Lecture 15

Parallel I/O

Overview

- What is parallel I/O
- MPI-I/O
- Parallel File Systems
- Application examples

Parallel I/O

- I/O for scientific computing
  - Not for databases, enterprise applications, the Internet
- Large data items
  - Arrays
  - Simulation results
- Shared across many processors
  - Hundreds or thousands

An Example Application

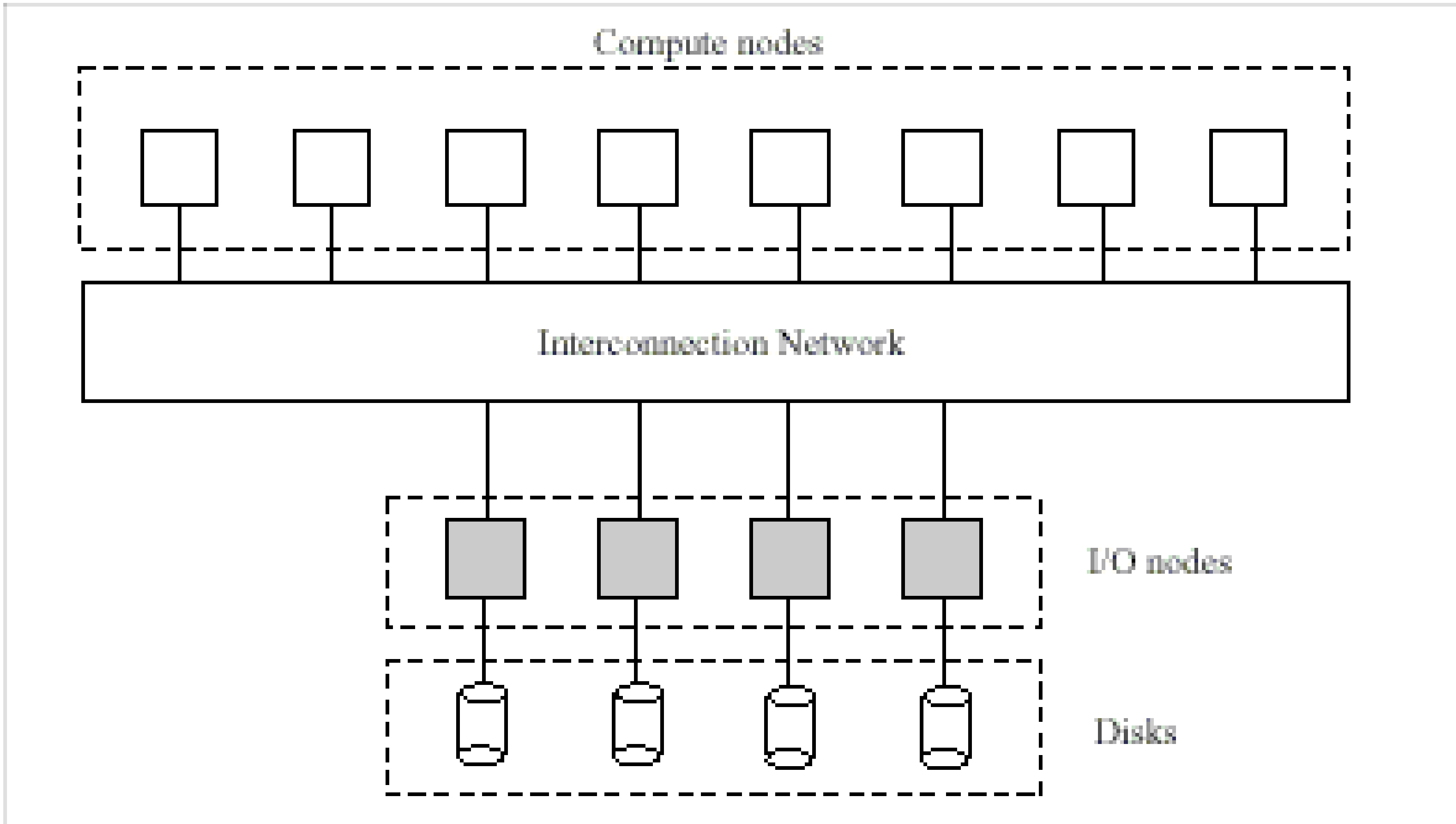
The I/O Gap in Parallel Applications

- The I/O gap between memory speed and average disk access stands at 10^5 (40 nanosecs versus 4 ms)
- Is the gap larger or smaller for parallel applications and why?

