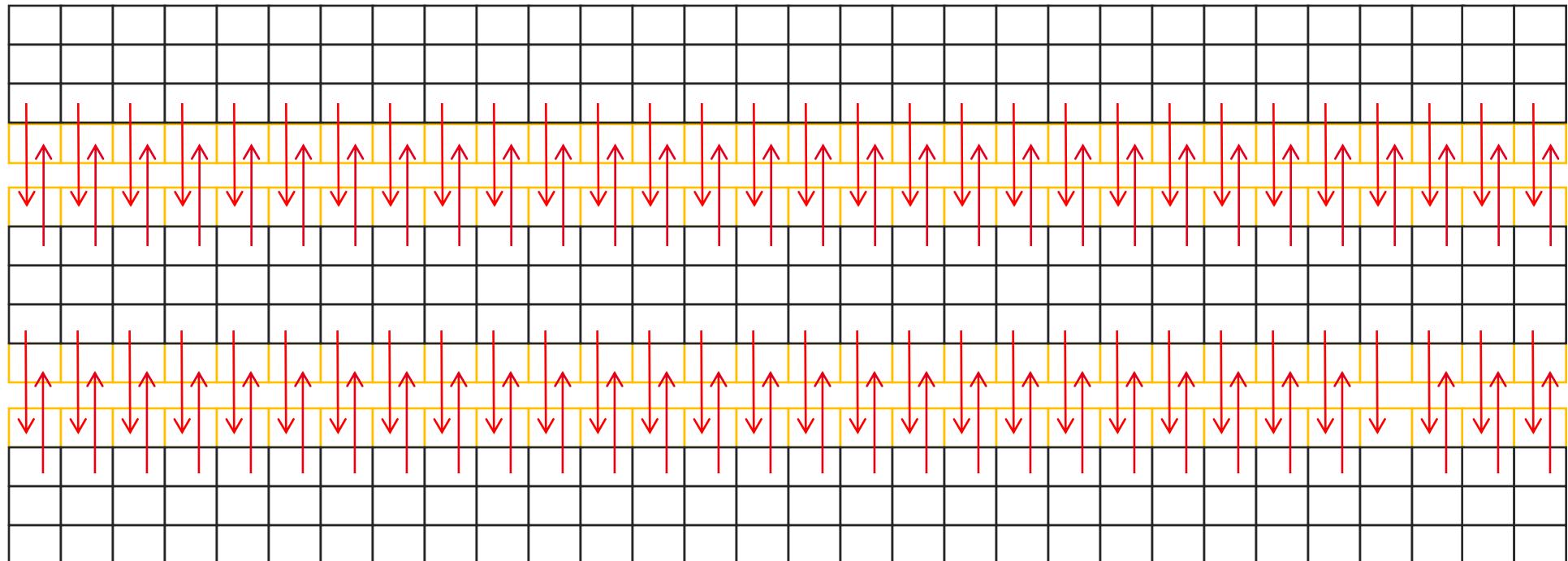
- ❖ Horizontal decomposition
- ❖ Ghost cells exchange
 - `MPI_Send(&(a[1][0], 30, MPI_INT, ...)`
 - `MPI_Send(&(a[H/3][1], 1, MPI_INT, ...)`





DD example: 2D matrix stencil

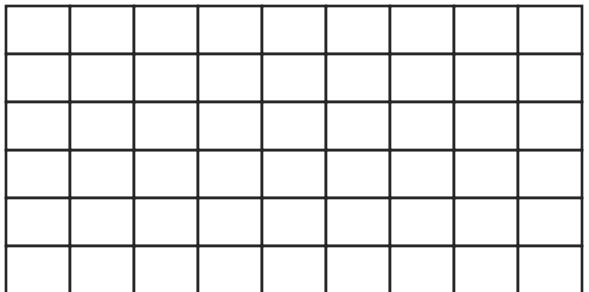
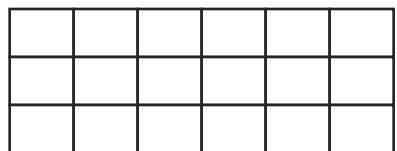
- ❖ Horizontal decomposition
- ❖ Ghost cells exchange: 2 sends
 - `MPI_Send(&(a[1][0], 30, MPI_INT, ...)`
 - `MPI_Send(&(a[H/3][1], 1, MPI_INT, ...)`





Ex. 2: stencils and Sgemm

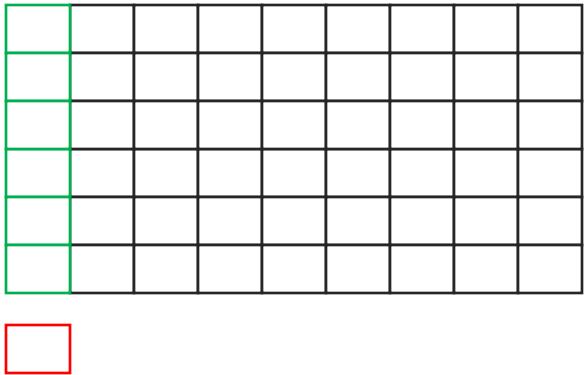
- ❖ Sgemm: simple general matrix multiplication





Ex. 2: stencils and Sgemm

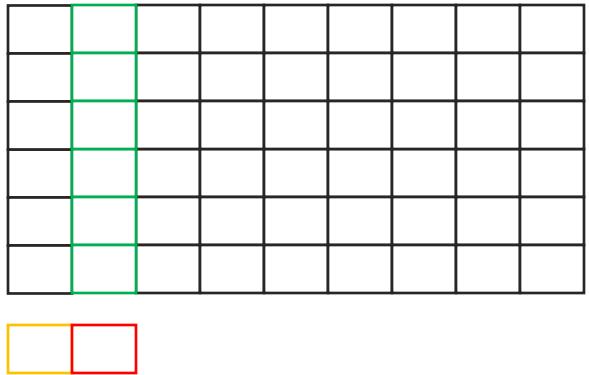
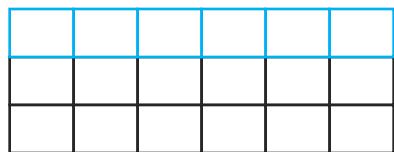
- ❖ Sgemm: simple general matrix multiplication
- ❖ Reminder





Ex. 2: stencils and Sgemm

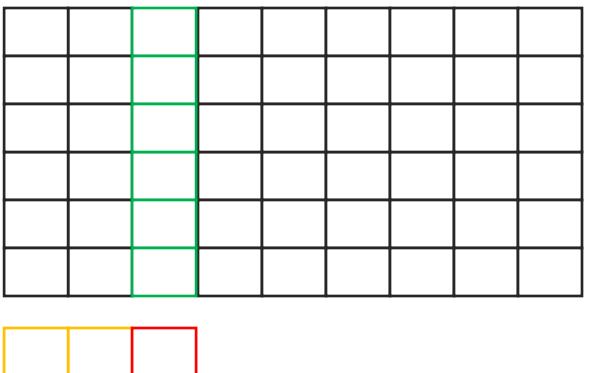
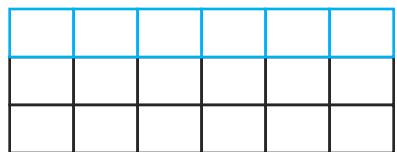
- ❖ Sgemm: simple general matrix multiplication
- ❖ Reminder





Ex. 2: stencils and Sgemm

- ❖ Sgemm: simple general matrix multiplication
- ❖ Reminder





Ex. 2: stencils and Sgemm

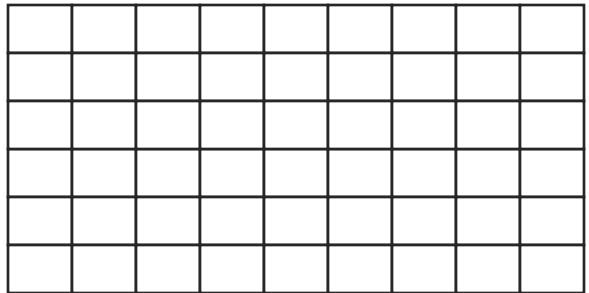
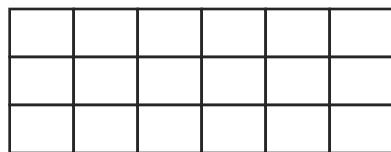
- ❖ Sgemm: simple general matrix multiplication
- ❖ Reminder





Ex. 2: stencils and Sgemm

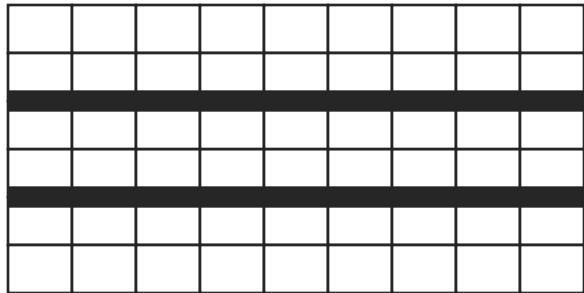
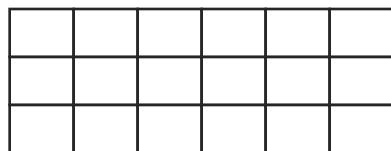
- ❖ Sgemm: simple general matrix multiplication
- ❖ Reminder





Ex. 2: stencils and Sgemm

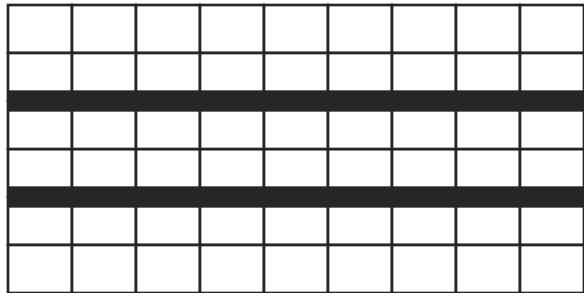
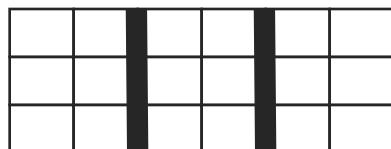
- ❖ Sgemm: simple general matrix multiplication
- ❖ Decomposing first matrix...





Ex. 2: stencils and Sgemm

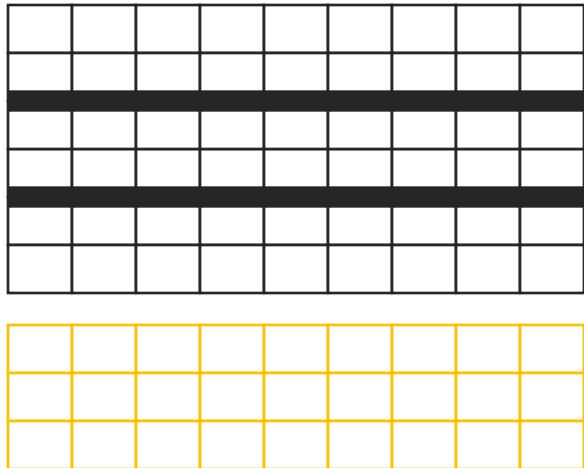
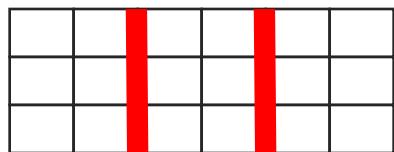
- ❖ Sgemm: simple general matrix multiplication
- ❖ Decomposing first matrix...
- ❖ ...forces the decomposition of the second





Ex. 2: stencils and Sgemm

- ❖ Second decomposition is not optimal
 - See first example





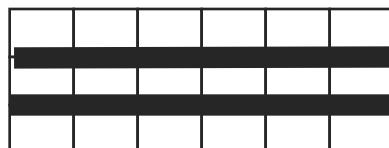
Ex. 2: stencils and Sgemm

- ❖ 2 solutions



Ex. 2: stencils and Sgemm

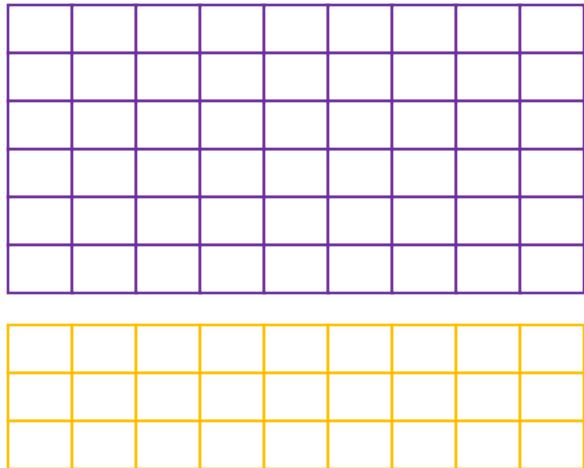
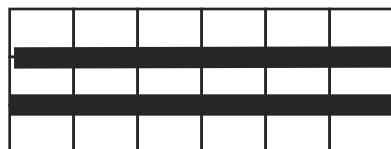
- ❖ 2 solutions
- Decompose only one matrix
 - To generate a cell (or several cells) per MPI process





Ex. 2: stencils and Sgemm

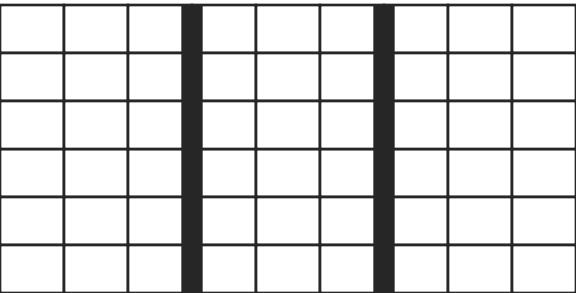
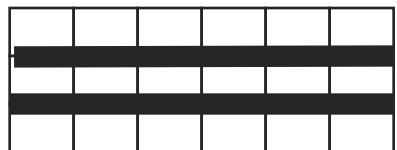
- ❖ 2 solutions
- Decompose only one matrix
 - Generate one or several cells per MPI process
 - Ok if one matrix can fit on memory (1 copy per MPI process)





Ex. 2: stencils and Sgemm

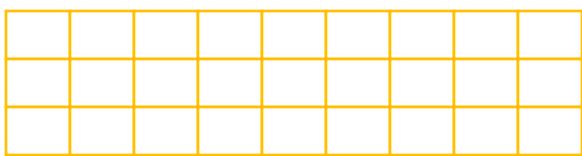
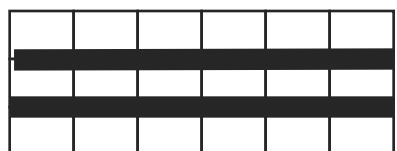
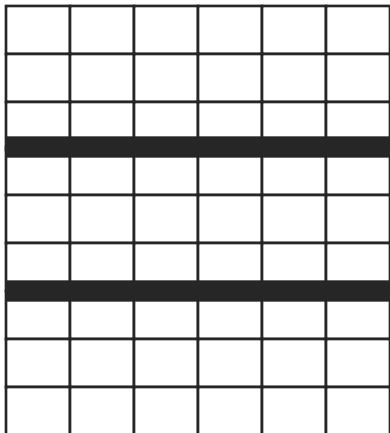
- ❖ 2 solutions
 - Decompose only one matrix
 - If the two matrices need to be decomposed





Ex. 2: stencils and Sgemm

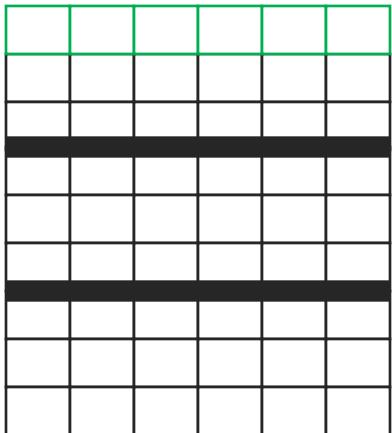
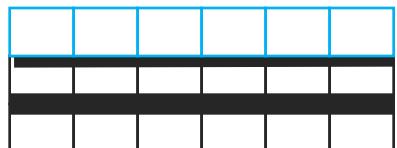
- ❖ 2 solutions
 - Decompose only one matrix
 - If the two matrices need to be decomposed Transpose!





Ex. 2: stencils and Sgemm

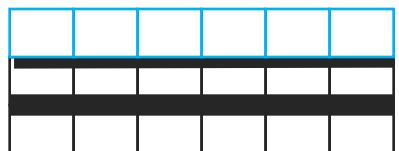
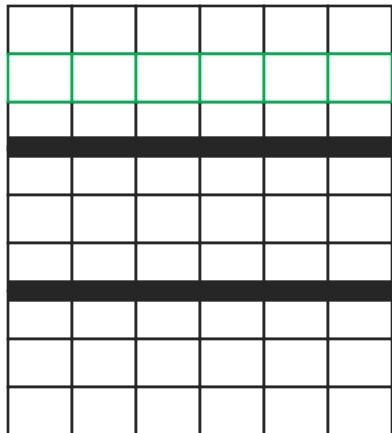
- ❖ 2 solutions
 - Decompose only one matrix
 - If the two matrices need to be decomposed Transpose!





Ex. 2: stencils and Sgemm

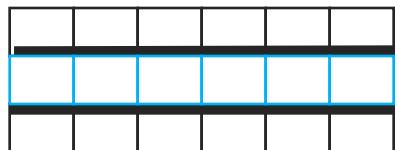
- ❖ 2 solutions
 - Decompose only one matrix
 - If the two matrices need to be decomposed Transpose!





Ex. 2: stencils and Sgemm

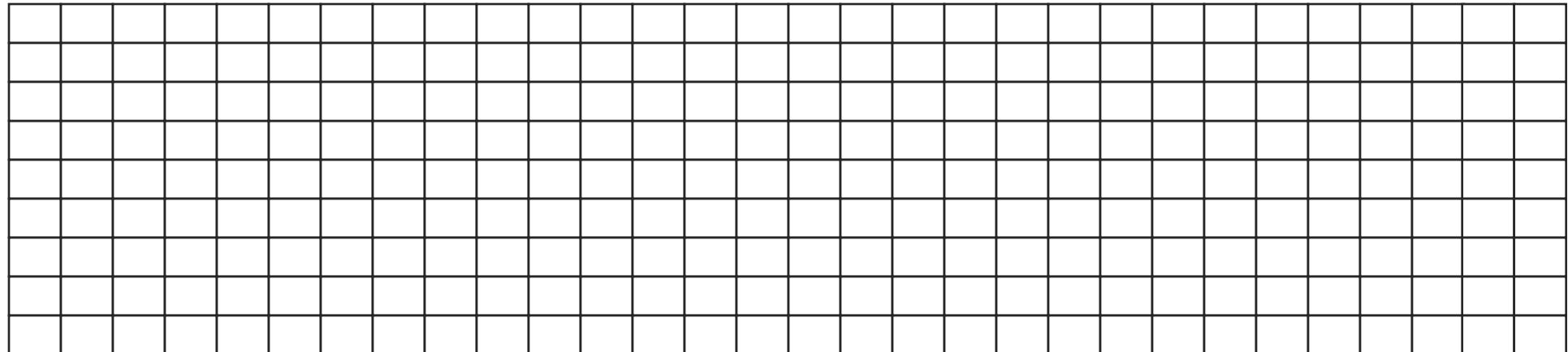
- ❖ 2 solutions
 - Decompose only one matrix
 - If the two matrices need to be decomposed Transpose!





Partitionning issues

- ❖ Communication cost is not the only issue when decomposing/partitionning a problem
 - Computation workload balance is also important

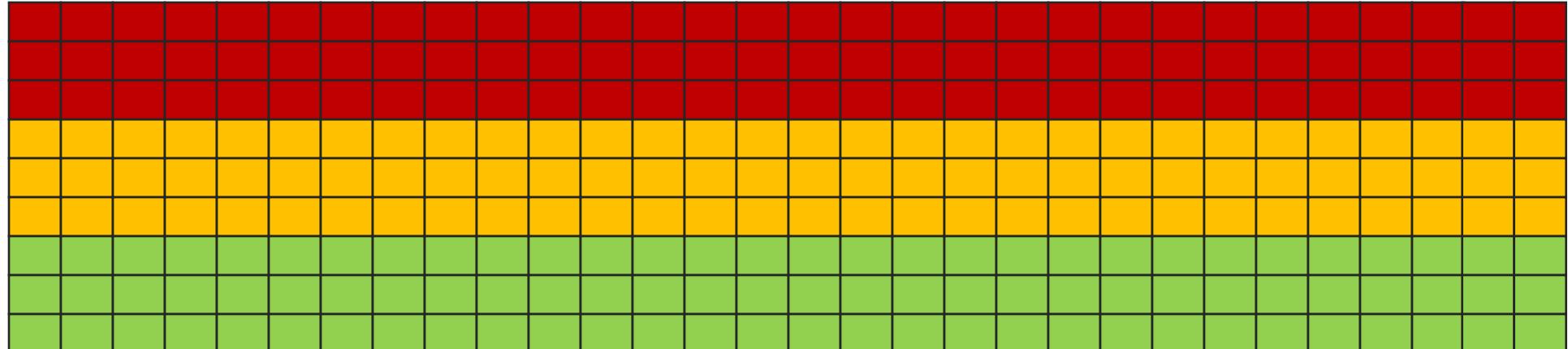




Partitionning issues

- ❖ Communication cost is not the only issue when decomposing/partitionning a problem
 - Computation workload balance is also important

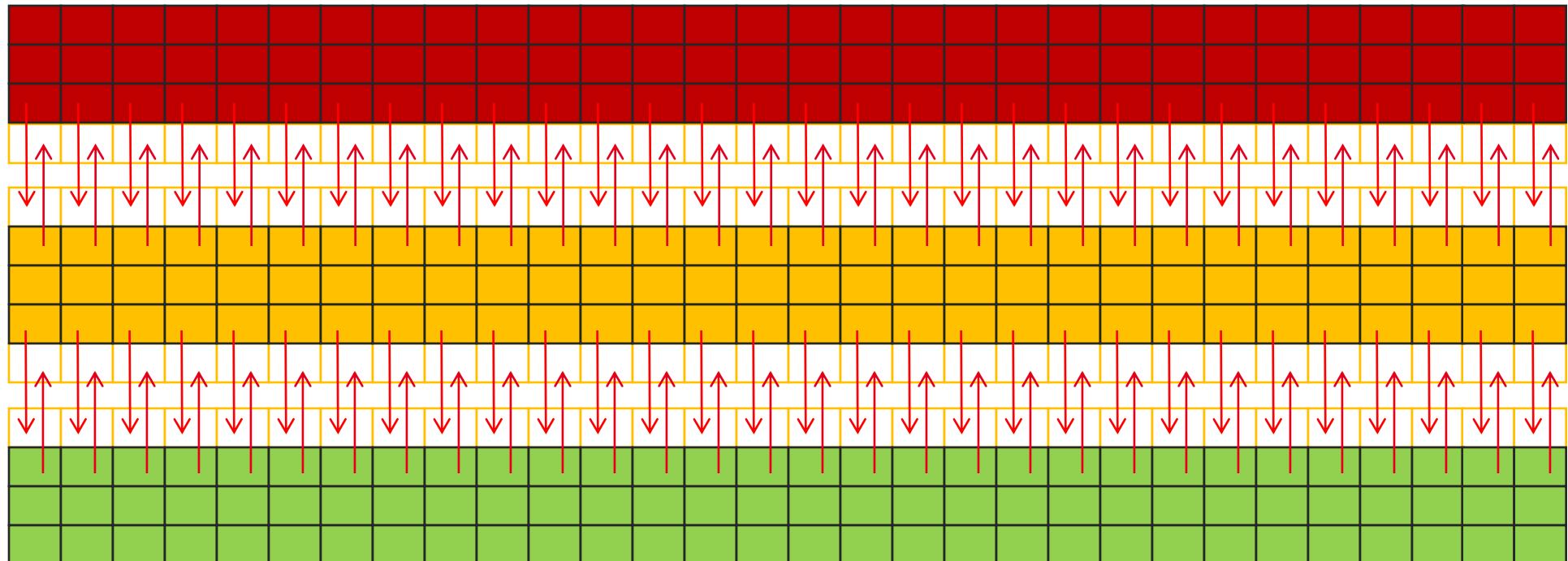
- ❖ Consider the same 2D matrix, but with weights on cells
 - Red > Orange > green (computation workload)





Partitionning issues

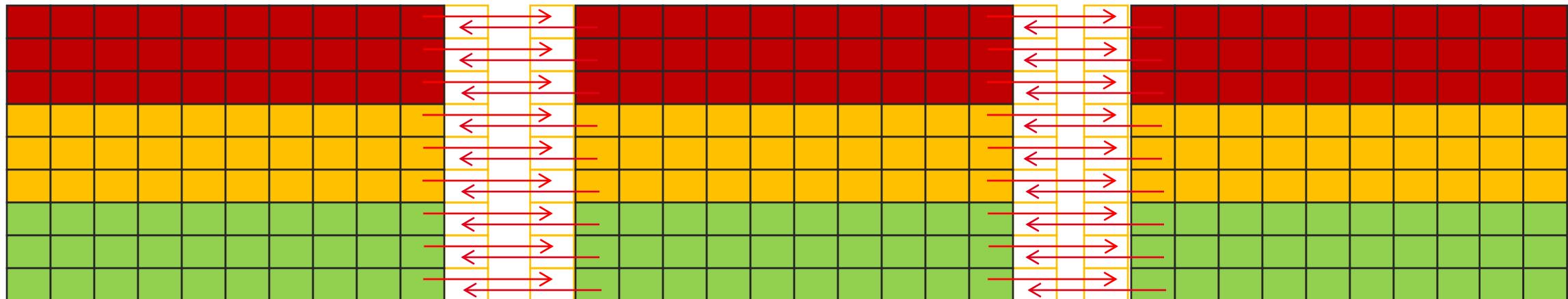
- ❖ If one uses horizontal decomposition
 - Best communication costs (2 MPI_Sends)
 - But compute workload imbalance
 - 1 process have all the heavyweight cells





Partiotonning issues

- ❖ Whereas with vertical decomposition
 - Computation workload is balanced between MPI processes
 - But the communication cost is higher





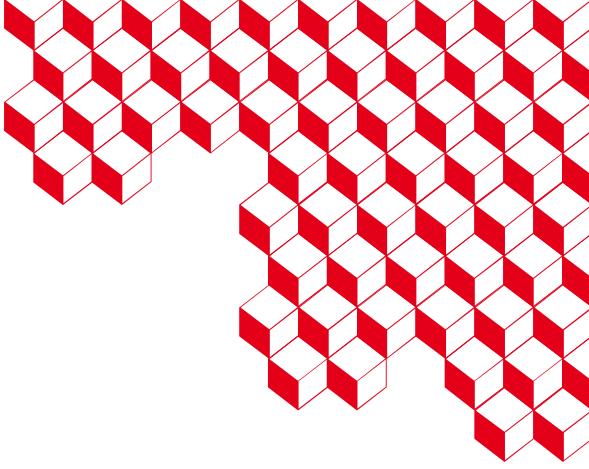
Automatic partitionner

- ❖ Partitionning/Decomposing a problem on MPI processes is not a simple tasks
 - Many metrics might need to be considered
- ❖ To help developers, automatic partitionners exist
 - Ex: ParMETIS
- ❖ Divide the problem according to the chosen metrics
 - single criterion: comm. cost, comp. load balancing
 - or multi criteria: load balancing + smallest frontiers surface



Small conclusion on user side

- ❖ The design of the code is of paramount importance
- ❖ If one is not careful, it can lead to heavy communications cost
 - Numerous calls
 - Dependance chains between calls
- ❖ Trade-off between communication cost and workload balancing
 - And other metrics can also be of interest



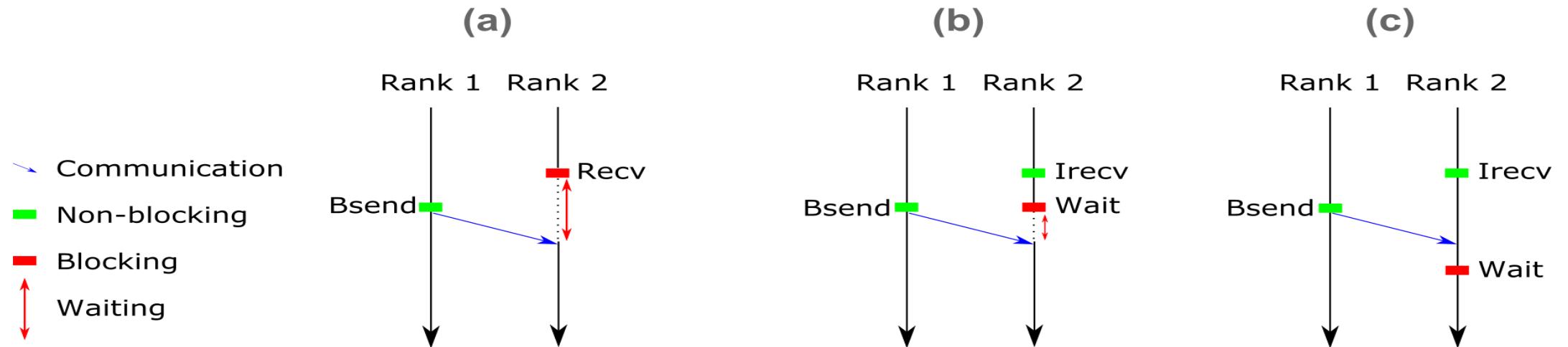
MPI optim.: implem. side



Asynchronous progression

❖ Allows to overlap communication with computation

- Example: blocking buffered send with
 - (a) blocking recv
 - (b) non-blocking recv partially overlapped
 - (c) non-blocking recv totally overlapped





The progression issue

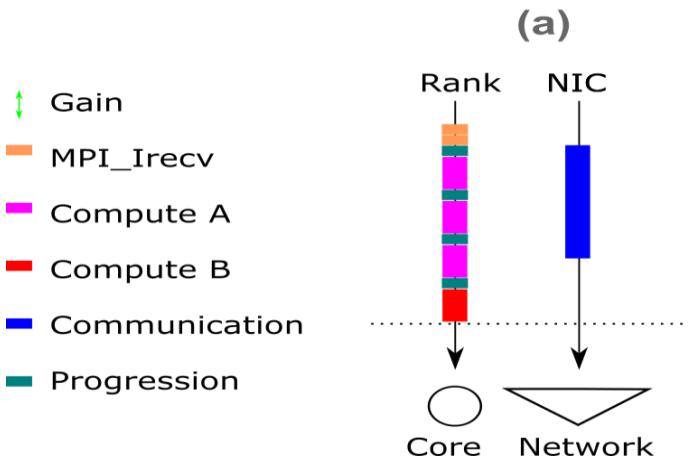
- ❖ How to progress messages?



