Parallel I/O
Issues and Solutions

Adrien Lèbre - Mines de Nantes

SCALUS - Barcelona Training School
Feb 2011, 23

Agenda

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

- Fundamentals
  - Context
  - Characterization
  - File decomposition
- Parallel I/O Solutions
  - Parallel FS
  - Dedicated I/O Libraries
  - Scheduling
- Let's be prospective (or let's have fun ;))

Context

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

- Scientific applications benefit from large scale infrastructures
  - Data is bigger and bigger / access patterns change
- Significant lack of performance from the I/O point of view
  - [amdahl67] The slowest sub system in a computer defines/limits the maximal performance reachable by a « parallelization » process.
  - tradeoff: 1MB of RAM, 1MIPS, 1Mb/s I/O ⇒ 2GHz /2Gb/s (250MB)
  - [Hennessy96] I/O 10%year vs CPU 40%/year

    SATA HDD: up to 100MB/s (average 70MB/s), average seek time 8ms
    Parsing the whole HDD requires more than 4 hours (hopefully, RAID policies exist ;))
- From CPU bound application to I/O bound application
- Access pattern characterization is mandatory
  - To analyze and understand how new behaviors impact storage systems
  - To suggest suited solutions

Access characterization

Marie Curie Initial Training Networks

SCALUS

SCALing by means of Ubiquitous Storage

- Application classification ("Scalable I/O initiative", 1995)

  - *Compulsive,* significant number of accesses at the same time,

    read accesses at the beginning / write accesses at the end

    Hard to make data accesses transparent (data is input and output of the app)

    ⇒ request aggregation, caches (prefetch, write behind, ...)

  - *Control,* a small amount of manipulated data (only few requests) but incoming during the whole execution of the application
    (« checkpoint » files , « temporary » files)

    ⇒ asynchronous requests, caches

  - *Out of bound,* « Out-of-core », data is bigger than RAM
    (virtual memory / « swap » issues),

Access characterization

- *Sequential,* the offset for the new request is bigger than the previous one.
- *Contiguous,* the request starts where the previous one ended
- *Overlapping,* contiguous data for several processes

  Sequential for one process,
  Contiguous for the application (the set of processes)

- Parallel I/O

  Definition: handling concurrent accesses to a same resource (a file)

  Main difficulties: accesses are different in size, in offset, …

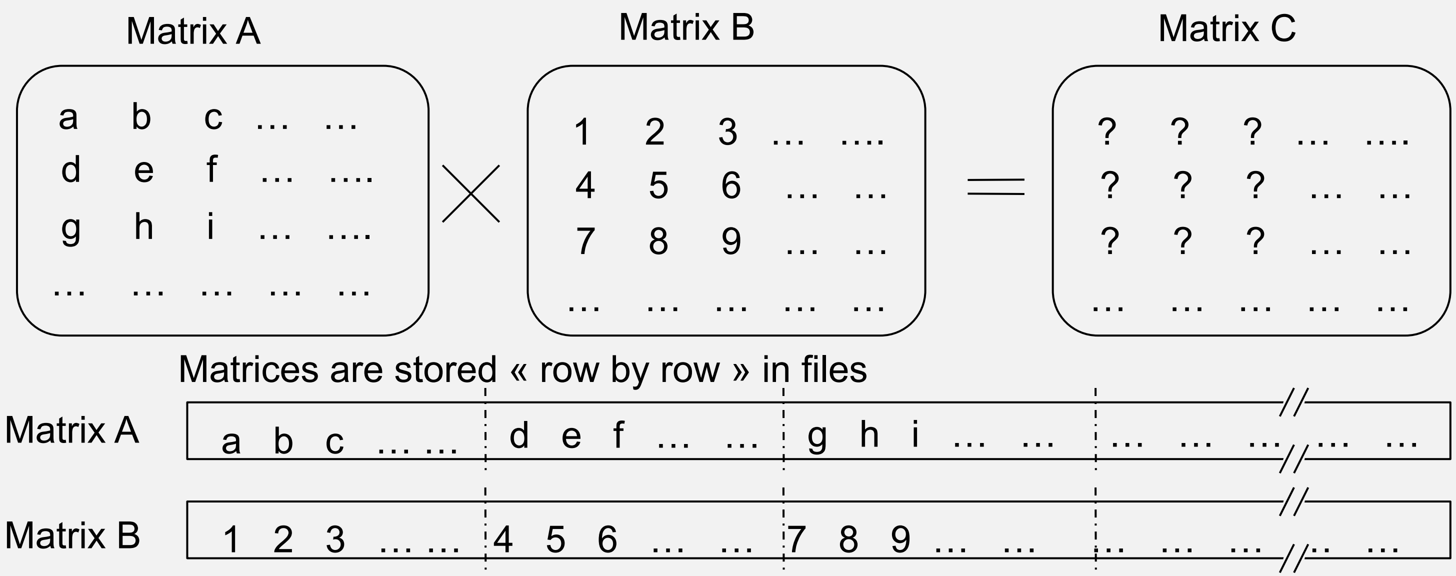
    Example: a parallel matrix product

- The objects (the matrices) vs logical view (the files)

$$
\underset{\text{Matrix A}}{\begin{pmatrix} a & b & c & \dots & \dots \\ d & e & f & \dots & \dots. \\ g & h & i & \dots & \dots. \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}}
\times
\underset{\text{Matrix B}}{\begin{pmatrix} 1 & 2 & 3 & \dots & \dots. \\ 4 & 5 & 6 & \dots & \dots \\ 7 & 8 & 9 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}}
=
\underset{\text{Matrix C}}{\begin{pmatrix} ? & ? & ? & \dots & \dots. \\ ? & ? & ? & \dots & \dots \\ ? & ? & ? & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix}}
$$

Parallel Matrix Product

- The objects (the matrices) vs logical view (the files)

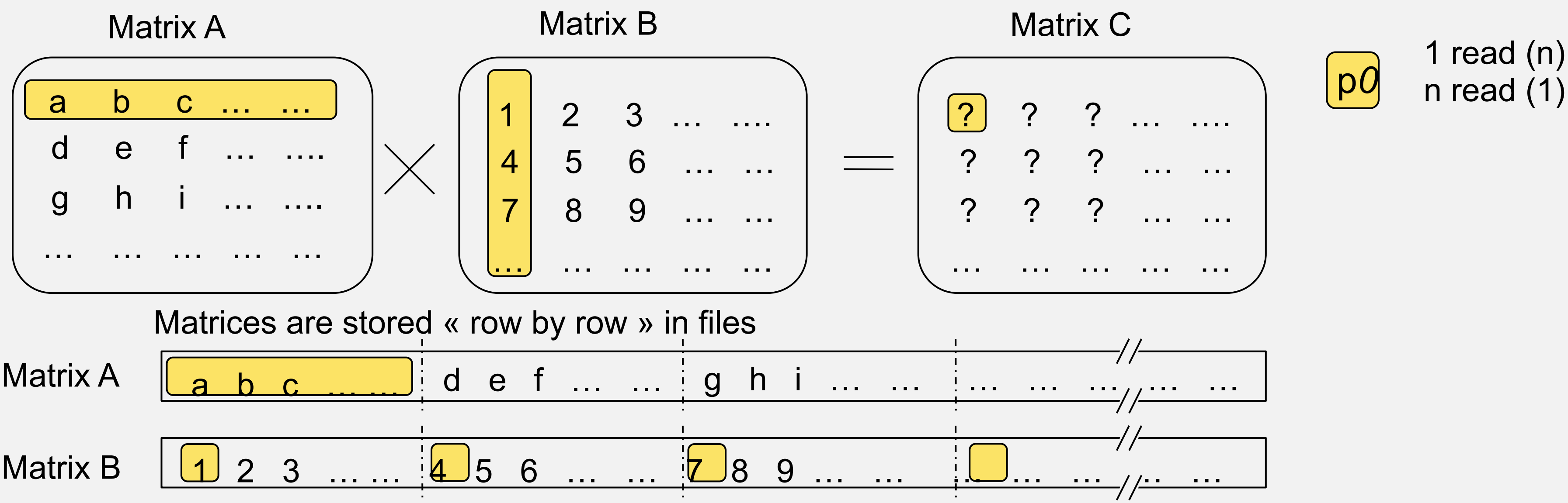


Parallel I/O
Parallel Matrix Product

Marie Curie Initial Training Networks

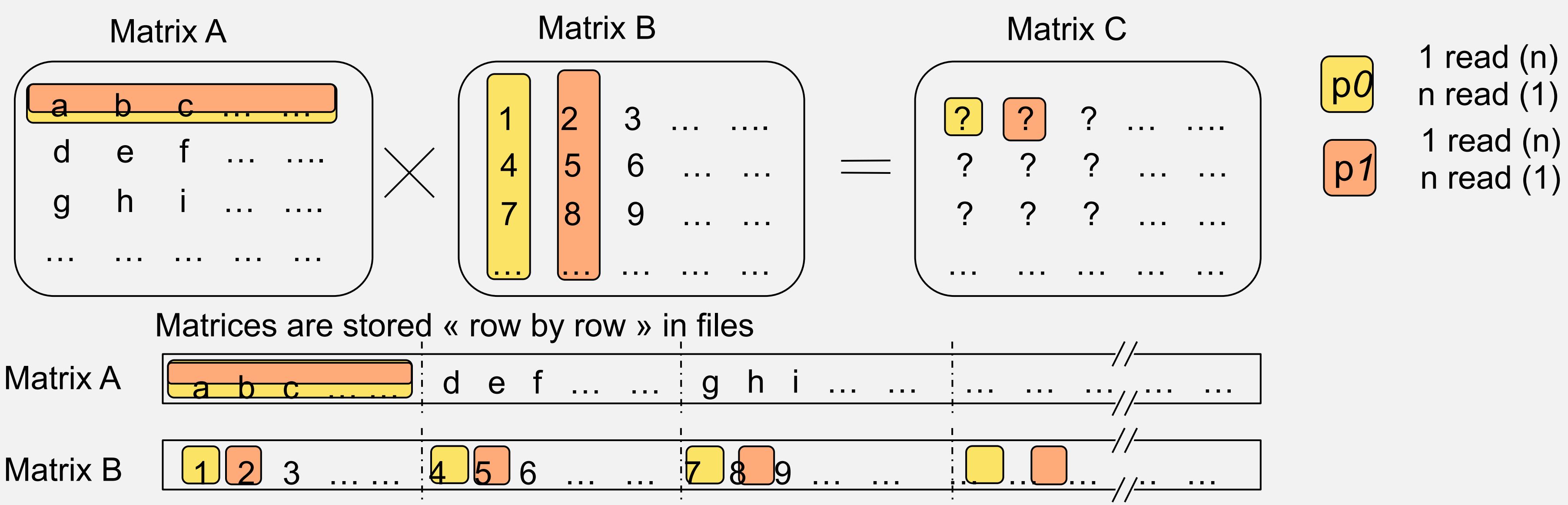
SCALing by means of Ubiquitous Storage

- The objects (the matrices) vs logical view (the files)
- Specific parts to fetch according to the data distribution (row/column,BLOCK/ BLOCK,…) ⇒ File decomposition



Parallel I/O
Parallel Matrix Product

- The objects (the matrices) vs logical view (the files)
- Specific parts to fetch according to the data distribution (row/column,BLOCK/BLOCK,…) ⇒ File decomposition

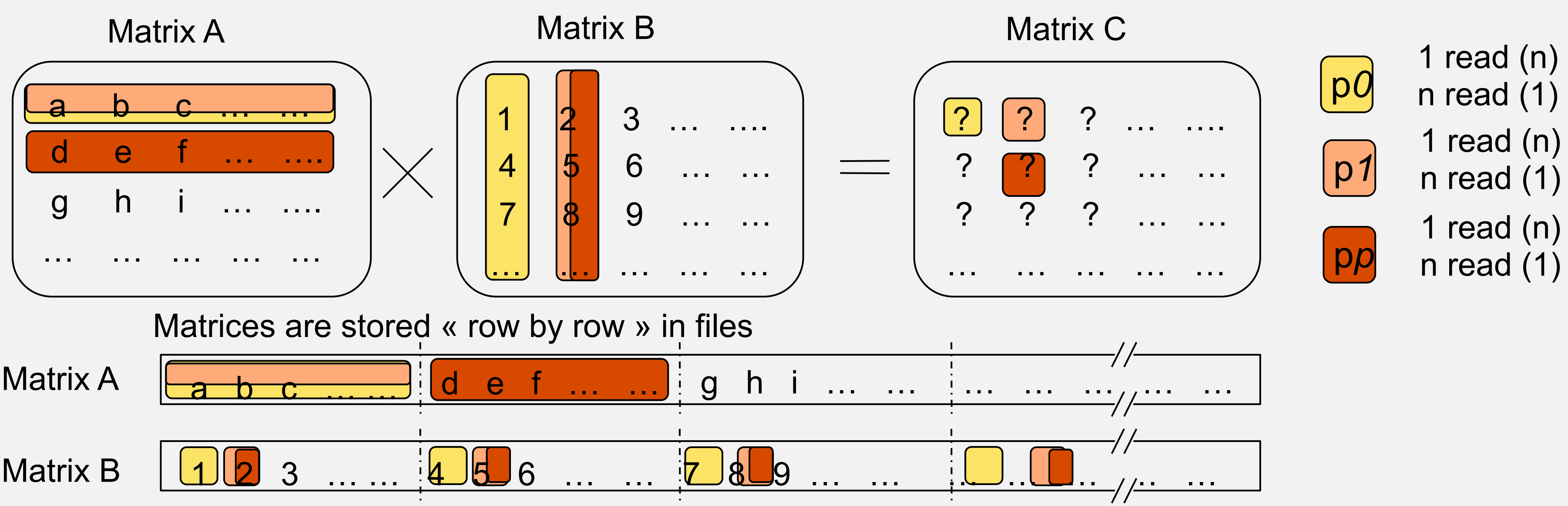


Parallel I/O
Parallel Matrix Product

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage

- The objects (the matrices) vs logical view (the files)
- Specific parts to fetch according to the data distribution (row/column,BLOCK/BLOCK,…) ⇒ File decomposition

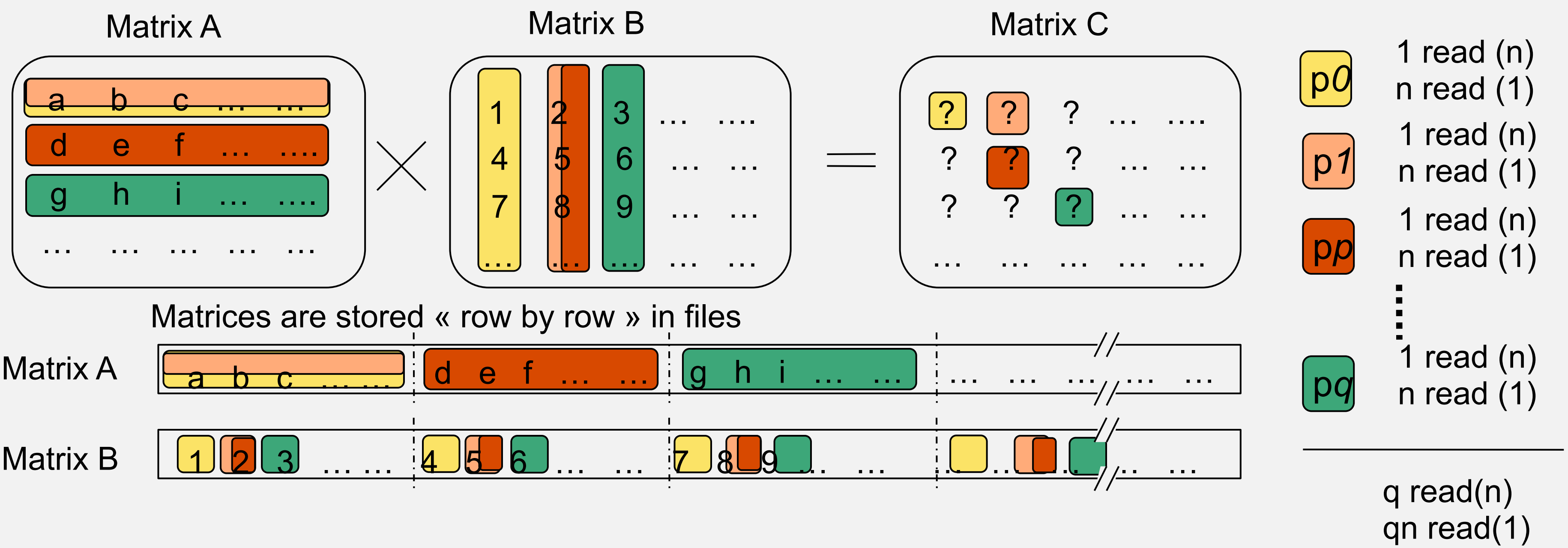


Parallel I/O
Parallel Matrix Product

Marie Curie Initial Training Networks

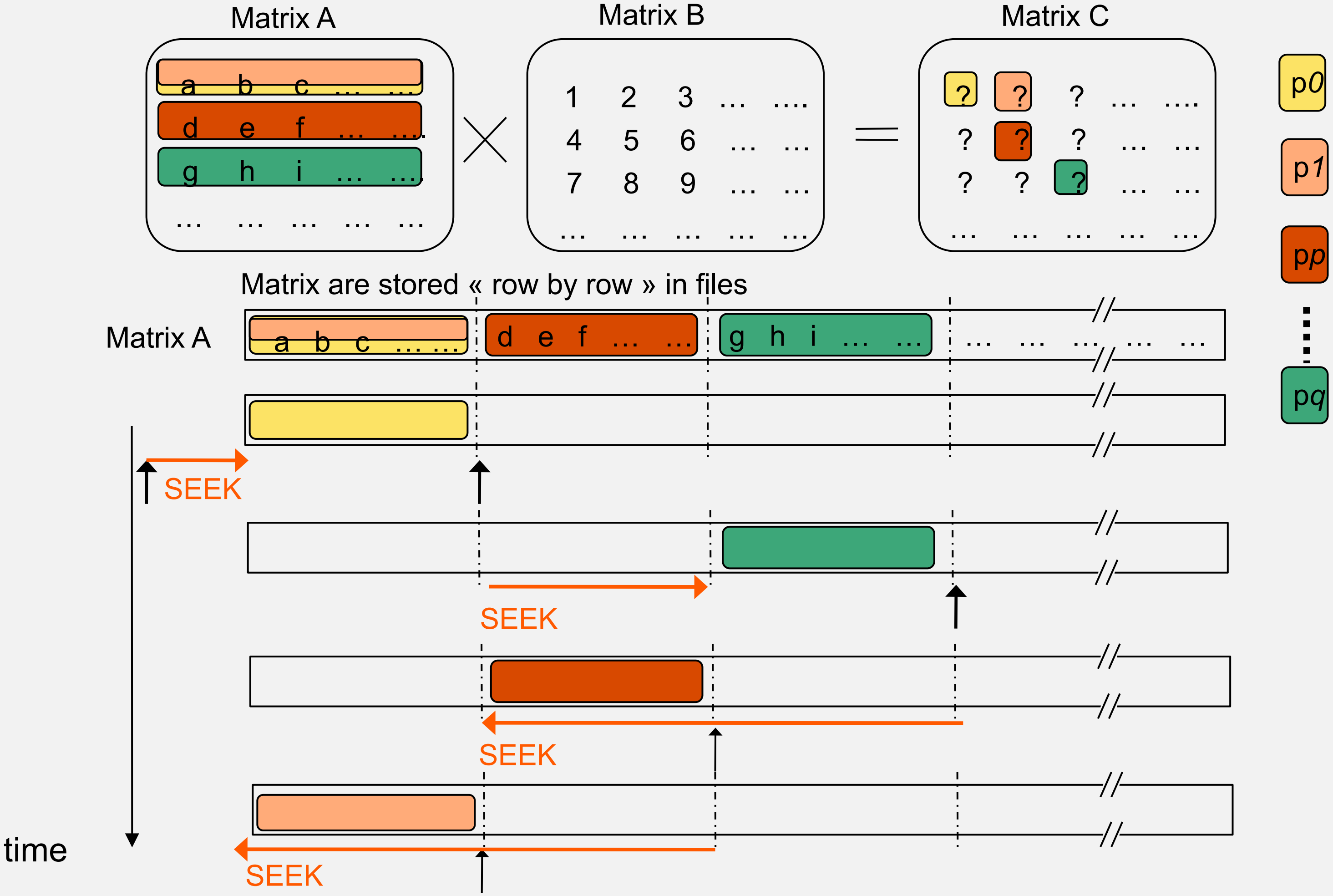
SCALing by means of Ubiquitous Storage

- The objects (the matrices) vs logical view (the files)
- Specific parts to fetch according to the data distribution (row/column,BLOCK/ BLOCK,…) ⇒ File decomposition
- Lot of disjoint/contiguous requests simultaneously
  ⇒"lethal" behavior for I/O subsystems