The I/O Gap in Parallel Applications

- The I/O gap between memory speed and average disk access stands at 10^5 (40 nanosecs versus 4 ms)
- Is the gap larger or smaller for parallel applications and why?
  - Better: caches are larger and shared through distributed memory systems?
  - Worse: many access patterns are sequential and therefore unshareable?
  - Worse: scientific applications tend to be write-oriented

Parallel I/O Architecture



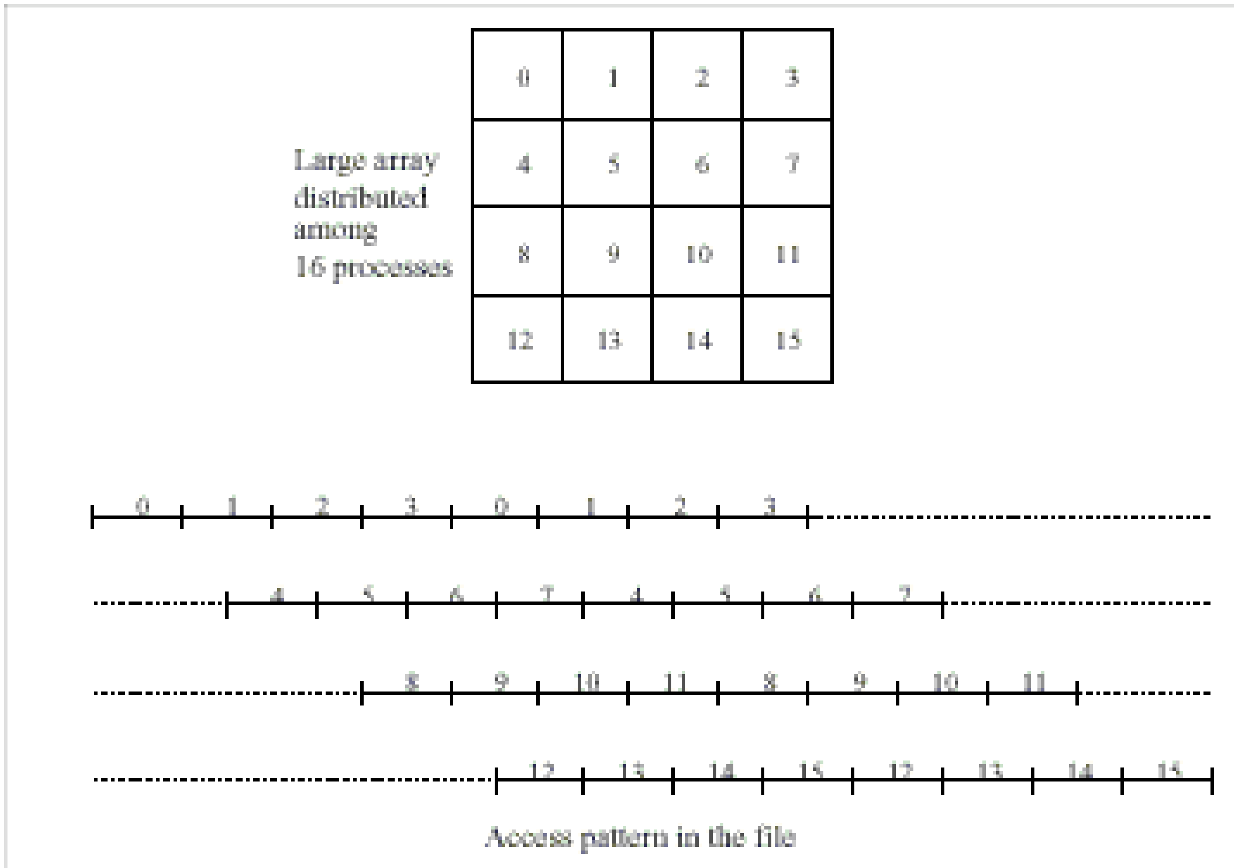
Parallel I/O Architecture

- I/O nodes
  - Block servers
  - Implement availability and caching (RAID)
  - Usually dedicated (no computation)
- Processors nodes
  - Sometimes have their own disk (not part of the compute system)
- High speed interconnect
  - Often switches
- Example systems
  - IBM SP/2 (ASCI White, BME has one)
  - Intel Tflops (ASCI Red)
  - Cray T3E
  - IBM BlueGene/L

The API Problem

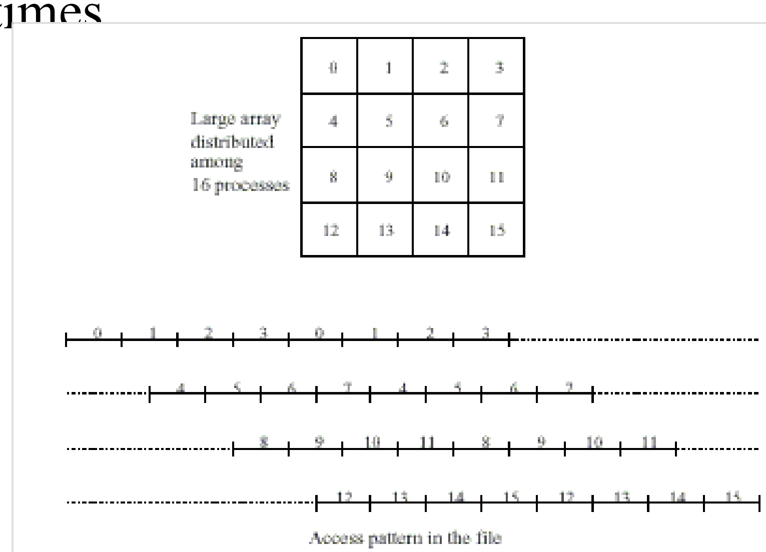
- UNIX has a "standard" API
  - open(), close(), read(), write()
- Parallel file systems often want to access:
  - Small data items from within a file
  - Non-contiguous in the file space
- Example
  - Two-dimensional array (spatial data)
  - Stored in row-major order
  - Partition (to processors) based on rectangles