The progression issue

❖ How to progress messages?

- a) on the same thread, need to call functions to progress

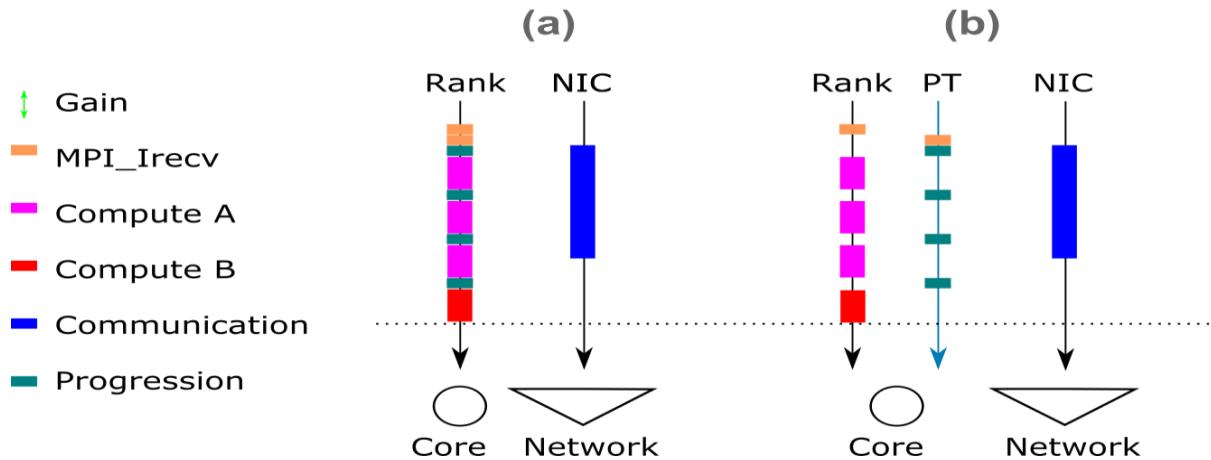




The progression issue

❖ How to progress messages?

- a) on the same thread, need to call functions to progress
- b) on another thread, only progression on thread but competition for resources

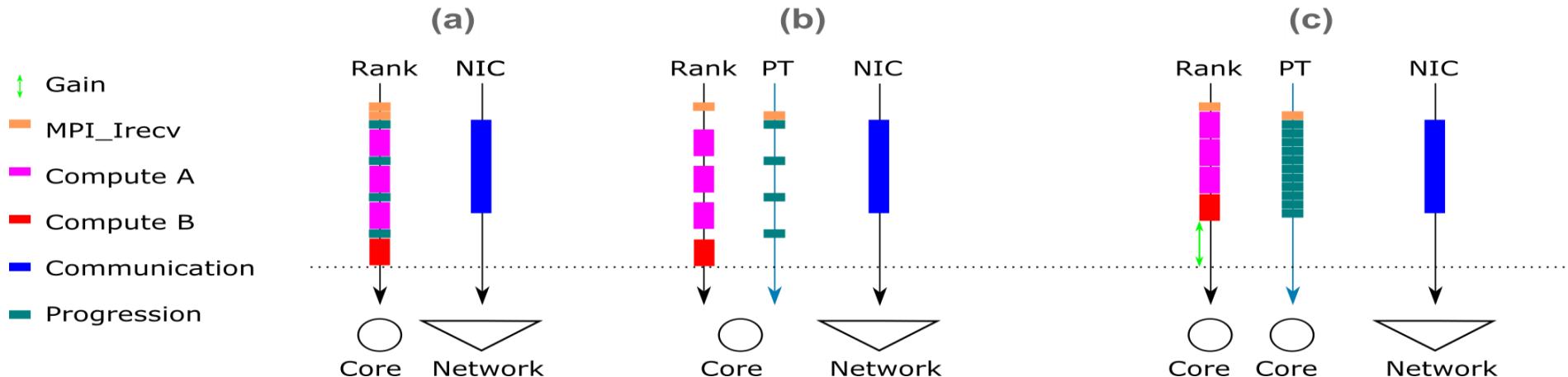




The progression issue

❖ How to progress messages?

- a) on the same thread, need to call functions to progress
- b) on another thread, only progression on thread but competition for resources
- c) on another core, start and finish communications as soon as possible

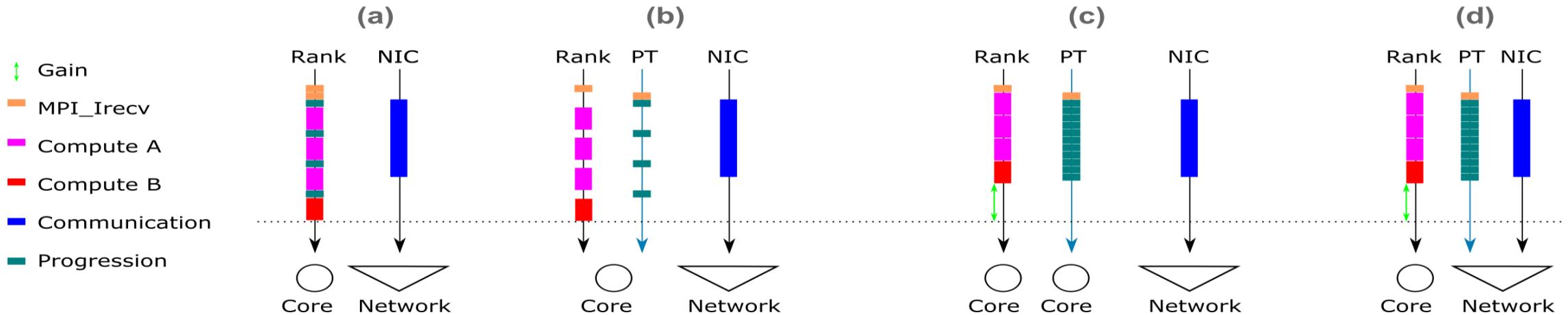




The progression issue

❖ How to progress messages?

- a) on the same thread, need to call functions to progress
- b) on another thread, only progression on thread but competition for resources
- c) on another core, start and finish communications as soon as possible
- d) on the network card, if the functionality is available



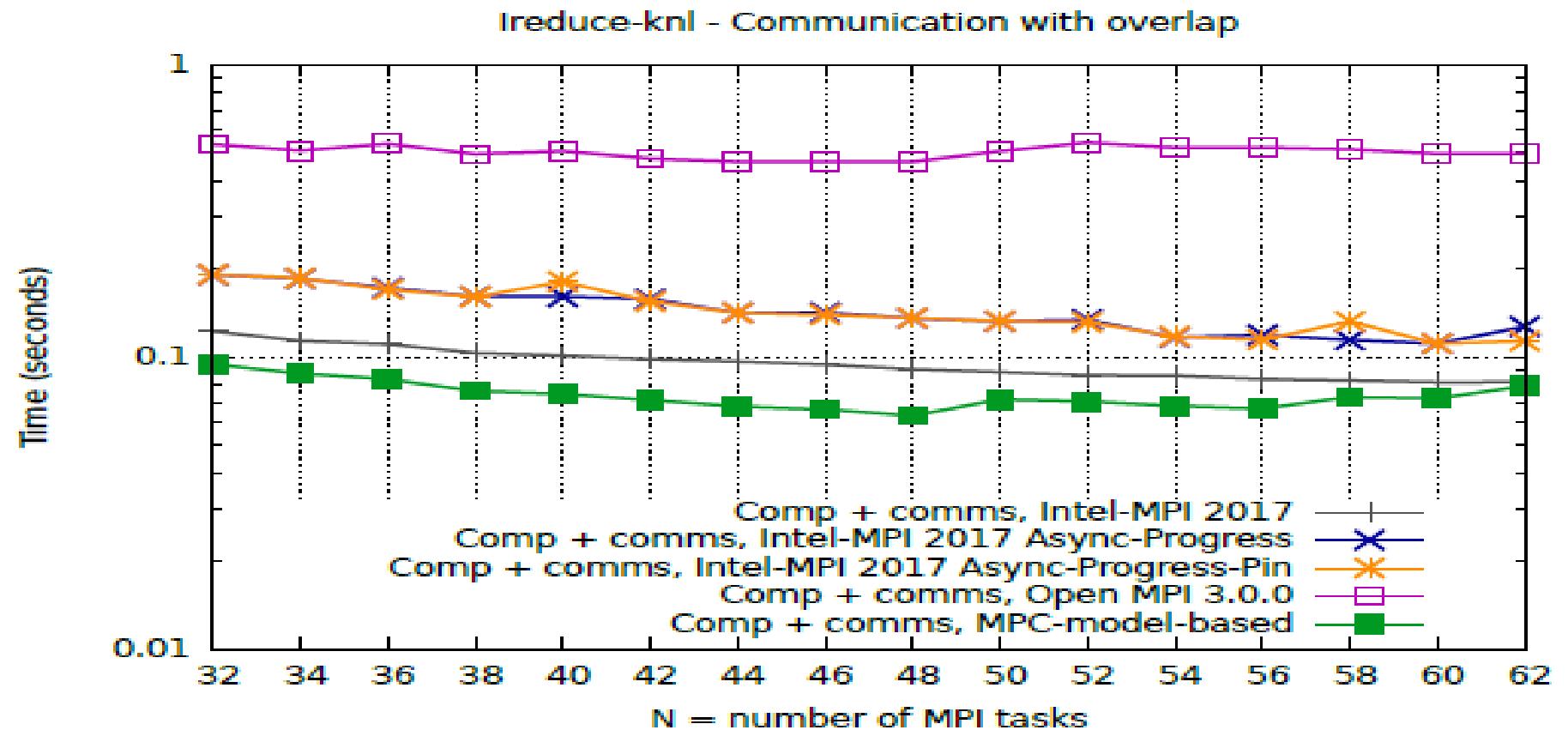


Implém. specifics

- ❖ Each MPI implementation provides its own specific options
- ❖ It may be really helpful to know the right options
 - Ensure there are no derived datatypes
 - Ensure there are no MPI I/O routines
 - Choose collective algorithms/patterns
 - Enable “real” asynchronous progress
 - Ex: In MPICH, 2 environment variables must be set to 1
 - `export I_MPI_ASYNC_PROGRESS=1`
 - `export I_MPI_ASYNC_PROGRESS_PIN=1`



Perf. comparison





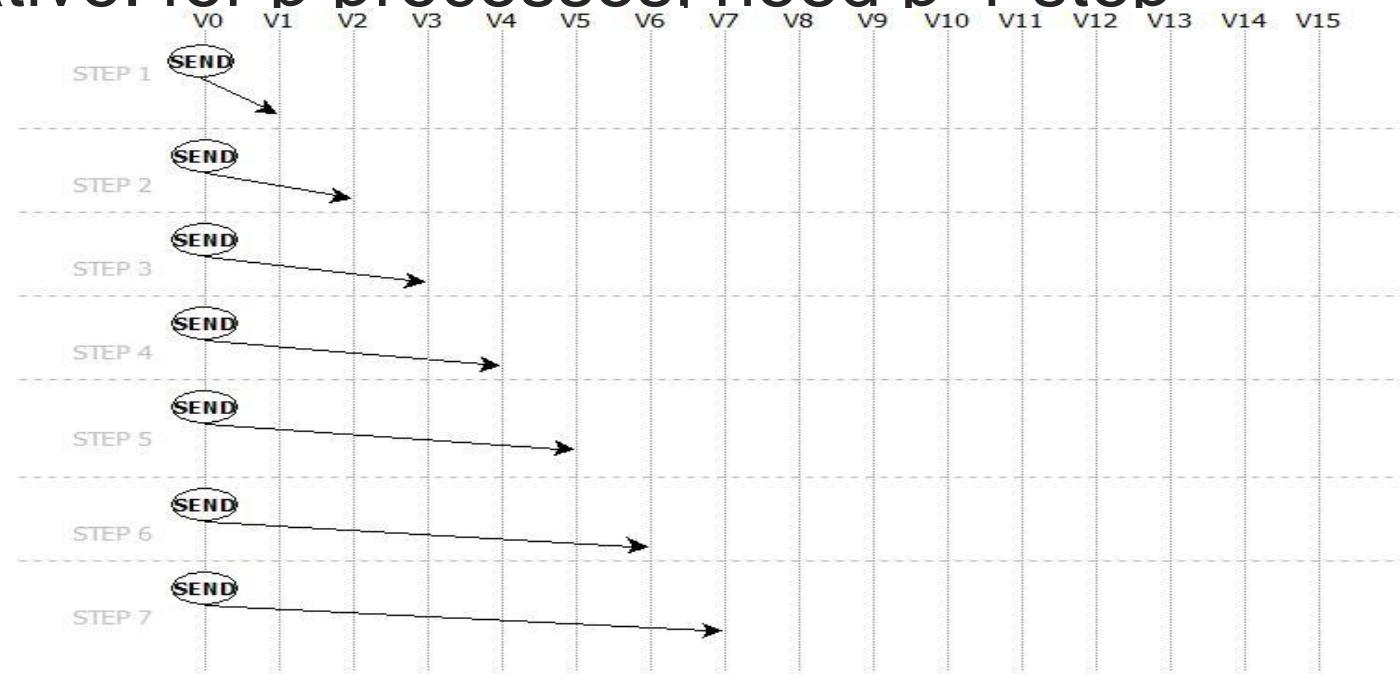
Communication algorithms

- ❖ Different algorithms are possible for most communication patterns
- ❖ Some are straightforward, and some are not
- ❖ Focus on collective communications algorithms
 - Provide the user with a set of commonly used communication pattern



One-to-all algorithms

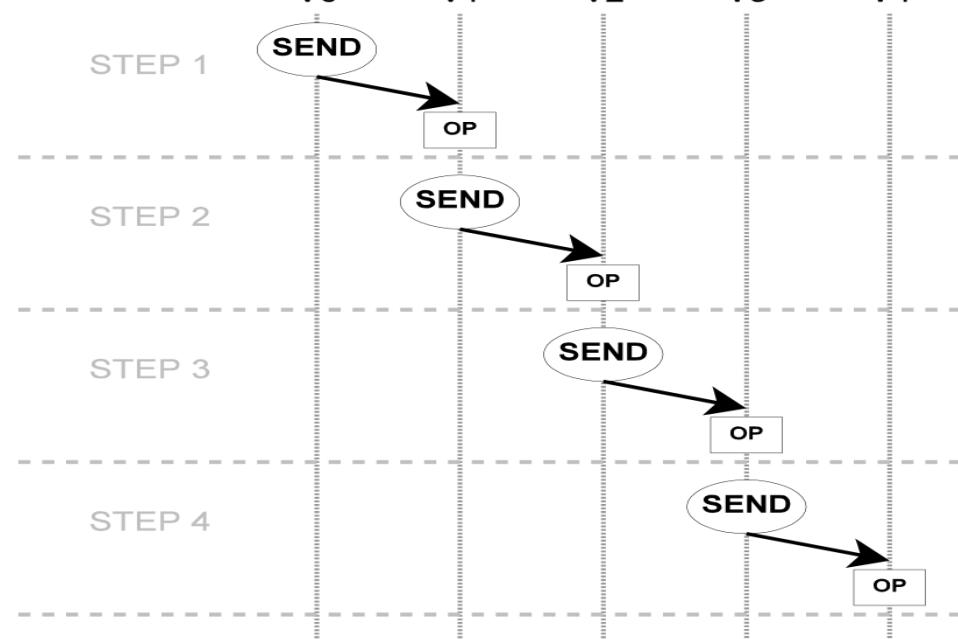
- ❖ Linear algorithm
- ❖ Root sends its data to each other process
- ❖ Iterative: for p processes, need $p-1$ step





One-to-all algorithms

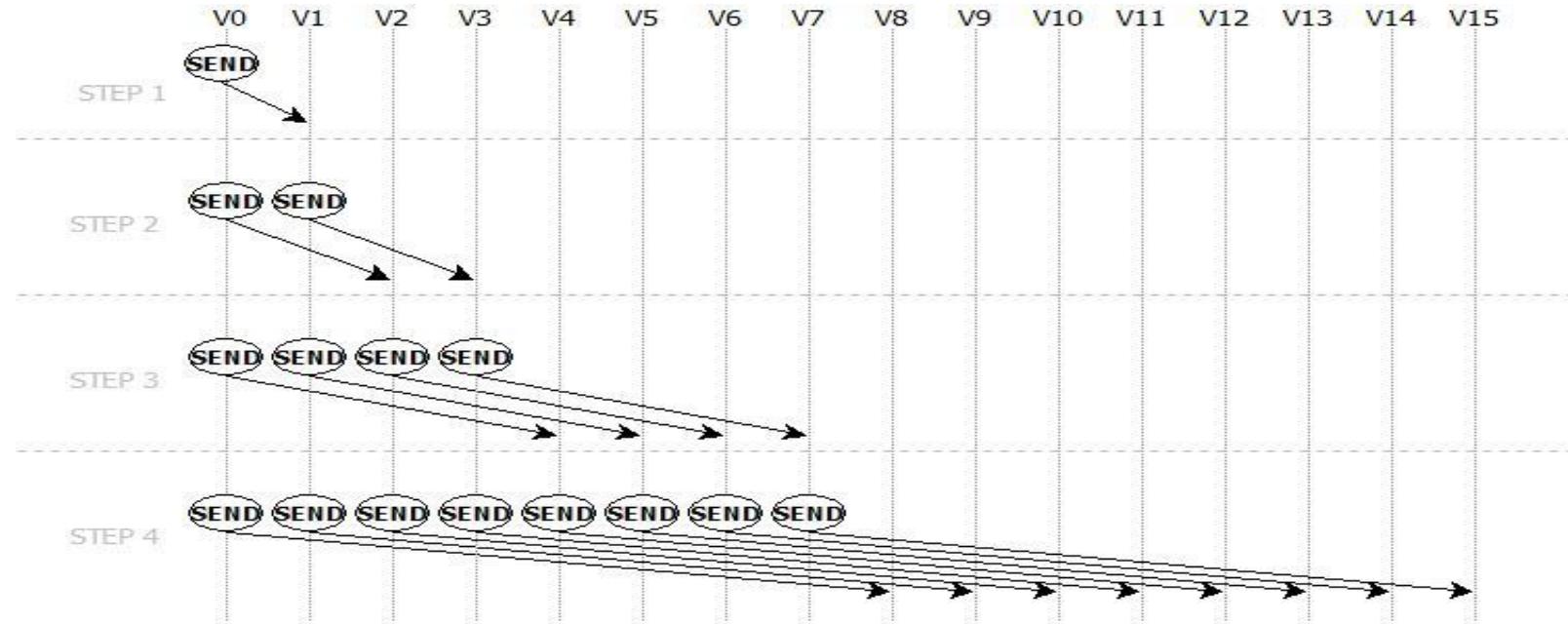
- ❖ Ring algorithm
- ❖ Each receiver sends the data to the next MPI process
- ❖ Iterative: for p processes, need $p-1$ step





One-to-all algorithms

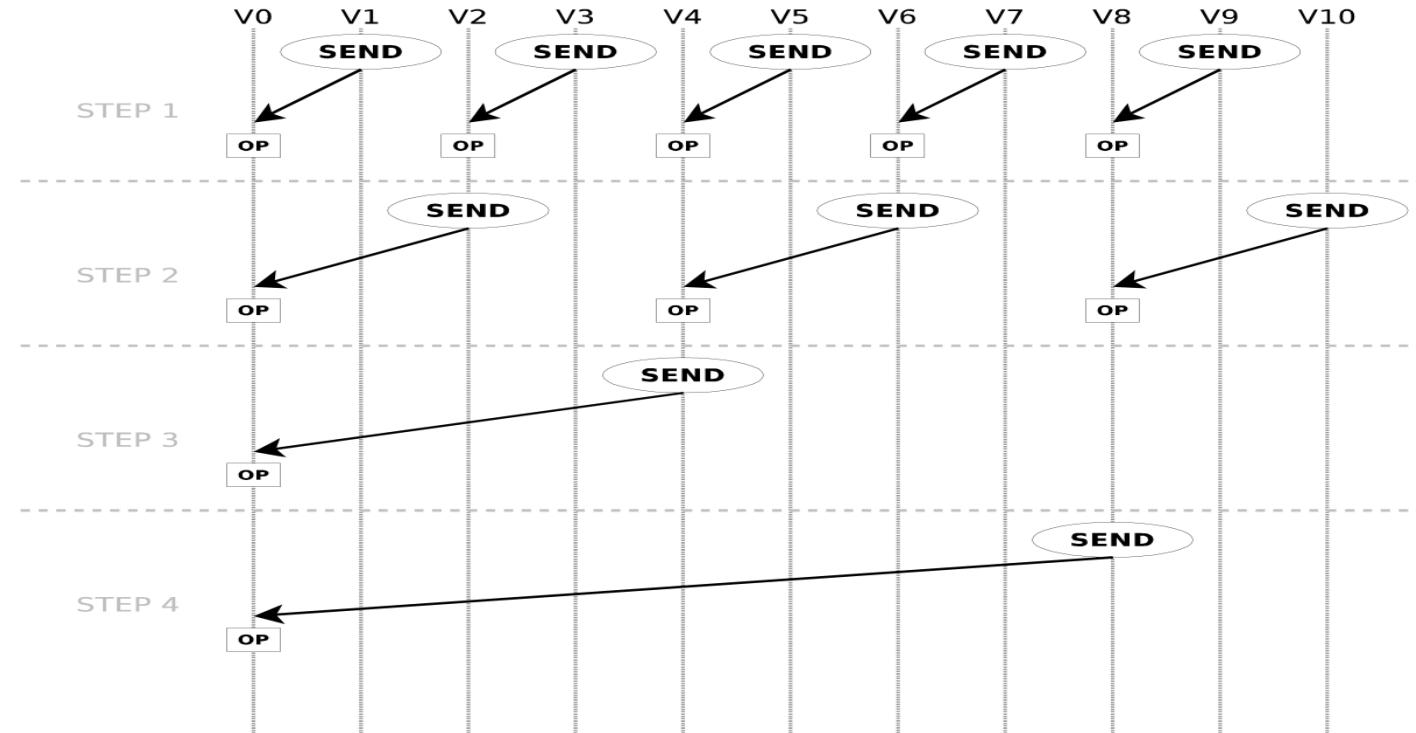
- ❖ Binomial tree: at each step, each process with the data to broadcast sends the data to another process
- ❖ Need $\log(p)$ steps





All-to-one algorithms

- ❖ Also implemented as linear, ring and binary tree
- ❖ Need p or $\log(p)$ steps





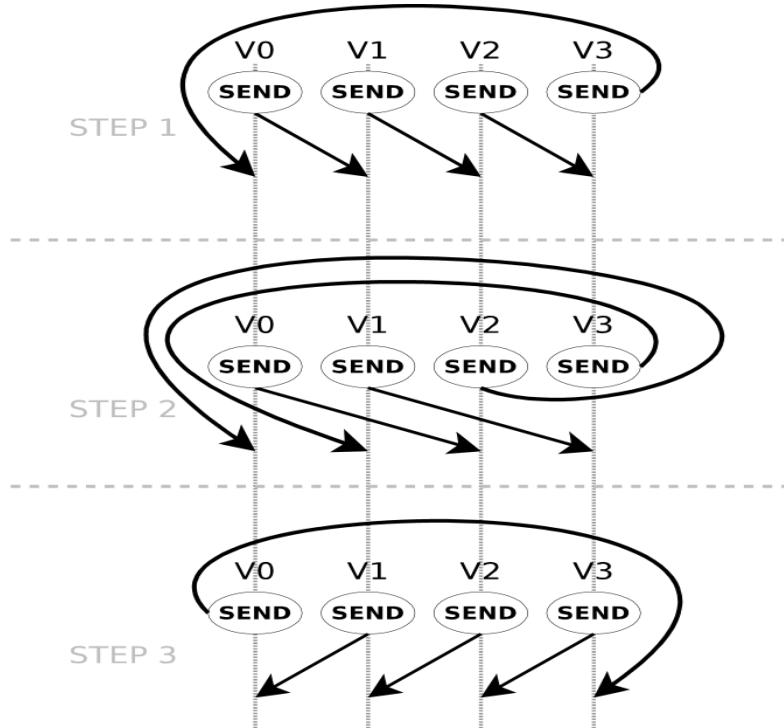
All-to-All algorithms

- ❖ Some all-to-all collectives can be implemented as an all-to-one collective followed by an one-to-all collective
 - Allreduce (reduce + broadcast)
 - Allgather (gather + broadcast)
 - Reduce_scatter (reduce + scatterv)
 - Reduce_scatter_block(reduce + scatter)
- ❖ Depends one the implementation of each part
 - linear + linear
 - tree + tree
 - tree + ring
 - ...