Parallel I/O
Parallel Matrix Product

- No defined order between requests ⇒ many disk head movements, no cache benefits

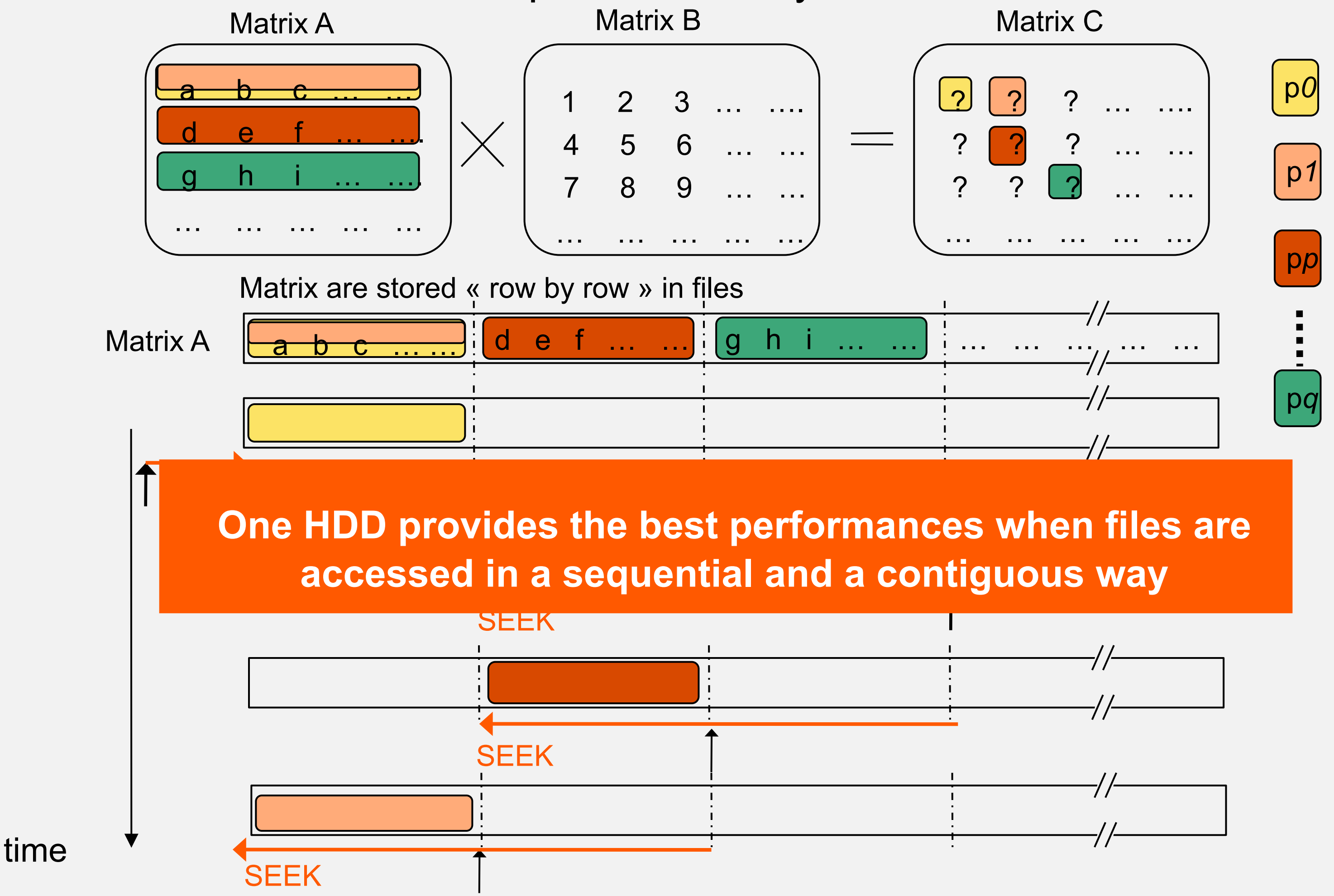


Parallel I/O
Parallel Matrix Product

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage

- No defined order between requests ⇒ many disk head movements, no cache benefits



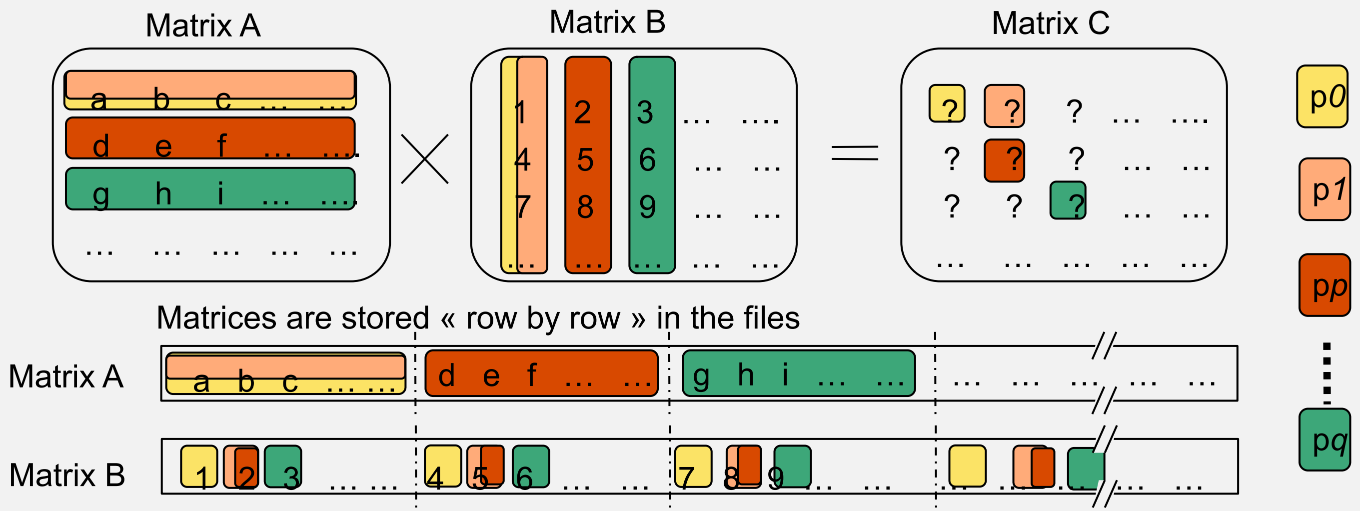
One HDD provides the best performances when files are accessed in a sequential and a contiguous way

Parallel I/O
Parallel Matrix Product

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

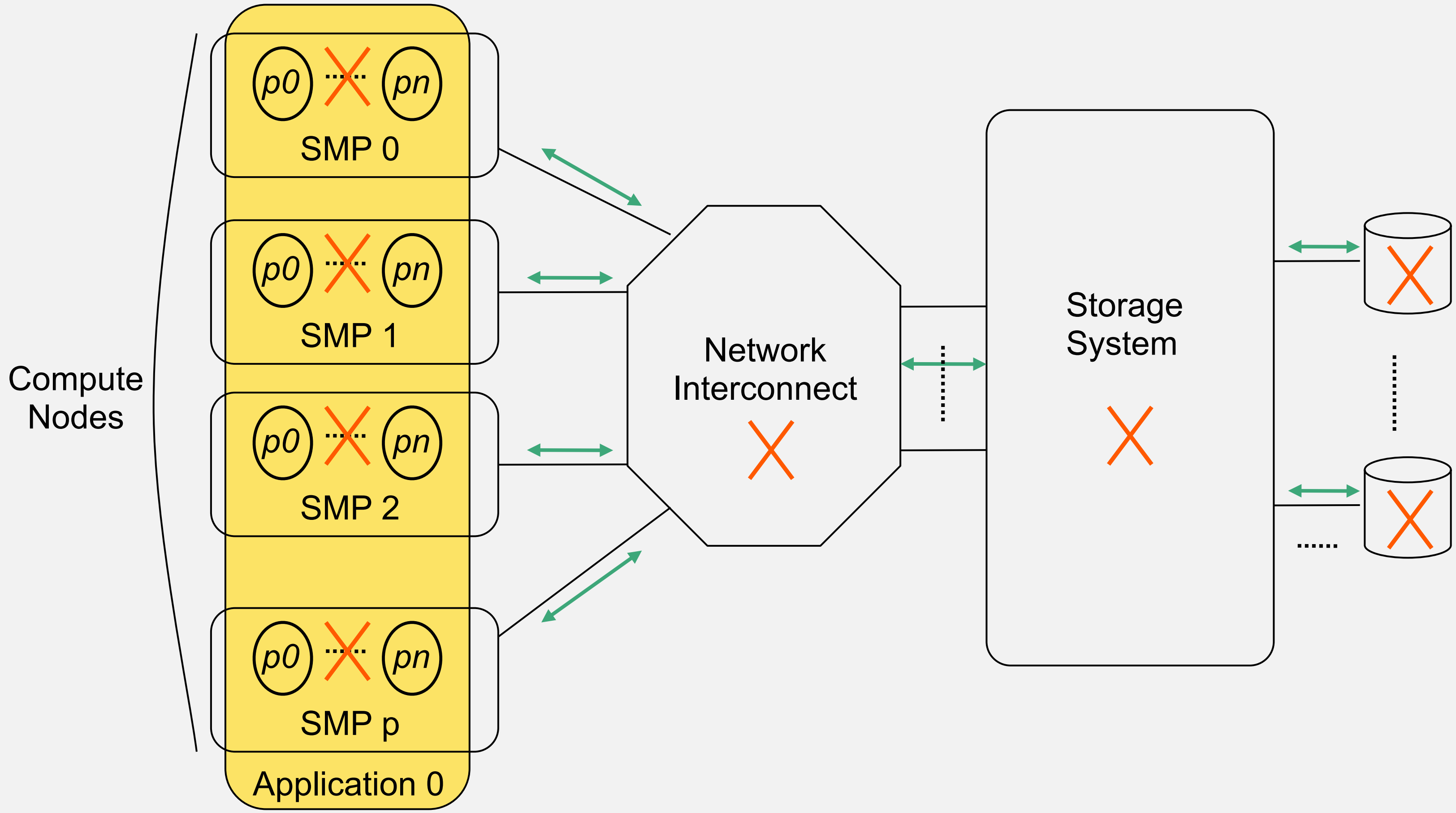
- No defined order between requests ⇒ many disk head movements, no cache benefits



Similar behavior for the matrix B
« random » order between requests

The bigger the number of requests
The bigger the potential number of seeks
⇒ Performance degradation

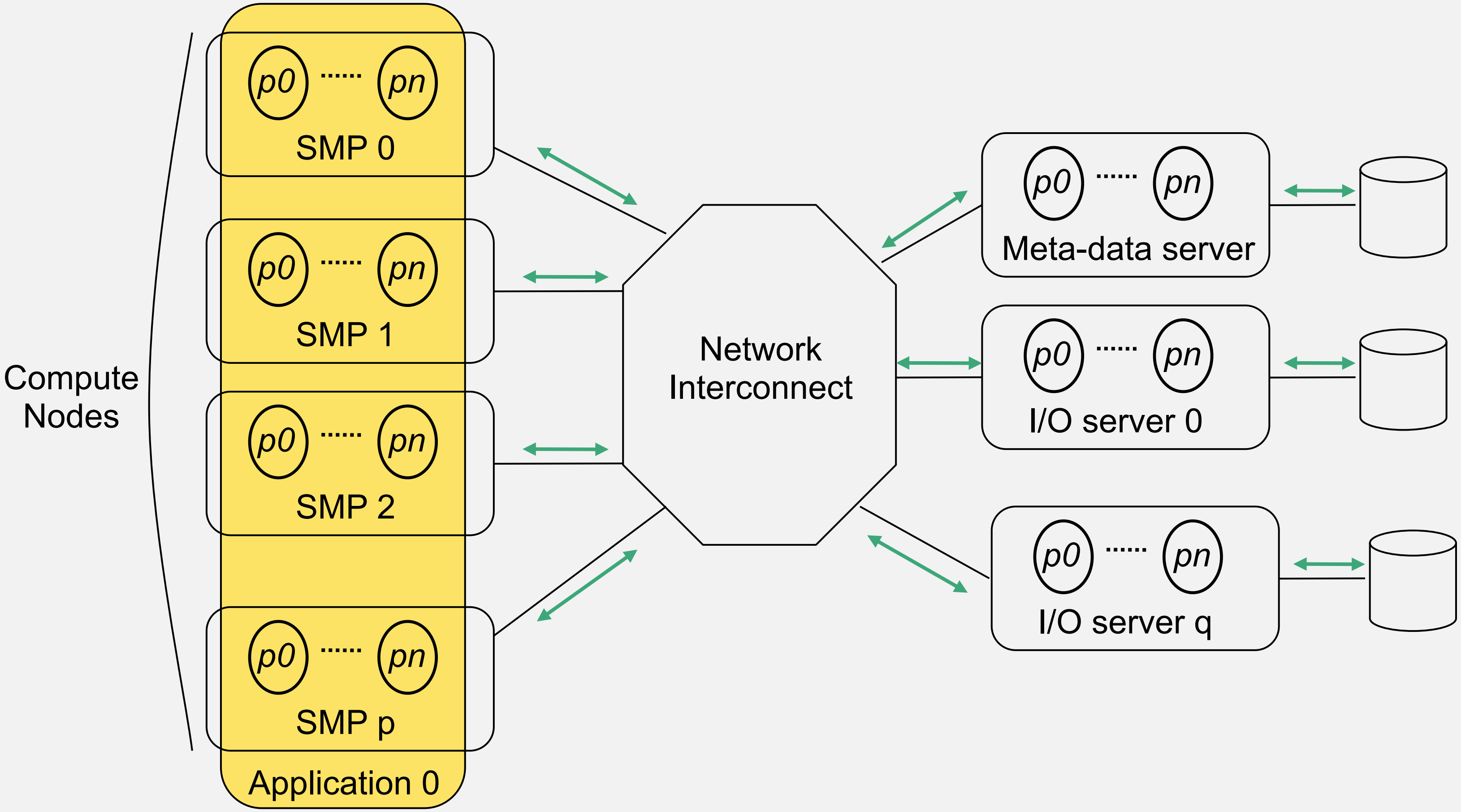
Parallel I/O ⇒Bottlenecks



Parallel I/O Requirements

- Performance constraints
  - *Reduce* the *number of requests*:
    Decrease the overhead implied by *syscalls*
  - Request *Scheduling*:
    Avoid expensive seeks and maximize large accesses
  - Exploit *cache* mechanisms:
    Benefit from read-ahead strategies, write behind…
    Taking into consideration *logical view* (objects/files) vs. *physical placement* (HDD blocks)

Solution 1: Parallel File Systems ⇒ Balance requests between several servers



Parallel File System

- Load balancing on several servers
- Two approaches :
  - Dedicated to parallel I/O : PIOUS, VESTA
    Logical view/physical placement, deprecated (gave up in the time)
  - More generic: PVFS, Lustre, NFS V4.1, ... GPFS
    (performance/coherence/fault tolerance….)
  - +/- finalized, +/- efficient, +/- intrusive (dedicated APIs at client side)
  - No real I/O scheduling policies
    (most of them rely on the low level scheduler and only on server-side)
- The performance depend on the striping policy of the file system
  (but in general, they do not take into account the striping of applications)