Typical Array Workload



Typical Array Workload

- UNIX interface requires seeks between each I/O
  - Interactive/blocking interface
- No way to encode, overall view of access pattern
  - Application has a priori knowledge of data usage
- Workload is not aligned to file layout (sequential)
  - Generally difficult to lay out sequentially, because used in different ways at different times
- Not interactive
  - Latency insensitive



The API Problem (again)

- UNIX has a "standard" API
  - open(), close(), read(), write()
- Parallel file systems often want to access:
  - Small data items from within a file
  - Non-contiguous in the file space
- Example
  - Two-dimensional array (spatial data)
  - Stored in row-major order
  - Partition (to processors) based on rectangles
- How do applications perform parallel I/O in a standard, portable way?
  - Well they don't really.

MPI-I/O

- An API standard for parallel I/O
  - Much different that POSIX/UNIX API
- But, as it gains traction, becomes easier to use

- Replaces many custom libraries and approaches
  - PANDA
  - Solar
  - ChemI/O
  - Others …

MPI-I/O History

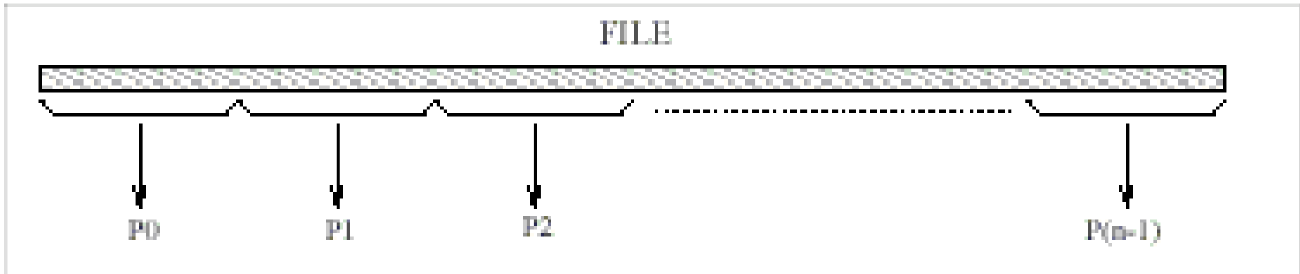
- Which of the following are true?
  - MPI-I/O grew out of a study that looked at how message passing architectures and standards affected I/O
  - MPI-I/O can be used for both good and evil
  - MPI-I/O was started at IBM Watson
  - MPI-I/O is easy to use
  - MPI-I/O maps I/O reads and writes to message passing sends and receives

MPI-I/O History

- Which of the following are true?
  - MPI-I/O grew out of a study that looked at how message passing architectures and standards affected I/O
  - MPI-I/O can be used for both good and evil
  - MPI-I/O was started at IBM Watson
  - MPI-I/O is easy to use
  - MPI-I/O maps I/O reads and writes to message passing sends and receives

MPI-I/O for Dum-Dummies

- If, *n* processors wish to read a file in *n* chunks
- MPI_File_Open – does an open for all *n* processors and indicates a "communicator" that defines the group
- Each processor does MPI_File_seek, MPI_File_Read, and MPI_File_Write
  - *i.e.*, each processor has its own seek pointer
- MPI_File_Read/Write operate on typed arrays of data
  - Meaningless semantic sugar