All-to-all algorithms

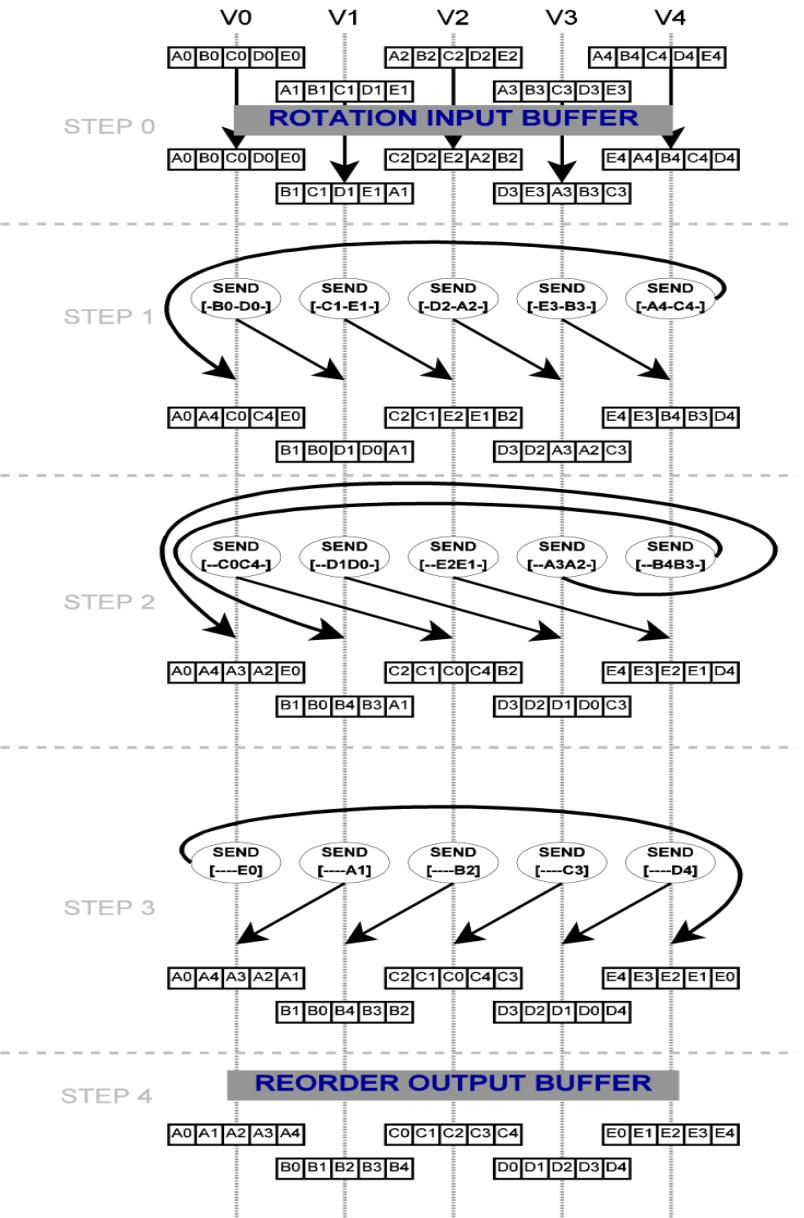
- ❖ Parallel linear broadcast algorithm
- ❖ Except that rank r will send to $(r+Step)\%p$ instead of $Step$
- ❖ Need $p-1$ steps, no contention





All-to-all algorithms

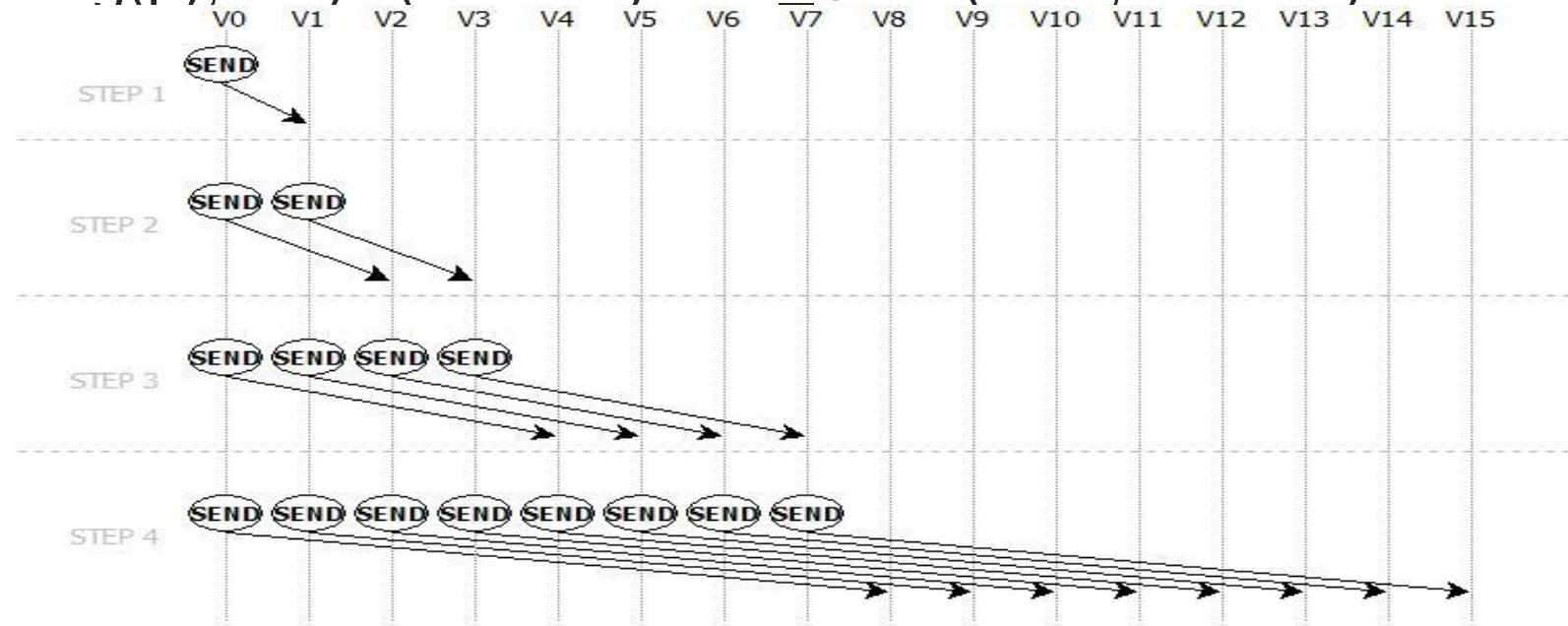
- ❖ Disymmetric ring
- ❖ Data shift of r elements
- ❖ Each step S , only elements with S^{th} bit to 1 are sent to next rank :
 - Step 1, elements 1 (001) and 3 (011) are sent
 - Step 2, elements 2 (010) and 3 (011) are sent
 - Step 3, elements 4 (100) is sent
- ❖ At the end, need to reorder output buffer
- ❖ Need $\log(p)$ steps (+shifts)





Broadcast tree: version 1

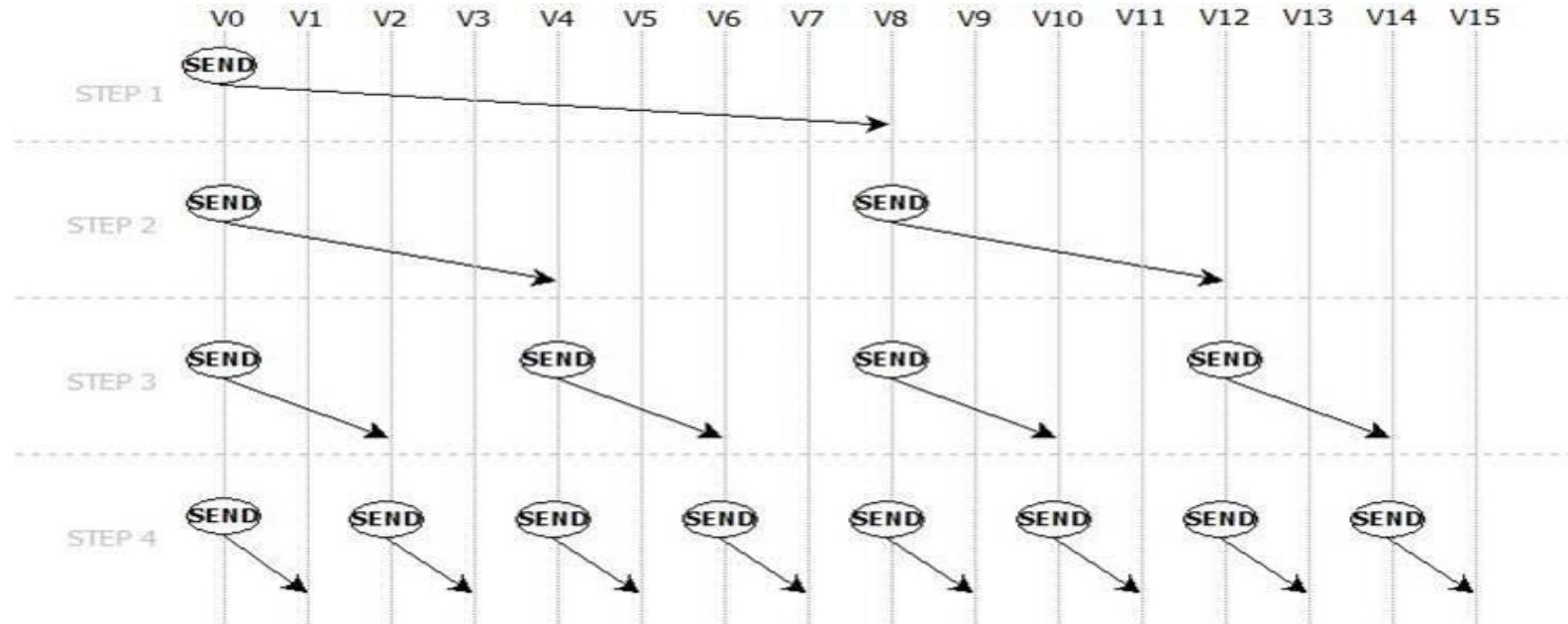
- ❖ Binomial tree: at each step, each process with the data to broadcast sends the data to another process
- ❖ Need $\log(p)$ steps
- ❖ `For(0; i<log(p); i++) if(rank<=i) MPI_Send(data, rank+2i)`





Broadcast tree: version 2

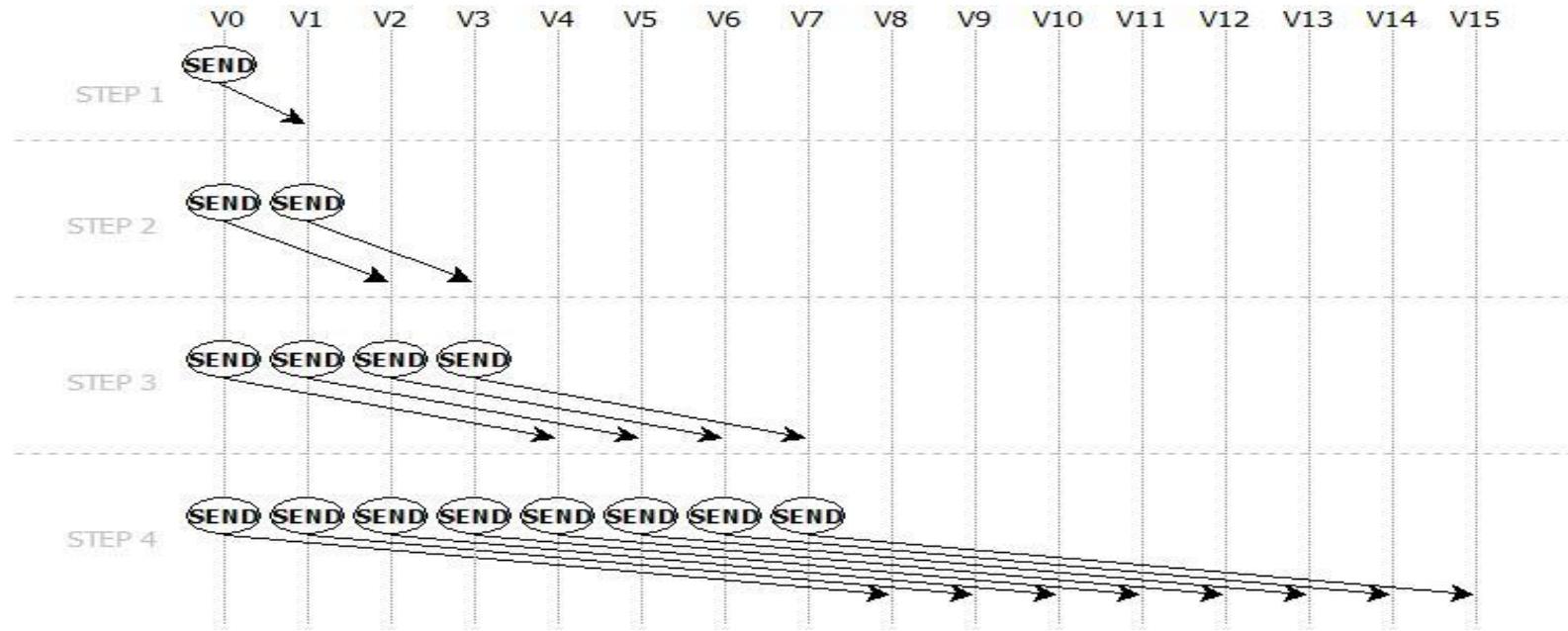
- ❖ Binary tree
- ❖ $\text{For}(\log(p); i>0; i--) \text{ if}(rank \% 2^i == 0) \text{ MPI_Send(data, rank+}2^{i-1})$
- ❖ Are those two algorithms equivalent?





Broadcast tree: version 1

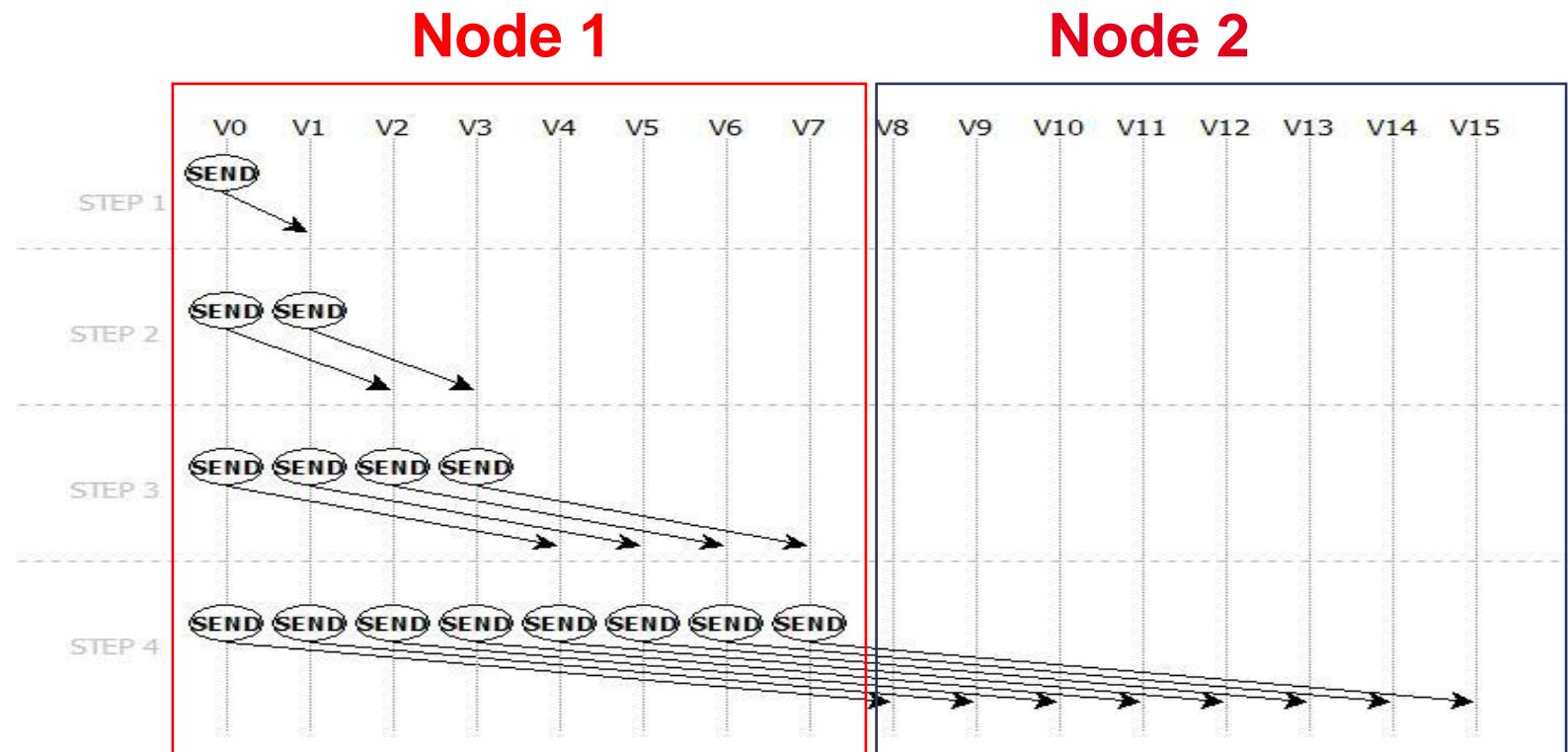
- ❖ If we consider two nodes





Broadcast tree: version 1

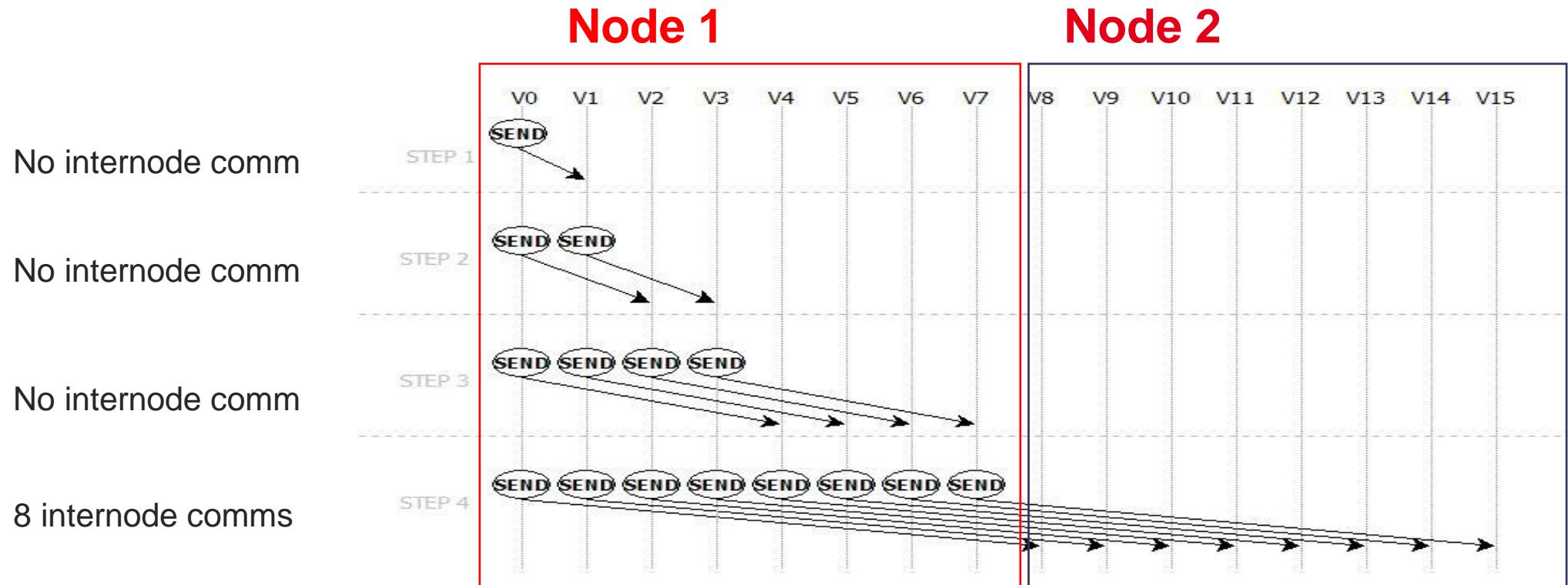
- ❖ If we consider two nodes





Broadcast tree: version 1

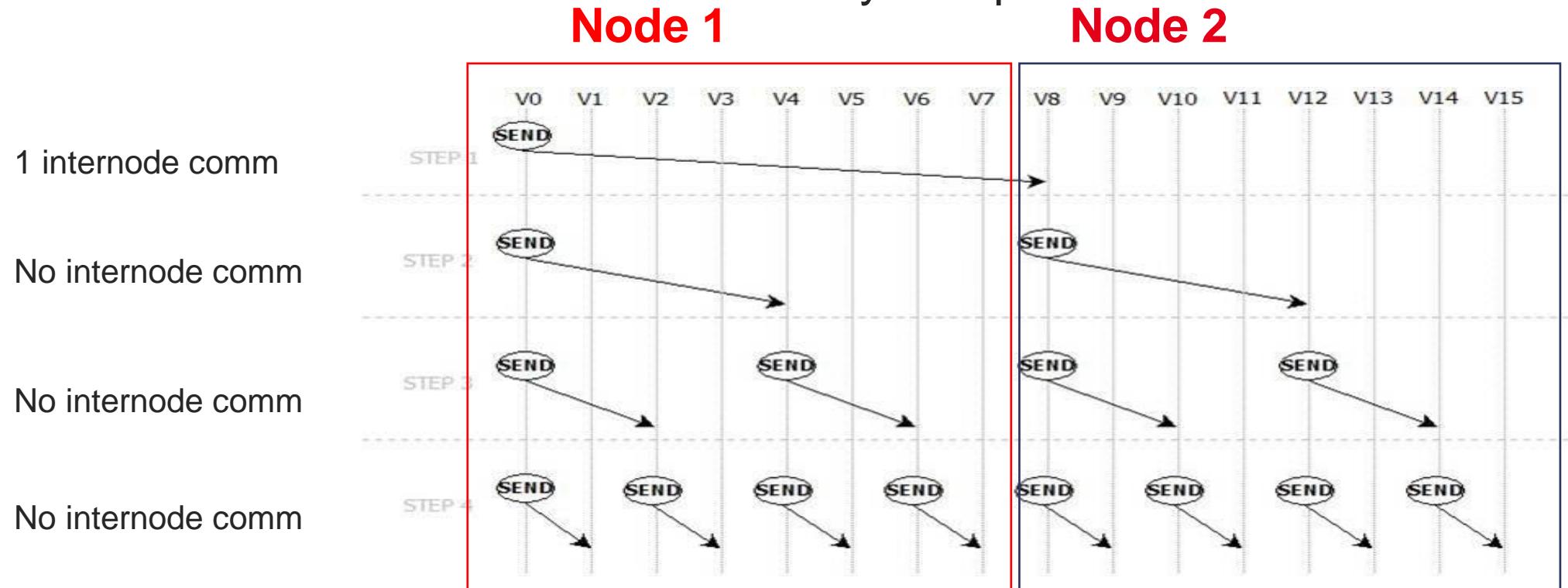
- ❖ If we consider two nodes
- ❖ All internode communications (network) happens at the same step





Broadcast tree: version 2

- ❖ Only one internode communication
- ❖ No contention on the network
- ❖ Intranode communications may be optimized





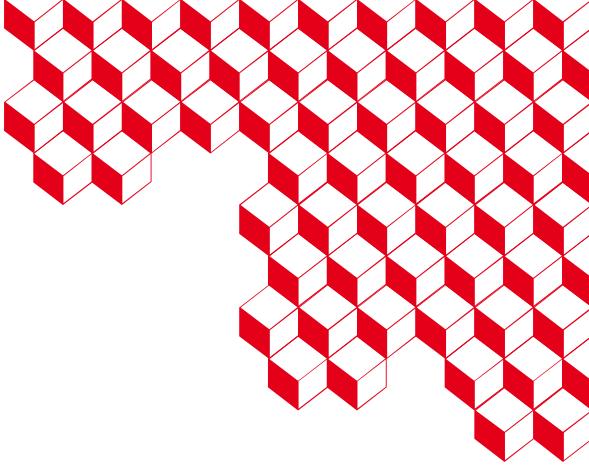
Collective algorithms

- ❖ Several algorithms may be implemented for one collective in a runtime
- ❖ Default selection of algorithms often depends on the size of the messages, and the number of MPI processes
- ❖ Selection may be available to user
 - launch options, environment variables, communicator attributes...



Collective algorithms

- ❖ The best collective algorithm also depends on the underlying topology
- ❖ Example: allgather
 - Ring algorithm have more steps ($2N-1$) but one comm. in each step
 - Parallel linear algorithm have less steps ($N-1$) but more comm. at each step
 - If the underlying topology cannot absorb all the messages, contention will make it slower



High-speed network topologies



Network topologies

- ❖ In a supercomputer, the compute nodes are connected through a high-speed network
- ❖ Each node doesn't have a direct link to all other nodes it needs to communicate with
 - Too expensive
- ❖ Network design must be optimized for a good trade-off cost / efficiency



Network topologies

- ❖ Two **performance criteria** for a network connecting n nodes:
 - Mean number of links to connect 2 nodes : n_L
 - Bissection Bandwidth (bandwidth between the two halves of the whole network): B_b

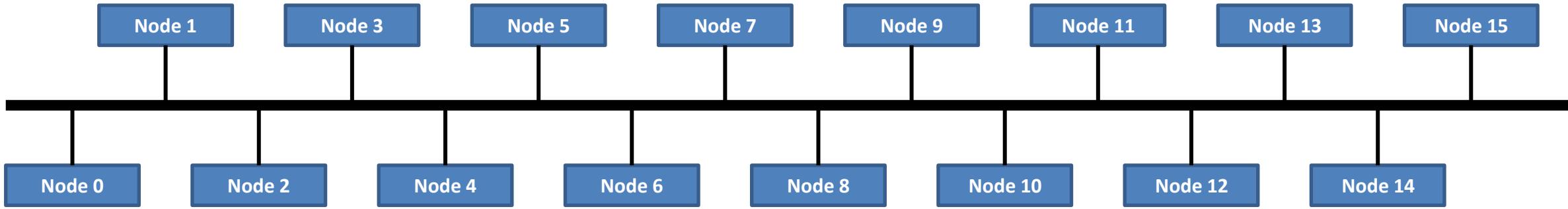


Network topologies

- ❖ Numerous different network topologies exist
 - Bus, Ring
 - Grid, Torus
 - Trees
 - Hypercube
 - Butterfly, Dragonfly
 - ...



Bus

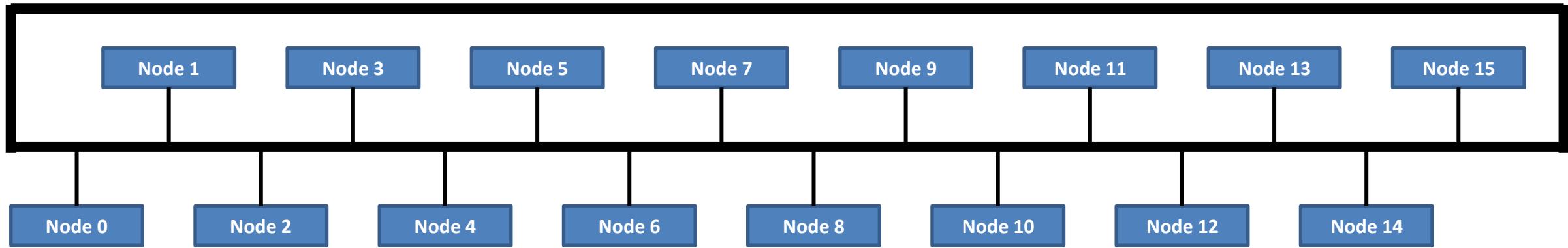


❖ $n_L \sim n/3$

❖ $B_b 1$



Ring



❖ $n_L \sim n/4$

❖ $B_b 2$



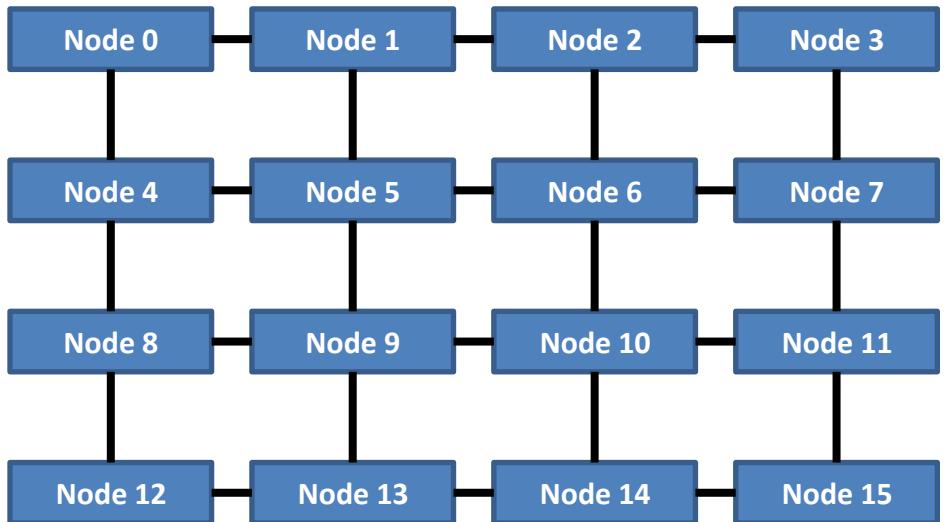
Characteristics

- ❖ Low cost
- ❖ Low adaptability
- ❖ Low bandwidth
- ❖ Low scalability
- ❖ High latency
- ❖ → Not used in HPC



Grid

- ❖ $n_L \sim \sqrt{n}$
- ❖ $B_b = \sqrt{n}$





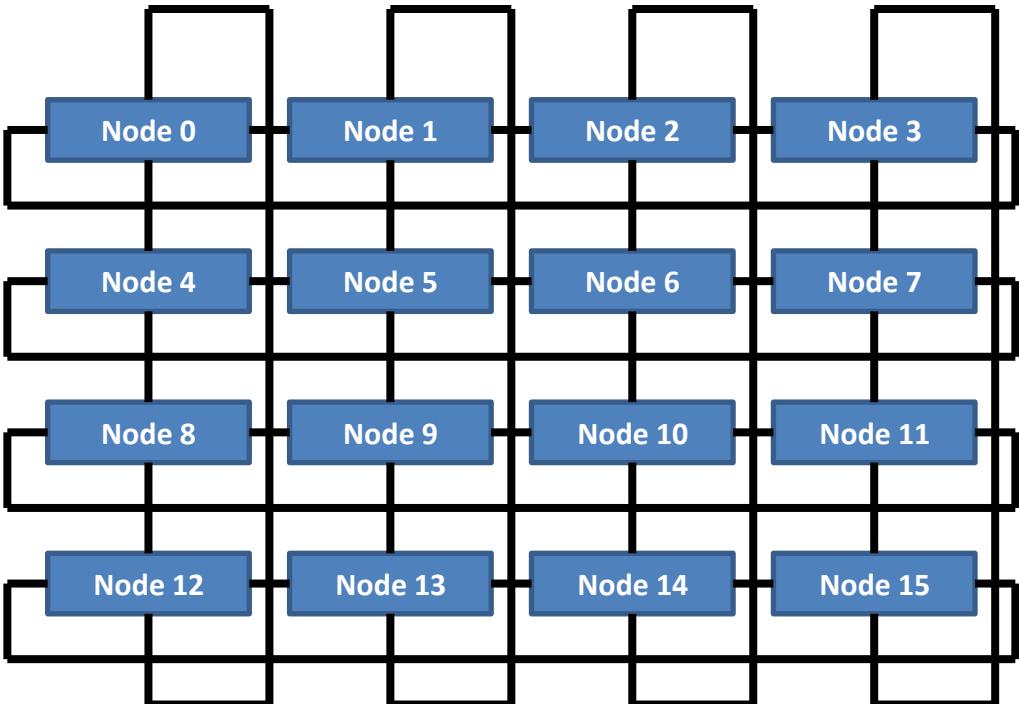
Characteristics

- ❖ Limitate number of neighbors:
 - Lower cost
 - Lower latency
- ❖ Easy to design
- ❖ Each link is the same
- ❖ Irregular on the boarders of the topology
 - Codes need to adapt to preserve communications locality
- ❖ Require an optimized policy from the ressource manager/allocator
 - Groups the nodes of the same application