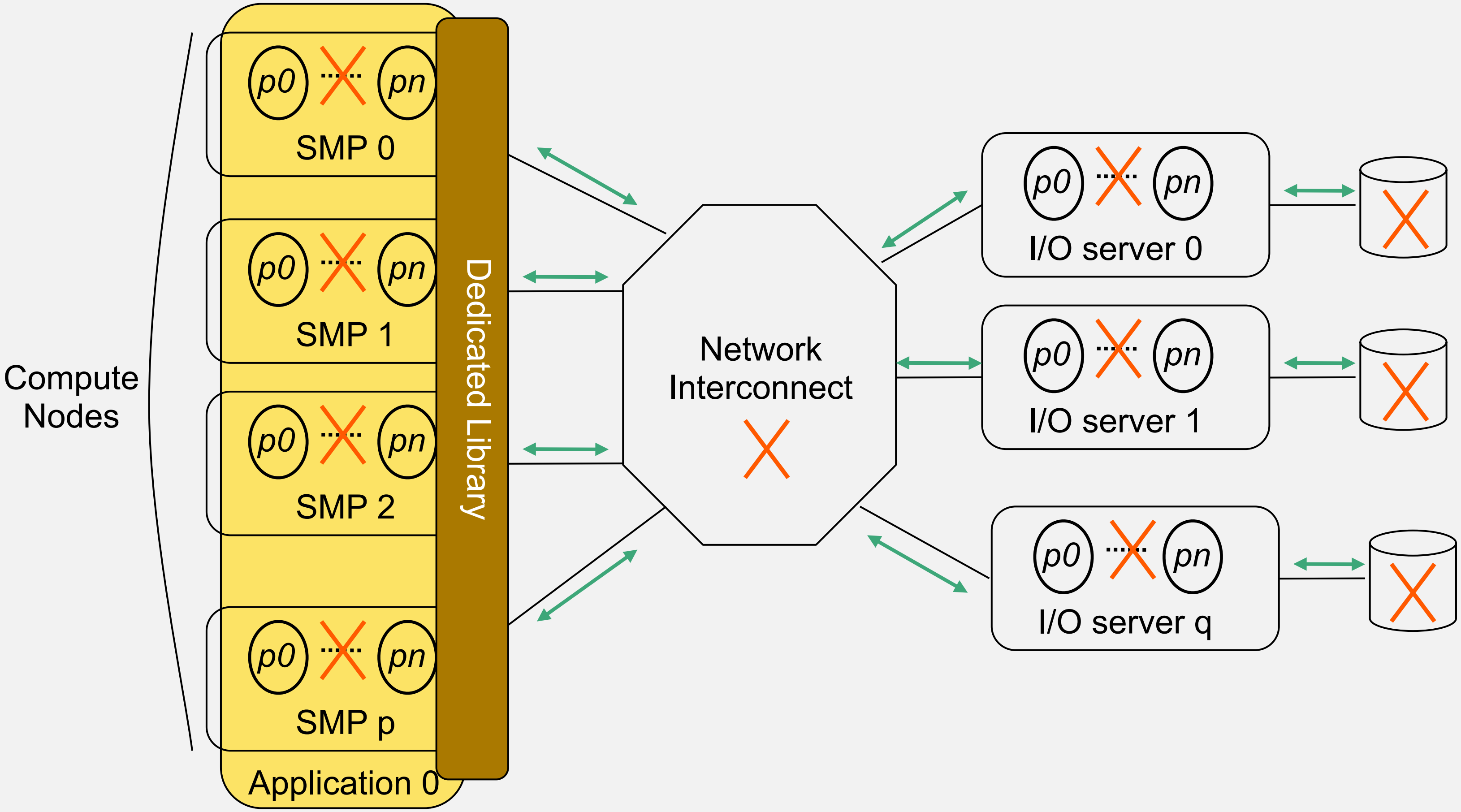
Linux
Low-level schedulers

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage

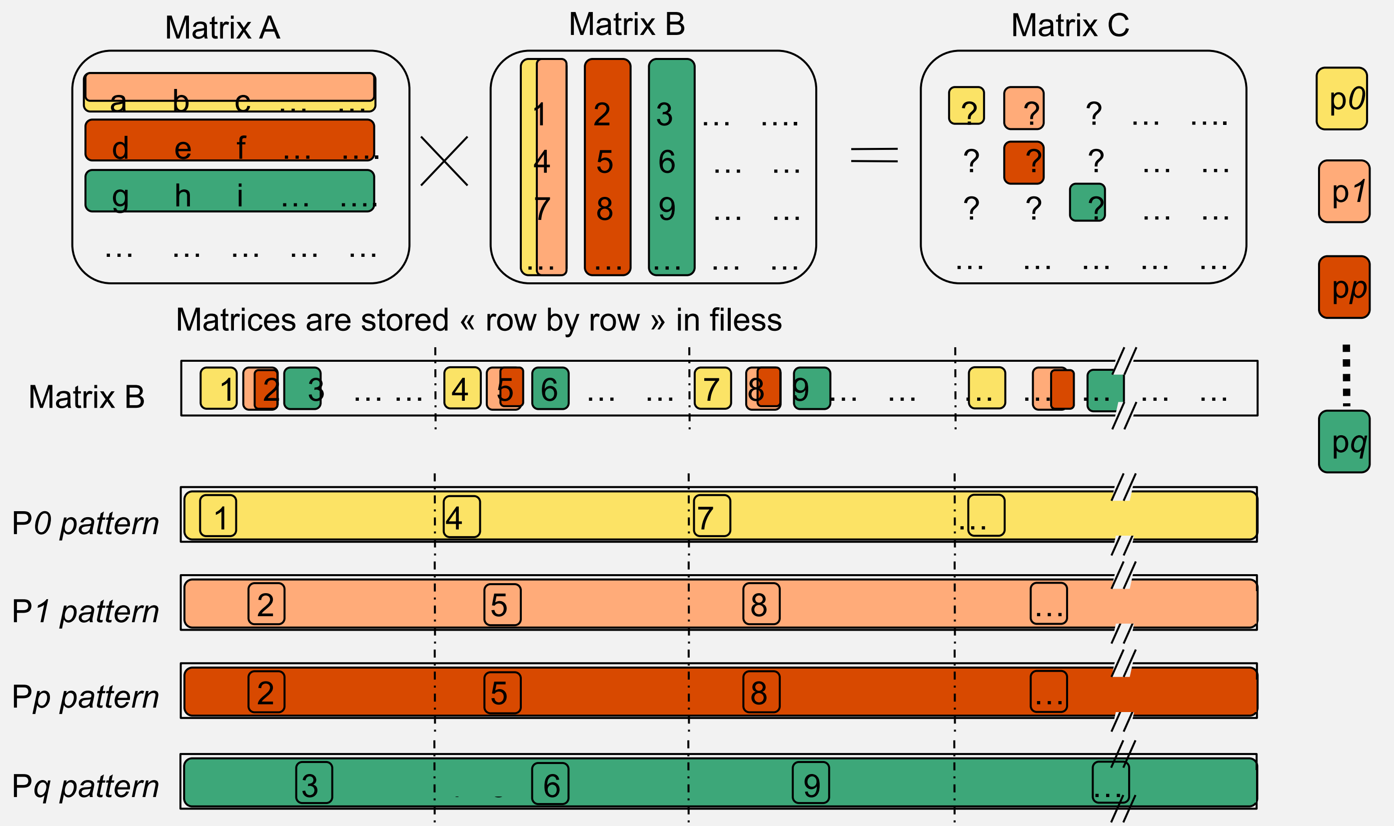
- 4 policies available in the Gnu/Linux system:
  - *Elevator / Deadline :*
    - *Linus elevator* : requests are sorted in a list based on the location of each block on the HDD ⇒ *starvation* issue
    - *Deadline* : each request is associated to a time-stamp (a "deadline")
    - According to the access type (read or write), each requests is inserted in a « deadline » list.
  - *Anticipatory :*
    - Deadline + non conservative algorithm
    - A short break is voluntary made before dealing the next request
A non conservative approach enables to receive new requests potentially contiguous
  - *Complete Fair Queuing*
    - Each process maintains a sorted list (similar to the elevator one)
The scheduler serves the requests in a round robin manner
  - *Noop :*
    - *« No operation », FIFO* policies (using aggregation mechanisms if possible)

Solution 2 : dedicated libraries, MPI I/O [MPI2, 1997]



Dedicated I/O Libraries

- Definition of access patterns
  (similar to the SQL views or access vectors like readv or writev POSIX calls)
  ⇒ Reduce number of requests

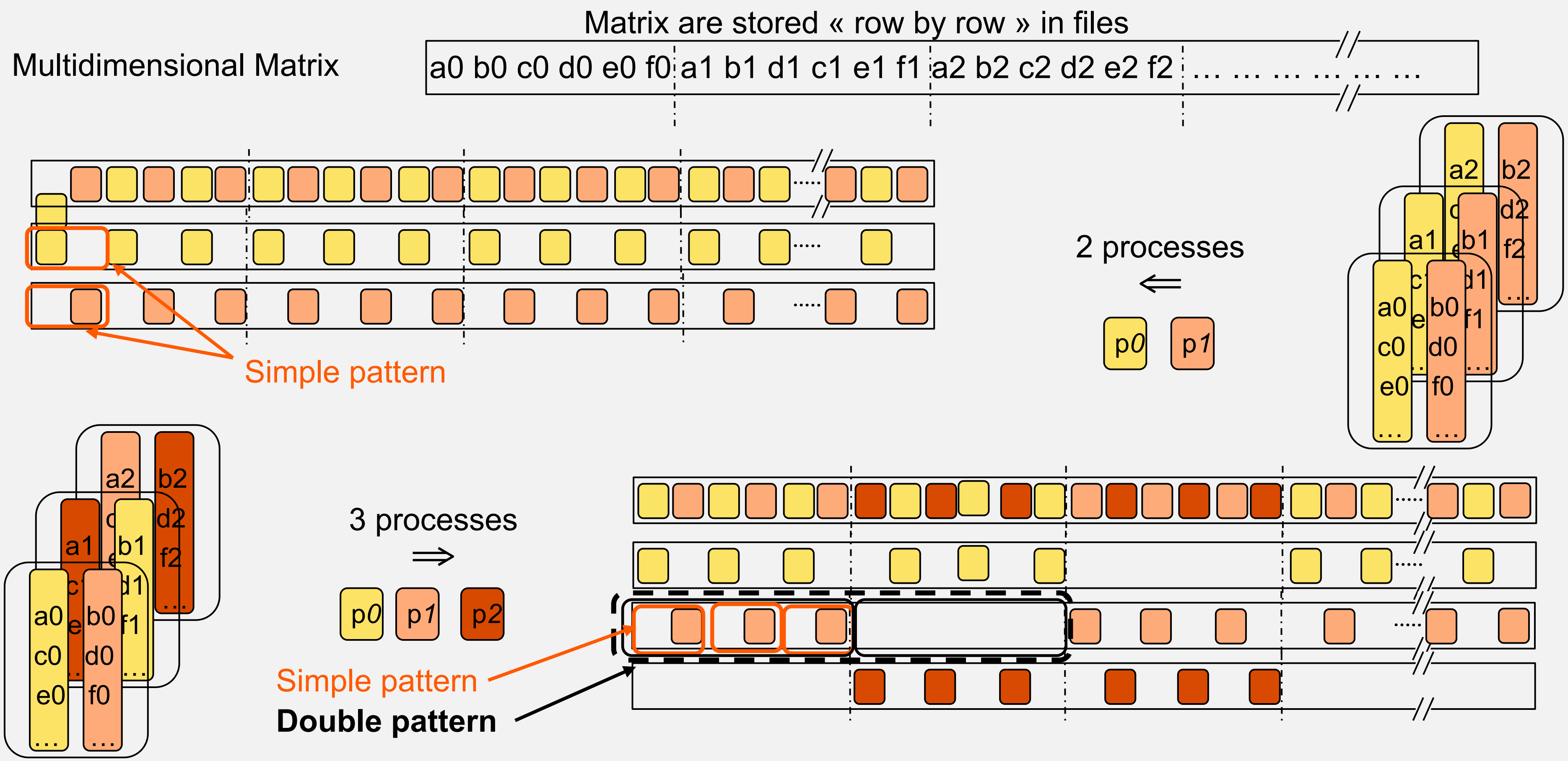


Dedicated I/O Libraries

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

- « Simple » access pattern vs « structured » access pattern
  (« *simple stride* » vs « *nested stride*»)



Dedicated I/O Libraries

- Take advantage of access patterns to used advanced mechanisms (mainly request aggregations)

- Access behavior: independent or collective operations
- Shared or traditional file pointer

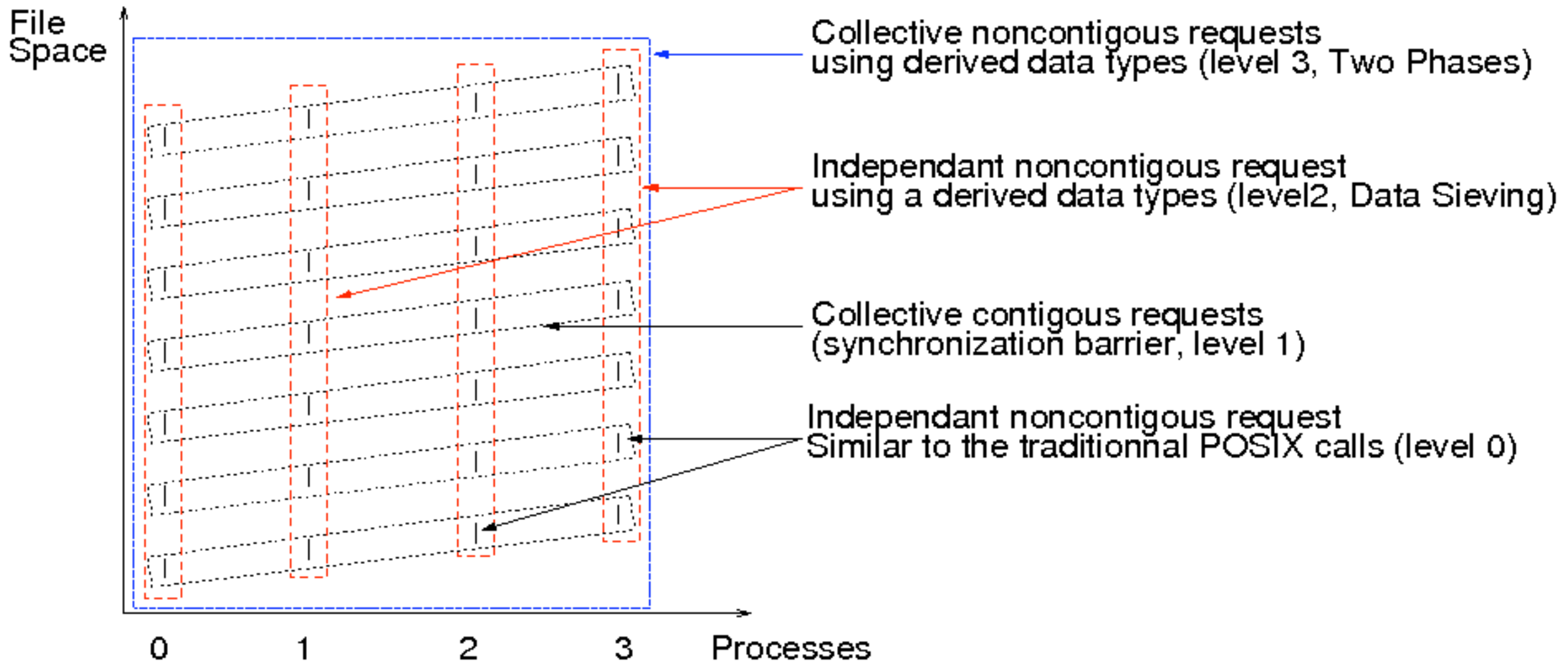
  ⇒ at node scale, group scale, cluster scale….
- 3 techniques :
  - *''Stream-Based I/O"*
    - Several requests are encapsulated in a composite one.
    - Requires a particular API *(PVFS, ''list I/O")*

  - Mono-process, ''*data-sieving*'',
    - Overlapped access (one larger access overlapping smaller ones)
    - Weakness : each write implies a read-modify-write operation

  - Multi-processes (collective approaches), process coordination to define an efficient access policy (''Two Phase'')

MPI I/O - Romio

- MPI I/O, Parallel I/O interface standardization (1997)
- Several implementations : ROMIO, the most famous (*mpich*, Open MPI)
  - 4 levels :



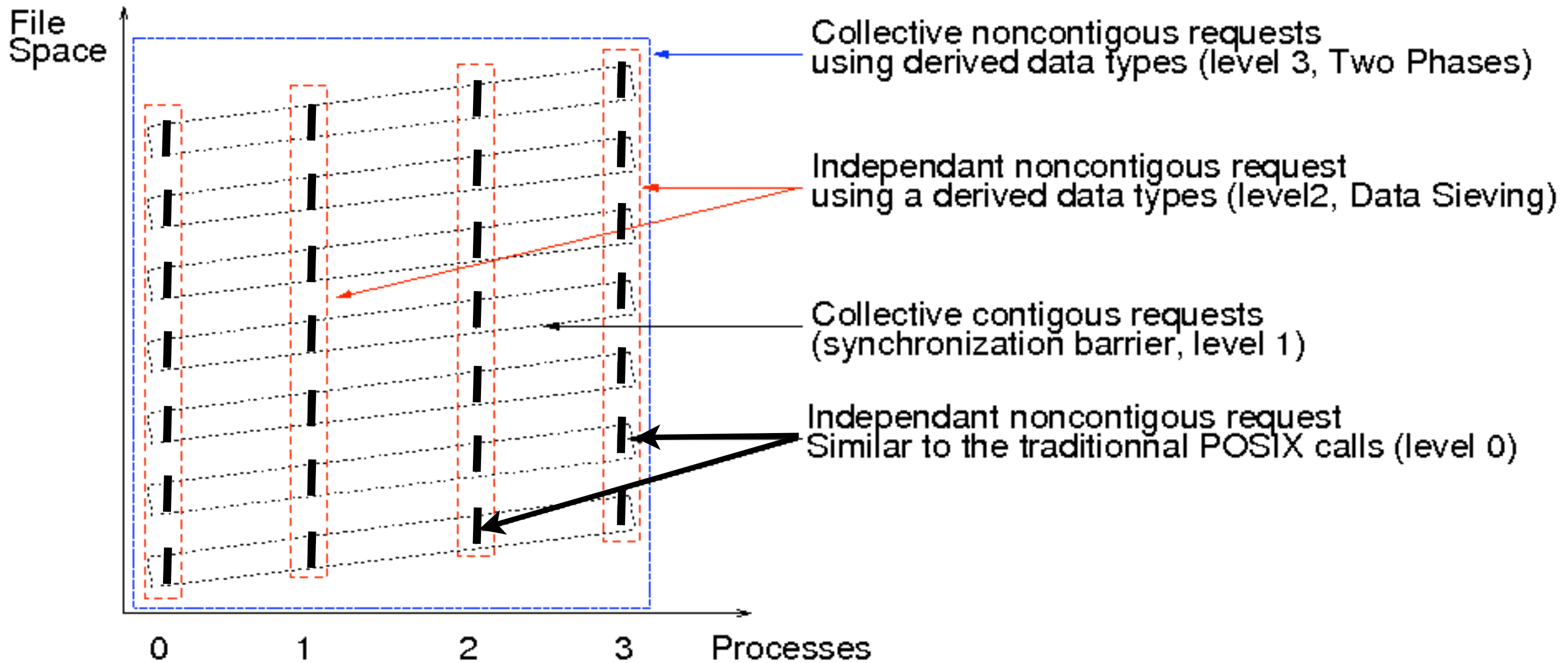
MPI I/O – Four levels [Thakhur/Gropp/Lusk02]
Representing increasing amounts of data per request

MPI I/O - Romio

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage

- MPI I/O, Parallel I/O interface standardization (1997)
- Several implementations : ROMIO, the most famous (*mpich*, Open MPI)
  - 4 levels :