MPI-I/O for Jane and John Smith

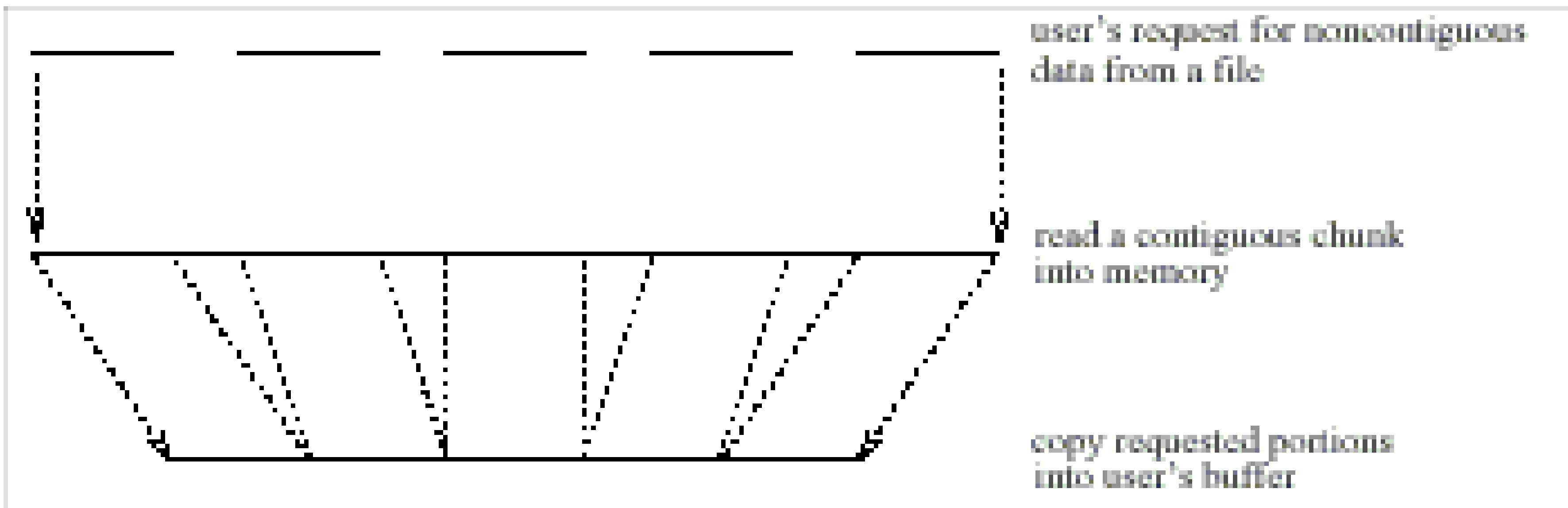
- Multiple I/O interfaces
  - Implicit offset (seek pointer)
  - Explicit offset
  - Shared file pointer (among all processes)
    - *E.g.*, a bunch of processes that read the next data item when completing work against a previous data item
- Multiple storage representations (recall I/Os are typed)
  - Native (no memory interpretation)
  - Internal (app provides interface for how to convert memory to persistent data)
  - External32 (network representation of data, similar to TCP)

MPI-I/O for Savants

- Many optimizations for groups of processes to do I/O
  - Collective I/O
  - Data sieving
- Hints
  - Information to help applications choose striping, prefetching, read-ahead and caching policies

Data Sieving

- Package a series of non-contiguous requests into a single large read
  - Use sequential capabilities of the underlying devices/systems
- Make as few requests as possible and make them big