Torus

- ❖ $n_L \sim \sqrt{n}/2$
- ❖ $B_b = 2\sqrt{n}$



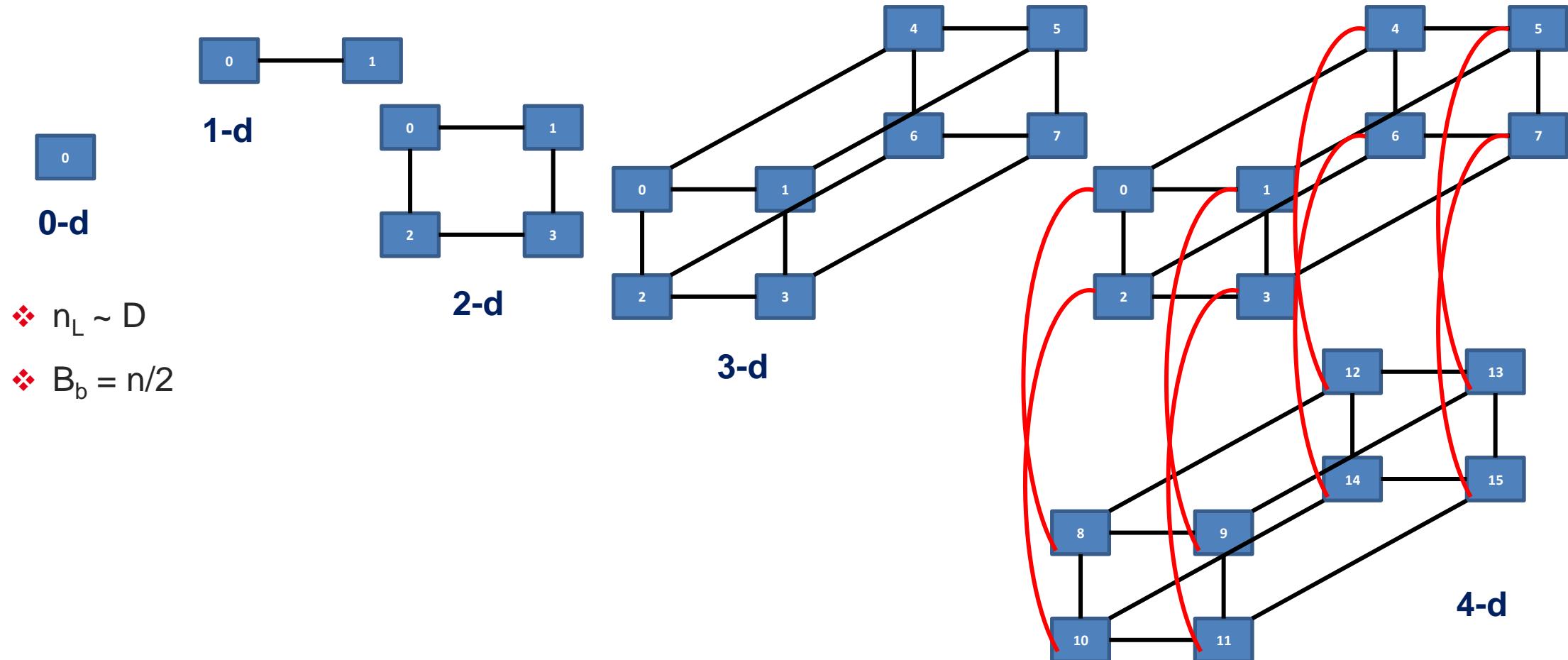


Characteristics

- ❖ Nearly the same as the Grid
- ❖ In addition, not irregular on the « boarders »...
- ❖ ... except the lenght of the connecting links
- ❖ Hard to design and to install
- ❖ Used in :
 - Fugaku Tofu-D(#4 top500)
 - BG/L, BG/P



Hypercube with D-dimensions



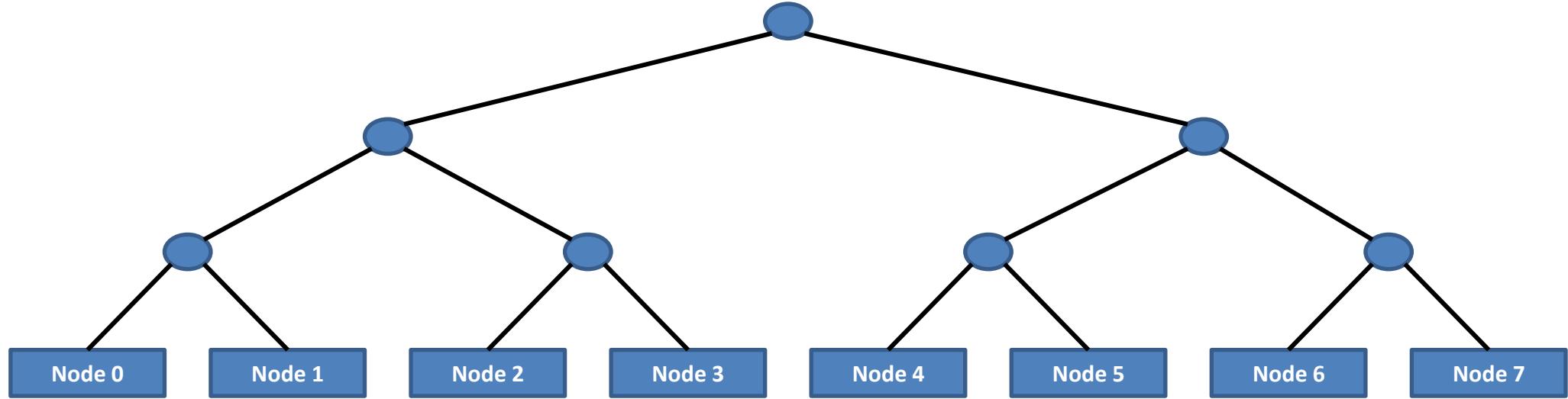


Characteristics

- ❖ Limitate number of neighbors:
 - Lower cost
 - Lower latency
- ❖ Irregular on the boarders of the topology
 - Codes need to adapt to preserve communications locality
- ❖ Require an optimized policy from the ressource manager/allocator
 - Groups the nodes of the same application
- ❖ Harder to design on a 2D or 3D chip
- ❖ Used in:
 - SGI



Tree



❖ $n_L \sim \log(n)$

❖ $B_b = 1$

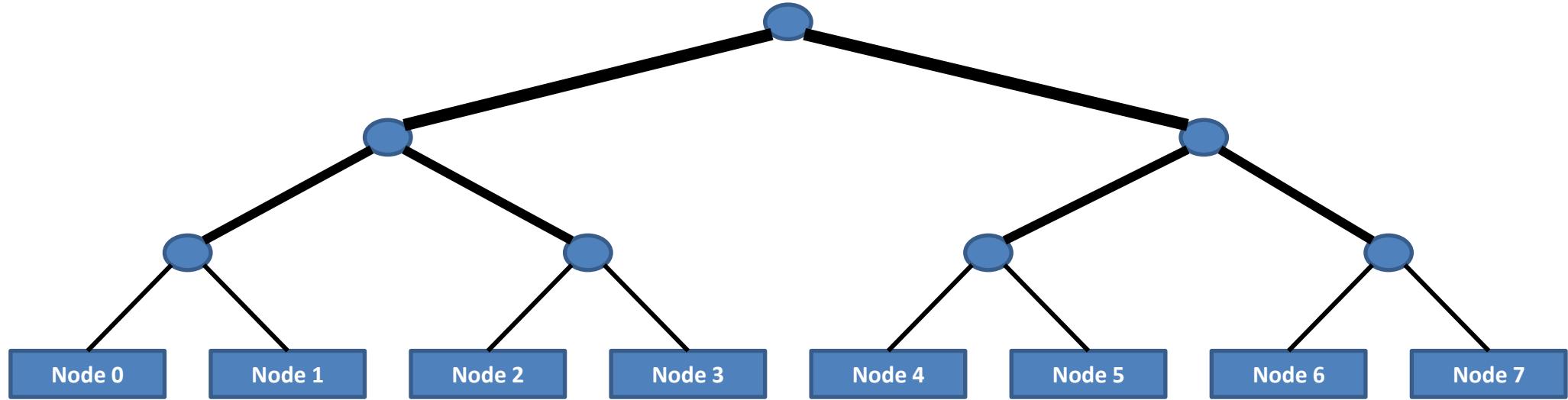


Characteristics

- ❖ Average cost
- ❖ Low latency
 - Few hops between nodes
- ❖ Easy to design
- ❖ Very good for local traffic...
- ❖ ... but the root can be overloaded if there are lots of long-distance messages
 - congestion



Fat-tree



- ❖ $n_L \sim \log(n)$
- ❖ $B_b = n/2 \dots ?$

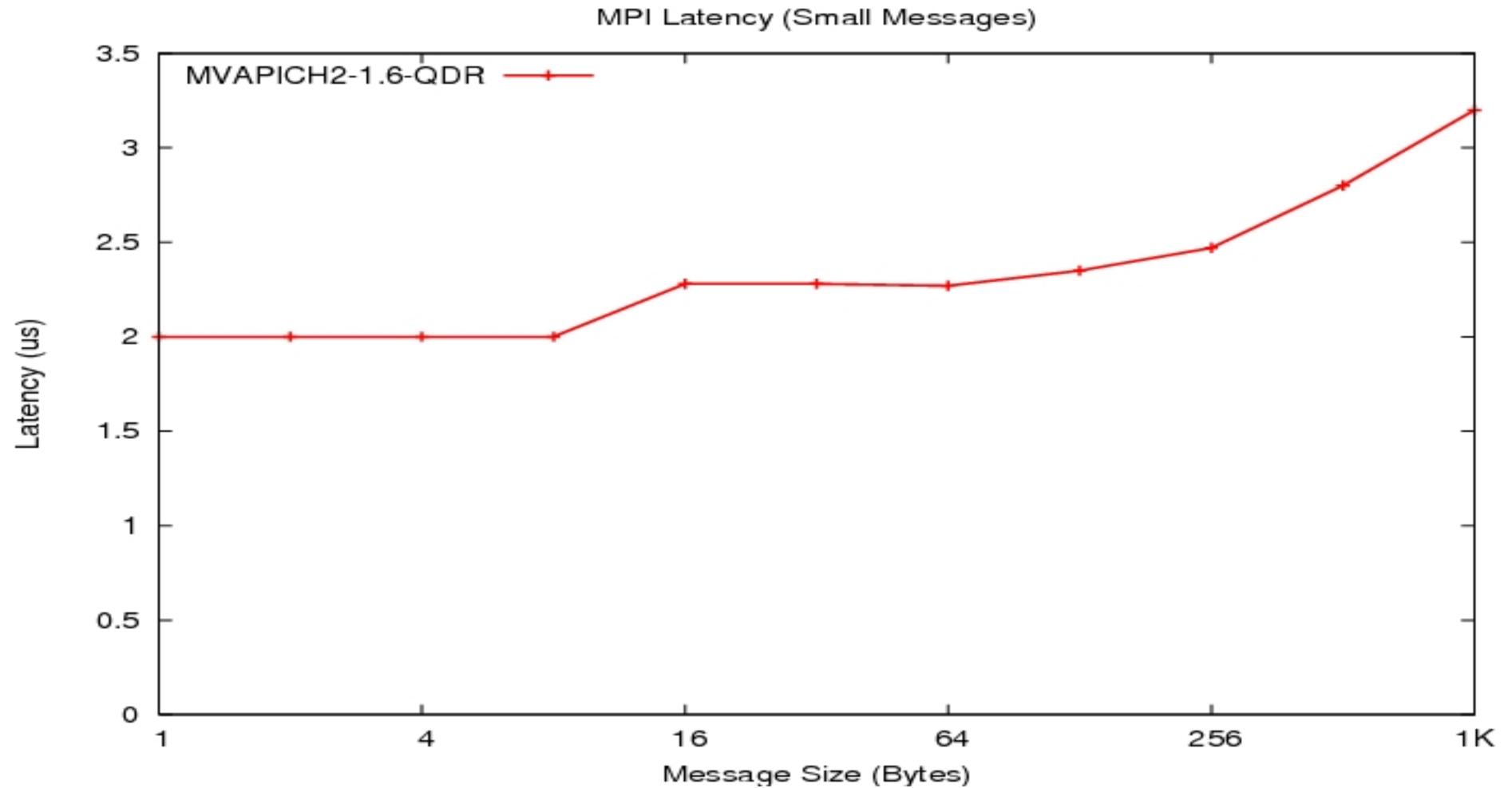


Characteristics

- ❖ High cost
- ❖ Each level has more physical links than the previous one
- ❖ → More bandwidth at the root to avoid congestion
- ❖ Greedy resource management policy
 - Often used because it reduces the user waiting time
- ❖ Reality less ideal than theory
 - Network switches are very loaded
- ❖ Used in
 - TERA 100, TERA 1000
 - Leonardo (#4), Summit (#5), Sierra (#6), Selene (#9) infiniband

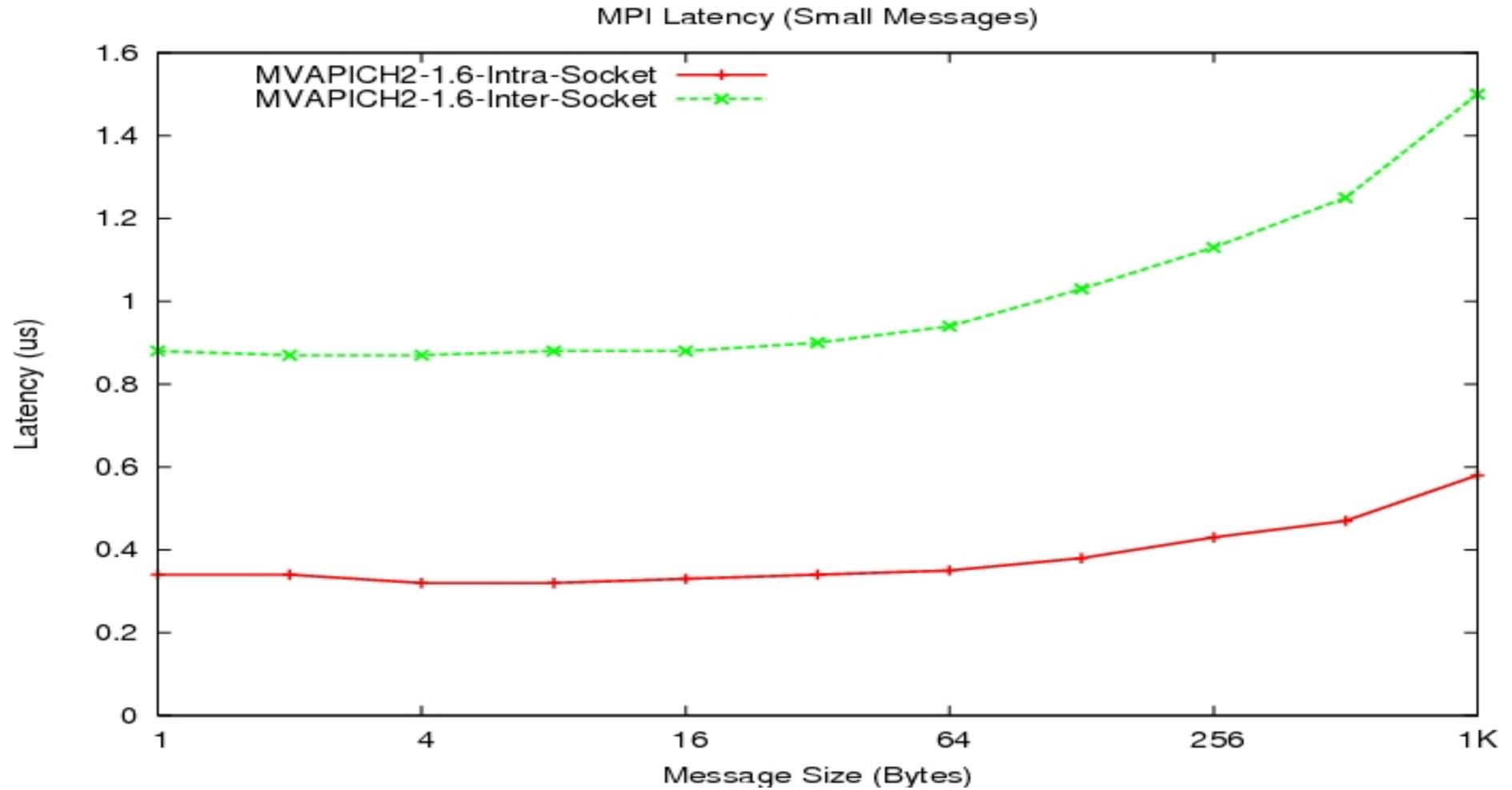


Latency evaluation



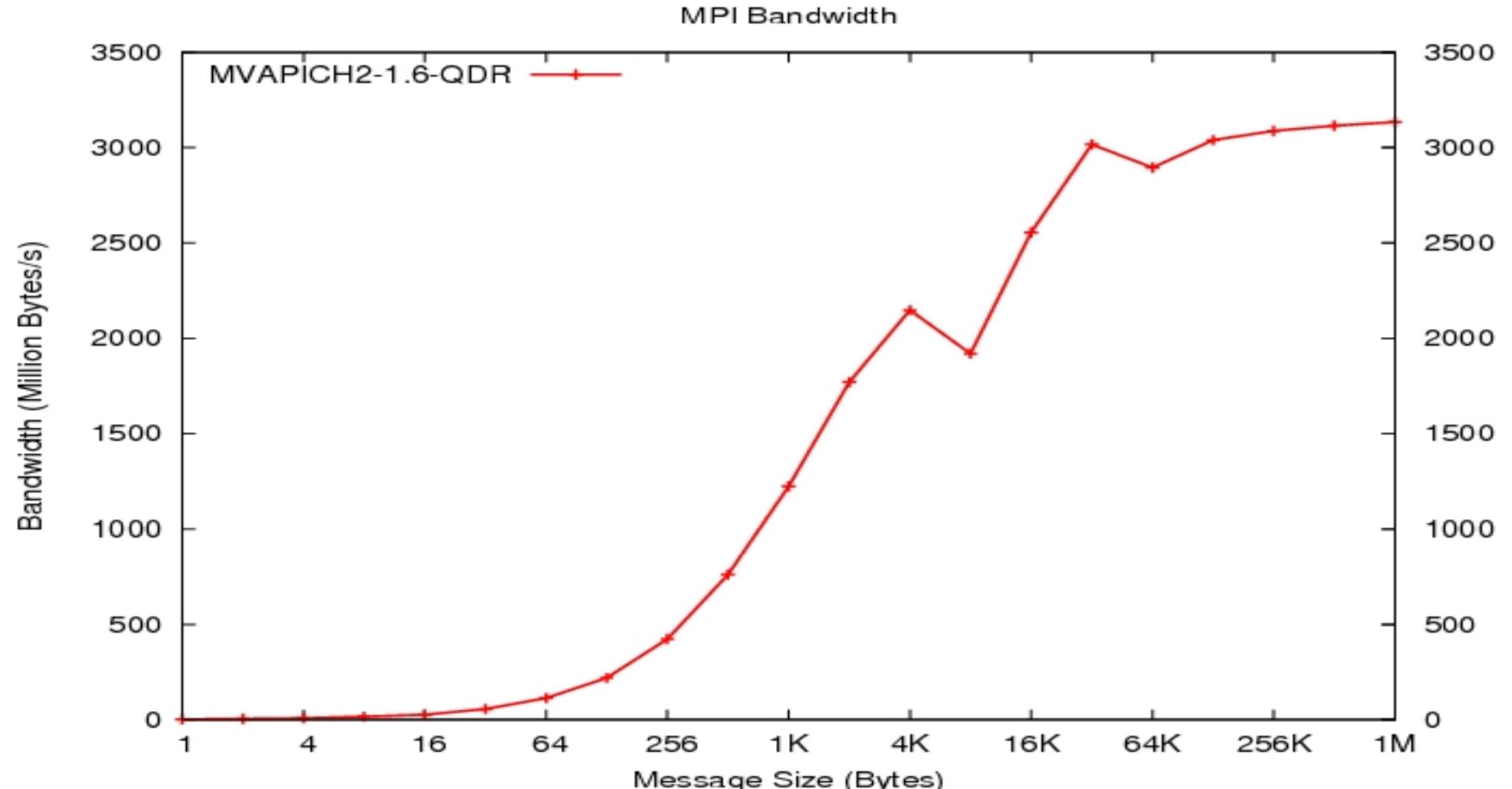


Latency evaluation



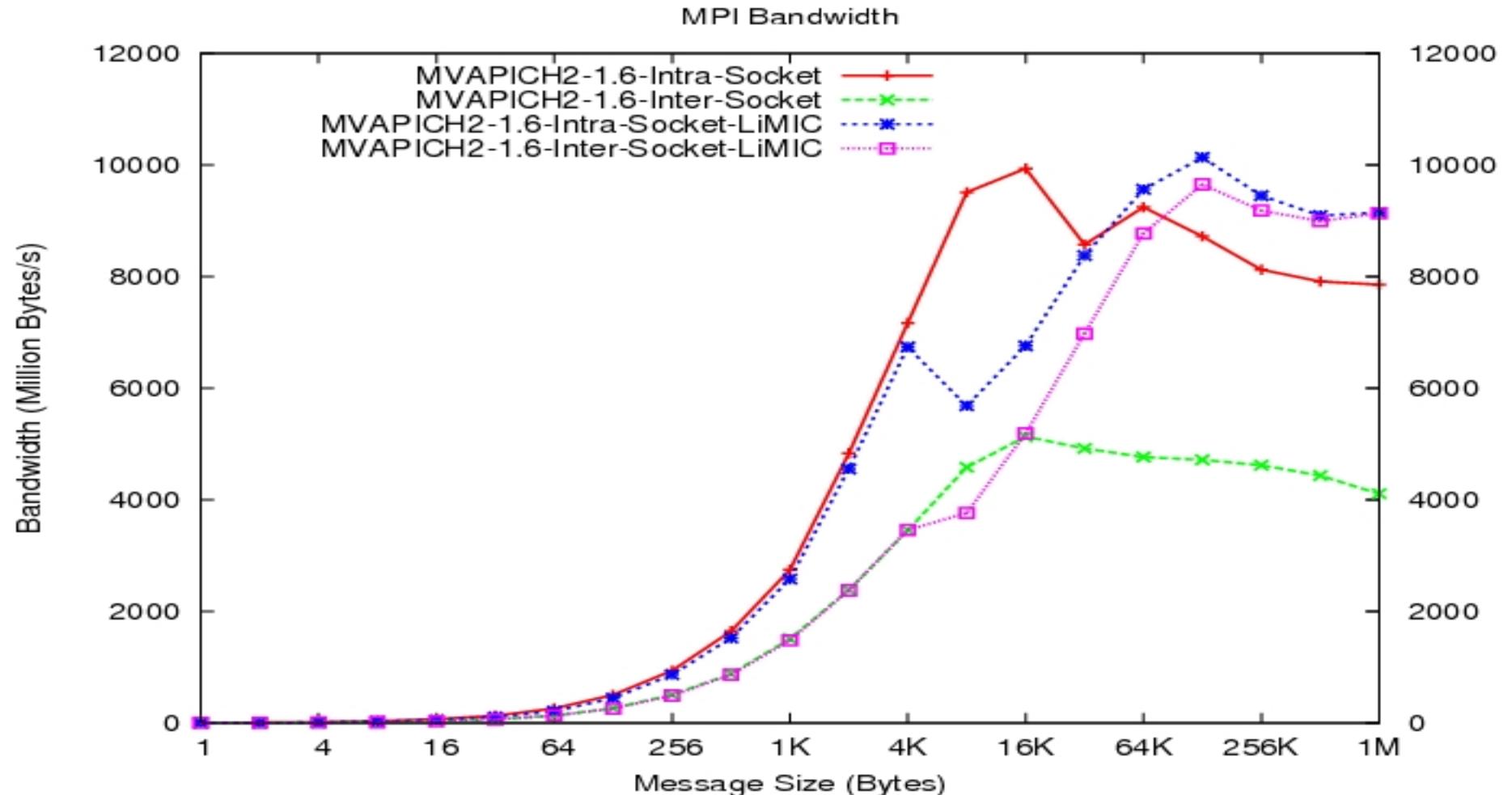


Bandwidth evaluation



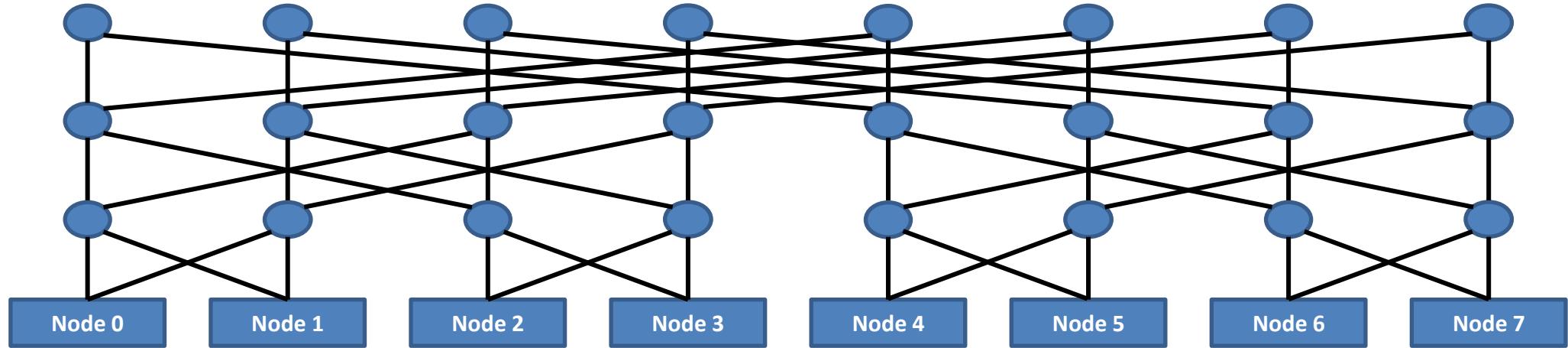


Bandwidth evaluation





Other topologies: Butterfly



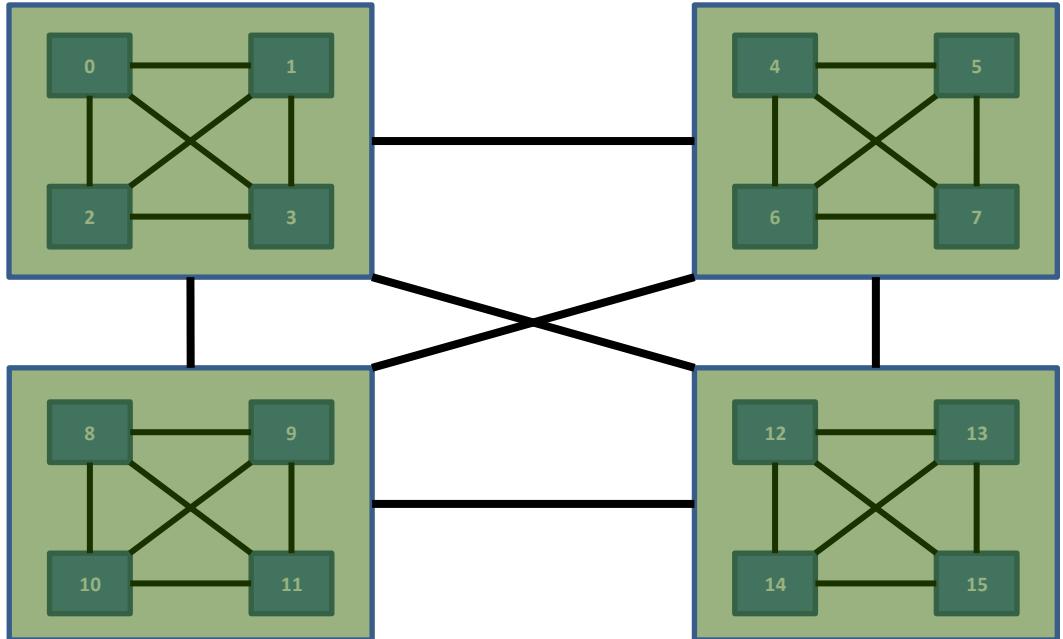
- ❖ $n_L \sim \log_k(n+1)$ (ici $\log_2(n+1)$)
- ❖ $B_b = n/2$



Other topologies: dragonfly

- ❖ Full-connected graph of full-connected graphs
- ❖ $n_L \sim 3$
- ❖ $B_b = (n\sqrt{n})/4$

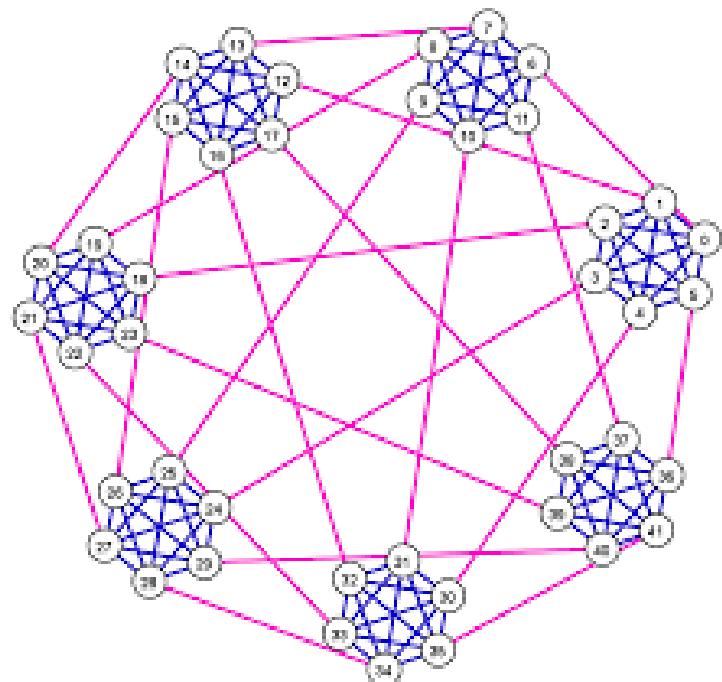
- ❖ Used in:
 - Slingshot : Frontier (#1), LUMI (#3), Perlmutter (#8), Adatra (#11)





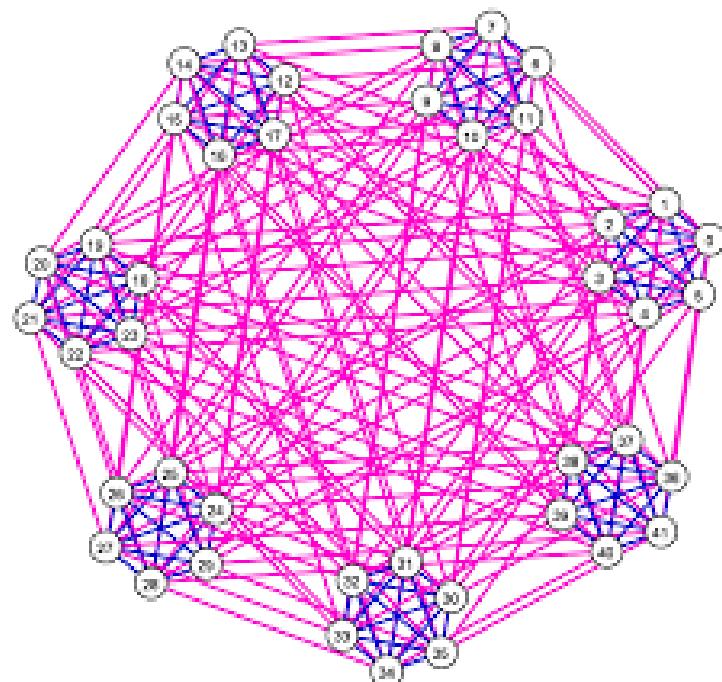
Other topologies: dragonfly

Simple Dragonfly



“Megafly”

dragonfly with replicated links





General remarks

- ❖ Network topology can impact which algorithm can be the best
 - Most MPI implementations don't consider hardware topology in the algorithm choice (too costly)
- ❖ Network topology also impacts resource allocation
 - Most MPI implementations also don't consider how resources (nodes) have been allocated to the current job
- ❖ Moreover, how ranks are given to MPI processes can depend on:
 - Network topology
 - Resource allocation



Resource manager (1)

- ❖ Slurm: Simple Linux Utility for Resource Management
 - *open source*
 - Allows to create sub-clusters (partitions) in Lunux with some fault-tolerance
 - ip-failover
 - Also schedules the job on the physical resources of the supercomputer.
- ❖ It can be used on clusters of various sizes, from two to hundred thousand nodes.