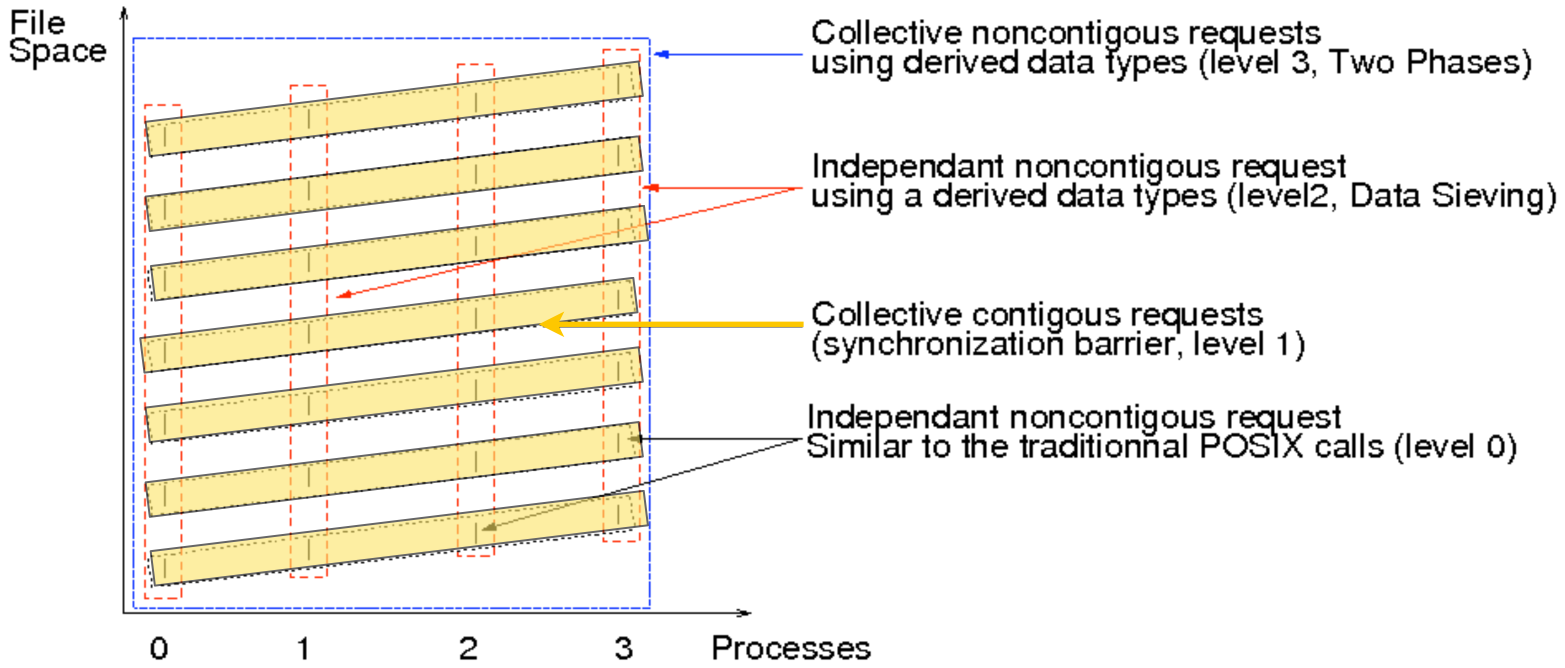
MPI I/O – Four levels [Thakhur/Gropp/Lusk02]
Representing increasing amounts of data per request

MPI I/O - Romio

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage

- MPI I/O, Parallel I/O interface standardization (1997)
- Several implementations : ROMIO, the most famous (*mpich*, Open MPI)
  - 4 levels :



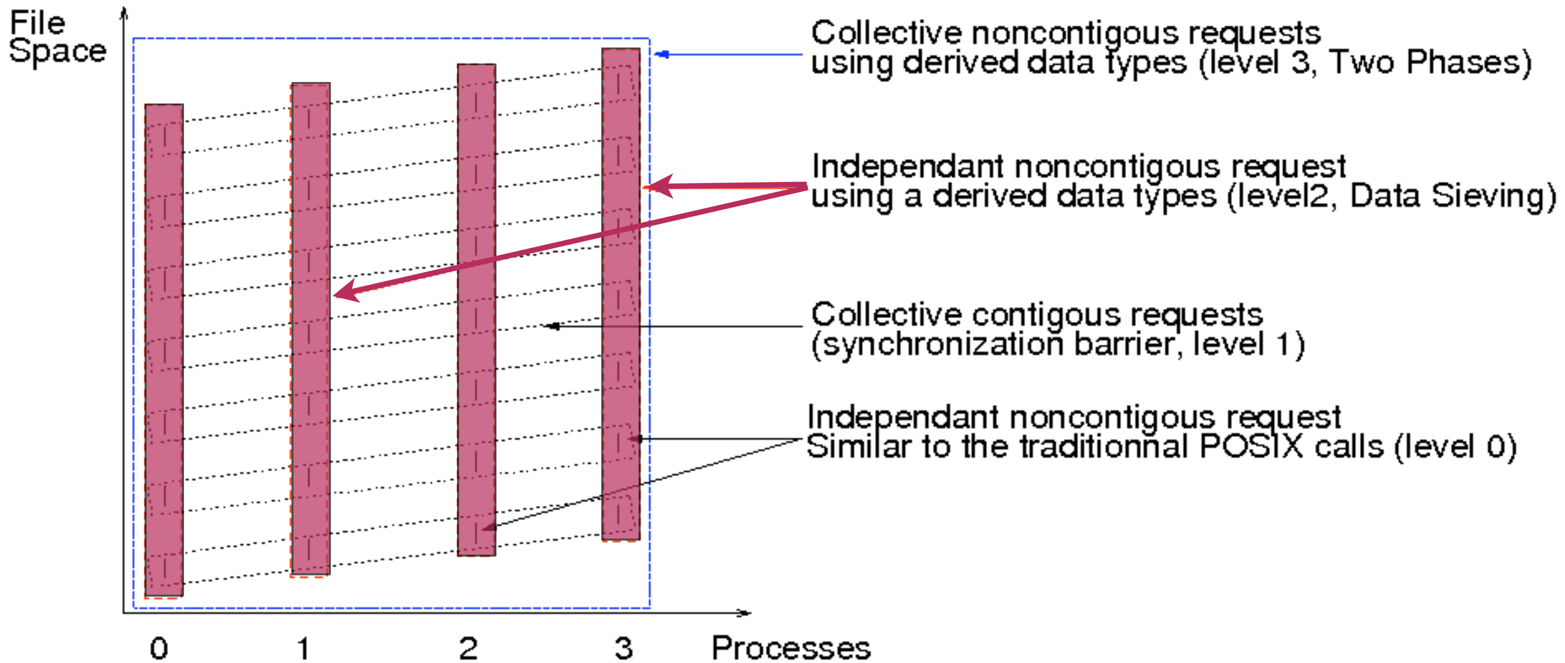
MPI I/O – Four levels [Thakhur/Gropp/Lusk02]
Representing increasing amounts of data per request

MPI I/O - Romio

SCALing by means of Ubiquitous Storage

Marie Curie Initial Training Networks

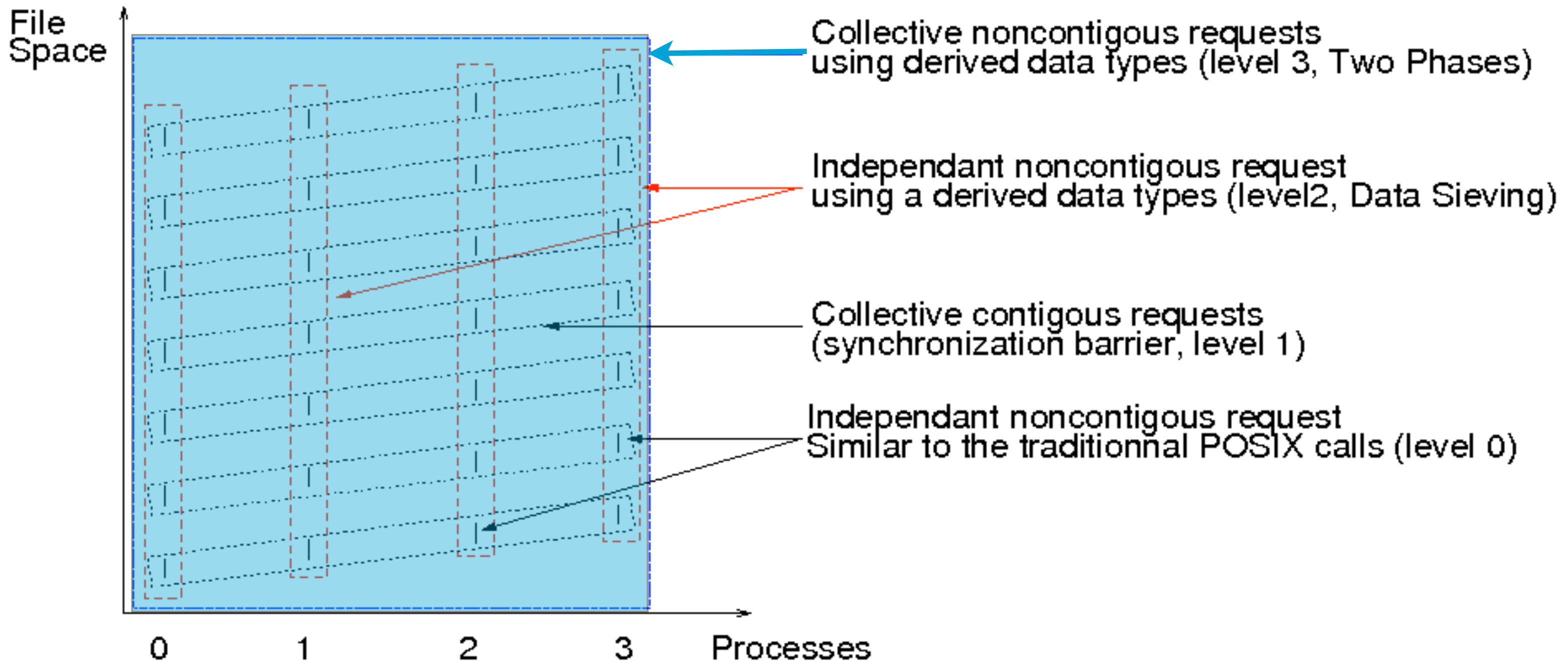
- MPI I/O, Parallel I/O interface standardization (1997)
- Several implementations : ROMIO, the most famous (*mpich*, Open MPI)
  - 4 levels :



MPI I/O – Four levels [Thakhur/Gropp/Lusk02]
Representing increasing amounts of data per request

MPI I/O - Romio

- MPI I/O, Parallel I/O interface standardization (1997)
- Several implementations : ROMIO, the most famous (*mpich*, Open MPI)
  - 4 levels :



MPI I/O – Four levels [Thakhur/Gropp/Lusk02]
Representing increasing amounts of data per request

MPI I/O - Romio

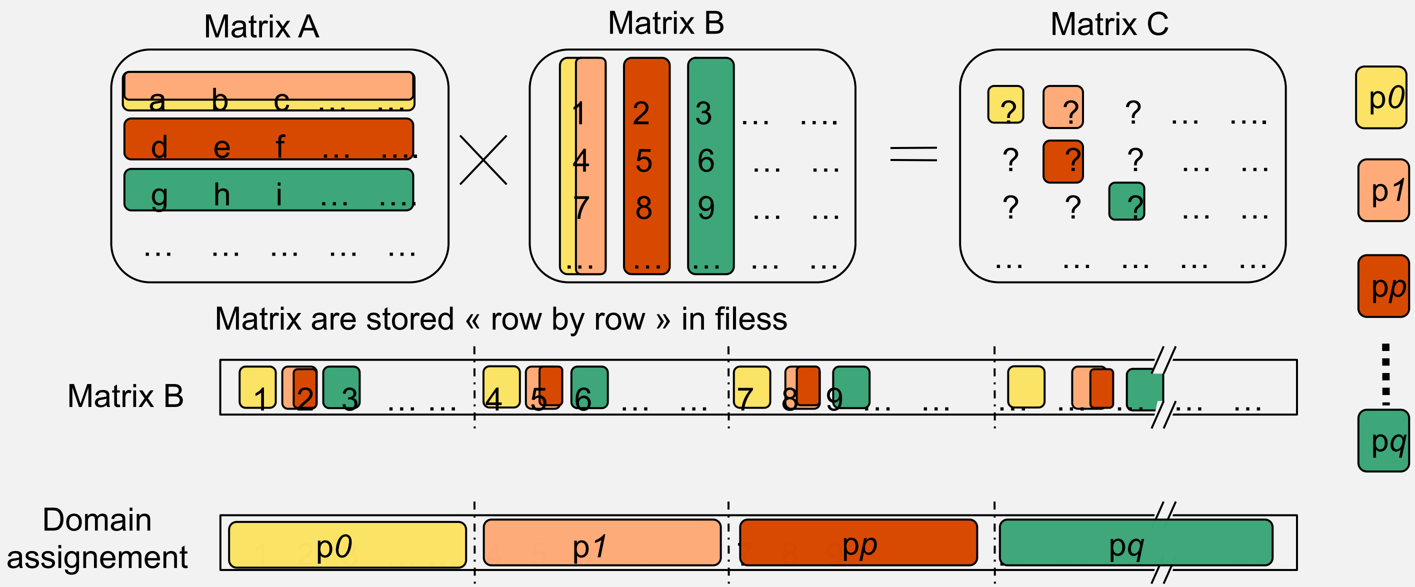
- ROMIO, ''Two phase'' approach (coordination at application level)

1./ Each process exchanges its own access pattern with other participants

«domain» assignment (accesses are sorted and then assigned to particular nodes)

Each node manipulates the data associated to its domain

2./ Each node forwards the data to the right nodes (message exchanges)



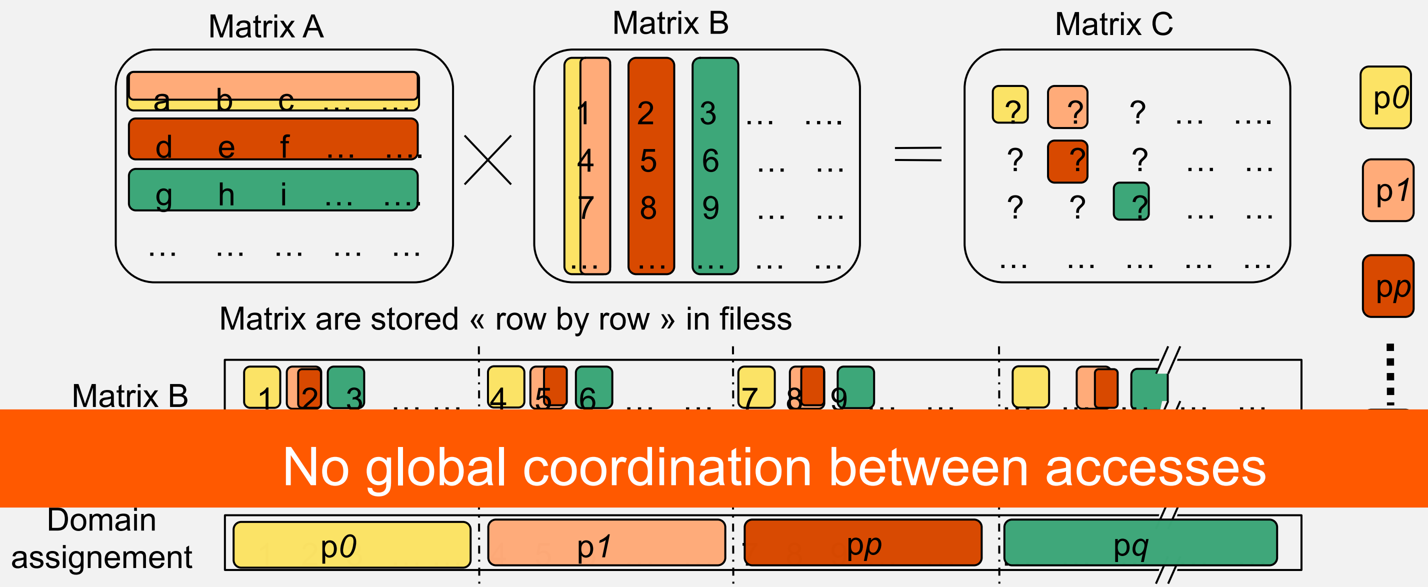
- ROMIO, "Two phase" approach (coordination at application level)

  1./ Each process exchanges its own access pattern with other participants

  «domain» assignment (accesses are sorted and then assigned to particular nodes)

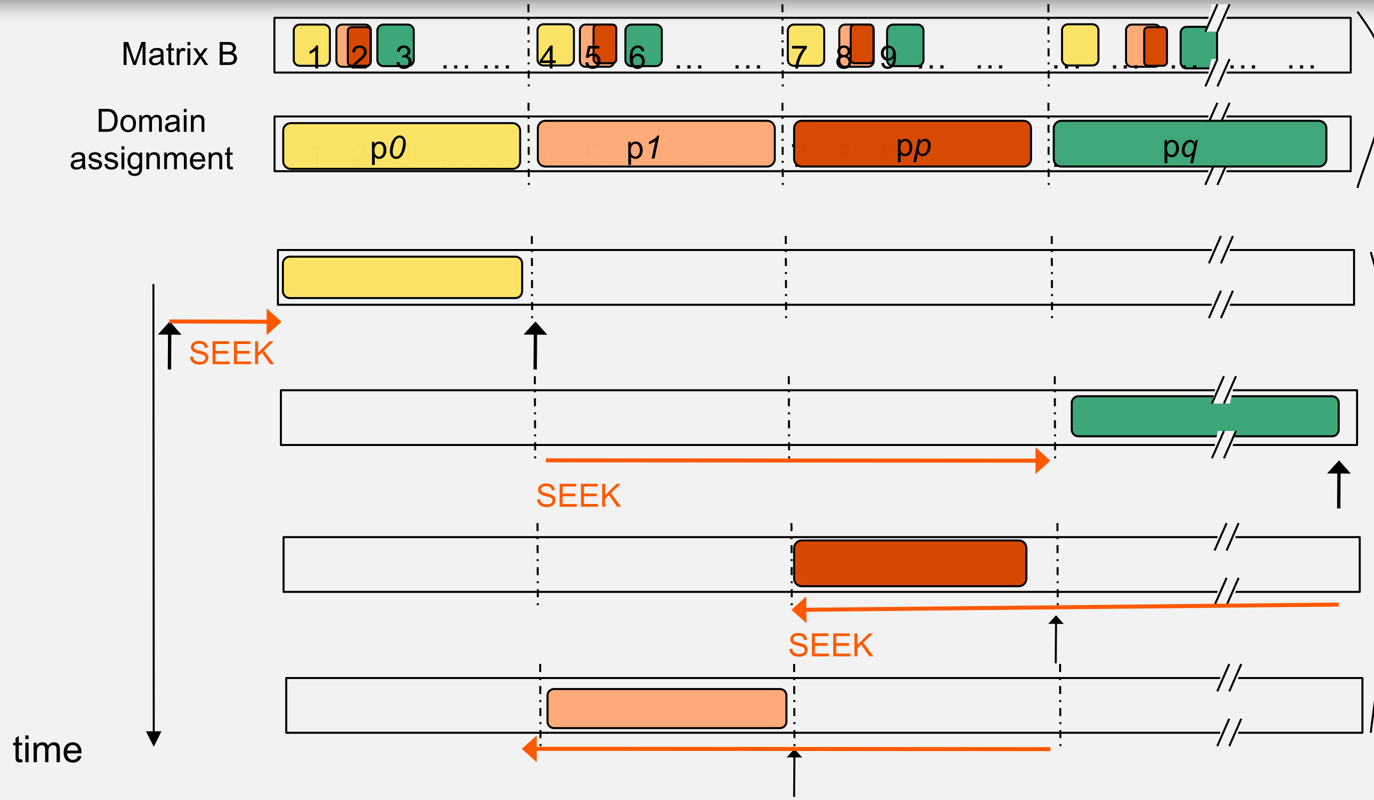
  Each node manipulates the data associated to its domain