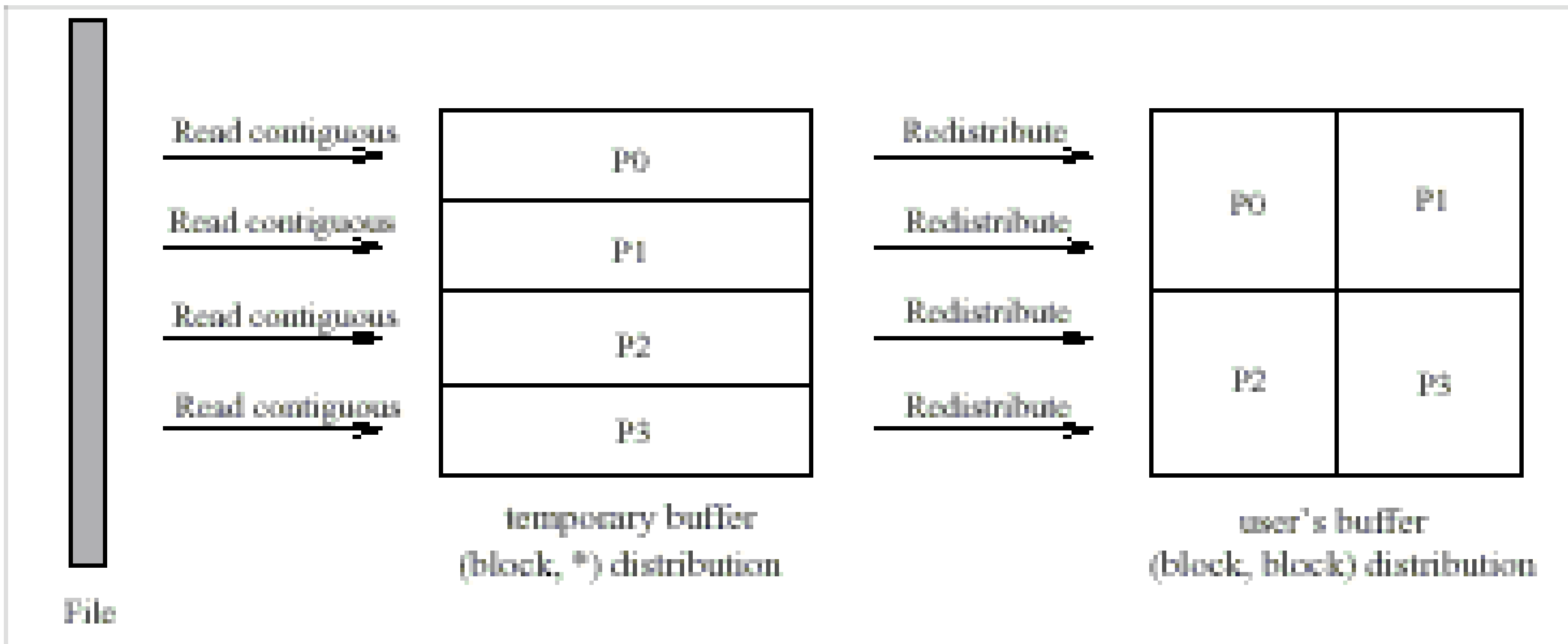


Collective I/O

- Create an I/O plan across many processors
  - Generally some communication to set-up the task
- Main technique – two-phase I/O
- Other techniques
  - Disk and server directed I/O
  - Similar, but limit the buffering requirements at I/O nodes

Two-Phase I/O

- I/Os are done in chunks consistent with the disk layout
- I/Os are redistributed into users buffers according to the application (in-memory) view of data
  - Redistributed among processors
- Same array example



Collective I/O

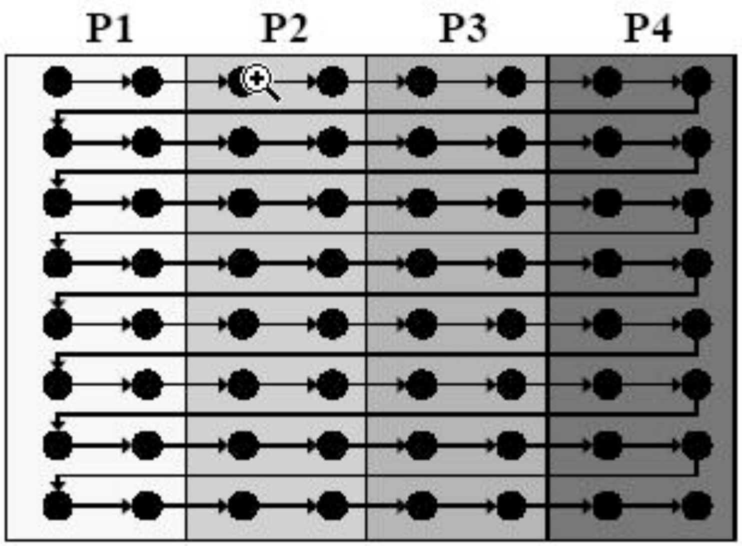


Figure 2: A file access by four processors.

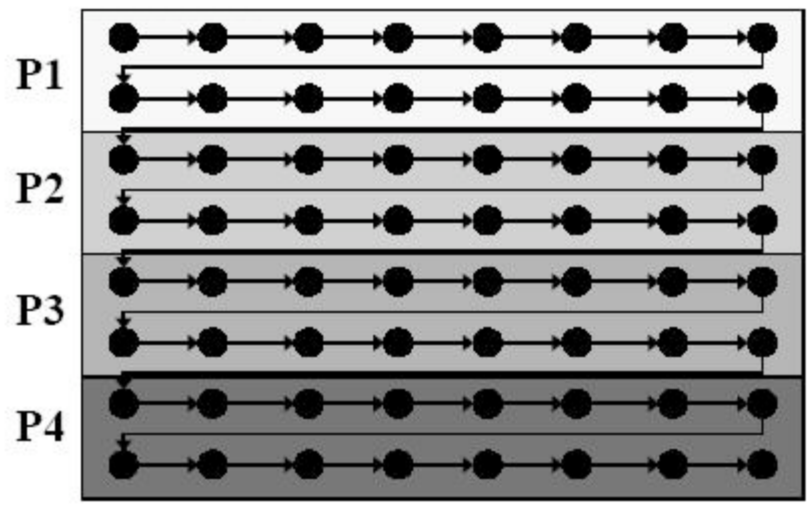


Figure 3: File access using the collective I/O.

How good is MPI I/O?