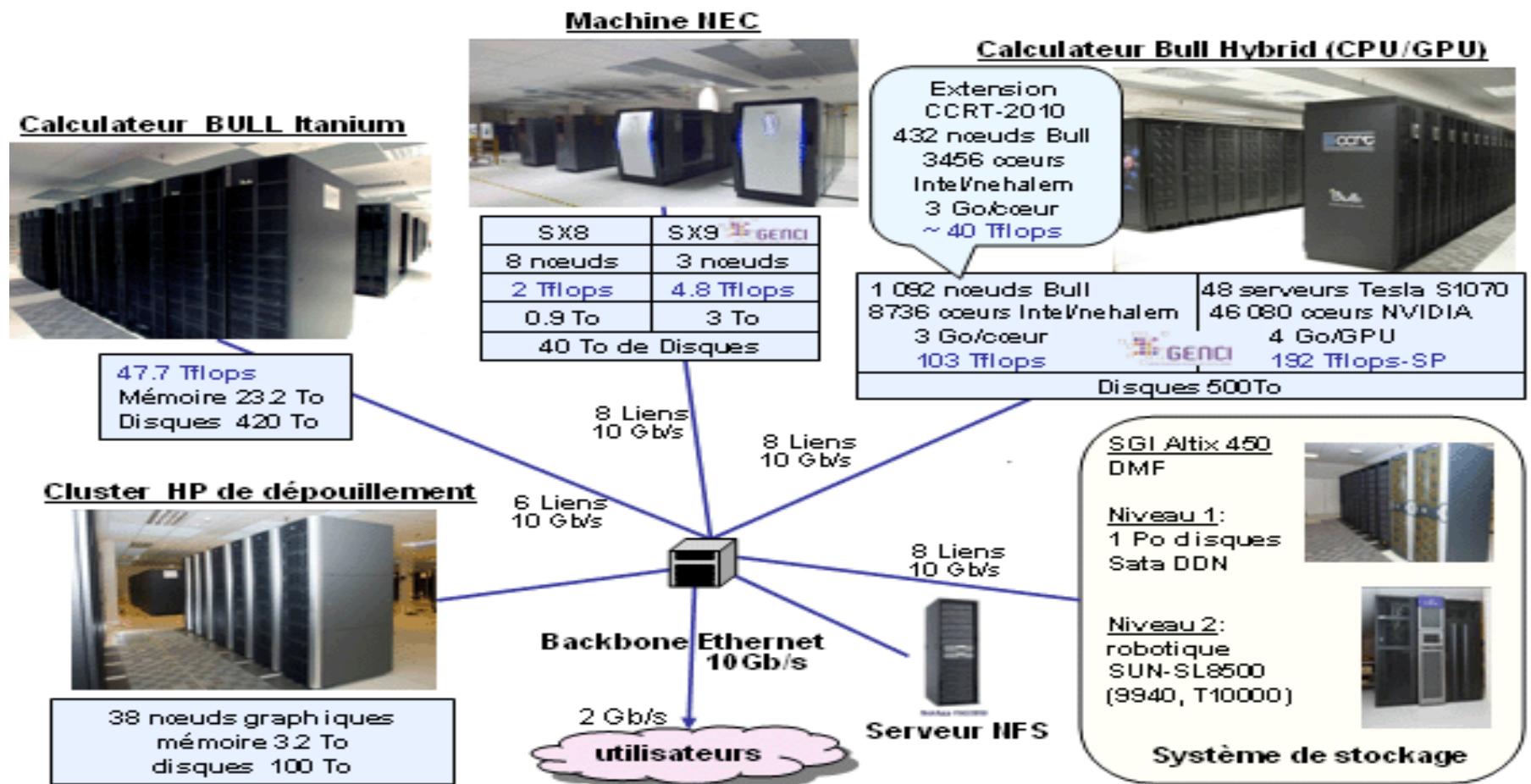


Resource manager (2)

- ❖ Slurm doesn't require any modification of the Linux kernel to work and is autonomous
- ❖ As resource manager on clusters, Slurm has three main functionnalities:
 - Allocate and give access to resources for all users, for a limited amount of time so they can run their jobs ;
 - Provide mechanisms to launch, execute and follow the status of the jobs on the allocated resources ;
 - Schedule the access to resources and the jobs by managing a waiting list.



Example: CCRT





Resource manager (3)

- ❖ The resource manager do its best regarding the scheduling of jobs and the demands for resources
- ❖ There are no guarantees that the provides resources (nodes) are adjacent and will benefit from the underlying topology
 - You get what is available
 - Not possible to deallocate/migrate running jobs to free resources and have a better locality
- ❖ It is the user's responsibility to detect and adapt to the actual topology of allocated resources



To conclude on MPI Optimisation

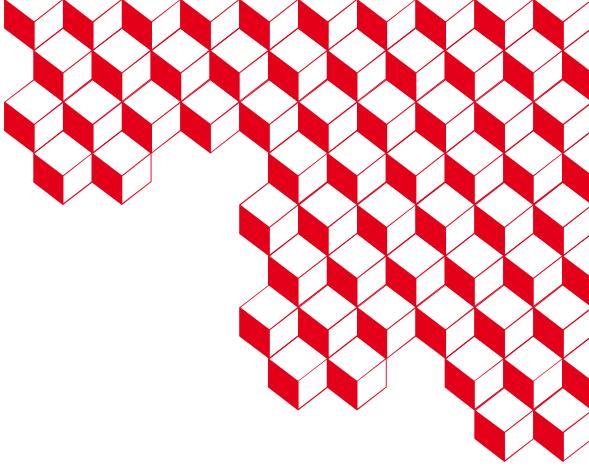
- ❖ Know your MPI implementation
- Which functions are optimized

- ❖ Be careful about your parallel complexity
- Domain decomposition
- Algorithms parallel steps concurrency

- ❖ Be aware of topology



10.■ Placement



OpenMP placement



OpenMP threads placement

- ❖ Some APIs exist to bind threads OpenMP to locations
- ❖ The OpenMP standard integrate its own mechanism to bind threads: Places

- ❖ Implementations of the OpenMP standard may also provide their own API
- INTEL: KMP_AFFINITY=[<modifier>, ...] <type> [,<permute>] [,<offset>]



OpenMP Places 1st step

- ❖ Define places list in environment variable OMP_PLACES
- ❖ OpenMP provide two ways to specify places in the environment variable
 - Abstract name describing a set of places
 - Explicit list of places described by non-negative integers
- ❖ Predefined name in OpenMP standard
 - Threads: each place correspond to one HW thread
 - Cores: each place correspond to one core
 - Sockets : each place correspond to one socket



OpenMP Places 1st step

- ❖ Extract of grammar to describe values in the OMP_PLACES environment variables
 - <p-list> |= <p-interval> | <p-list>,<p-interval>
 - <p-interval> |= <place>:<len>:<stride> | <place>:<len>
 - See full grammar in OpenMP v5.0 p 606, Section 6.5 OMP_PLACES

- ❖ The following examples are equivalent

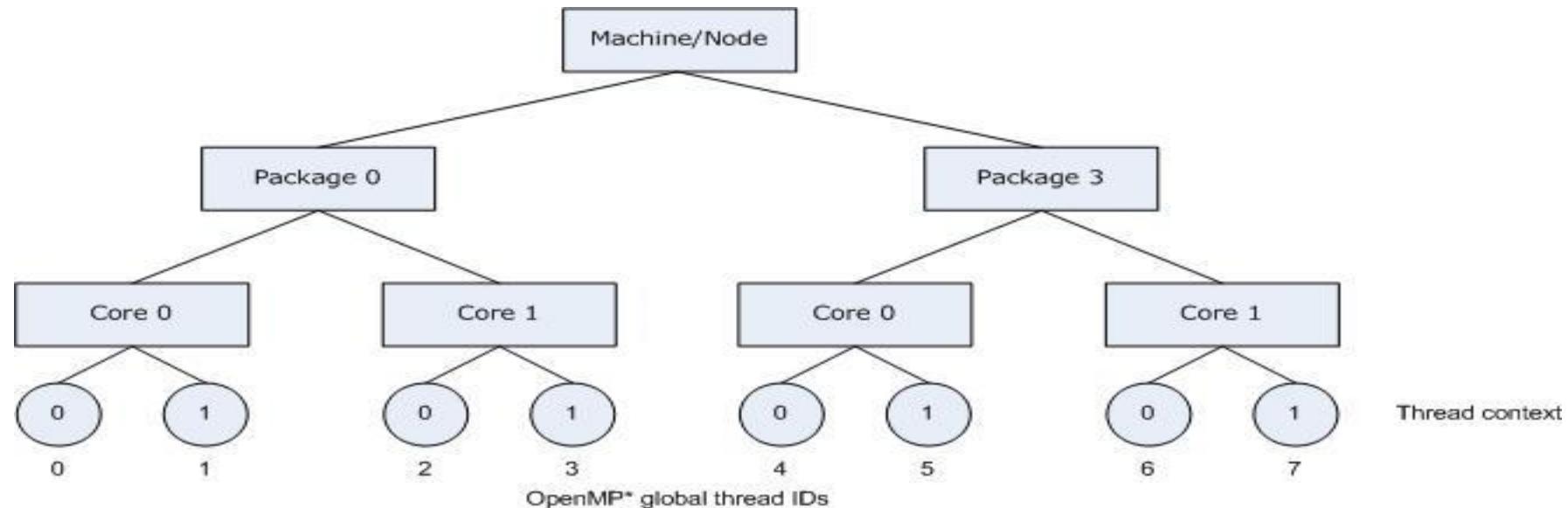
```
setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"
setenv OMP_PLACES "{0:4}:4:4"
```



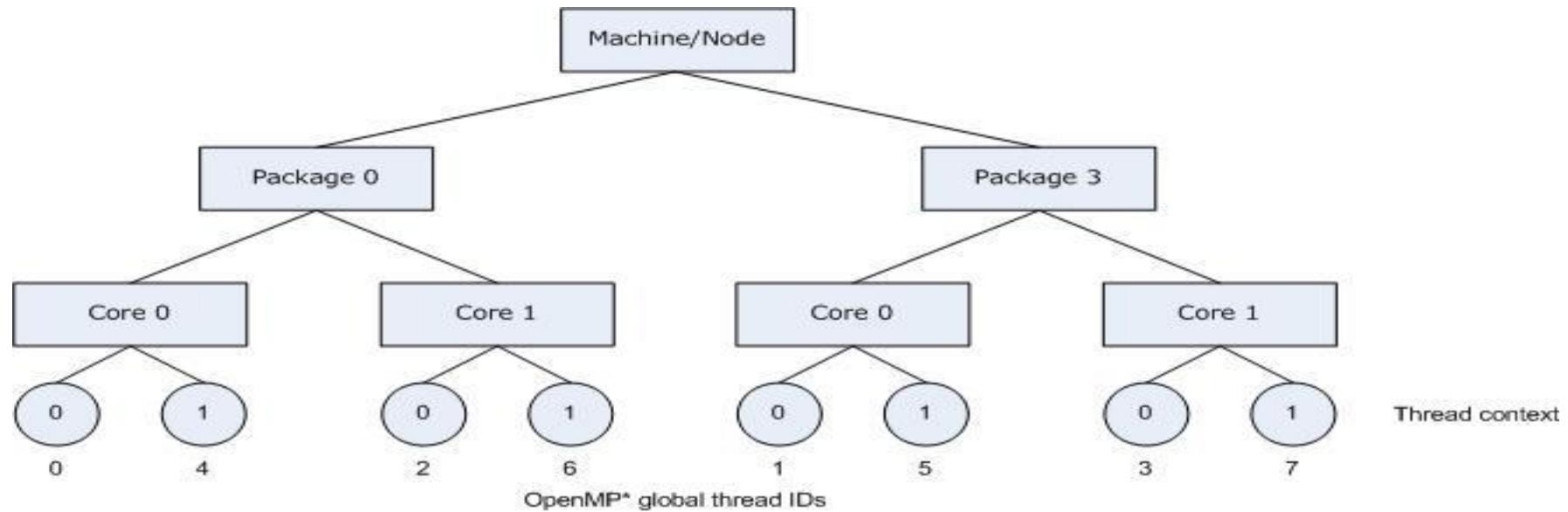
OpenMP Places 2nd step

- ❖ Define the binding policy to apply on those places
- ❖ Two possibilities:
 - OMP_PROC_BIND environment variables
 - Spread, close, master, true, false
 - Meaning is implementation defined
 - *proc_bind* clause
 - Master: each thread in the team is bound to the same place as the master
 - Close: place the threads in the team close to the parent
 - Spread: spread the threads on the available places

API KMP_AFFINITY: Compact



API KMP_AFFINITY: Scatter





OMP places example

- ❖ Same code as bandwidth example
- ❖ Thread 1 initializes the arrays
- ❖ Thread 0 computes the addition in parallel

- ❖ OMP places = cores
 - ❖ Binding hints apply at core level
- ❖ 2 tests
 - ❖ Close: thread 1 is next to thread 0
 - ❖ Spread: thread 1 is on the other socket



OMP places example

```
#pragma omp parallel proc_bind(spread)
{
    int i;
    int thid = -1;

    thid = omp_get_thread_num();

    if(thid == THREAD_ALLOC)
    {
        printf("thread allocated == %d\n", thid);
        for(i=0; i<GSIZE; i++)
        {
            a[i] = i+1;
            b[i] = GSIZE-i-1;
            r[i] = 0;
        }
    }
    struct timeval tv_start;
    struct timeval tv_stop;

    gettimeofday(&tv_start, NULL);

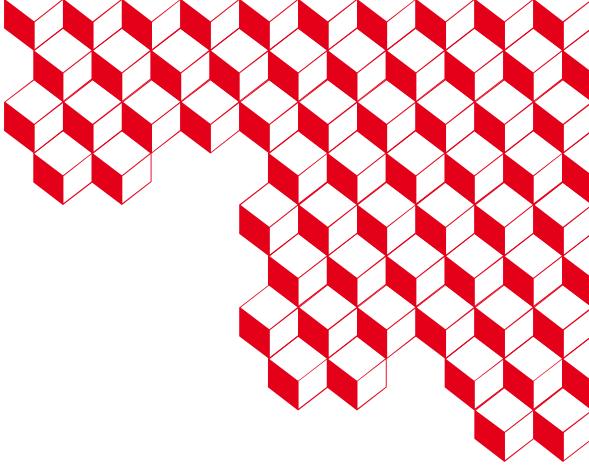
    #pragma omp parallel proc_bind(spread)
```

```
#pragma omp parallel proc_bind(spread)
{
    int i;
    int j;
    int thid = -1;
    thid = omp_get_thread_num();

    if(thid < (thnb/2))
    {
        int loop_b = (GSIZE / (thnb/2)) * thid;
        int loop_e = (GSIZE / (thnb/2)) * (thid+1);

        for(j=0; j< REPET; j++)
        {
            for(i=loop_b; i<loop_e; i++)
            {
                r[i] = a[i] + b[i];
            }
        }
    }
}
```

```
(c-inti)[jaegerj@inti2216 bandwidth_numa] $ OMP_NUM_THREADS=2 OMP_PLACES=cores ./bandwidth_test_local_close.pgr
thread allocated == 1
time for r[i]=a[i]+b[i] with allocation by thread 1 is: 7s 626206us
(c-inti)[jaegerj@inti2216 bandwidth_numa] $ OMP_NUM_THREADS=2 OMP_PLACES=cores ./bandwidth_test_local_spread.pgr
thread allocated == 1
time for r[i]=a[i]+b[i] with allocation by thread 1 is: 8s 80657us
(c-inti)[jaegerj@inti2216 bandwidth_numa] $
```



MPI placement



MPI and Slurm

- ❖ Srun !

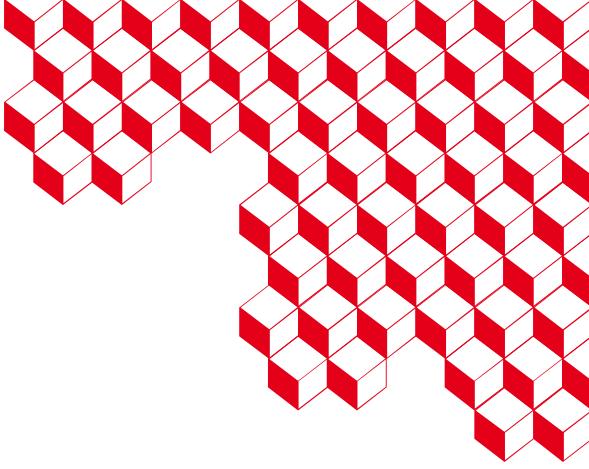


11 ■ Scalability



Parallel Optimisation

- ❖ Writing and optimizing a parallel program is not trivial
- ❖ A few keys to avoid crippling issues for your parallel performance
 - **Think parallel:**
 - use a parallel efficient algorithm instead of trying to parallelize a sequential algorithm
 - **Follow Amdahl:**
 - Reduce the amount of sequential parts in you programs
 - Avoid synchronizations in your programs



Amhdal Law

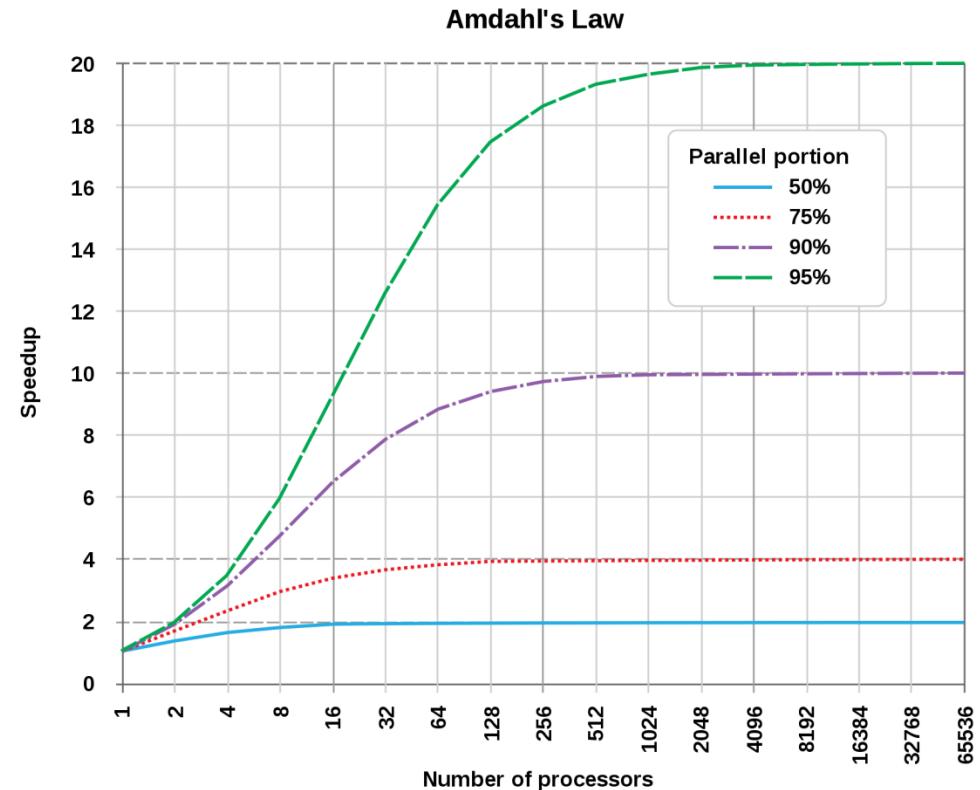


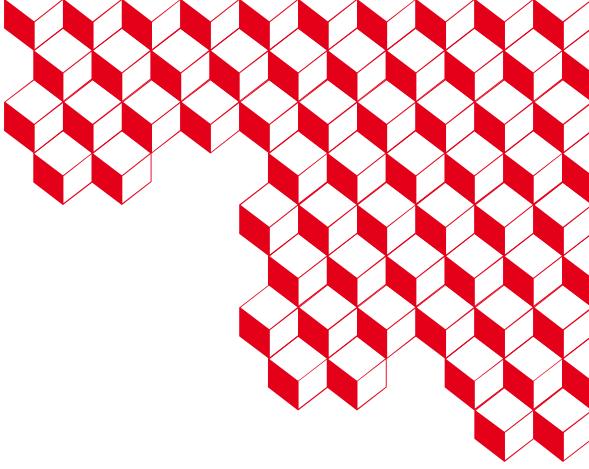
Amhdal Law (Wikipedia example)

- ❖ Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours to complete using a single thread, but a one-hour portion of the program cannot be parallelized, therefore only the remaining 19 hours' ($p = 0.95$) execution time can be parallelized, then regardless of how many threads are devoted to a parallelized execution of this program, the minimum execution time is always more than 1 hour. Hence, the theoretical speedup is less than 20 times the single thread performance

- N : number of processors
- p : proportion of parallel code

$$\text{Speedup} = \frac{1}{(1 - p) + p/N}$$





Strong Scaling and Weak scaling



Parallel scalability

- ❖ The scalability of an algorithm is linked to its parallel efficiency
- ❖ Two ways of evaluating the parallel efficiency
 - Strong scaling
 - Weak scaling



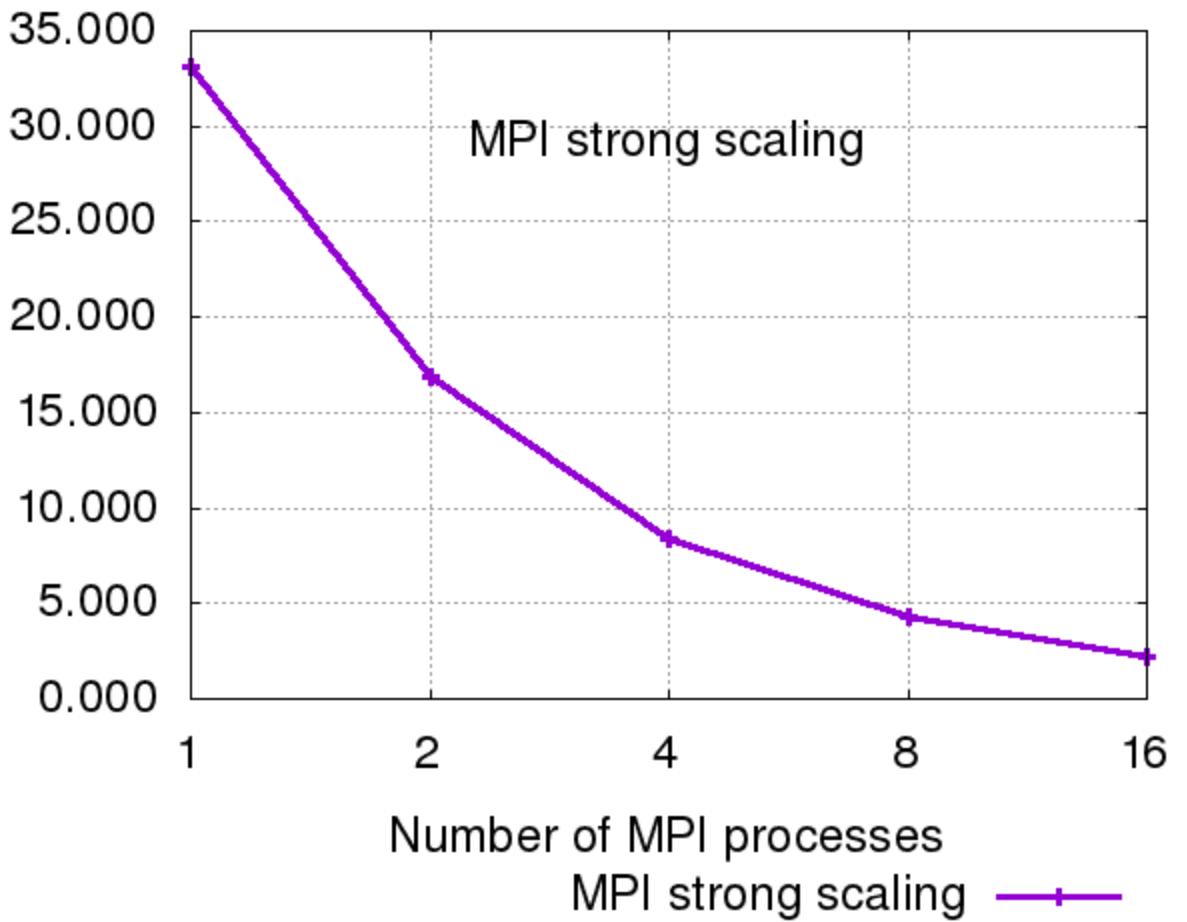
Strong scaling

- ❖ Strong scaling: keep the same global amount of work
 - When increasing the number of compute units, decrease the amount of work per compute units
- ❖ With perfect strong scaling, the speed-up is directly linked to the new compute units available
 - If compute units x2, work done per compute units / 2 => performance x2



Strong scaling Example

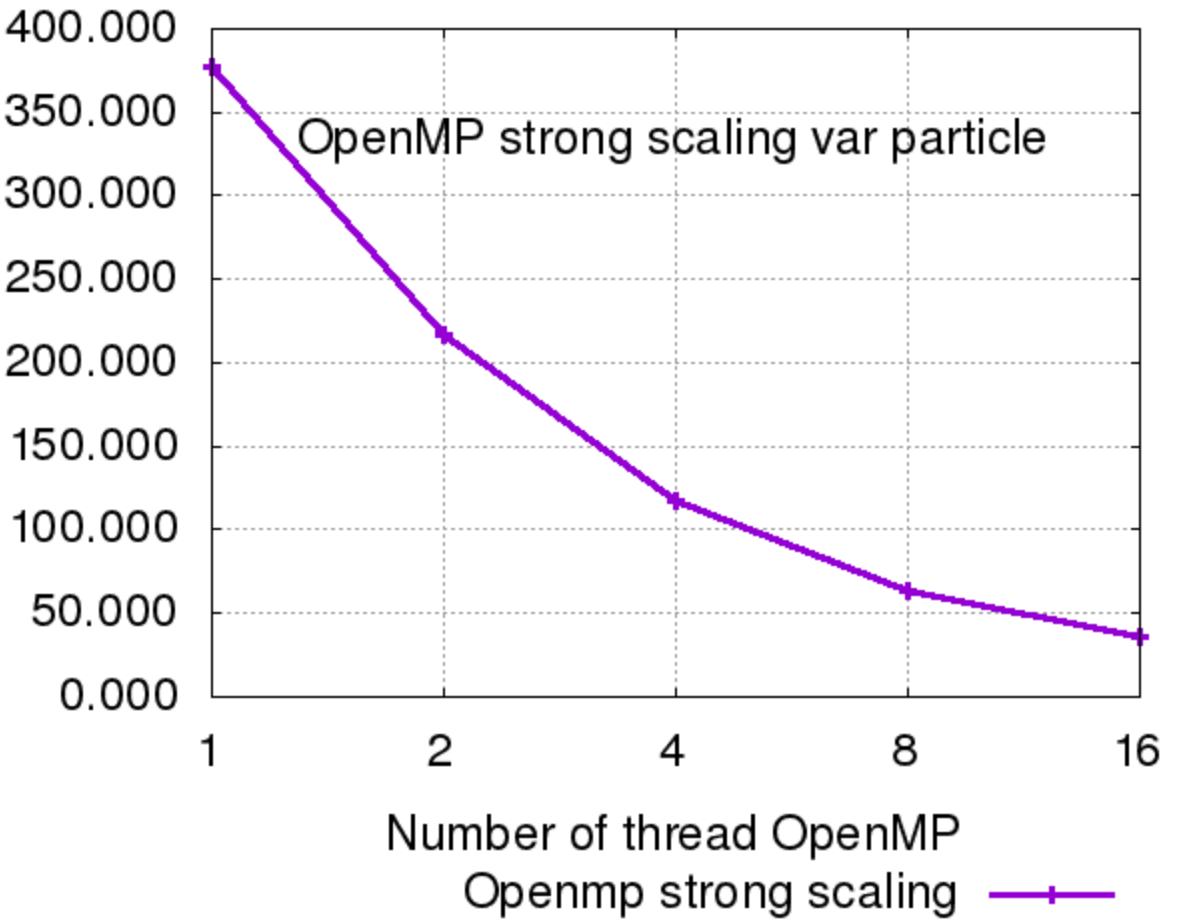
- ❖ Example of nearly perfect strong scaling
 - MPI very simple in the code of the example