  2./ Each node forwards the data to the right nodes (message exchanges)



No global coordination between accesses

MPI I/O - Romio



At client side 'Two Phase' enables to query larger requests

At server side

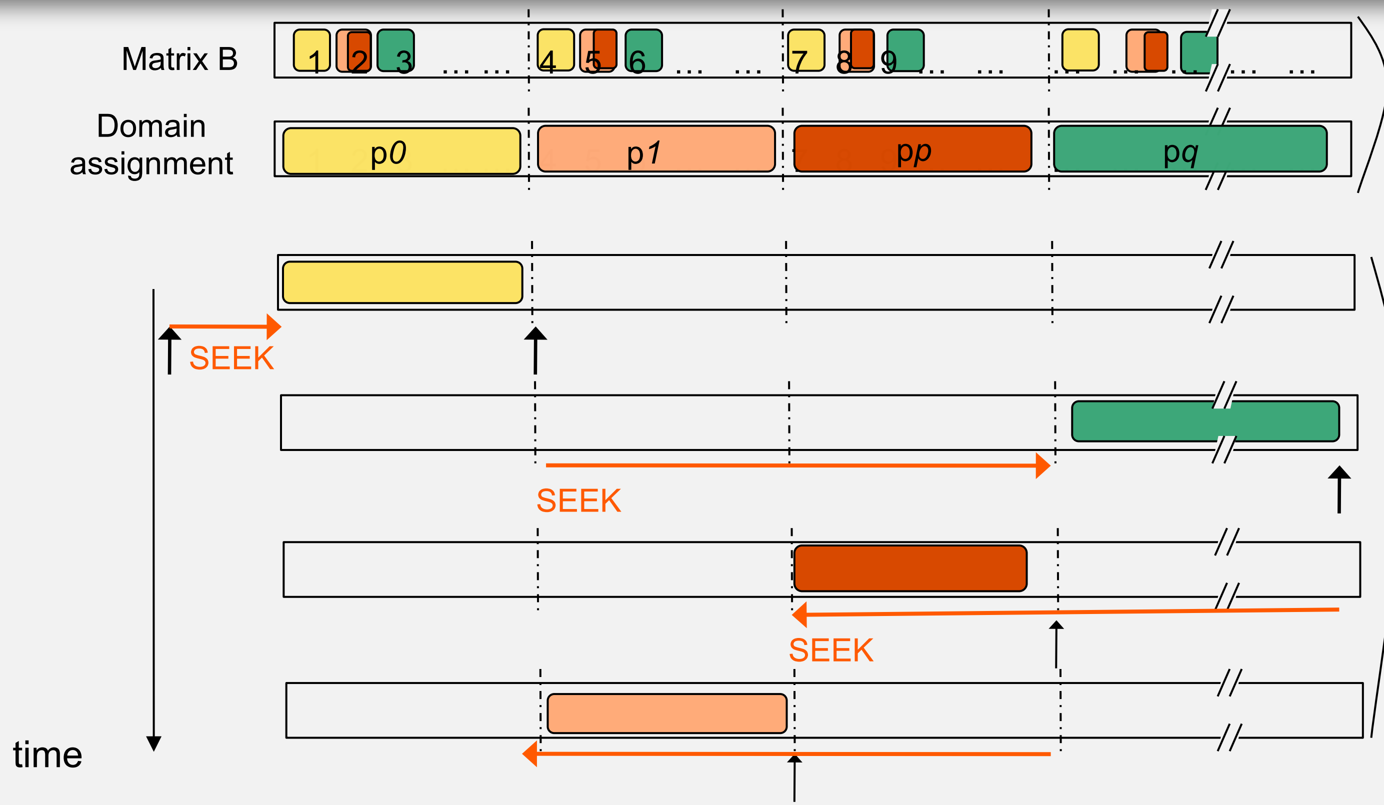
Each access is sent in an independent and a parallel way.

No global coordination between accesses

MPI I/O - Romio

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage



At client side 'Two Phase' enables to query larger requests

At server side

Each access is sent in an independent and a parallel way.

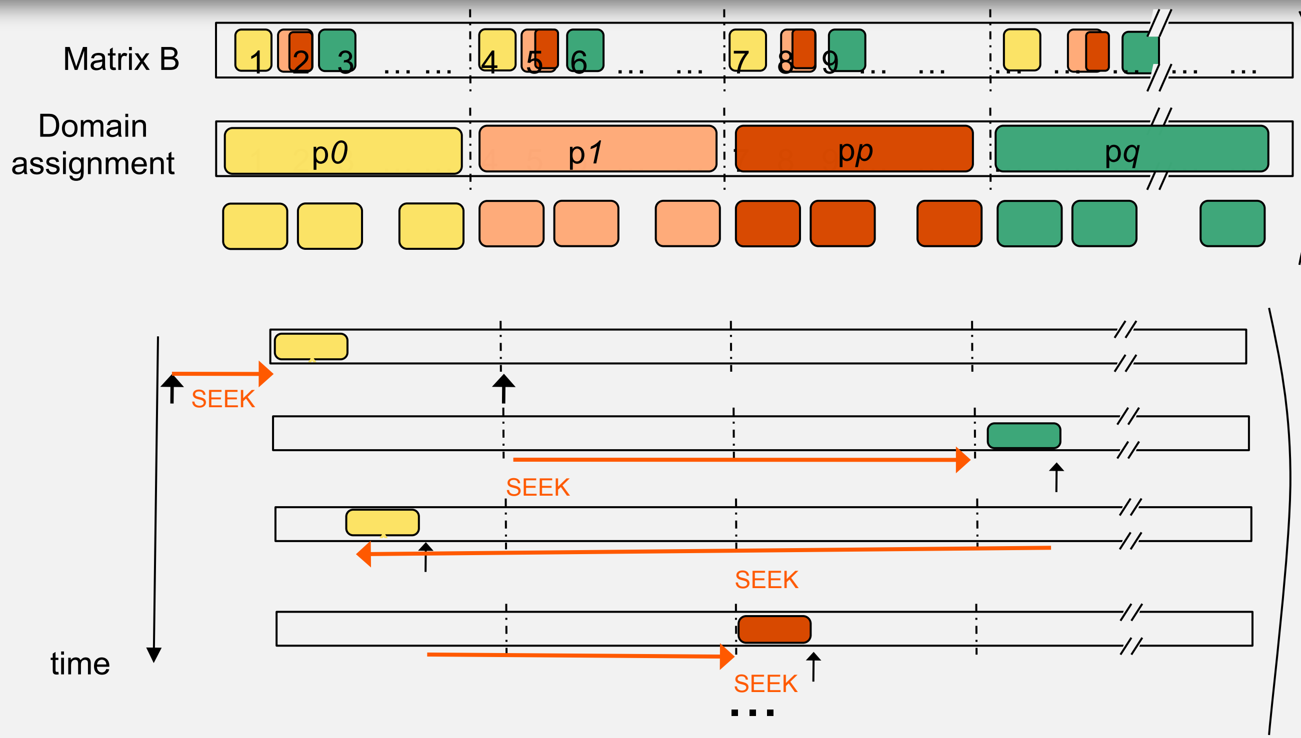
No global coordination between accesses

It can be even worse !

MPI I/O - Romio

Marie Curie Initial Training Networks

SCALing by means of Ubiquitous Storage



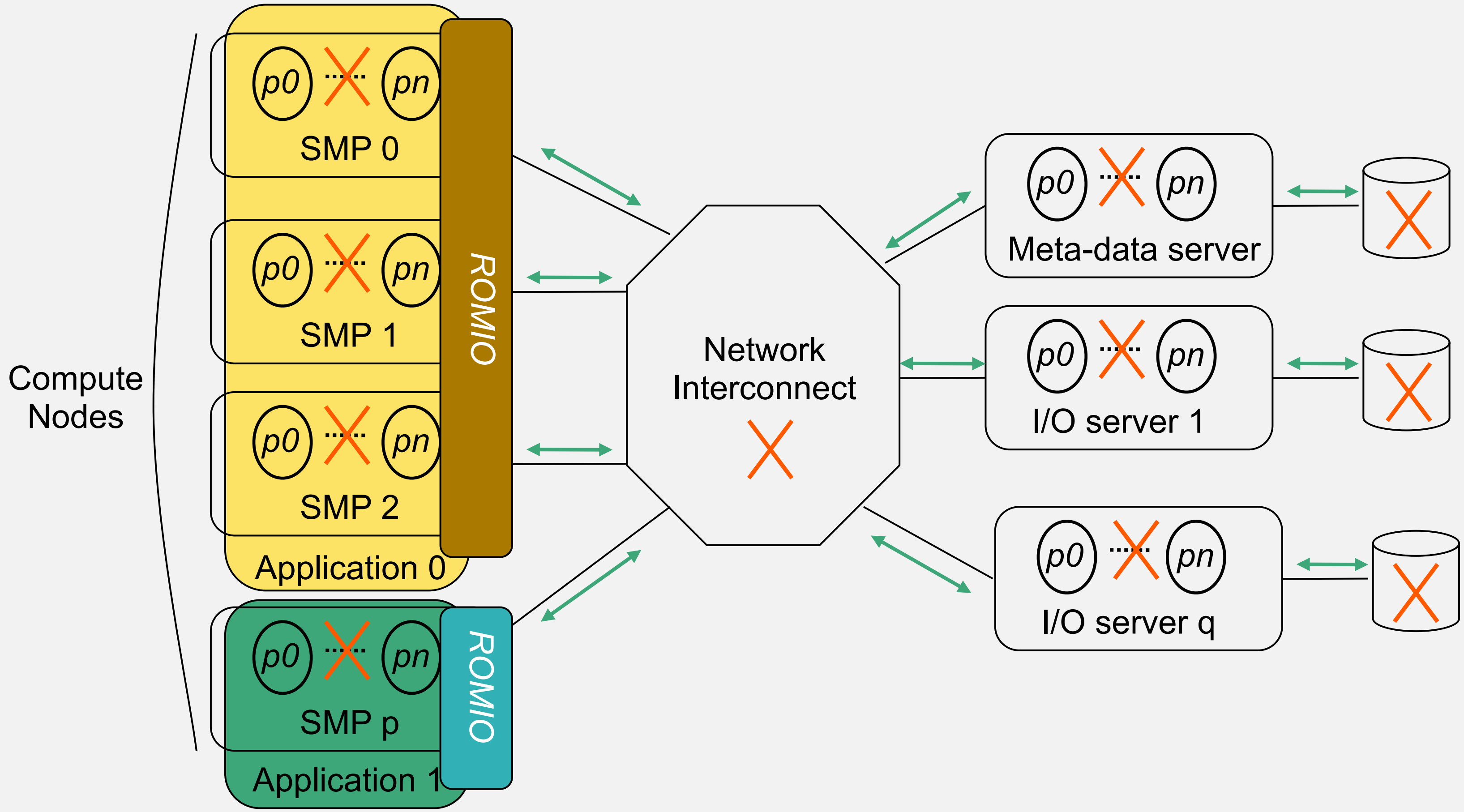
At client side 'Two Phase' requests may be divided in smaller ones according to the FS granularity

At server side

Each access is sent in an independent and a parallel way.

- Use of additional calls to define a particular order («hints»)
- From a global point of view, the performance may be improved but:
  - Sophisticated API ⇒ Development Overhead / Language Bindings
  - Requires to know the whole details of the I/O path (FS granularity, caches, ...)

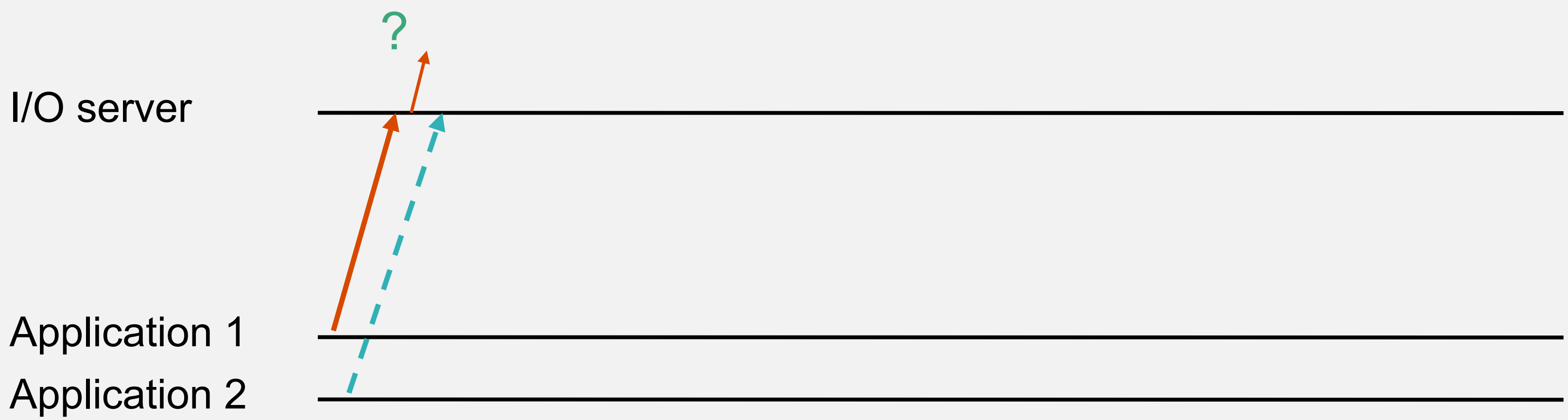
Dedicated I/O libraries and multi-application environments



Multi-apps Environment

- Synchronous behavior

(2 concurrent applications execute a "cat" like operation)



No informations about applications, only about files

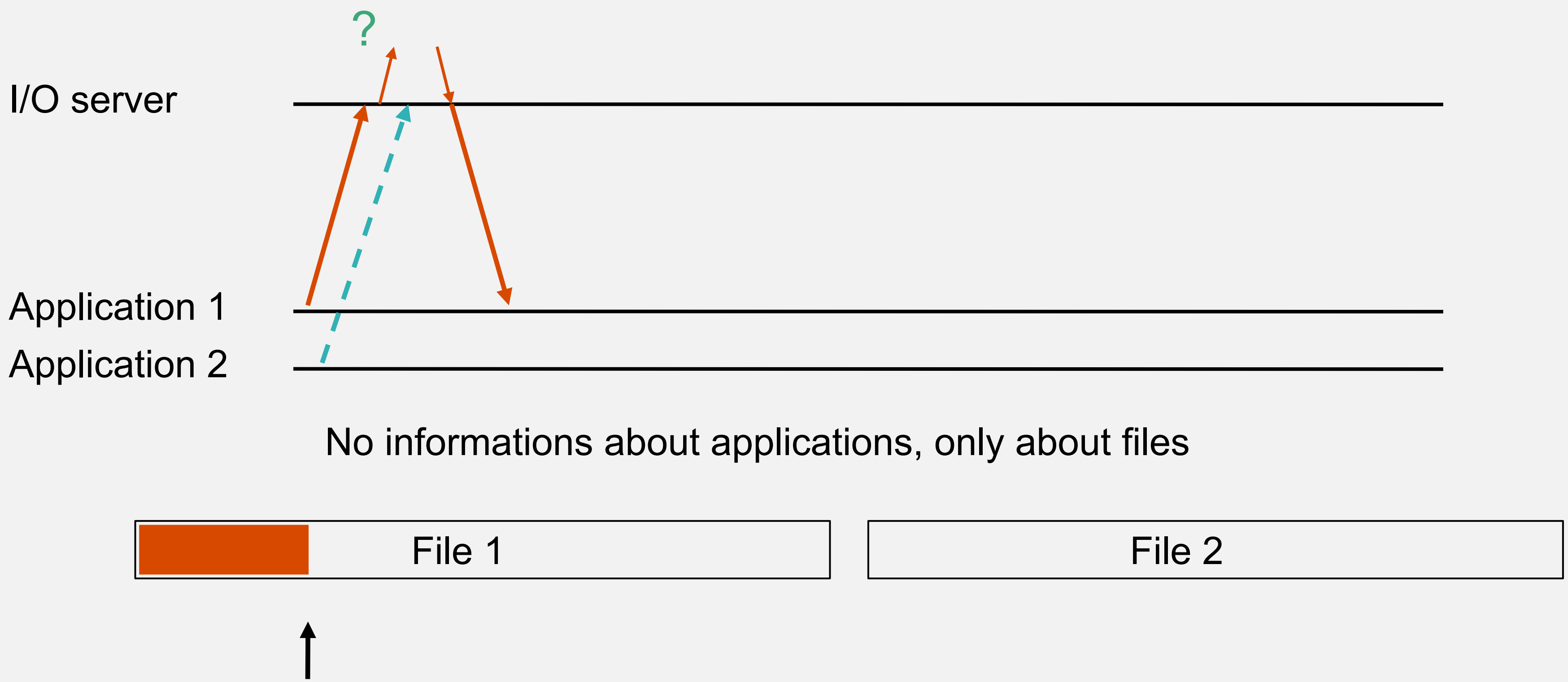
File 1 | File 2

SEEK ?