- Depends on
  - How it is used?
  - What the implementation does with the information it has?
- MPI I/O is an interface
  - Many implementations are possible
  - Popular implementation is ROMIO, from the guys who make the graphs/figures

Level #0 – Ogres Have Layers

- Many independent, contiguous requests
  - No access information available to MPI system at runtime

```
MPI_File_Open ( …, "filename", …, &fh );
for ( i=0; i<n_rows); i++ )
{
  MPI_File_Seek ( fh, … );
  MPI_File_read ( fh, row[i], … );
}
```

Level #1 – Ogres Have Layers

- Many collective, contiguous requests
  - MPI implementation expects to see same access pattern at multiple sites
  - Can lead to good read-ahead, prefetching decisions when implementation sees patterns repeat at different processors

```
MPI_File_Open ( MPI_COMM_WORLD, “filename”, …, &fh );
for ( i=0; i<n_rows); i++ )
{
  MPI_File_Seek ( fh, … );
  MPI_File_read ( fh, row[i], … );
}
```

Level #2 – Ogres Have Layers

- Single independent, non-contiguous requests
  - Data sieving can be used
  - Based on an application defined data type

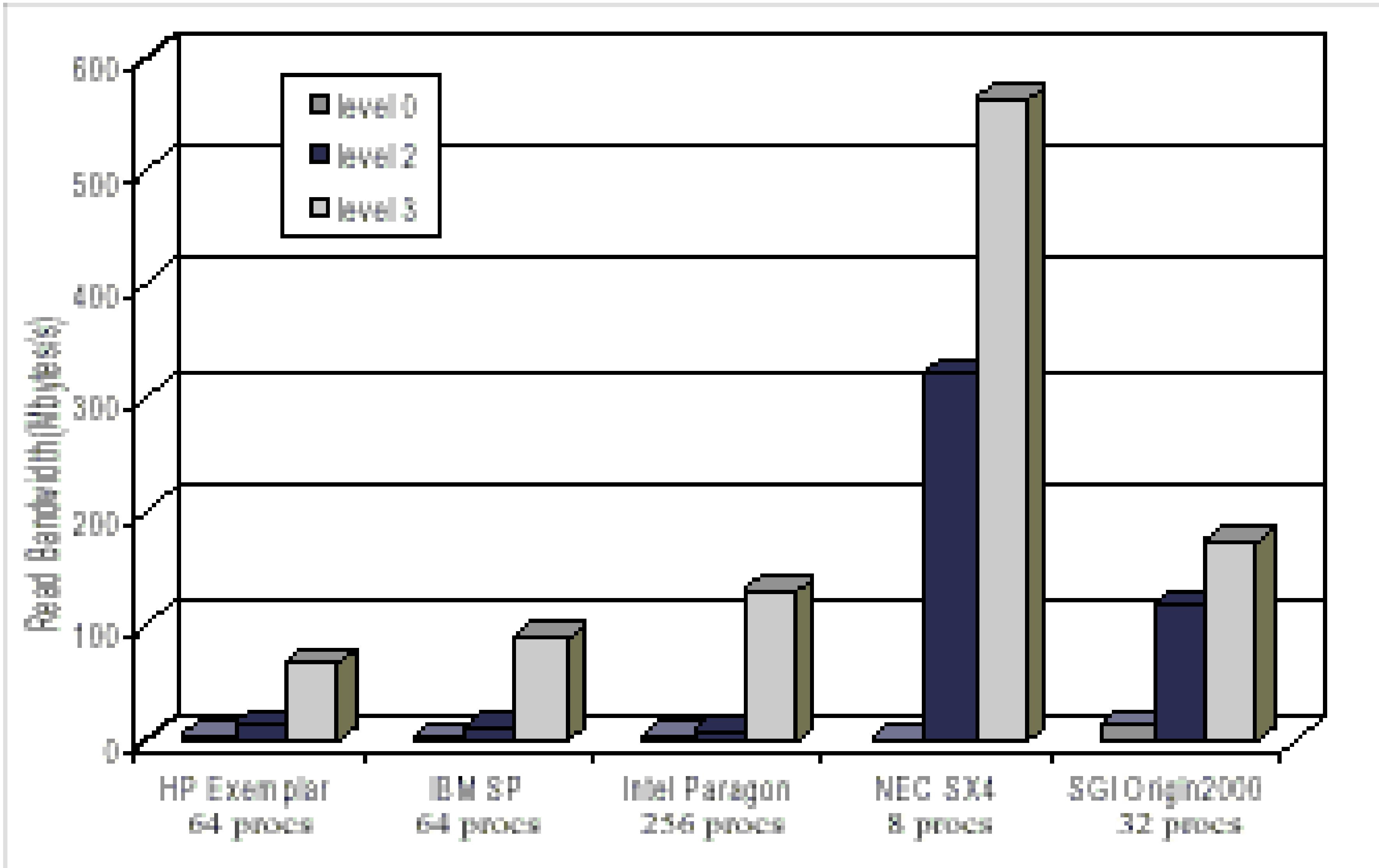
```
MPI_Type_create_subarray (…, &subarray, …. );
MPI_Type_commit ( &subarray );
MPI_File_open ( …, “filename”, …, &fh );
MPI_File_set_view ( fh, …, &subarray, …. );
MPI_File_read ( fh, local_array, … );
```