Strong scaling Example

- ❖ Example of less perfect strong scaling
 - OpenMP has load-balancing issues in this example





Weak scaling

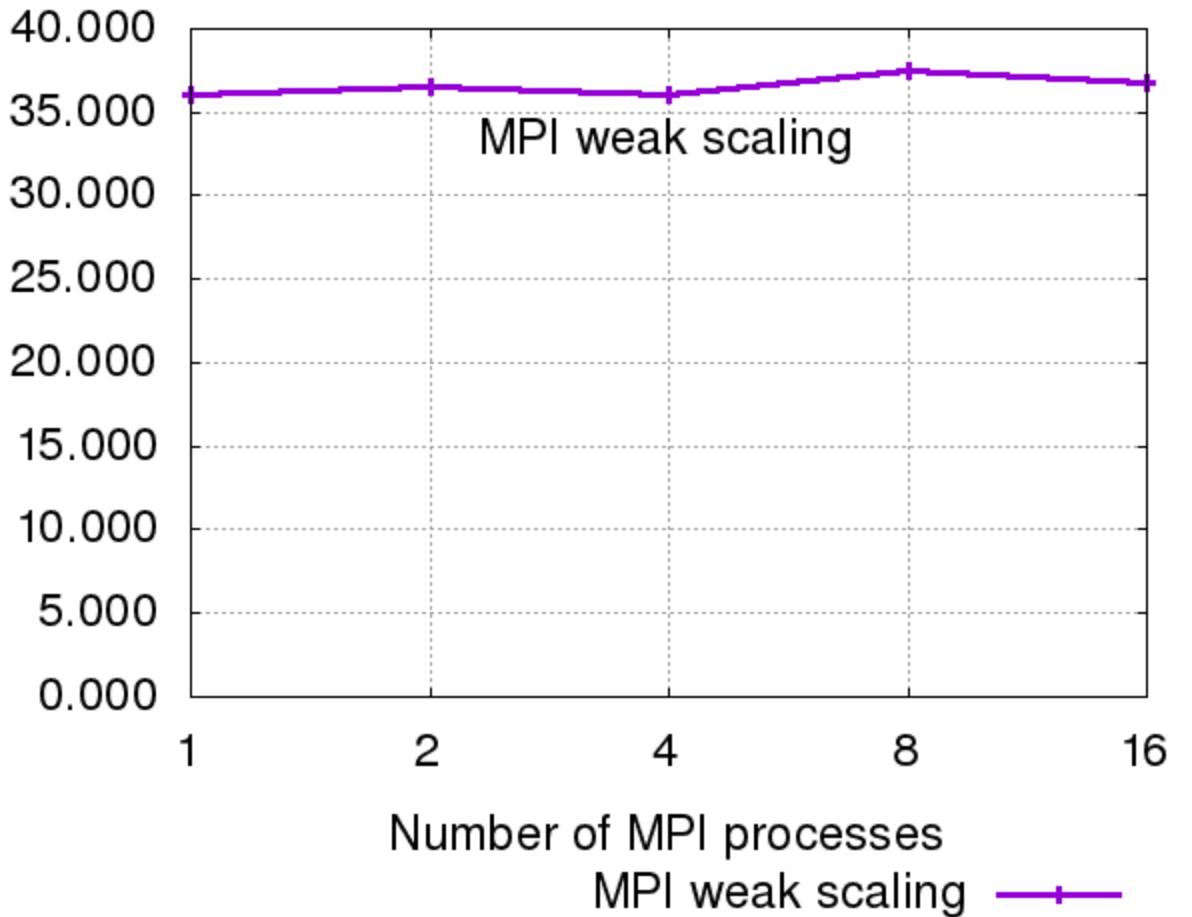
- ❖ Weak scaling: keep the same amount of work per compute units
 - When increasing the number of compute units, increase the global amount of work

- ❖ With perfect weak scaling, the execution time stays the same while the work done is increased
 - If compute units x2, global work done x2 in the same time => performance x2



Weak scaling

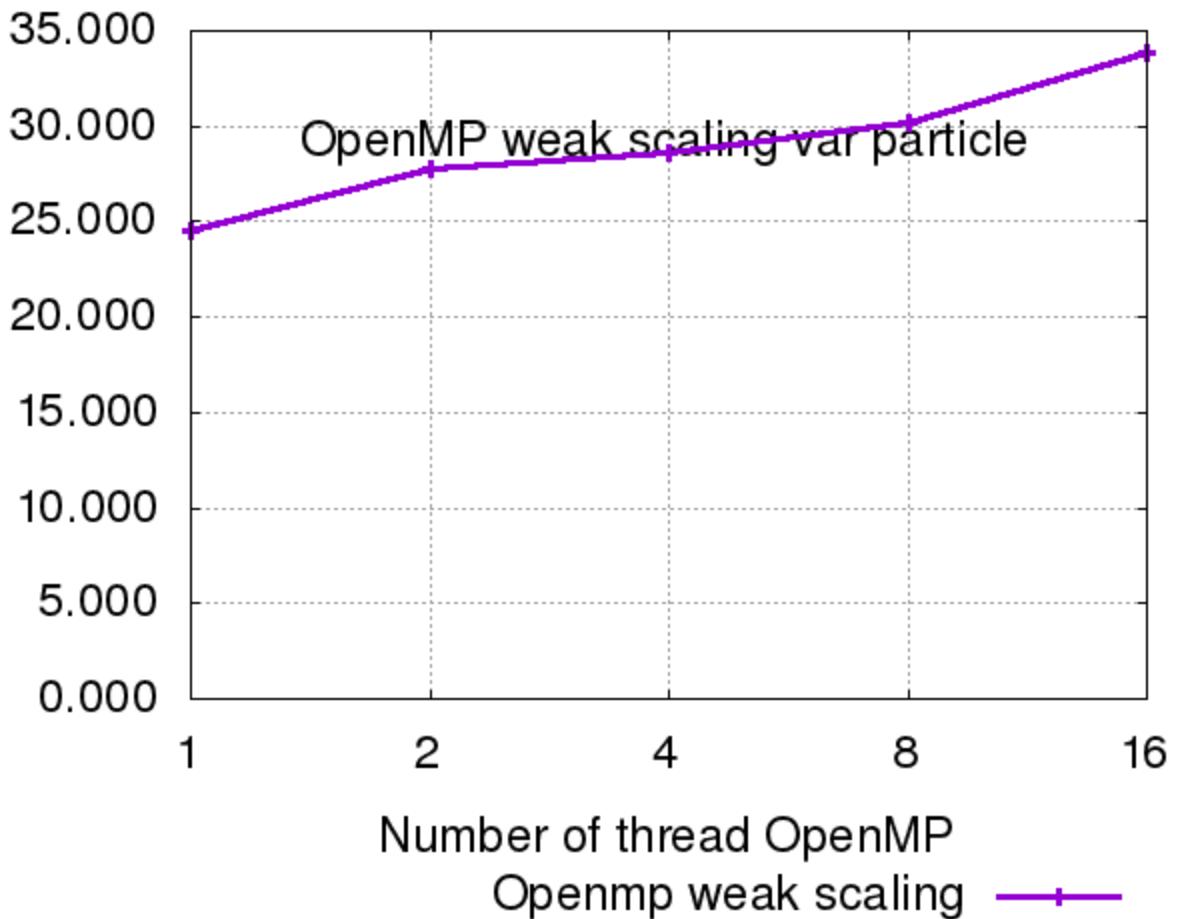
- ❖ Example of nearly perfect weak scaling

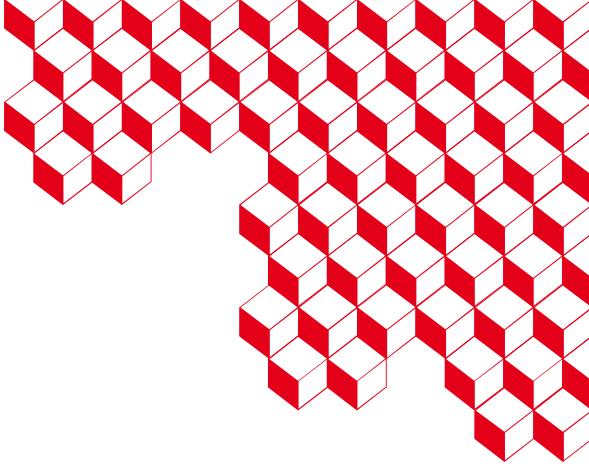




Weak scaling

- ❖ Example of less perfect weak scaling
 - OpenMP has load-balancing issues in this example





Example of a scaling algorithm



A small example of scalability

- ❖ Let's consider an MPI program
- ❖ Each rank should send a value to the next rank

If(rank==0)

MPI_Send(&a, 1, MPI_INT, (rank+1)%P,)

MPI_Recv(&b, 1, MPI_INT, rank-1,...)

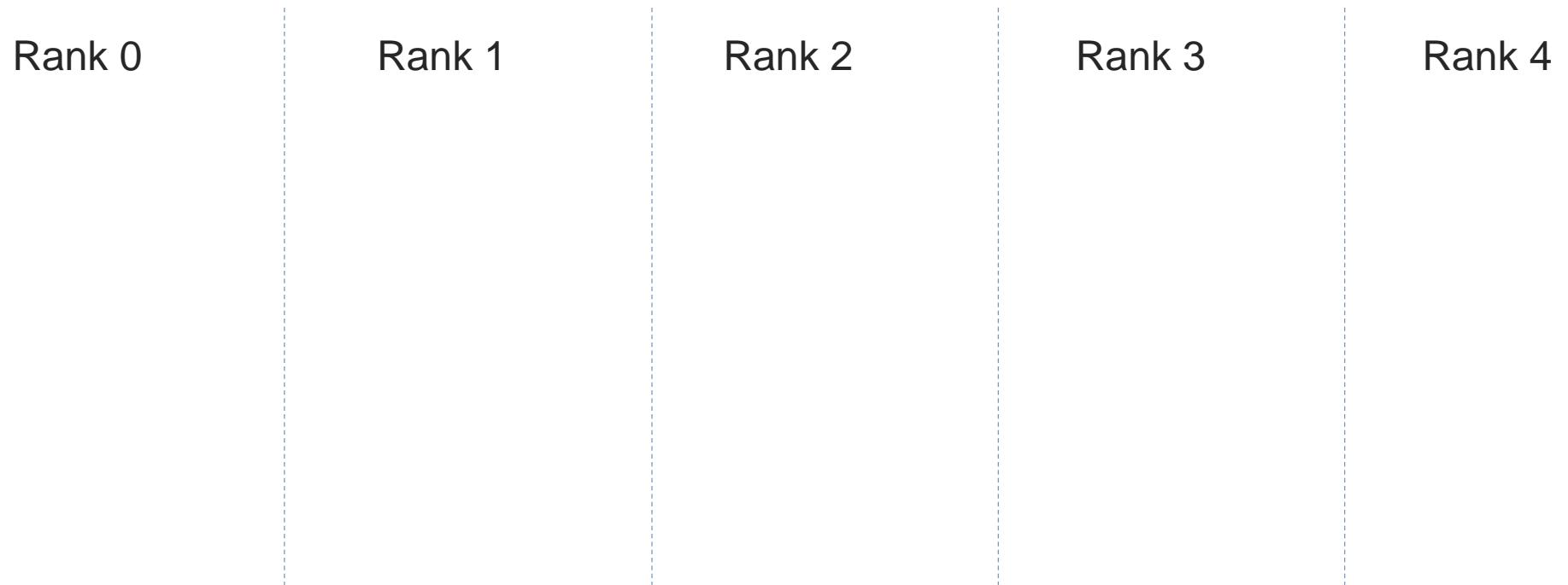
else

MPI_Recv(&b, 1, MPI_INT, rank-1)

MPI_Send(&a, 1, MPI_INT, (rank+1)%P,...)

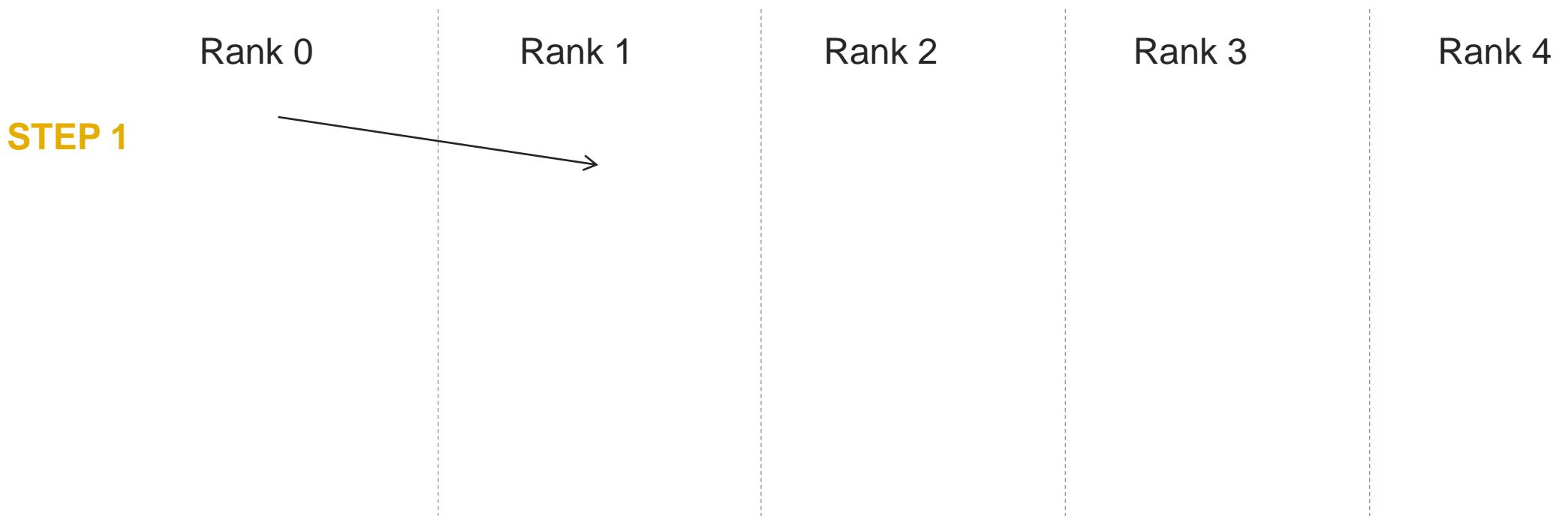


A small example of scalability



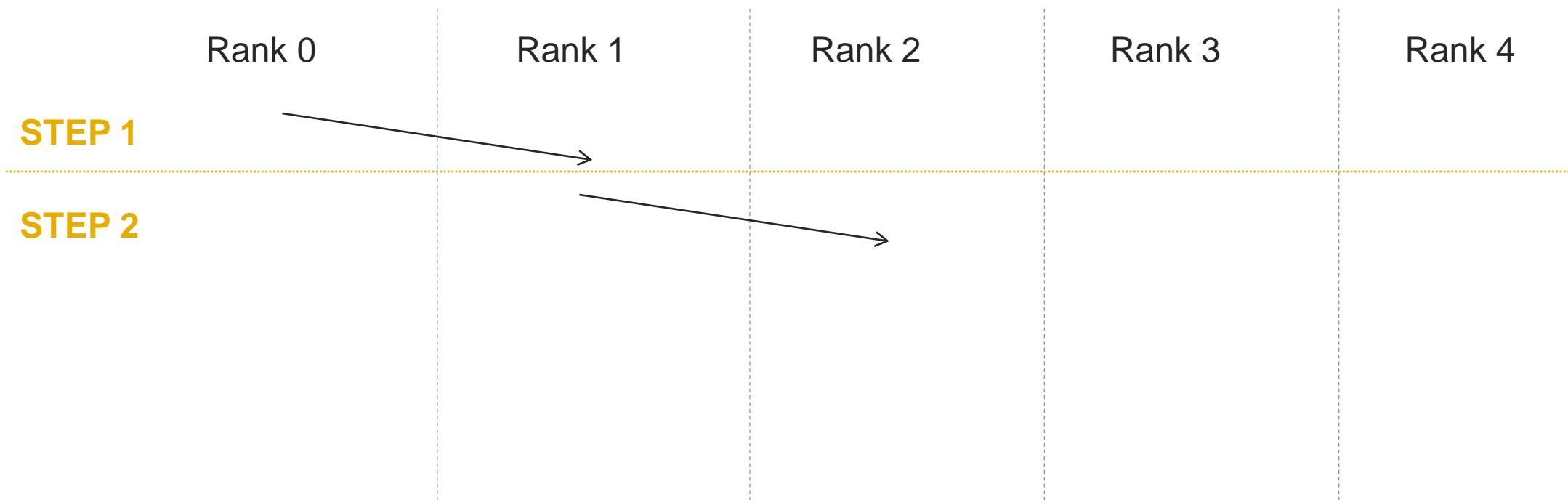


A small example of scalability



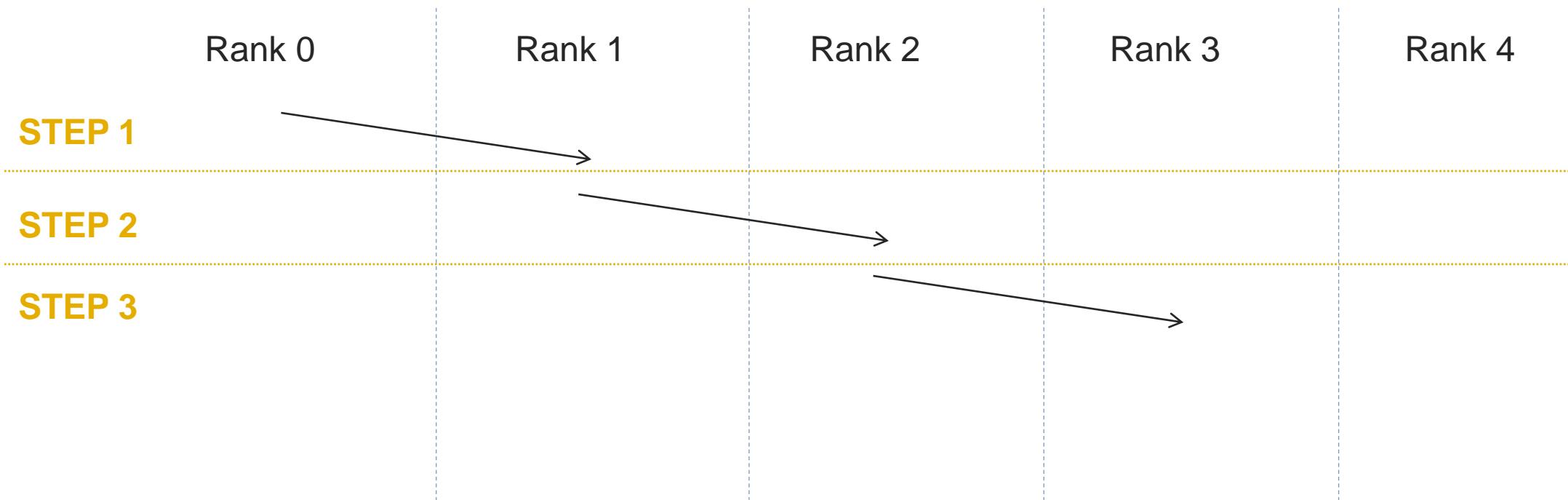


A small example of scalability



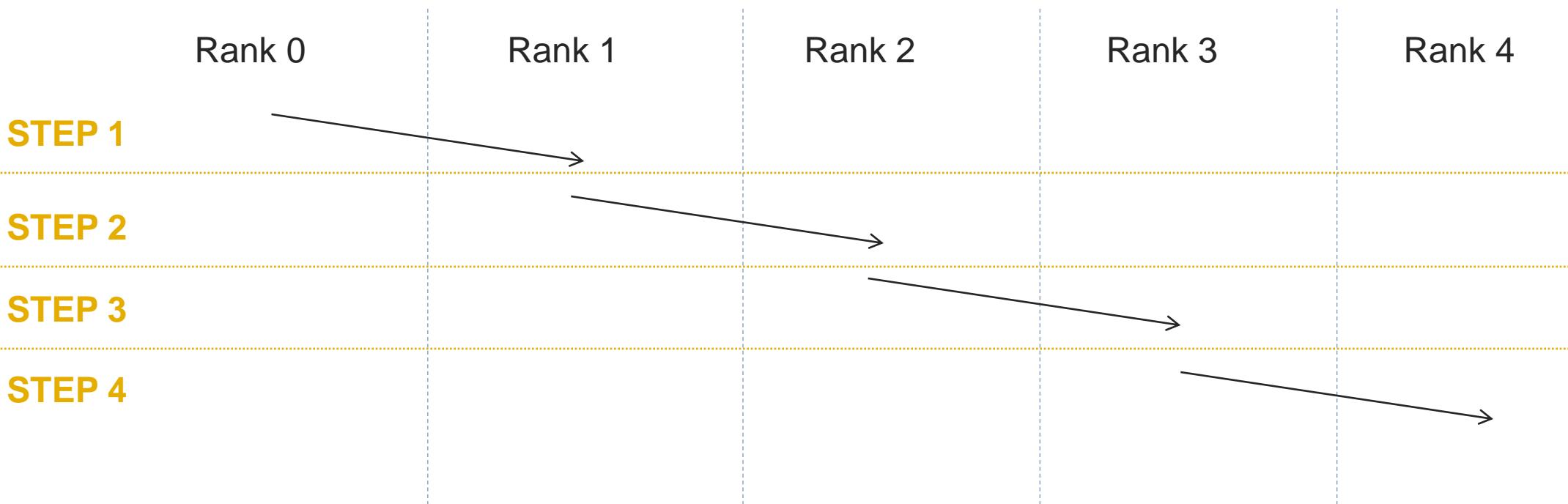


A small example of scalability



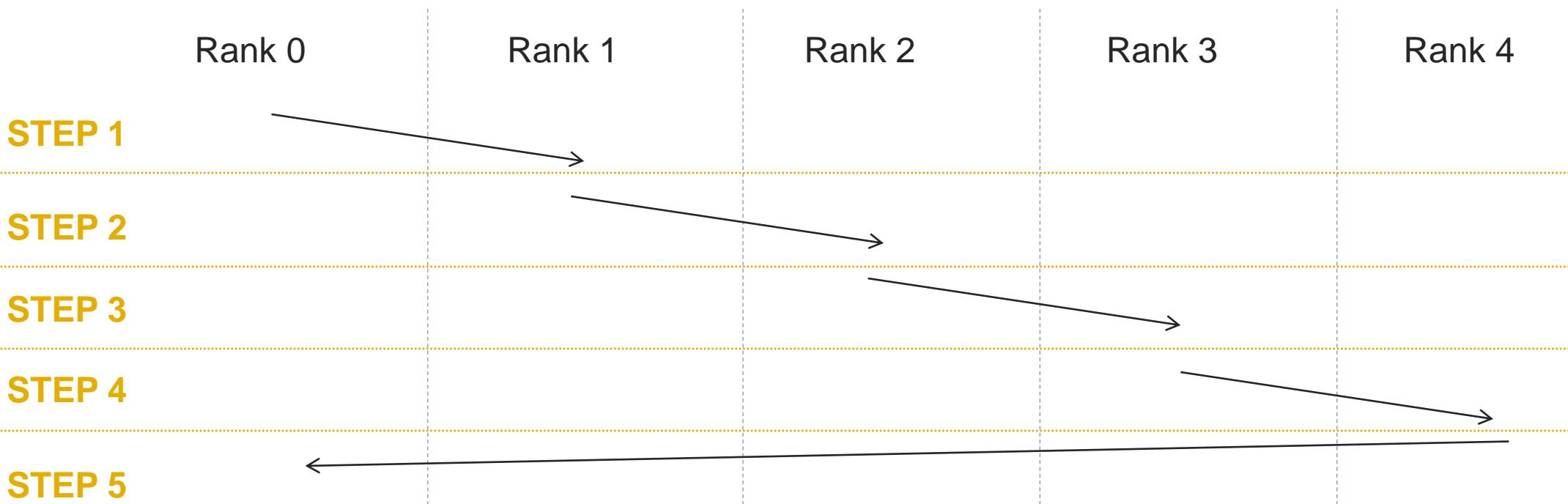


A small example of scalability



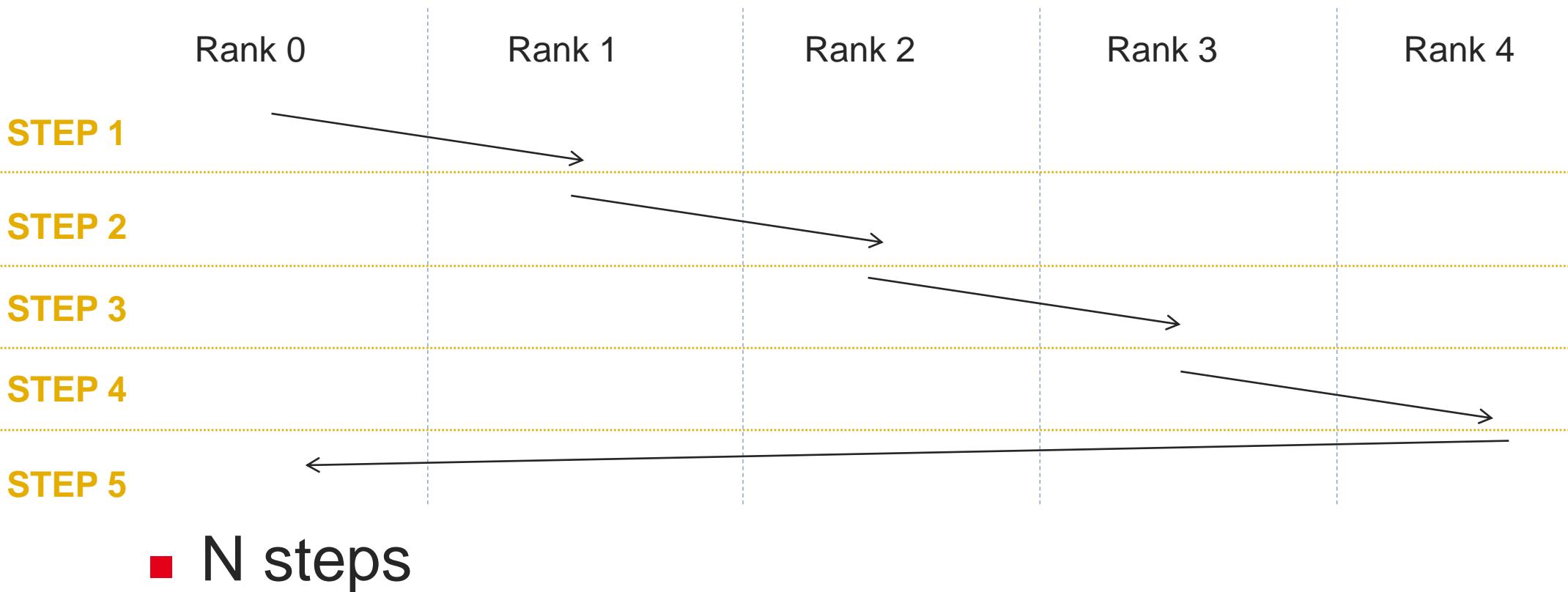


A small example of scalability





A small example of scalability





A small example of scalability

- ❖ Let's consider an MPI program
- ❖ Each rank should send a value to the next rank

```
If(rank==0)  
    MPI_Send(&a, 1, MPI_INT, (rank+1)%P, ....)  
    MPI_Recv(&b, 1, MPI_INT, rank-1,...)  
  
else  
    MPI_Recv(&b, 1, MPI_INT, rank-1)  
    MPI_Send(&a, 1, MPI_INT, (rank+1)%P,...)
```



A small example of scalability

- ❖ Let's consider an MPI program
- ❖ Each rank should send a value to the next rank

If(rank%2==0)

```
MPI_Send(&a, 1, MPI_INT, (rank+1)%P, ....)  
MPI_Recv(&b, 1, MPI_INT, rank-1,...)
```

else

```
MPI_Recv(&b, 1, MPI_INT, rank-1)  
MPI_Send(&a, 1, MPI_INT, (rank+1)%P,...)
```