Multi-apps Environment

- Synchronous behavior

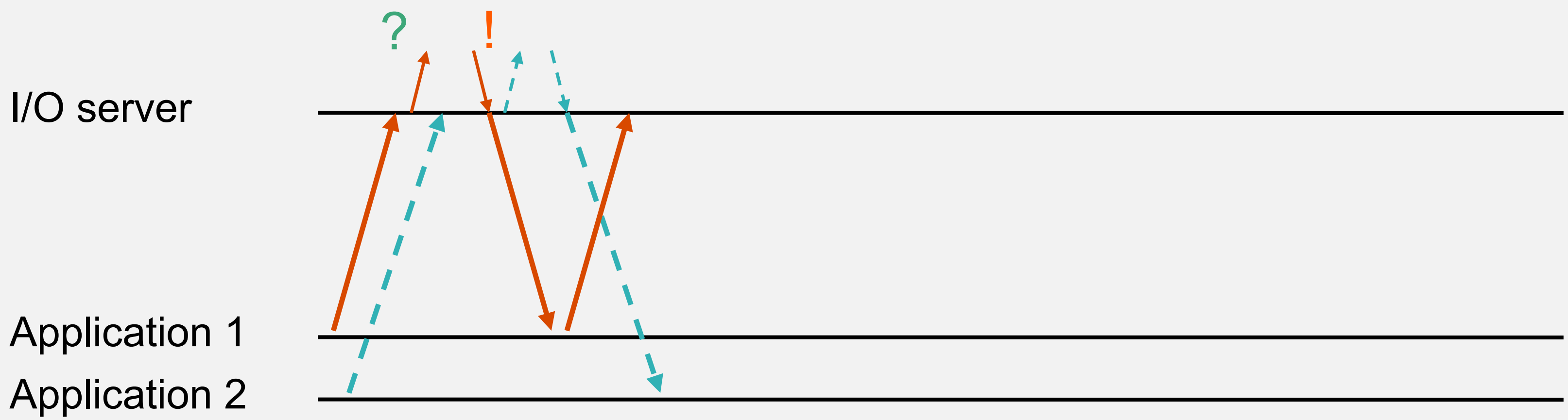
(2 concurrent applications execute a “cat” like operation)



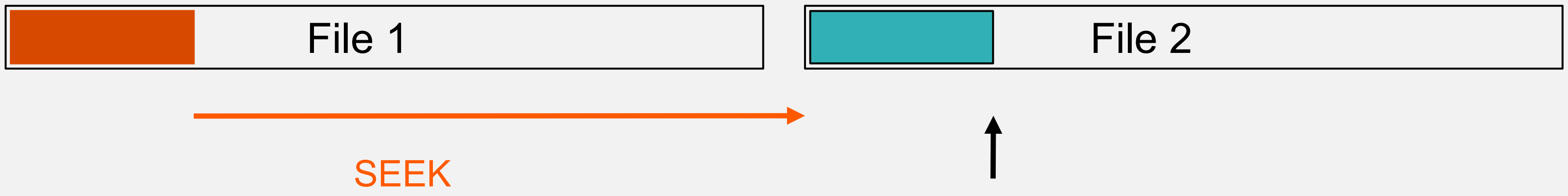
Multi-apps Environment

- Synchronous behavior

(2 concurrent applications execute a “cat” like operation)



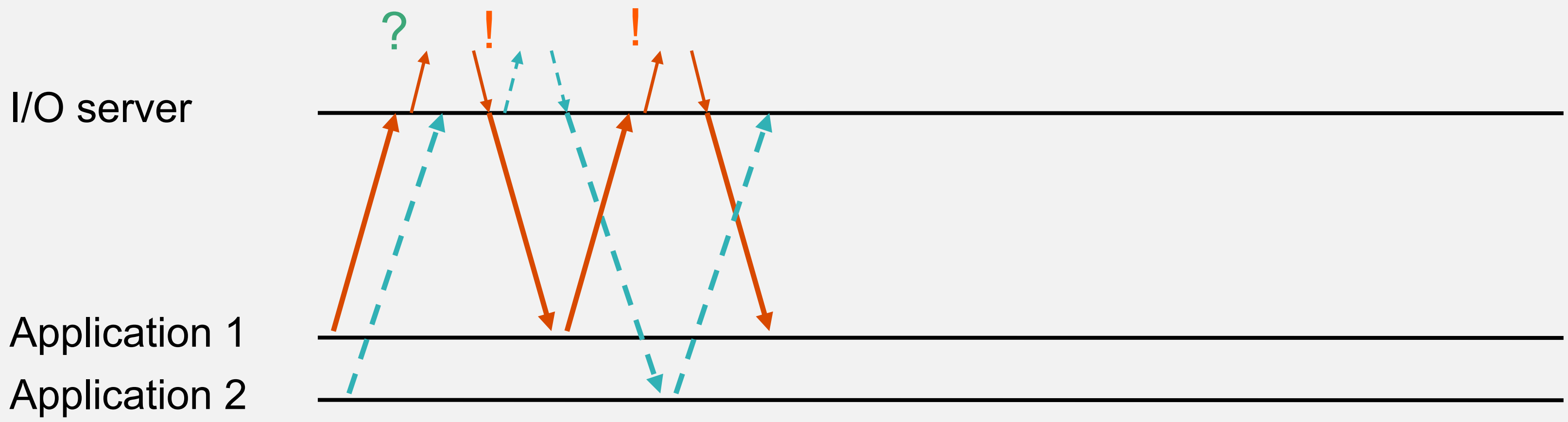
No informations about applications, only about files



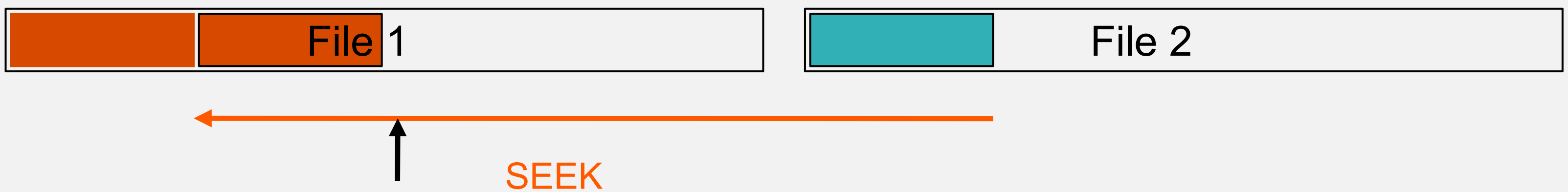
Multi-apps Environment

- Synchronous behavior

(2 concurrent applications execute a “cat” like operation)



No informations about applications, only about files



Multi-apps Environment

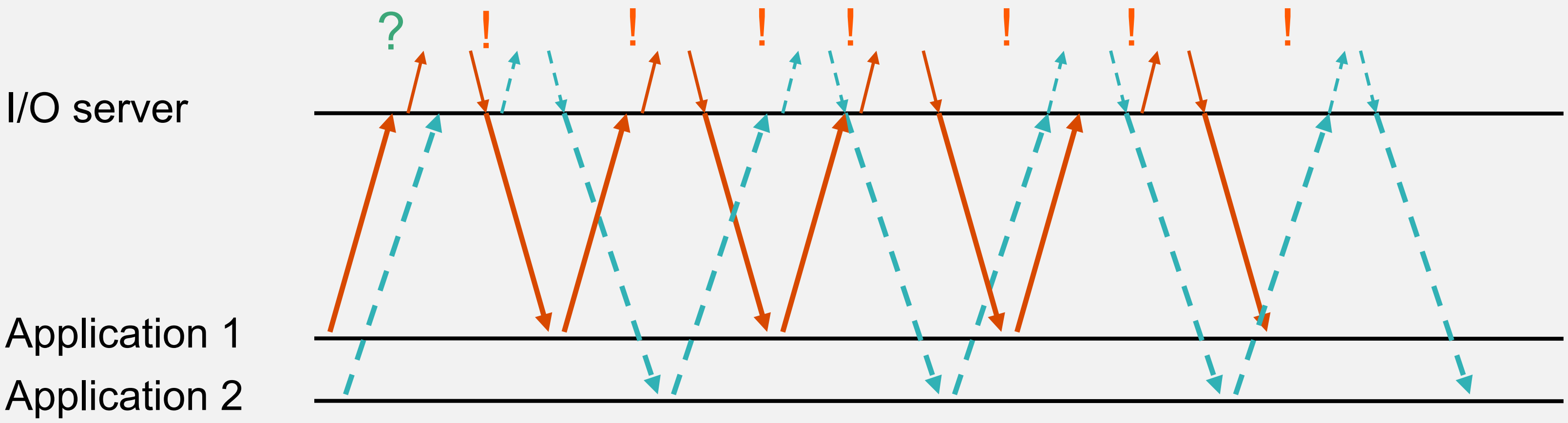
Marie Curie Initial Training Networks

SCALUS

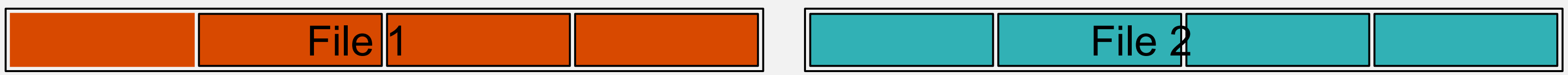
SCALing by means of Ubiquitous Storage

- Synchronous behavior

(2 concurrent applications execute a "cat" like operation)



No informations about applications, only about files



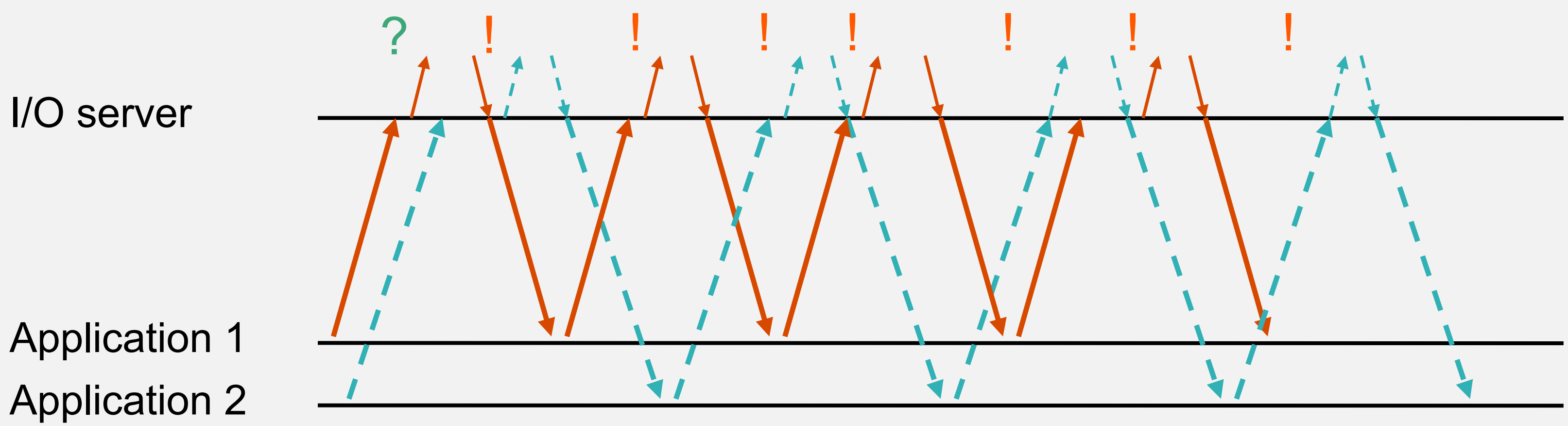
Multi-apps Environment

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

- Synchronous behavior

(2 concurrent applications execute a “cat” like operation)



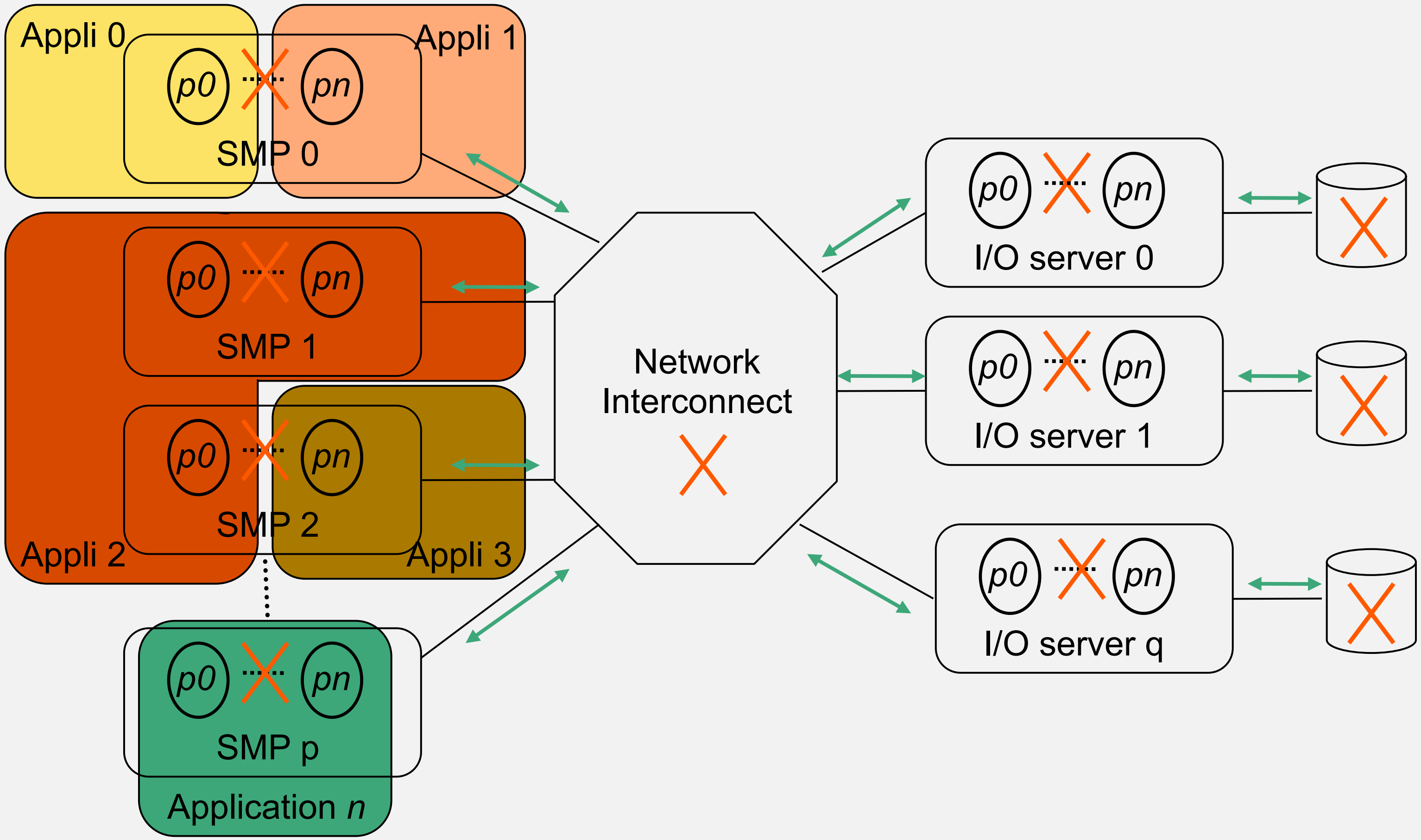
No informations about applications, only about files

Switching from one file to another one may significantly decrease performances
Global synchronization is required ⇒ Libraries are not suited

Multi-apps Environment

- Requirements/Objectives :
  - Exploit parallel I/O algorithms
    Scheduling / aggregating / overlapping accesses ⇒ mono-application efficiency
  - Only through the use of ubiquitous POSIX calls:
    `open/creat/lseek/read/write/close` ⇒ portability / simplicity
  - Address requests in a global manner ⇒ multi-application efficiency
    - Naive approach: processing all requests from one application before serving another one.
      ⇒ not suited for a cluster
- aIOLi, an I/O scheduler for HPC: tradeoff between ''fairness'' and performance