Level #3 – Ogres Have Layers

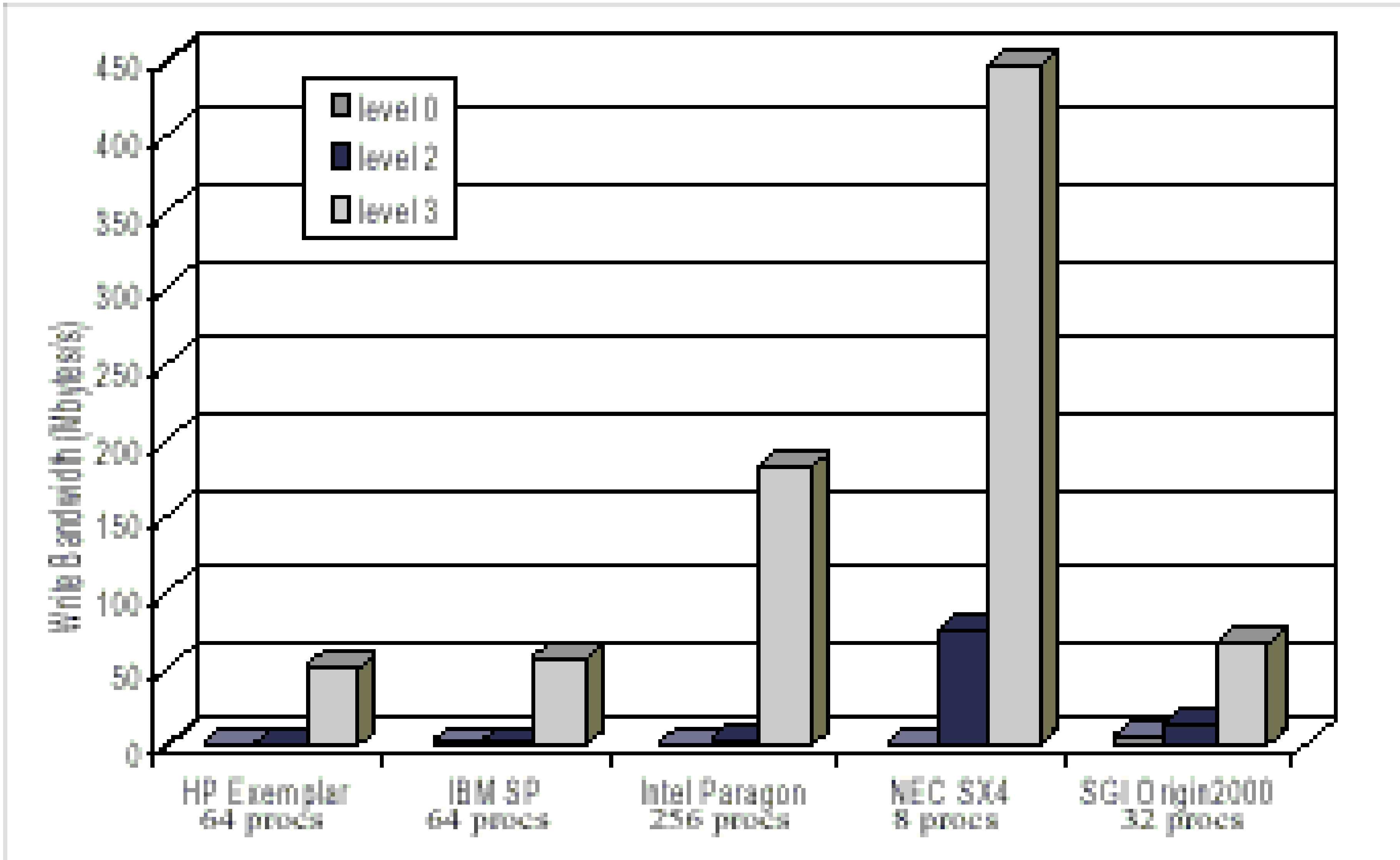
- Multiple collective, non-contiguous requests
  - Data sieving can be used
  - Collective I/O can be used

```
MPI_Type_create_subarray (…, &subarray, …. );
MPI_Type_commit ( &subarray );
MPI_File_open ( MPI_COMM_WORLD,  “filename”, …, &fh );
MPI_File_set_view ( fh, …, &subarray, …. );
MPI_File_read ( fh, local_array, … );
```

MPI I/O Performance (Read)



MPI I/O Performance (Write)



Performance Observations

- Sieving (level #2) performs well on read
- Write requires collective I/O to get high bandwidth
- Scaling is highly dependent on system architecture