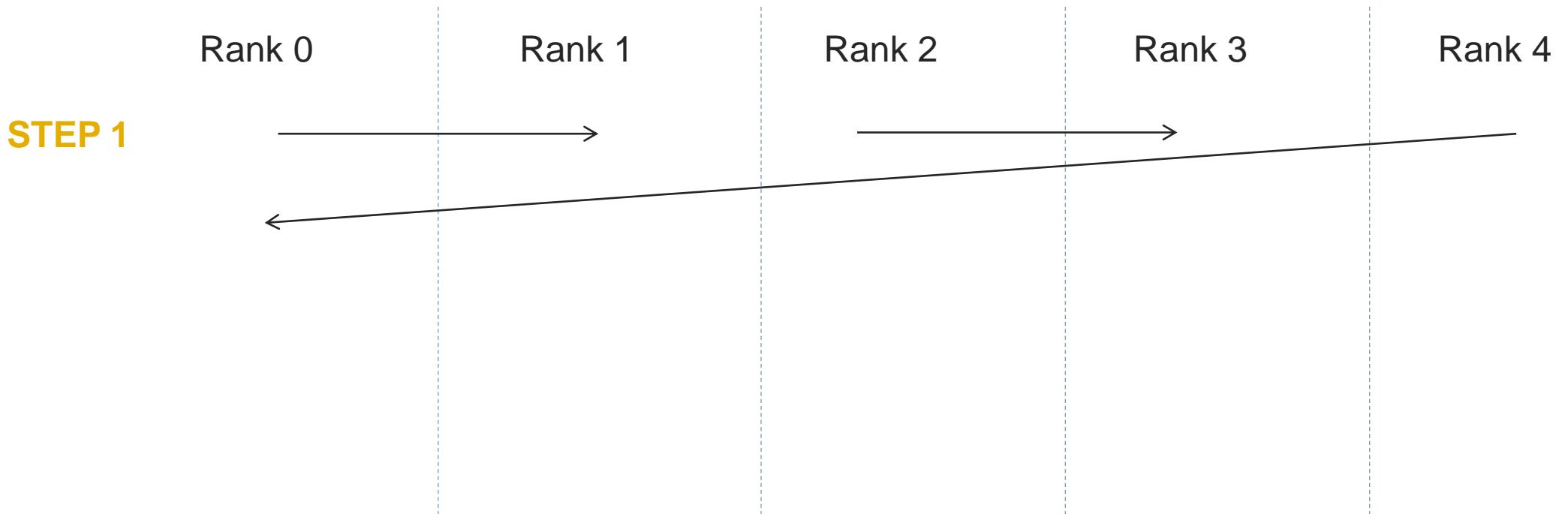


A small example of scalability



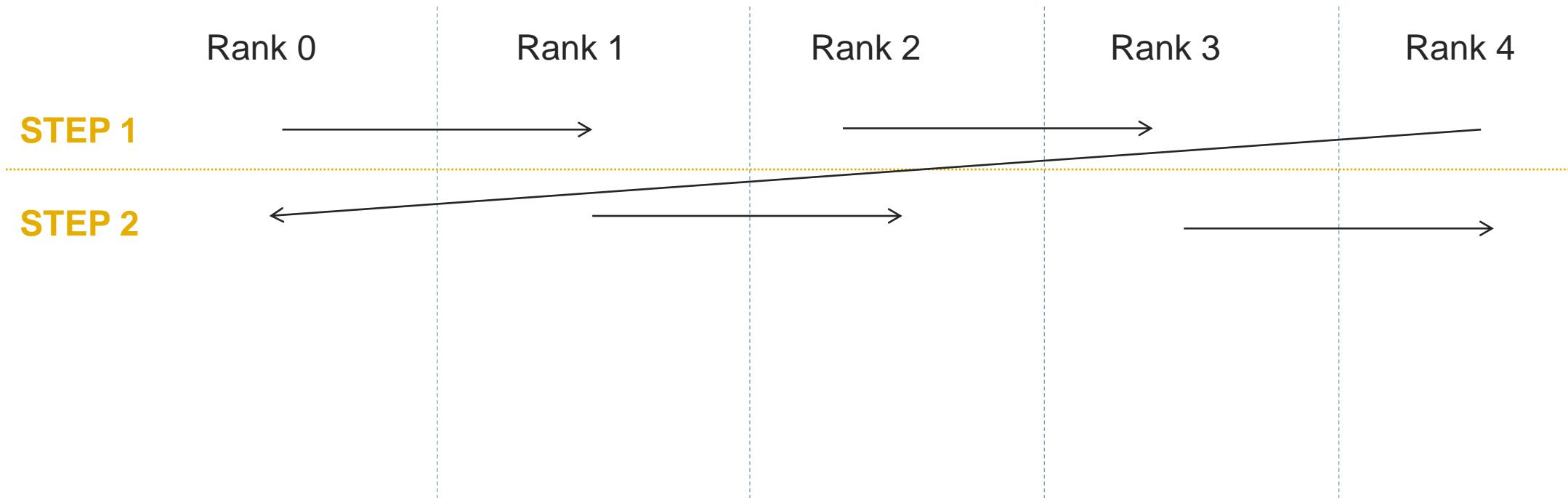


A small example of scalability



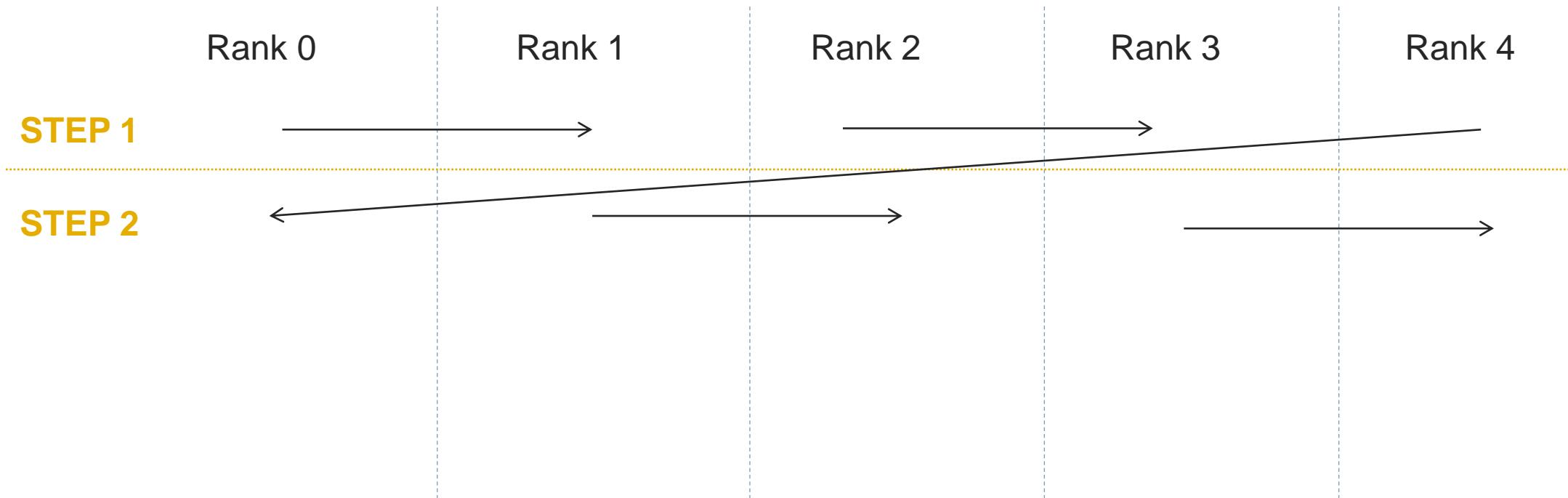


A small example of scalability





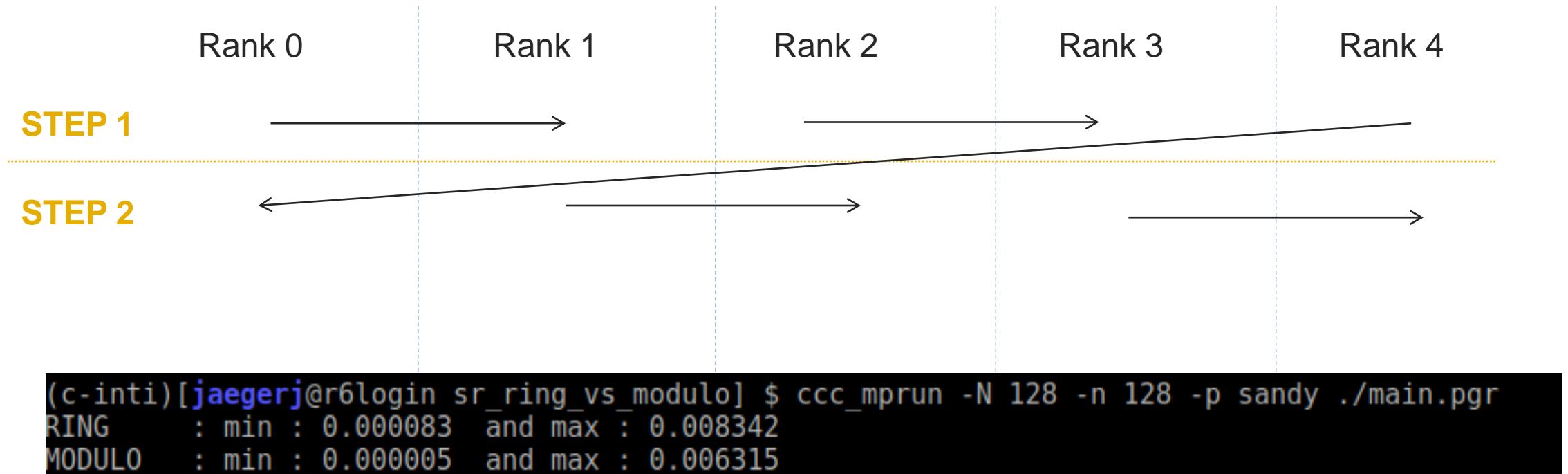
A small example of scalability



- 2 steps



A small example of scalability



- 2 steps



Know what you do

- ❖ A small difference in the code can have huge performance influence
- ❖ It is important to know the parallel algorithm/pattern of your communications
- ❖ A good scalability is a trade-off between number for communications and concurrency of these communications



Know what you will do

- ❖ The number of communications and the communication pattern may also depend on the code design
- ❖ It is important to know how to distribute the data among your MPI processes
 - ❖ Also depends on what will be sent
 - ❖ And to which rank



12. Threads ■ Synchronisation



Optimisation multithread

- ❖ Synchronisation
 - ❖ Barriers and nowait clause
 - ❖ locks
 - ❖ Atomicity
- ❖ Algorithm lock free
- ❖ Parallel for schedule and chunks



OpenMP synchronizations

- ❖ OpenMP offers several directives to synchronize between threads
 - ❖ Barriers
 - ❖ Either explicit (#pragma omp barrier)
 - ❖ Or implicit



Implicit barrier

- ❖ Most OpenMP directives have an implicit barrier at the end
- ❖ When a thread completes its own part, it cannot continue
- ❖ It has to wait for the other threads to finish their work too
 - ❖ And reach the implicit barrier



Directives with implicit barrier

- ❖ #pragma omp for
- ❖ #pragma omp sections
- ❖ #pragma omp workshare
- ❖ #pragma omp simd
- ❖ #pragma omp target *
- ❖ #pragma omp single...???



Focusing of single directive

- ❖ #pragma omp single
- ❖ Same as Master directive but for any thread, not only the master thread
- ❖ The first thread to enter the single directive will execute the code in the scope
- ❖ Thus, only one thread deals with the code in the pragma...
- ❖ ...yet, all threads have to wait for the code to be finished.



Solution: the nowait clause

- ❖ OpenMP provides a way to avoid waiting on the implicit barrier
- ❖ The nowait clause
- ❖ Threads arriving at the end of a directive with a nowait clause continue their execution
 - ❖ Without having to wait for the others
- ❖ Very useful if there are no dependencies between the work of the threads



Example with single directive

```
codes : vi — Konsole
File Edit View Bookmarks Settings Help
#include <omp.h>
#include <unistd.h>

int main()
{
    int nb_th = -1;
#pragma omp parallel
    {
        nb_th = omp_get_num_threads();
        int i;
        for(i=0; i<nb_th; i++)
        {
            #pragma omp single
            {
                sleep(1);
            }
        }
    }
    return 0;
}
"main_single.c" 21L, 232C written
codes : vi
```

- ❖ Omp program with 4 threads
 - ❖ As many single pragmas as there are threads
 - ❖ Each single do sleep(1s)
 - ❖ Take ~4s user time

```
codes : bash — Konsole
File Edit View Bookmarks Settings Help
└ $ time ./main_single.pgr
real 0m4.004s
user 0m0.495s
sys 0m0.000s
jaegerj @jaegerj-Precision-5530 ~/Bureau/Cours/2019-2020/UVSQ/TOP/cours4_figs/codes
└ $
```



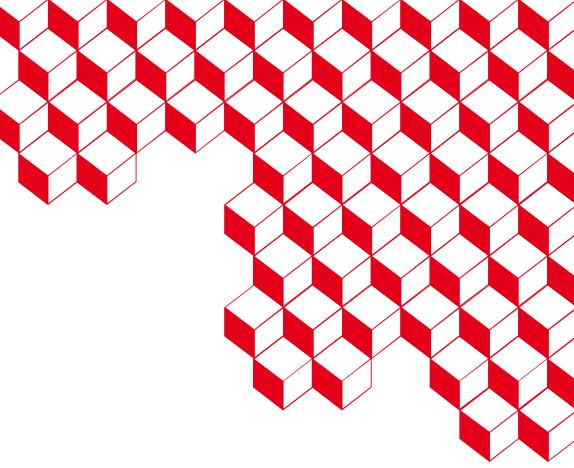
Example with single directive

```
codes : vi — Konsole
File Edit View Bookmarks Settings Help
#include <omp.h>
#include <unistd.h>

int main()
{
    int nb_th = -1;
#pragma omp parallel
    {
        nb_th = omp_get_num_threads();
        int i;
        for(i=0; i<nb_th; i++)
        {
            #pragma omp single nowait
            {
                sleep(1);
            }
        }
    }
    return 0;
}
"main_single_nowait.c" 21L, 239C
codes : vi
```

- ❖ Omp program with 4 threads
 - ❖ Same example
 - ❖ + nowait clause on single directives
 - ❖ Take ~1s user time

```
codes : bash — Konsole
File Edit View Bookmarks Settings Help
└ $ time ./main_single_nowait.pgr
real 0m1.004s
user 0m0.001s
sys 0m0.004s
jaegerj @jaegerj-Precision-5530 ~/Bureau/Cours/2019-2020/UVSQ/TOP/cours4_figs/codes
└ $
```



Locks



OpenMP synchronizations

- ❖ OpenMP offers several directives to synchronize between threads
 - Barriers
 - Either explicit (#pragma omp barrier)
 - Or implicit
 - Atomic
 - Allows to perform the atomic operation in the statement just after the directive
 - Critical
 - Generalization of atomic
 - non-atomic operations or set of operations
- ❖ OpenMP also offers direct access to a locking scheme



Lock Definition

- ❖ Locks: object for mutual exclusion
 - Allow shared-data protection
 - Enable critical sections
-
- ❖ Actions
 - Acquire the lock (lock)
 - Release the lock (unlock)



Lock Definition

- ❖ States
 - locked (owned by a thread)
 - unlocked (no ownership)

- ❖ State update
 - Lock on locked status → wait for release
 - Lock on unlocked status → update status to locked
 - Unlock on unlocked status → nothing (not recommended)
 - Unlock on locked status → release



OpenMP Locks: Initialization

- ❖ Opaque type for locks
- `omp_lock_t`
- ❖ Lock initialization
- `void omp_init_lock(omp_lock_t *lock);`
- ❖ Lock destruction
- `void omp_destroy_lock(omp_lock_t *lock);`



OpenMP Locks: Actions

- ❖ Acquire a lock

- `void omp_set_lock(omp_lock_t *lock);`

- ❖ Release a lock

- `void omp_unset_lock(omp_lock_t *lock);`

- ❖ Try to acquire a lock

- `int omp_test_lock(omp_lock_t *lock);`



OpenMP Nested Locks: Initialization

- ❖ Extension to nested locks: same thread can acquire the same lock multiple times
- ❖ Opaque type for locks
 - `omp_nest_lock_t`
- ❖ Lock initialization
 - `void omp_init_nest_lock(
 omp_nest_lock_t *lock);`
- ❖ Lock destruction
 - `void omp_destroy_nest_lock(
 omp_lock_t *lock);`



OpenMP Nested Locks: Actions

- ❖ Acquire a lock

- `void omp_set_nest_lock(omp_nest_lock_t *lock);`

- ❖ Release a lock

- `void omp_unset_nest_lock(`
`omp_nest_lock_t *lock);`

- ❖ Try to acquire a lock

- `int omp_test_nest_lock(`
`omp_nest_lock_t *lock);`

OpenMP Lock Example

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int n ;
    int c ;
    omp_lock_t l ;
    omp_init_lock( &l ) ;
    ...  
    omp_set_lock(&l) ;  
    c++ ;  
    omp_unset_lock(&l) ;  
}  
    ...  
    omp_destroy_lock( &l ) ;  
  
#pragma omp parallel  
{  
    #pragma omp single  
    {  
        n = omp_get_num_threads() ;  
        ...  
        printf( "Number of threads function:%d  
                count:%d\n", n, c ) ;  
        return 0 ;  
    }  
}
```

Generalized way to perform atomic operations and critical sections

368



Waiting

- ❖ In parallel programming, there always are waiting times between workers
- ❖ In OpenMP, threads wait in:
 - Barriers
 - Locks
- ❖ Two types of waiting policy:
 - Active waiting
 - Passive waiting



Active waiting

- ❖ Active waiting consists in polling indefinitely a variable to see if another thread has modified it
- ❖ While the variable hasn't the expected value, the continue polling
- ❖ Advantages:
 - Very reactive
- ❖ Drawbacks:
 - Use a lot of compute resources and keep the core busy
 - Don't give back the hand to the scheduler



Passive waiting

- ❖ Passive waiting consists in handing back to the scheduler
- ❖ Other threads can then be scheduled and use the computing resource
- ❖ Advantages:
 - Don't block the compute resource for the other threads
- ❖ Drawbacks:
 - Less reactive

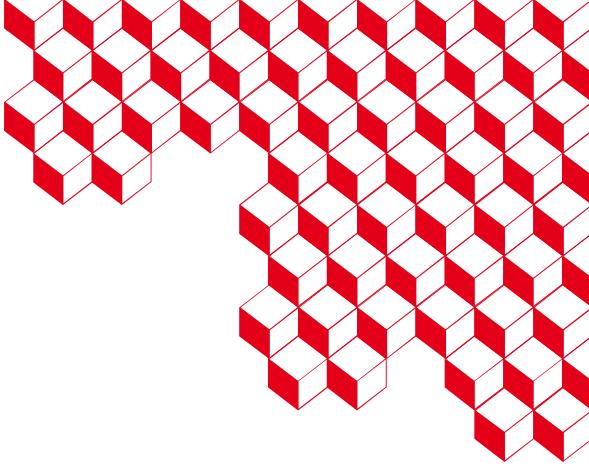


Lock vs barrier

- ❖ Barrier synchronizes all threads
- The fastest thread has to wait for the latest
- Worst, all threads has to wait for the slowest

- ❖ Lock only synchronizes two threads
- Without impacting the other threads

- ❖ Locks have less impact than barrier
- But more complex to use has the user needs to deal with each thread interaction



Atomicity



Atomic operations

- ❖ If multiple threads try to modify the same memory cell at the same time:
 - No guaranty about the results
 - Keep HW simple to diminish complexity and costs
- ❖ User need to synchronize to ensure correct ordering
 - Barrier, locks....
- ❖ Atomic operations allow to do such synchronization on simple operations
 - Less costly than other synchronization methods



Atomic operations

- ❖ Real atomicity: operations performed in 1 cycle on a core
- It ensures that the operation is done before any other operation can touch the values
- Hence, no need to do extra synchronization

- ❖ OpenMP atomic operations:
 - Either an atomic instruction exists for it on the target architecture
 - Or atomicity is emulated with software



Atomic operations

❖ Bit Test:

- Atomically Write a bit
- Return the initial value of this bit

❖ Load Lock/Store Conditional (LL/SC):

- Load begins the transaction
- Store ends it
- Store returns success only if no other threads modified the memory in the meanwhile
- Operation can then be tried again if it failed



Atomic operations

❖ Compare-and-Swap (CAS):

- Conditional write of a value
- Return the pre-existing value
- Very useful to implement spinlocks

❖ Atomic arithmetic:

- Usable only on x86 and x86-64
- Atomic arithmetic and logical operations
- Not supported by RISC processors



Atomic operations

- ❖ An architecture either support LL/SC or CAS, but not both
- ❖ The two approaches are equivalent
- ❖ They are the base of implementing atomic arithmetic operations



Atomic addition

- ❖ Example of atomic addition

```
int curval;
int newval;
do {
    curval = var;
    newval = curval + addend;
} while (CAS(&var, curval, newval));
```

```
int curval;
int newval;
do {
    curval = LL(var);
    newval = curval + addend;
} while (SC(var, newval));
```

Atomic increment

- Multiple atomic increment can be available
 - With different implementations
- Performance can heavily depend on atomic increment implementation on x86 architecture

Exchange add	Add fetch	CAS
0,23s	0,21s	0,73s



Atomic operations

- ❖ Why CAS is more costly
 - There are two memory operations
 - CAS operation is more complicated than necessary
 - Necessary to have a while loop in case of concurrent operations
- ❖ Possible with CAS to implement all atomic operations
 - ❖ But non-optimal implementation



Atomic operations

- ❖ Very useful to implement « lock-free » lists and algorithms
- ❖ Non-portable
- ❖ There are atomic operations in the OpenMP specification
- ❖ Library implementing atomic operations: Open Portable Atomics
 - <https://trac.mcs.anl.gov/projects/openpa/>
 - Used in MPICH, OpenMPI, MPC



Lock-free algorithm

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:   loop                                     # Keep trying until Dequeue is done
D2:     head = Q->Head                         # Read Head
D3:     tail = Q->Tail                          # Read Tail
D4:     next = head->next                       # Read Head.ptr->next
D5:     if head == Q->Head                      # Are head, tail, and next consistent?
D6:       if head.ptr == tail.ptr                 # Is queue empty or Tail falling behind?
D7:         if next.ptr == NULL                    # Is queue empty?
D8:           return FALSE                        # Queue is empty, couldn't dequeue
D9:         endif
D10:        CAS(&Q->Tail, tail, <next.ptr, tail.count+1>) # Tail is falling behind. Try to advance it
D11:      else                                # No need to deal with Tail
D12:        # Read value before CAS, otherwise another dequeue might free the next node
D13:        *pvalue = next.ptr->value          # Try to swing Head to the next node
D14:        if CAS(&Q->Head, head, <next.ptr, head.count+1>) # Dequeue is done. Exit loop
D15:          break
D16:        endif
D17:      endif
D18:    endloop
D19:    free(head.ptr)                         # It is safe now to free the old dummy node
D20:    return TRUE                           # Queue was not empty, dequeue succeeded
```



Lock-free algorithm

```
enqueue(Q: pointer to queue_t, value: data type)
E1:      node = new_node()                                # Allocate a new node from the free list
E2:      node->value = value                            # Copy enqueueued value into node
E3:      node->next.ptr = NULL                          # Set next pointer of node to NULL
E4:      loop                                         # Keep trying until Enqueue is done
E5:          tail = Q->Tail                           # Read Tail.ptr and Tail.count together
E6:          next = tail.ptr->next                     # Read next ptr and count fields together
E7:          if tail == Q->Tail                         # Are tail and next consistent?
E8:              if next.ptr == NULL                      # Was Tail pointing to the last node?
E9:                  if CAS(&tail.ptr->next, next, <node, next.count+1>)  # Try to link node at the end of the linked list
E10:                 break                               # Enqueue is done. Exit loop
E11:                 endif
E12:             else                                 # Tail was not pointing to the last node
E13:                 CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Try to swing Tail to the next node
E14:             endif
E15:         endif
E16:     endloop                                     # Enqueue is done. Try to swing Tail to the inserted node
E17:     CAS(&Q->Tail, tail, <node, tail.count+1>)
```



13. ▪ Parallel I/O



What are I/O ?

- ❖ I/O stands for Input/Output
- ❖ I/O usually regroups every operations applied on files by a program
 - Reading (Input)
 - Writing (Output)
 - File Management
 - Resizing
 - Splitting/Merging
 - Organizing
 - Searching



Why do we need I/O ?

- ❖ Simulations produce lot of data
- Not meant to be directly read by a person

- ❖ These data need to be stored
- To be used by other programs (visualizer, ...)
- To keep history of simulations
 - You don't want to replay the simulation any time you need some result
- Checkpoint/restart



Sequential I/O

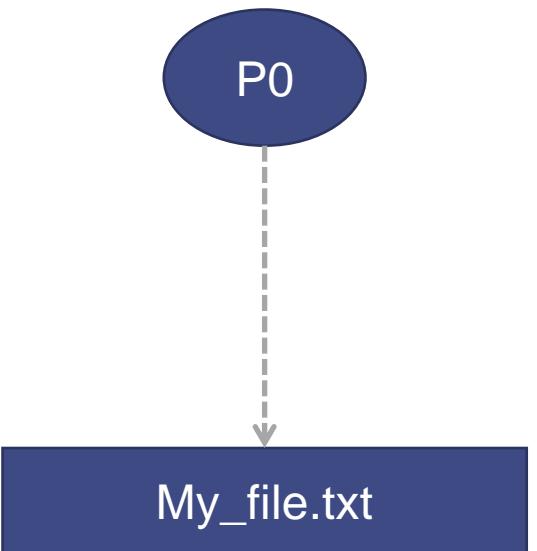
- ❖ Original paradigm
- Example: Posix I/O
- ❖ How to do I/O?

P0



Sequential I/O

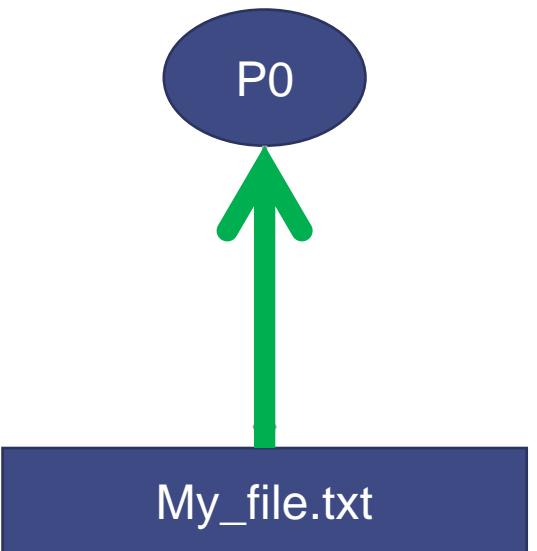
- ❖ Original paradigm
 - Example: Posix I/O
- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name





Sequential I/O

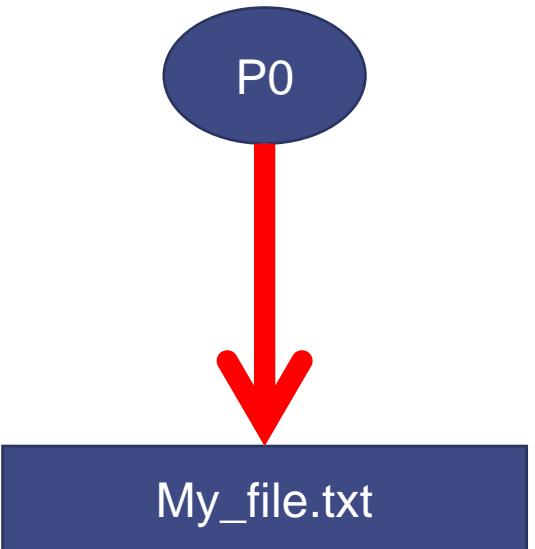
- ❖ Original paradigm
 - Example: Posix I/O
- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....