Toward a global coordination

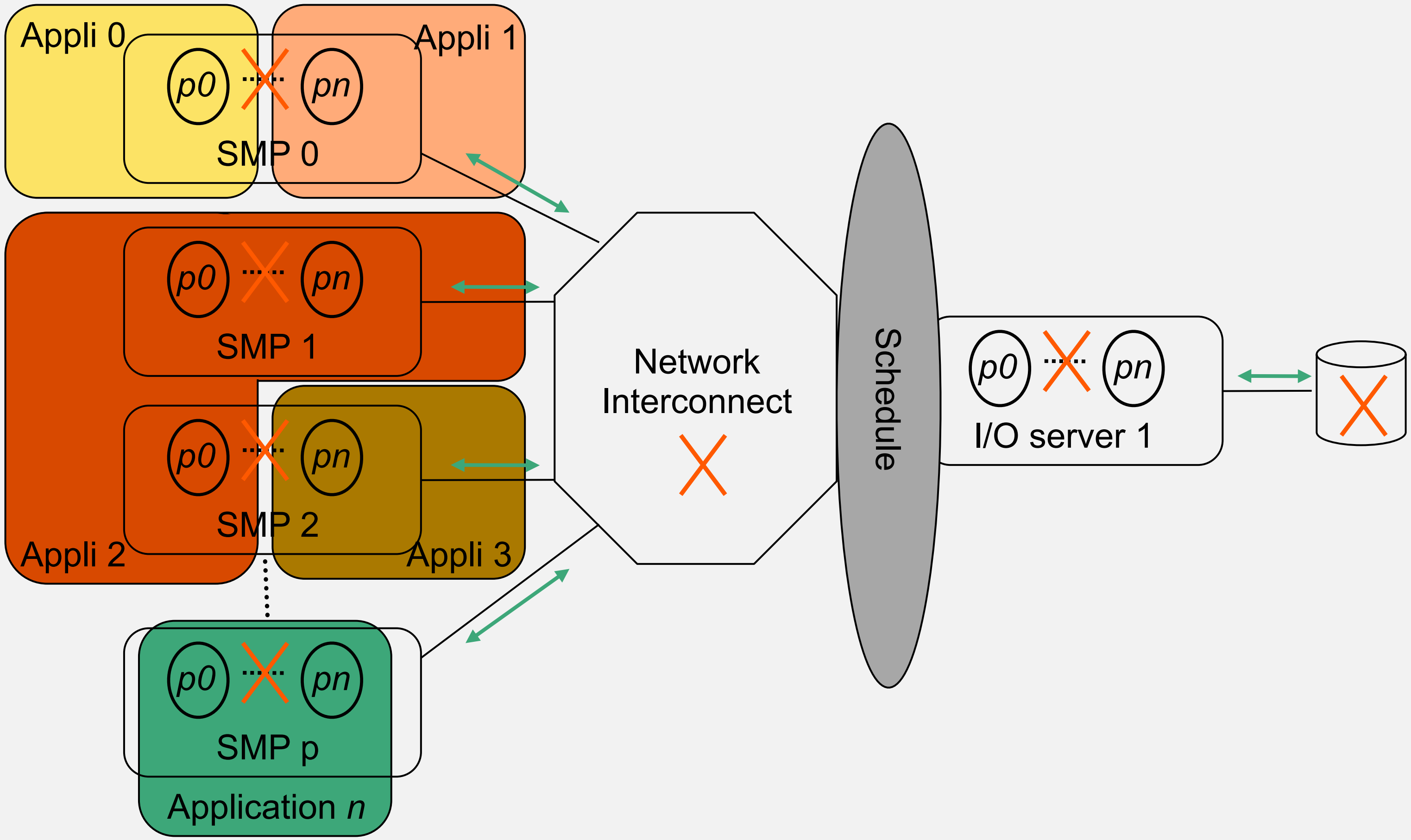


Multi-apps Scheduling

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

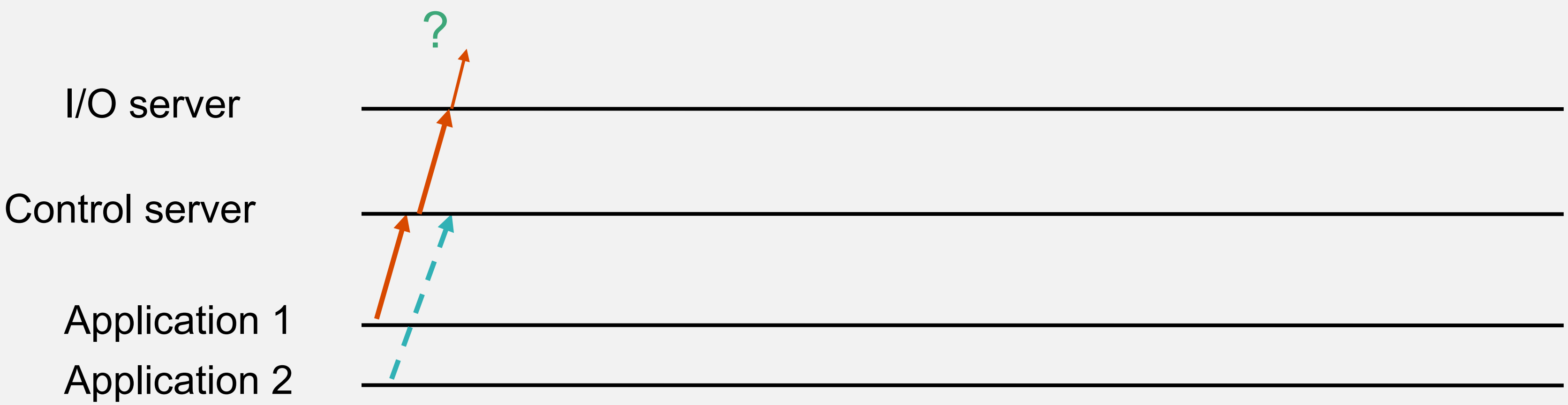
Toward a global coordination



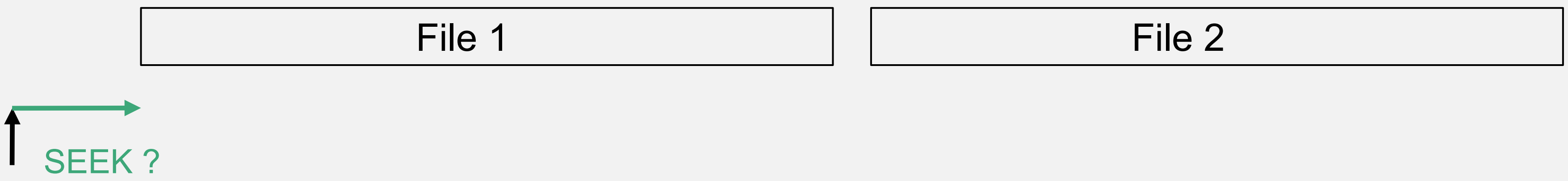
Multi-apps Scheduling

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



No information about applications only on files



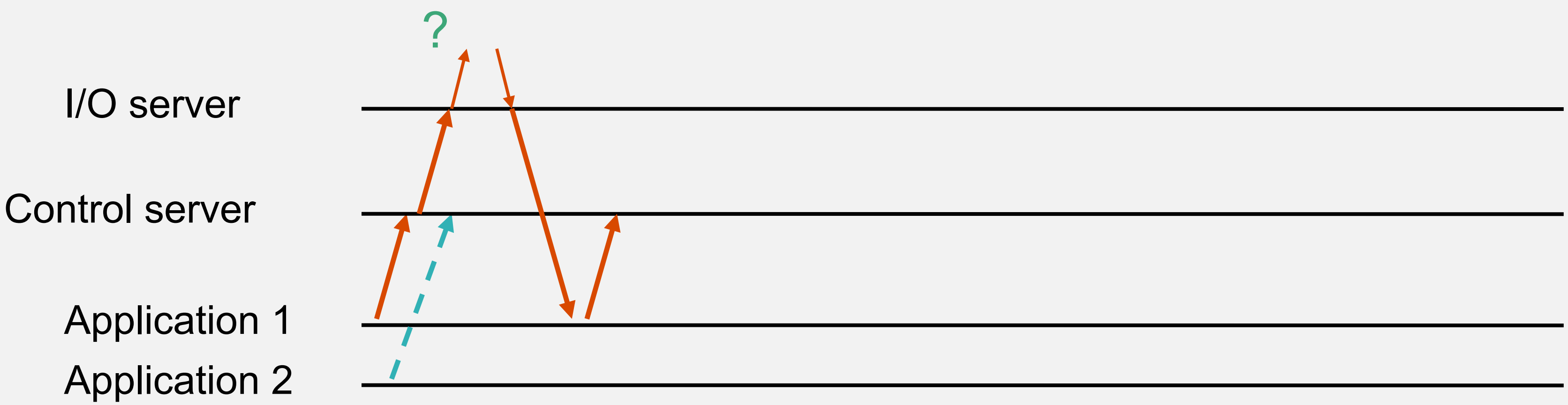
Multi-apps Scheduling

Marie Curie Initial Training Networks

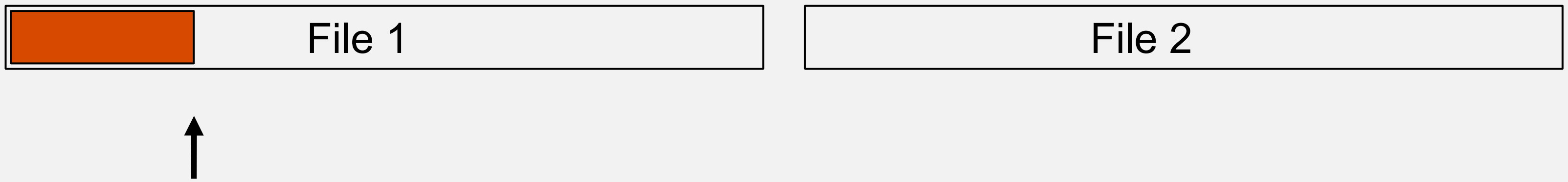
SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



No information about applications only on files



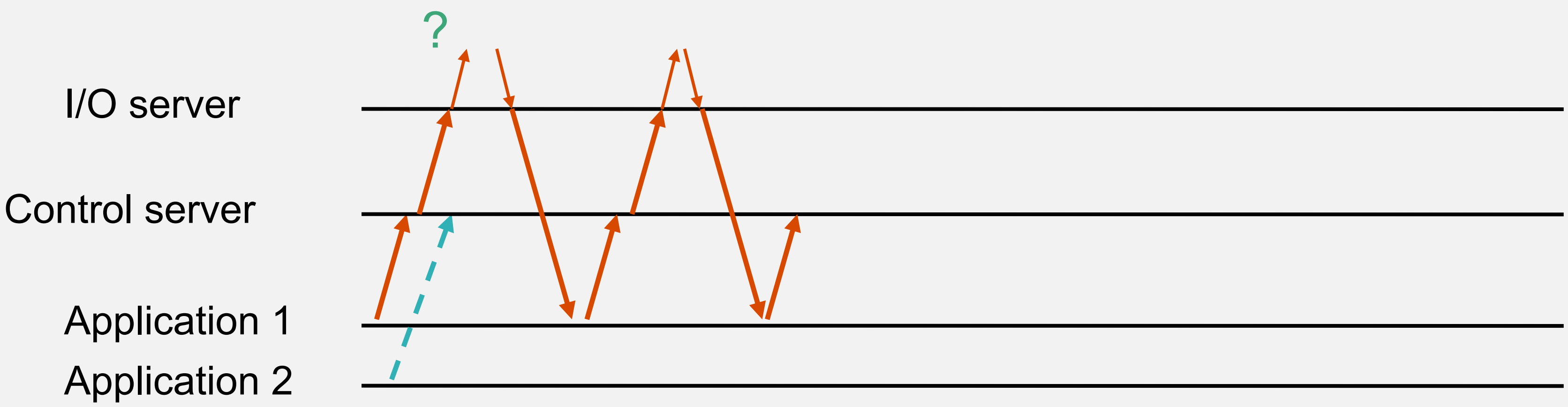
Multi-apps Scheduling

Marie Curie Initial Training Networks

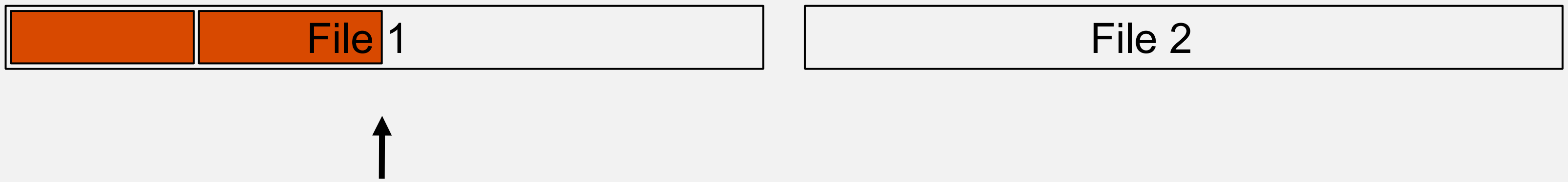
SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



No information about applications only on files



Multi-apps Scheduling

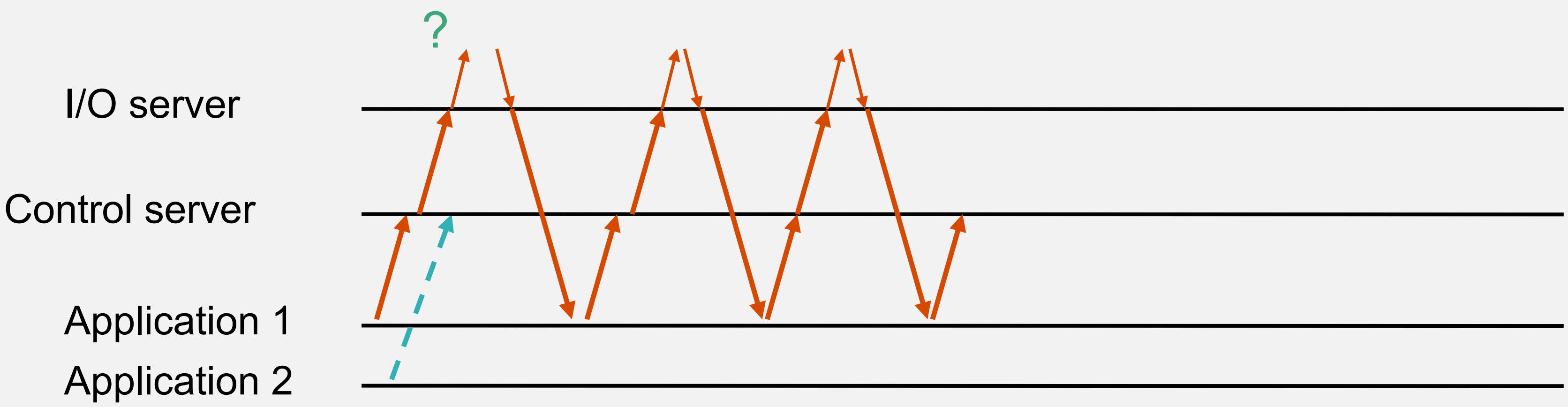
Marie Curie Initial Training Networks

SCALUS

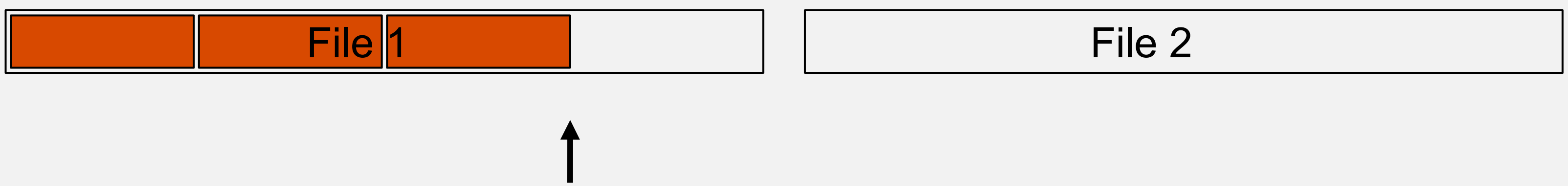
SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



No information about applications only on files



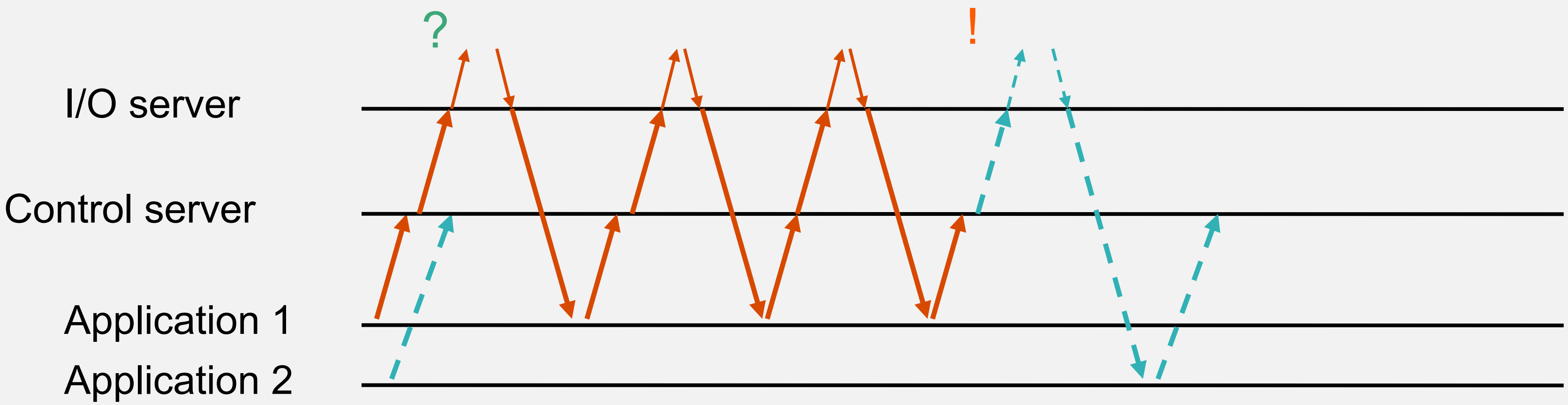
Multi-apps Scheduling

Marie Curie Initial Training Networks

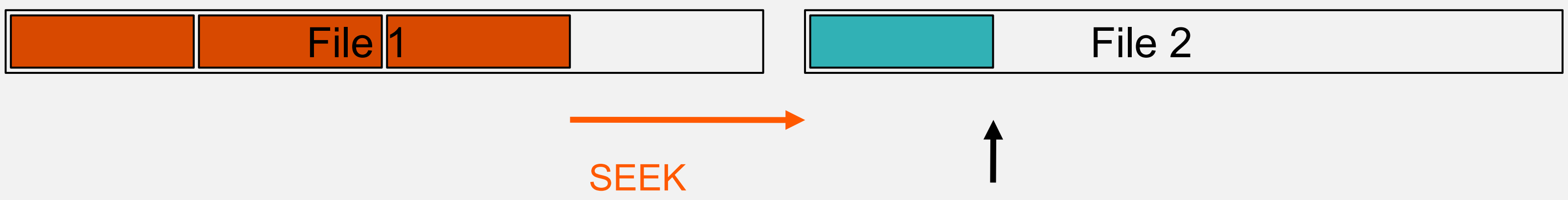
SCALUS SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



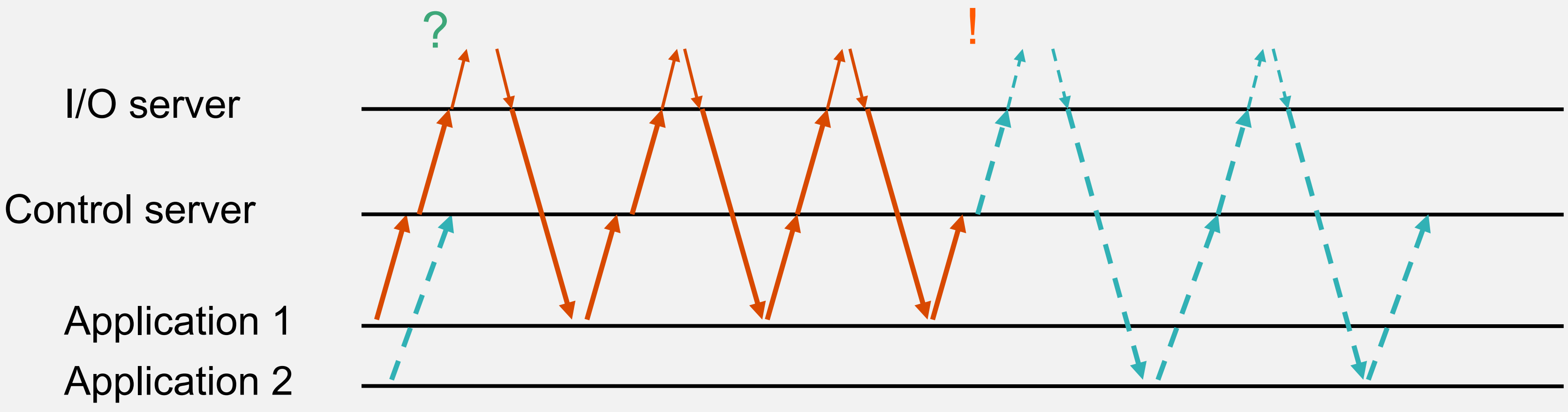
No information about applications only on files



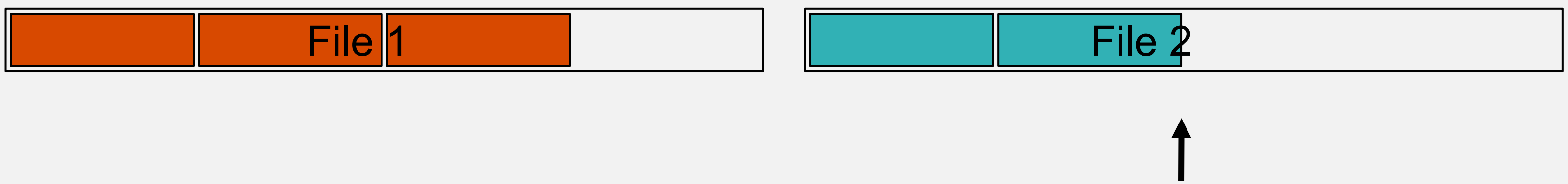
Multi-apps Scheduling

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



No information about applications only on files

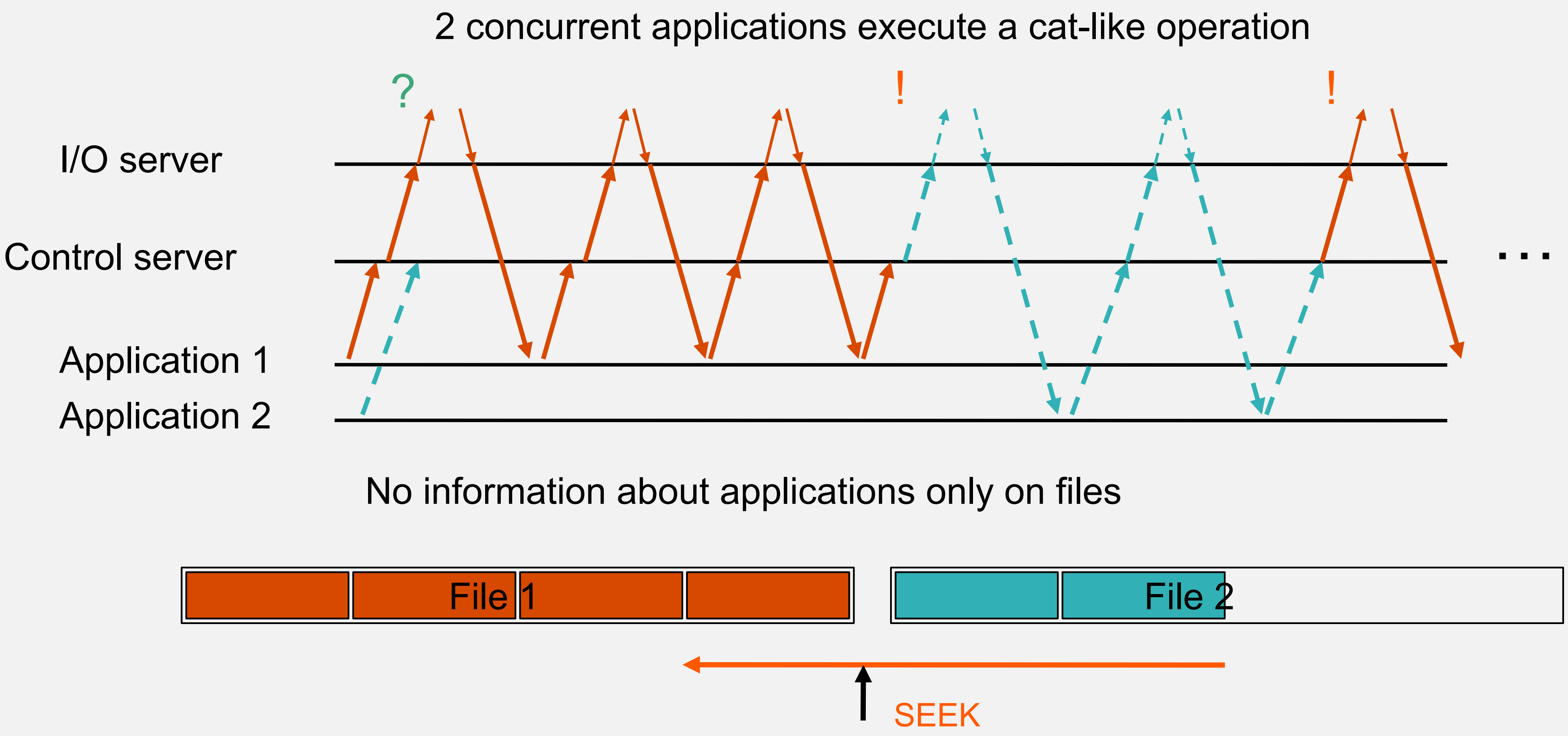


Multi-apps Scheduling

Marie Curie Initial Training Networks

SCALUS
SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way



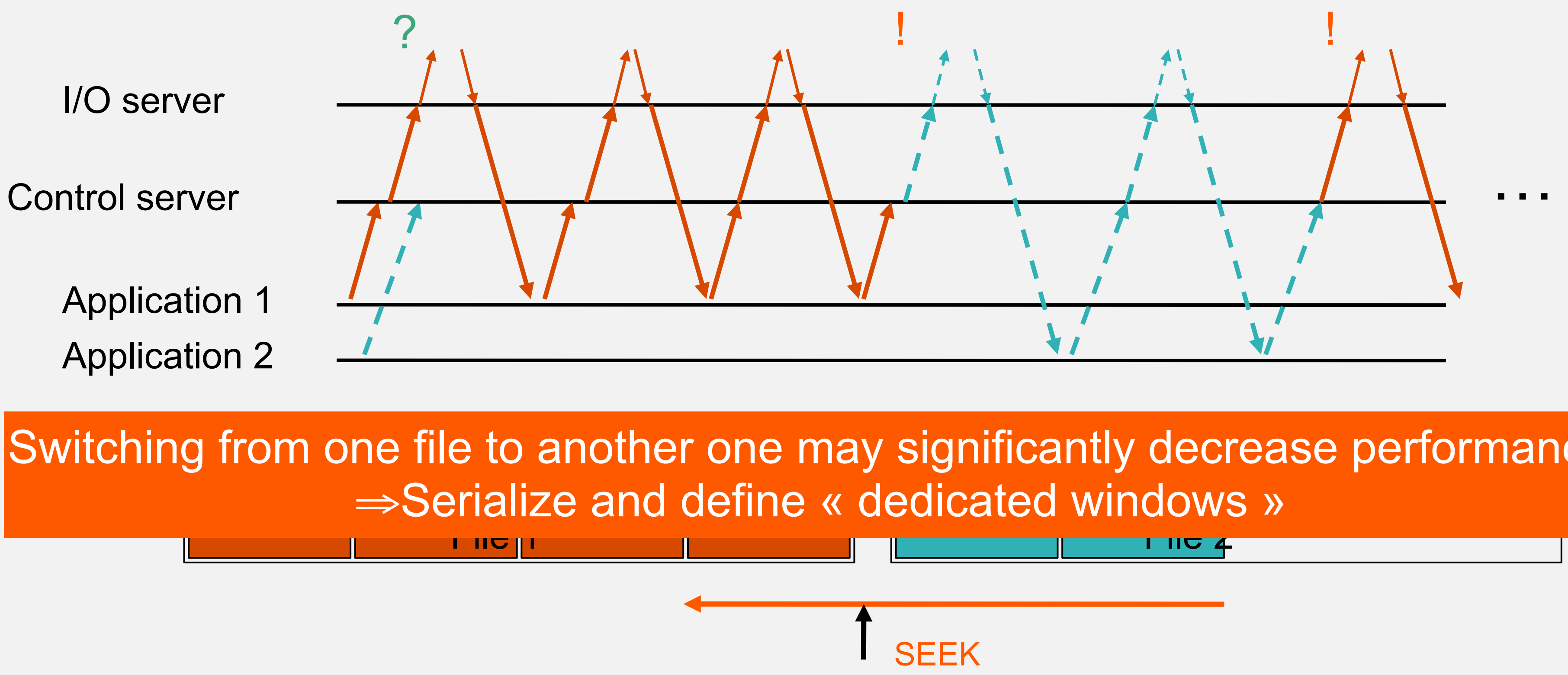
Multi-apps Scheduling

Marie Curie Initial Training Networks

SCALUS — SCALing by means of Ubiquitous Storage

- Manage synchronous behaviors in a efficient way

2 concurrent applications execute a cat-like operation



Switching from one file to another one may significantly decrease performances
⇒Serialize and define « dedicated windows »

Multi-apps Scheduling

Marie Curie Initial Training Networks

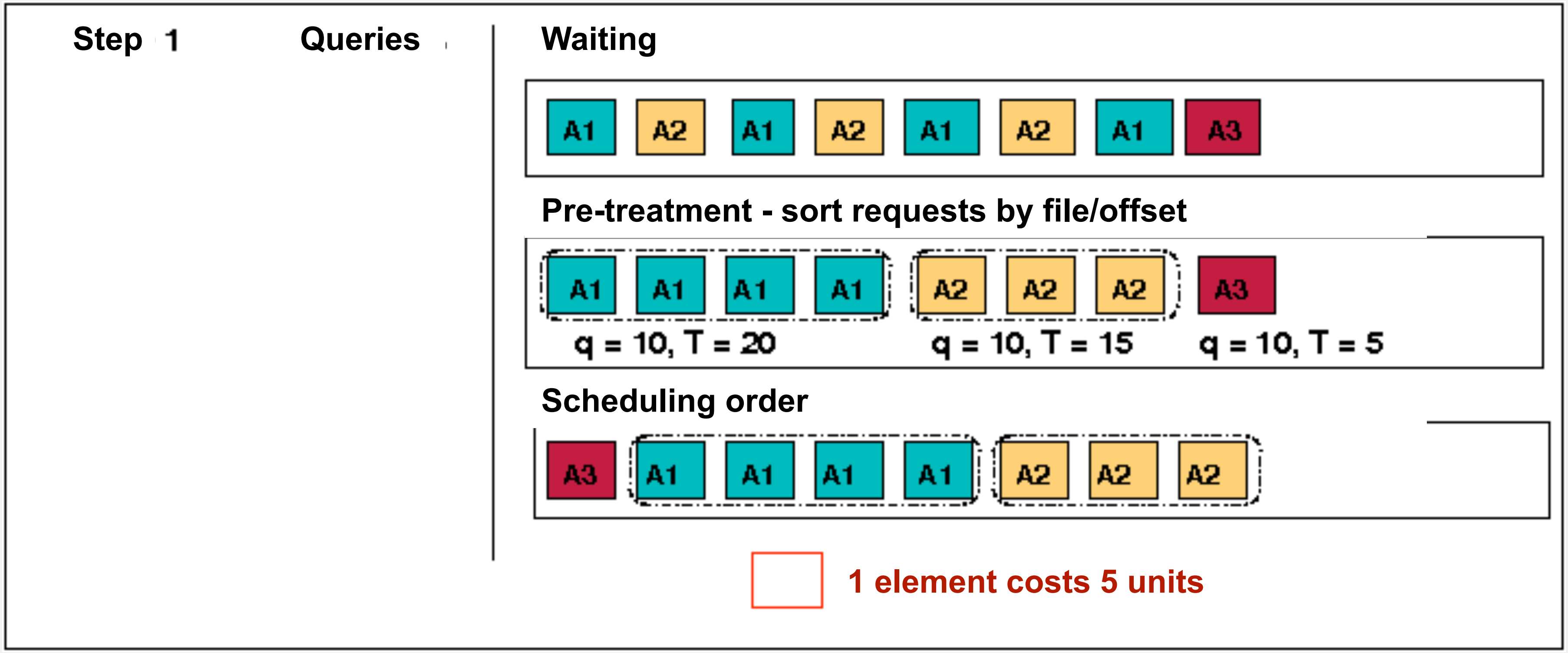
SCALUS

SCALing by means of Ubiquitous Storage

- Challenges:
  - "online" problem (from the scheduling point of view)
    Several requests from distinct applications are delivered to the file system
  - Transparently detect application access patterns to define appropriated "control" windows
  - Wished criteria: "Efficiency" with "fairness" constraints (avoid starvation issue)
  - Distributed system : a global vision is mandatory to propose an appropriated policy
- First case: a global scheduler for NFS :
  - Algorithm : « multi-level feedback » variant
    - Quantum approach (the quantum defines the size of the « control » window)
    - The grow of a quantum could be set for each applications (QoS)

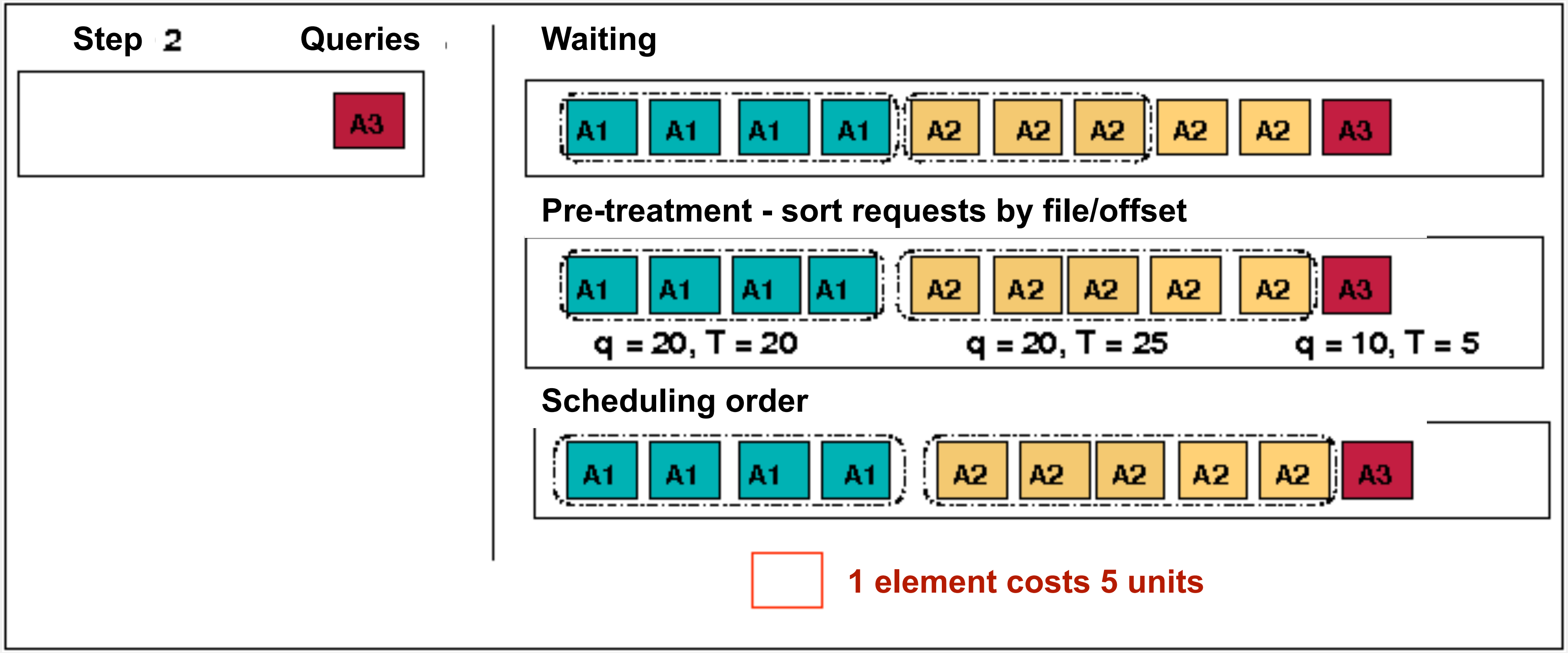
Multi-apps Scheduling

- scheduling policy (aIOLi proposal):



Multi-apps Scheduling

- scheduling policy (aIOLi proposal):



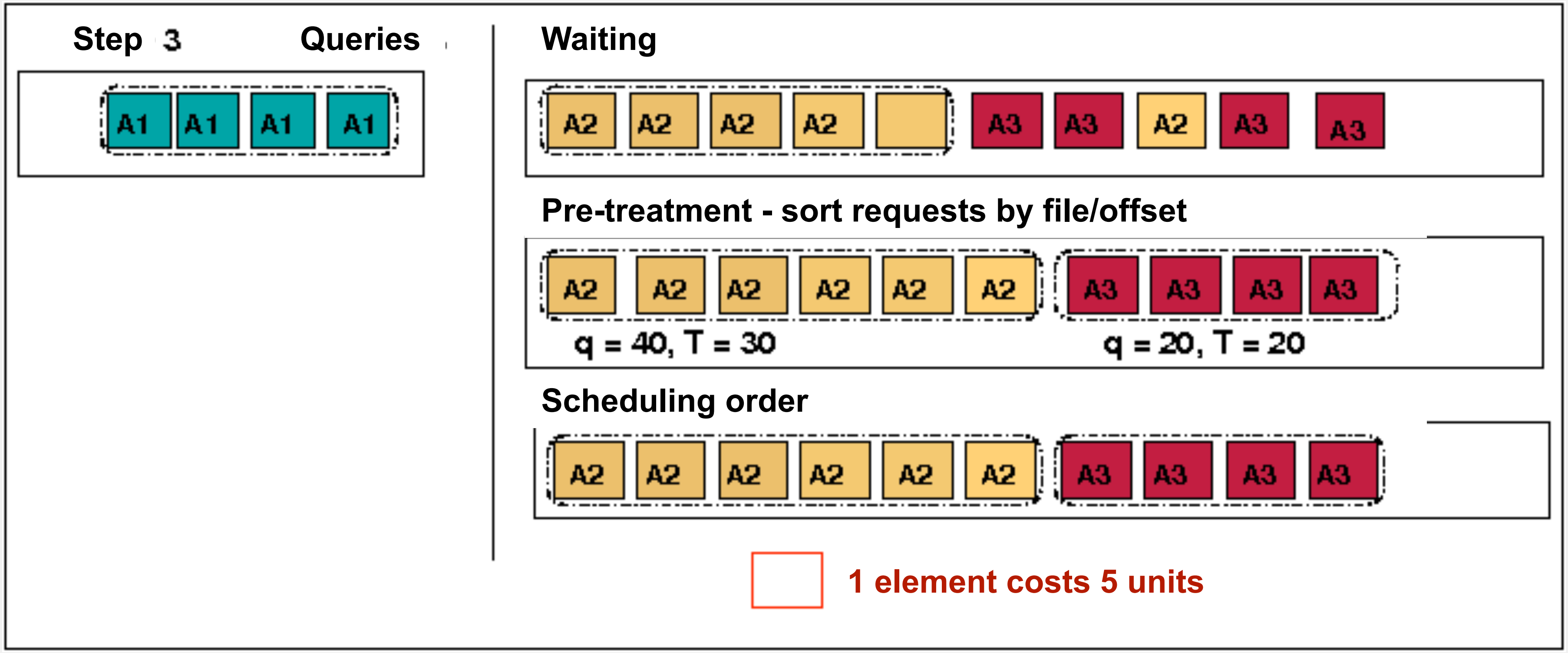
Multi-apps Scheduling

Marie Curie Initial Training Networks

SCALUS

SCALing by means of Ubiquitous Storage

- scheduling policy (aIOLi proposal):



- Significant improvement from performance point of view
- Distributed scheduling is quite difficult
(Few developments have been proposed)

Parallel I/O is still a concern !

- Scientific applications manipulate huge amounts of data (bigger and bigger)
- Parallel File Systems solutions
  - Reduce the impact from performance point of view
  - However, they do not take into account application access patterns and cannot exploit the whole bandwidth provided by the storage units.
- Dedicated Parallel I/O Libraries
  - Reduce the impact from performance point of view
  - However, they imply tedious and complex development in order to significantly improve performances
  - Major weaknesses: require a dedicated FS (no global coordination)
- Scheduling approaches
  - Improvements are significant (from performance point of view)
  - Benefits are strongly linked to the implementation:
    - At block level: good knowledge of physical placement but nothing about applications (logical objects)
    - At application level: good knowledge of access patterns but nothing about physical placements (xFS, PVFS, Lustre, …
  - Global coordination implies global view (distributed scheduling issues)

Bibliography

Marie Curie Initial Training Networks

SCALUS — SCALing by means of Ubiquitous Storage

- I/O Characterization:
  - ''**Input/Output characteristics of scalable parallel application**'' by P. E. Crandalll, R. A. Aydt, A.A. Chien and D. A. Reed. In P*roceedings* of Supercomputing'95, IEEE computer Society Press, 1995
  - ''***Parallel I/O for High Performance Computing***'' by J. May, Morgan Kaufmann, 2001
- MPI I/O
  - ''***Overview of the MPI IO parallel I/O interface***'' P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. P. Prost, M. Snir, B. Traversat and P. Wong. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, IEEE Computer society, 2001
  - ROMIO, http://www.mcs.anl.gov/romio
- I/O scheduling
  - ''**I/O Scheduling Service for Multi-Application Clusters**'' A. Lèbre, G. Huard, Y. Denneulin and P. Sowa, In Proceedings of IEEE Cluster 2006
- kDDM and kDFS

  ''**Reducing Kernel Development Complexity in Distributed Environments**'' by A; Lèbre, R. Lottiaux, E. Focht and Christine Morin. In Proceedings of Europar conference, 2008.

  http://www.kerrighed.org/wiki/index.php/KernelDevelKdFS

Thanks

- Questions ?