Sequential I/O

- ❖ Original paradigm
 - Example: Posix I/O
- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....
 - ... or write data

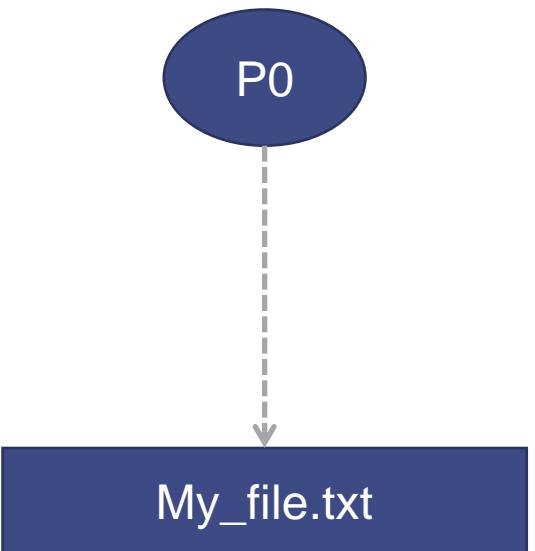




Sequential I/O

- ❖ Original paradigm
 - Example: Posix I/O

- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....
 - ... or write data
 - Once it is done

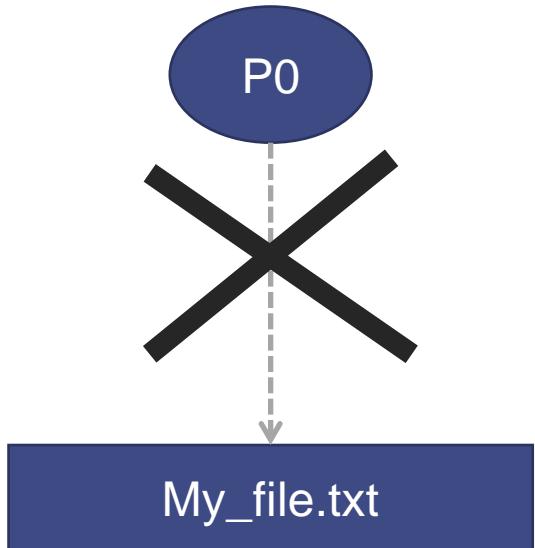




Sequential I/O

- ❖ Original paradigm
 - Example: Posix I/O

- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....
 - ... or write data
 - Once it is done, P0 close file handler





Sequential I/O

- ❖ Original paradigm
 - Example: Posix I/O

- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....
 - ... or write data
 - Once it is done, P0 close file handler

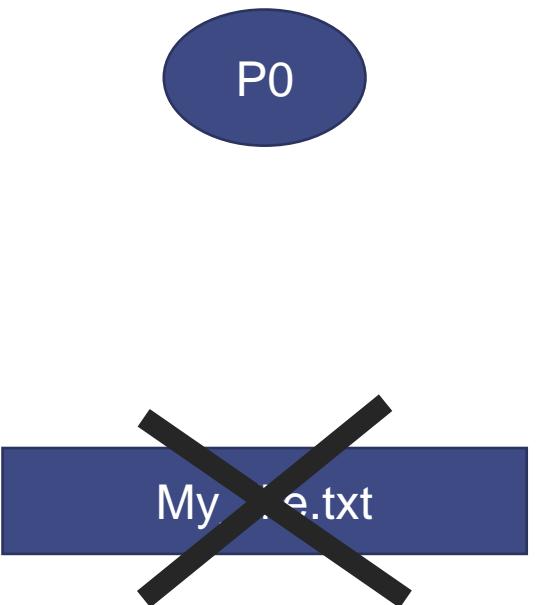
P0

My_file.txt



Sequential I/O

- ❖ Original paradigm
 - Example: Posix I/O
- ❖ How to do I/O?
 - Process P0 open a file handler
 - Associated with a name
 - May create it if it does not already exist
 - P0 either read data....
 - ... or write data
 - Once it is done, P0 close file handler
 - May also delete the file if asked





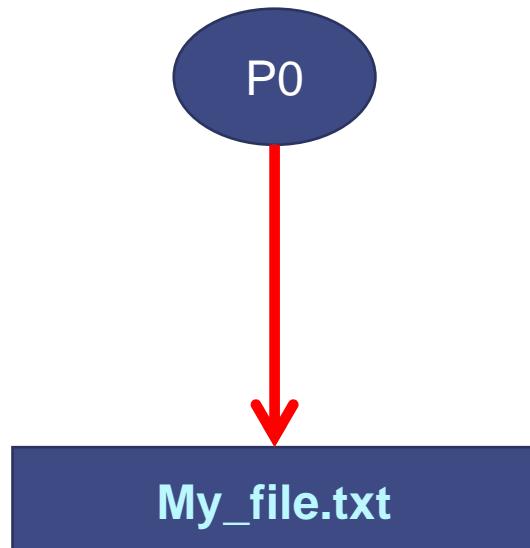
Parallel I/O

- ❖ Easy when there is only one process
- ❖ How to manage that with multiple processes which all want to read/write?
- ❖ Solution:
 - One-to-One
 - All-to-All
 - One-to-All
 - All-to-One



Parallel One-to-One I/O

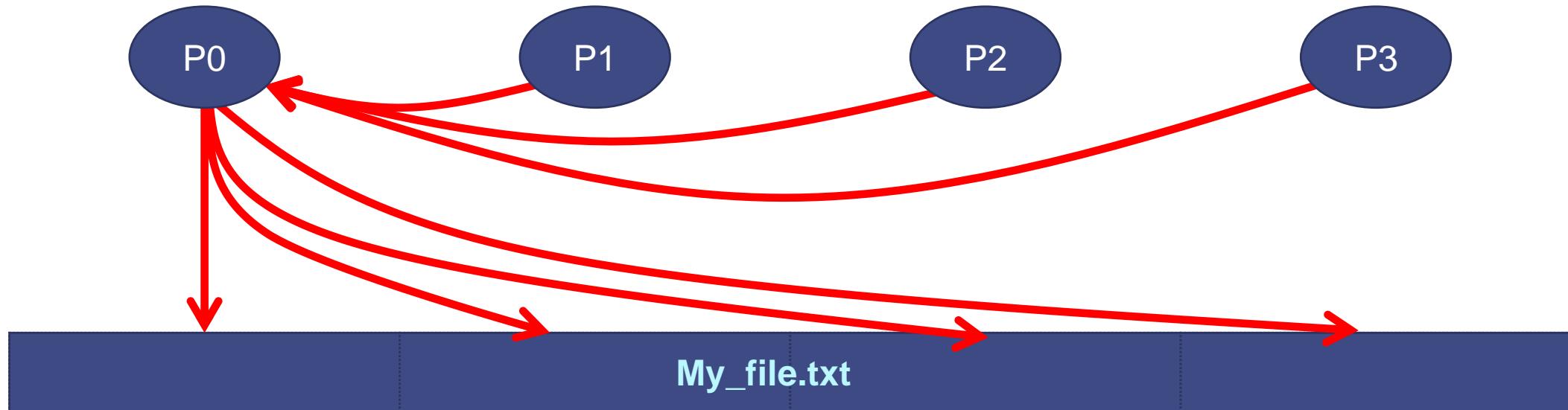
- ❖ Based on legacy sequential I/O
- One process manage the I/Os (Open File handler, read, write)





Parallel One-to-One I/O

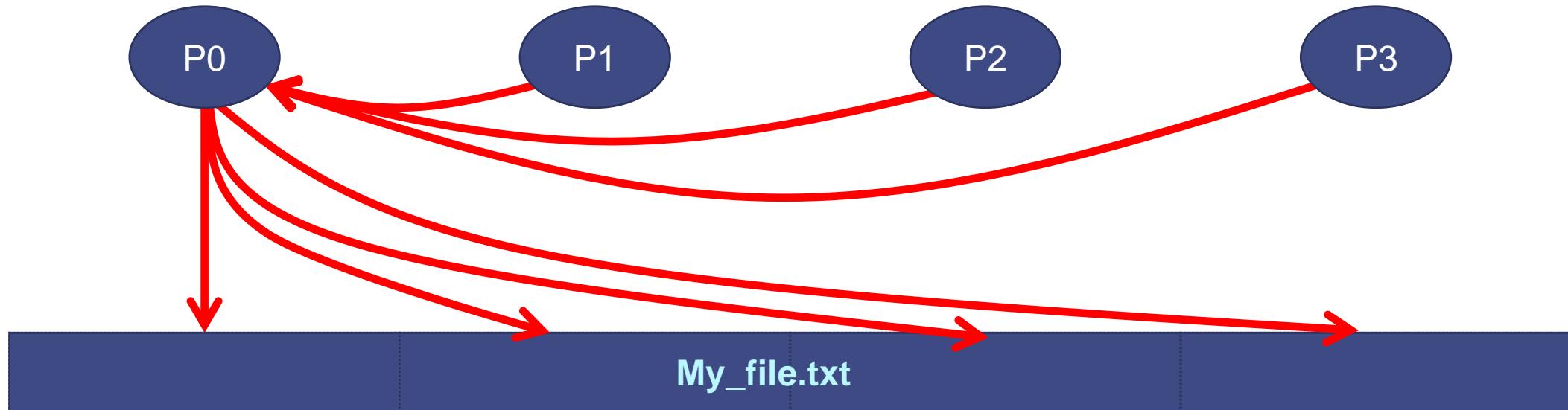
- ❖ Based on legacy sequential I/O
- One process manage the I/Os (Open File handler, read, write)
- ❖ The other processes “delegate” their I/Os to the chosen one





Parallel One-to-One I/O

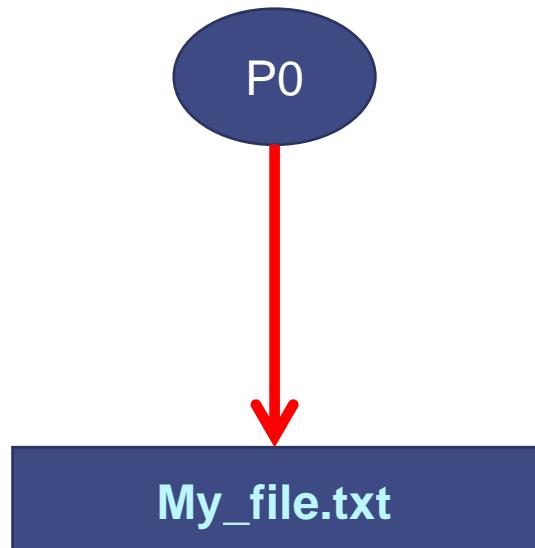
- ❖ Based on legacy sequential I/O
 - One process manage the I/Os (Open File handler, read, write)
- ❖ The other processes “delegate” their I/Os to the chosen one
 - Fit for parallel systems and I/O library not supporting parallel I/O
 - Only one result file: easy to read, to move...
 - No parallelism: poor scalability -> poor performances





Parallel All-to-All I/O

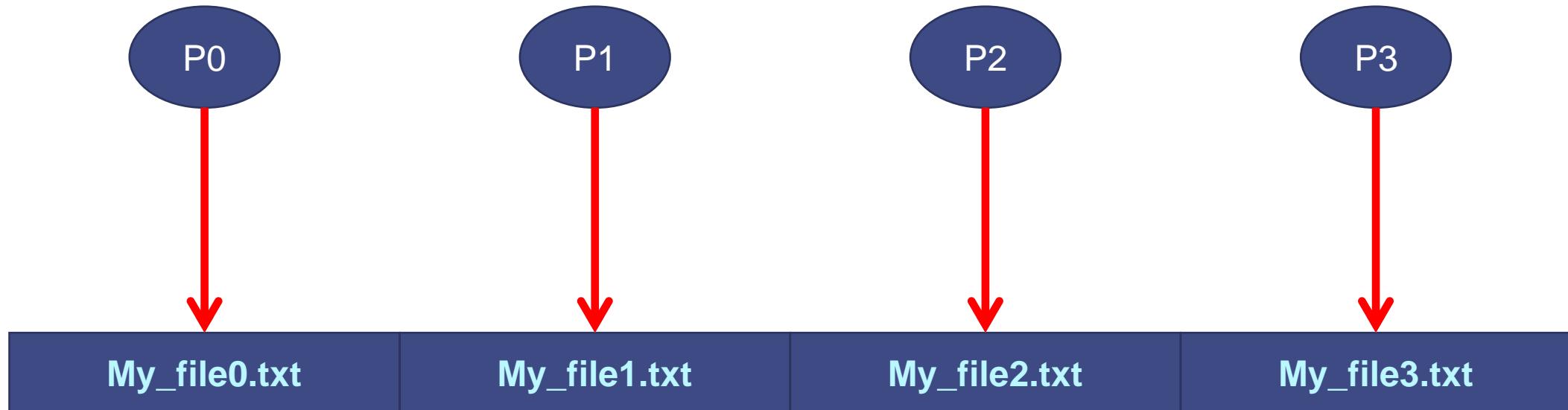
- ❖ Based on legacy sequential I/O
- One process manages its own I/O





Parallel All-to-All I/O

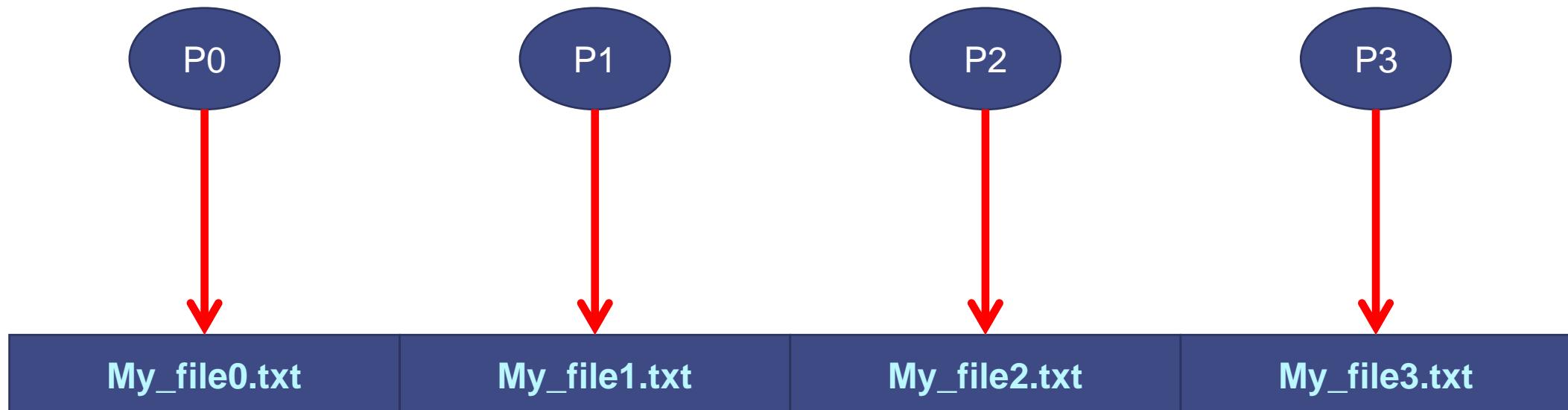
- ❖ Based on legacy sequential I/O
- One process manages its own I/O
- ❖ Every process only manages its own handler file





Parallel All-to-All I/O

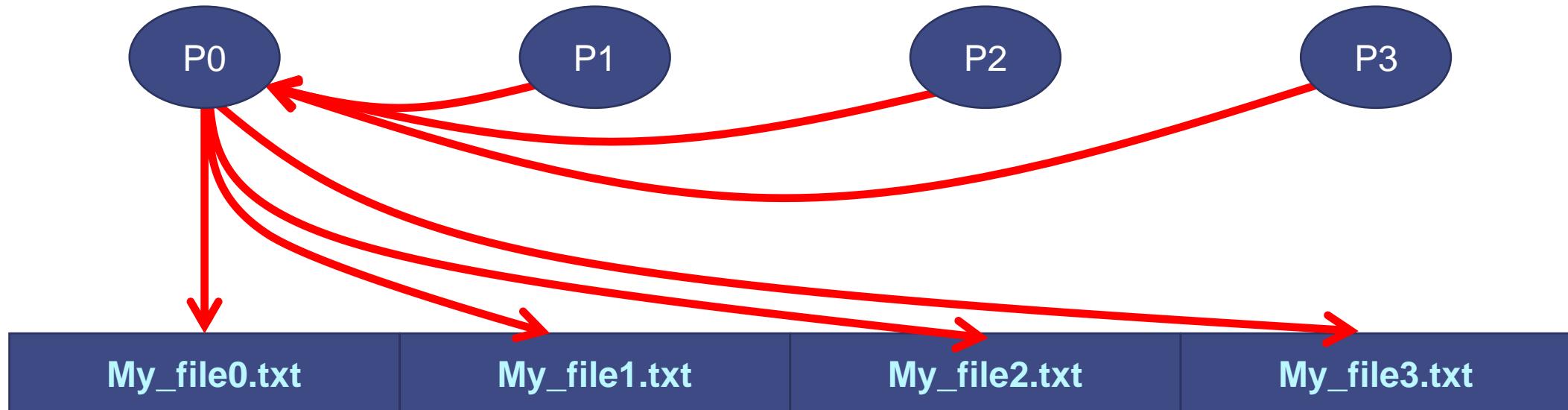
- ❖ Based on legacy sequential I/O
 - One process manages its own I/O
- ❖ Every process only manages its own handler file
 - Parallelism: no bottlenecks on processes
 - Lots of small files to manage
 - Difficult to read back from different number of processes





Parallel One-to-All I/O

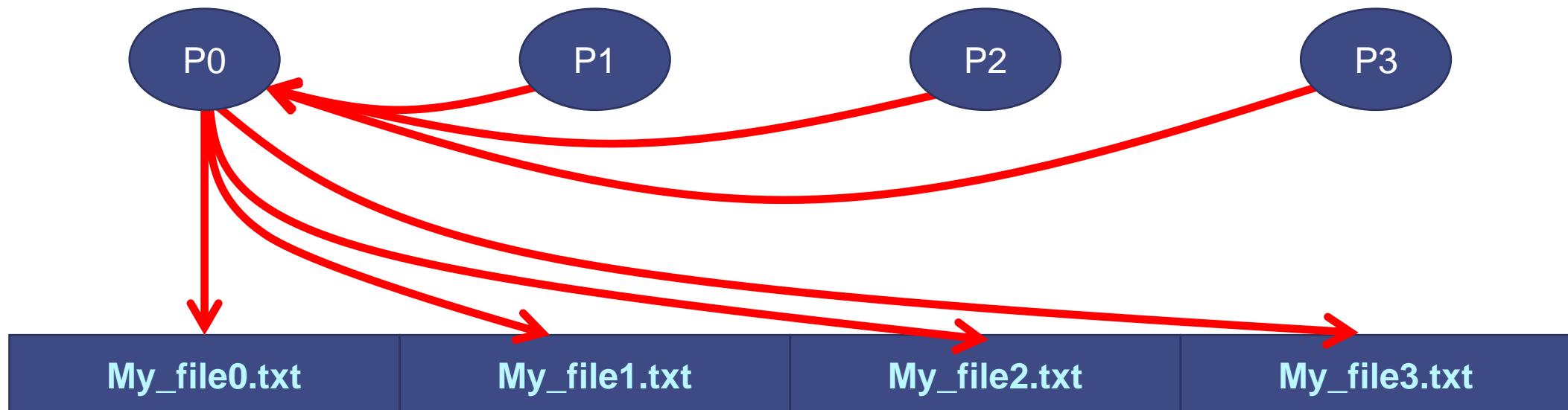
- ❖ One process manage the I/Os (Open File handler, read, write)
- ❖ The other processes “delegate” their I/Os to the chosen one
- ❖ The I/Os are spread across multiple files





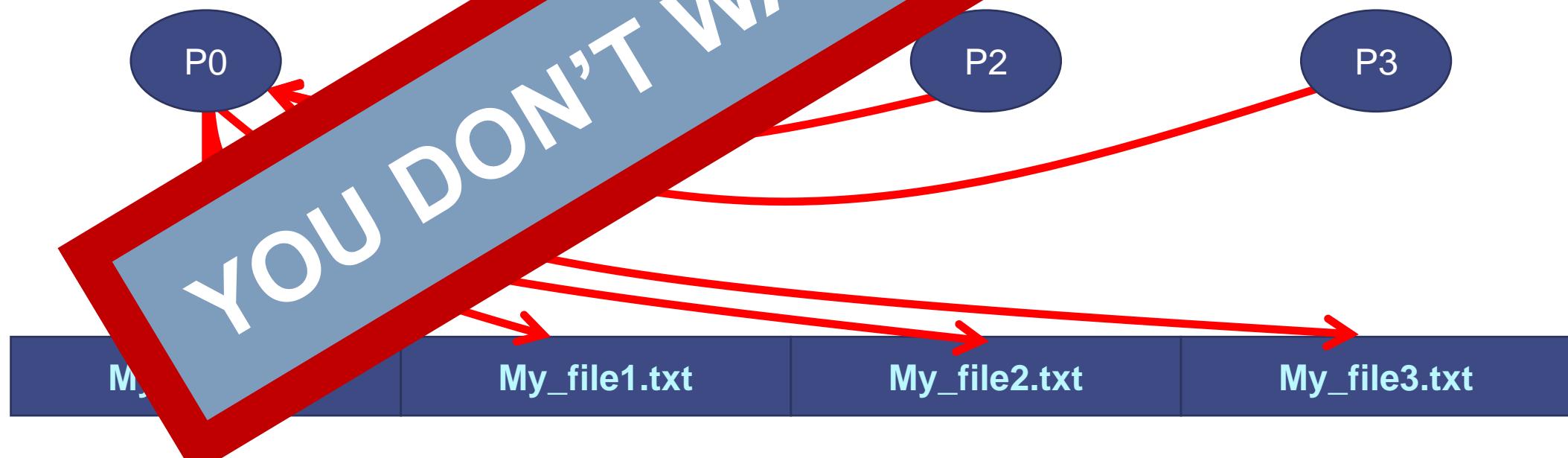
Parallel One-to-All I/O

- ❖ One process manage the I/Os (Open File handler, read, write)
- ❖ The other processes “delegate” their I/Os to the chosen one
- ❖ The I/Os are spread across multiple files
 - Bottleneck on the I/O manager process
 - Multiple file handlers to manage by chosen process
 - Multiple result files to manage



Parallel One-to-All I/O

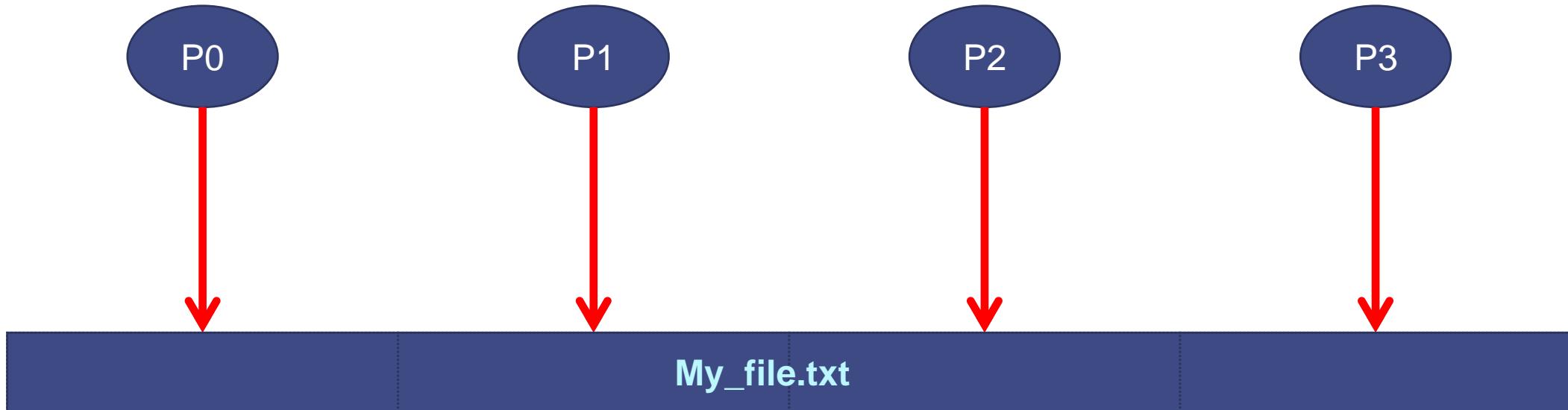
- ❖ One process manage the I/Os (Open File handles)
- ❖ The other processes “delegate” their I/Os to it
- ❖ The I/Os are spread across multiple files
 - Bottleneck on the I/O manager process
 - Multiple file handlers to manage
 - Multiple result files to manage





Parallel All-to-one I/O

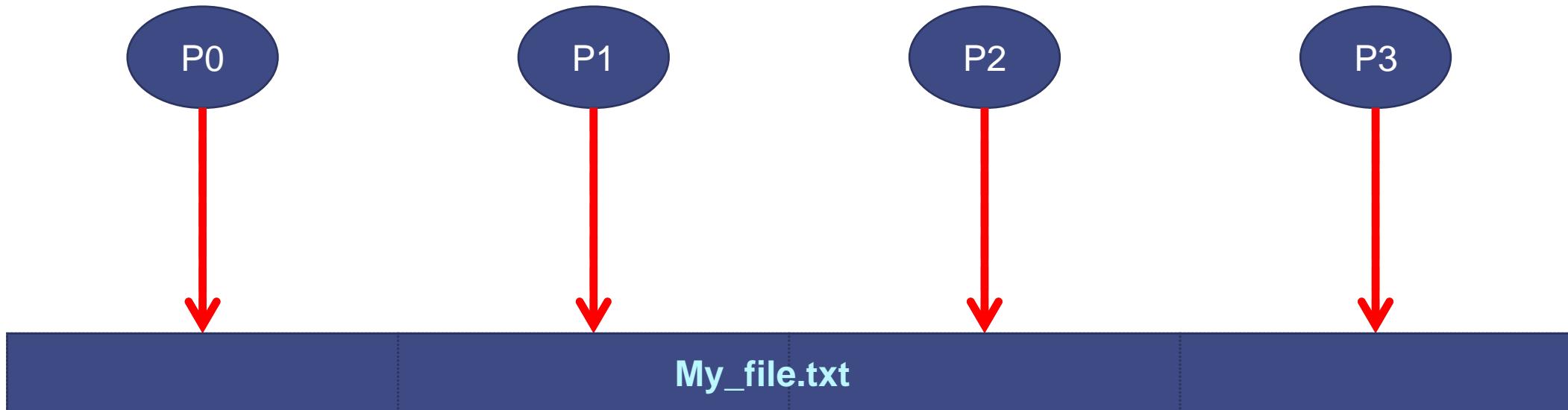
- ❖ Every process manages its own I/Os
- ❖ Cooperative access to the same file to write data in parallel





Parallel All-to-one I/O

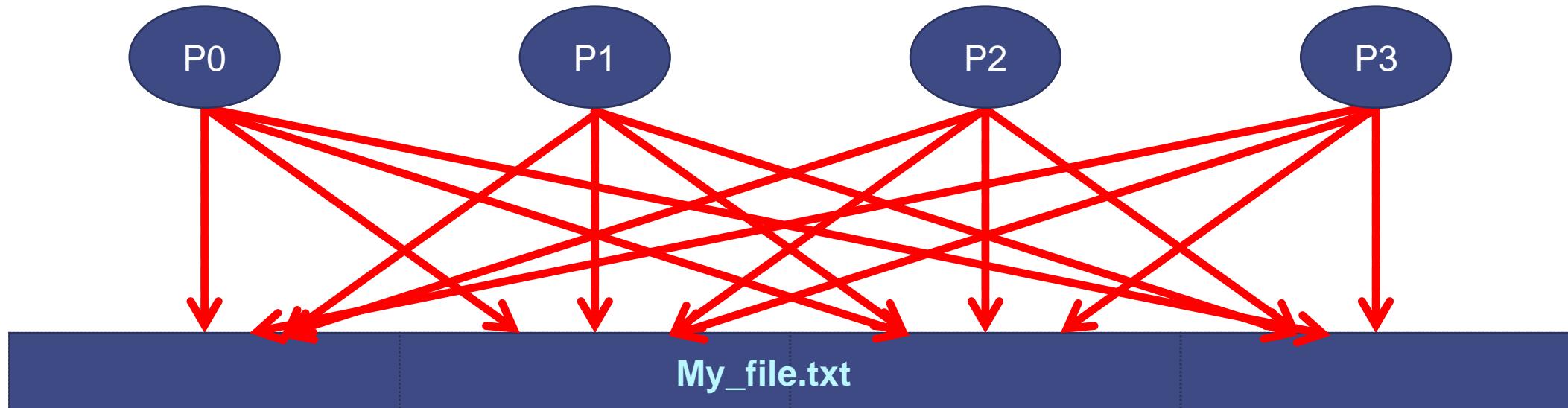
- ❖ Every process manages its own I/Os
- ❖ Cooperative access to the same file to write data in parallel
 - Parallelism: no bottlenecks on processes
 - Only one result file to manage
 - $/!l$ ordering when multiple processes are writing to the same address





Parallel All-to-one I/O

- ❖ Every process manages its own I/Os
- ❖ Cooperative access to the same file to write data in parallel
 - ❖ Parallelism: no bottlenecks on processes
 - ❖ Only one result file to manage
 - ❖ /!\ ordering when multiple processes are writing to the same address
 - ❖ If handled correctly, all processes should be able to write all over the file





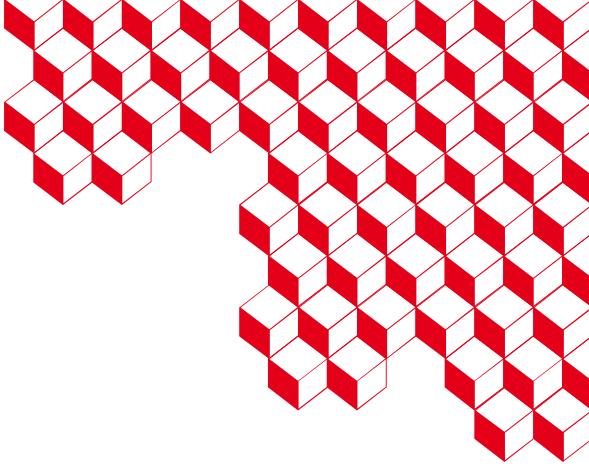
I/O impact on performances

- ❖ Problem: I/O can have huge impact on performance
- ❖ I/O use system calls
- ❖ Workers performing I/O are blocked until I/O are completed
- ❖ Every I/O will impact parallel performances of the application



I/O delegation

- ❖ I/O delegation is a solution to limit I/O impact
- ❖ Compute workers don't perform their own I/O
 - They delegate them to specialized I/O workers
- ❖ Specialized I/O workers block when performing I/O
 - Compute workers can still continue to run the application
- ❖ Several I/O delegation libraries used in HPC
 - netCDF, HDF5
 - Usually use I/O threads for the I/O delegation



Thank you !

Hugo Taboada

CEA/LIHPC