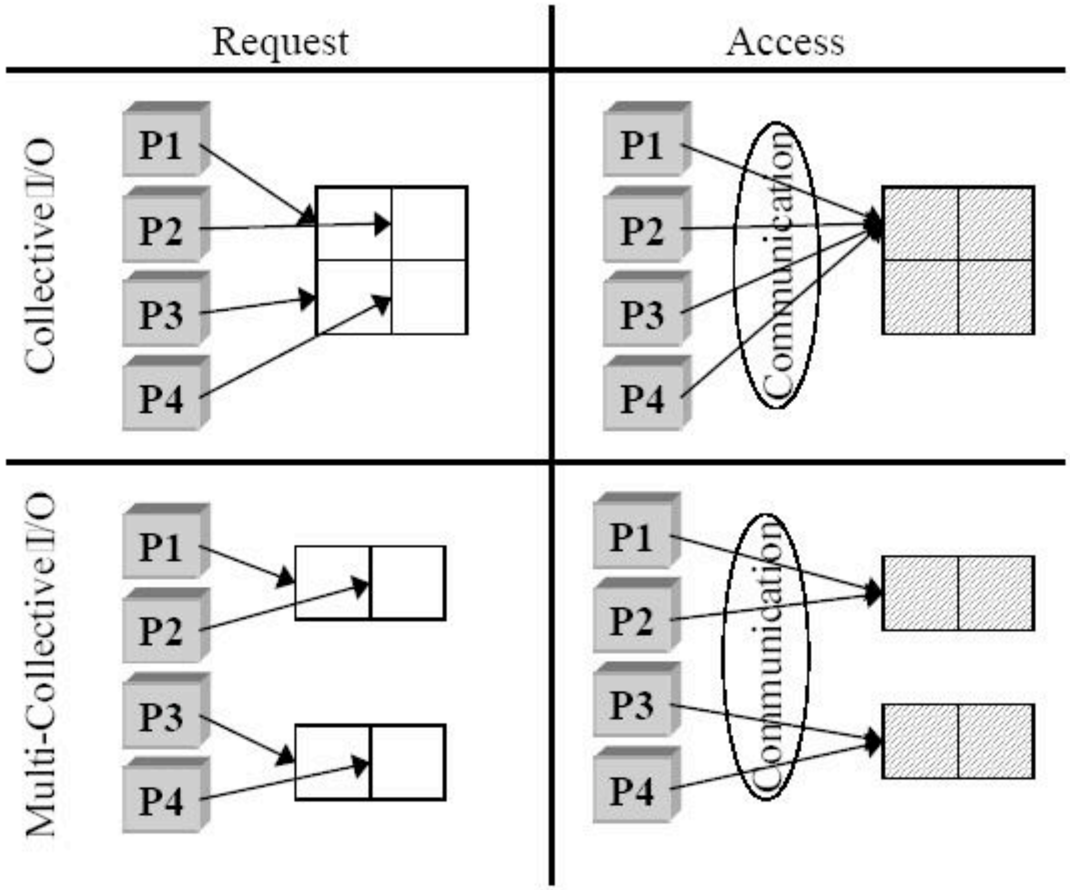
Is MPI-I/O Important

- The developers of GPFS tell me that their users almost always use the POSIX interface, not MPI-I/O
- Can storage system technologies mimic the function of MPI-I/O implicitly?
  - Data sieving?
  - Collective I/O?

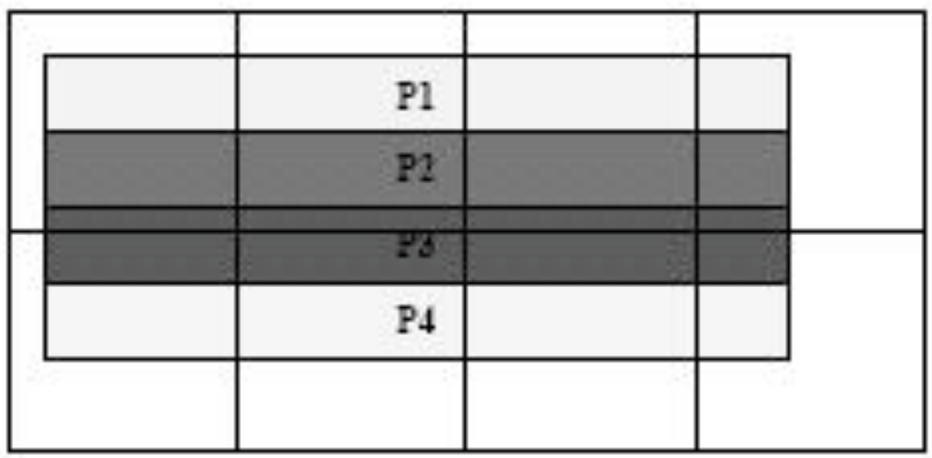
SANs and High-Perf. Computing

- FC/iSCSI SANs create block storage endpoints
  - Similar to the storage servers in the HP model
- Can SAN storage replace storage servers
  - In GPFS for the POSIX interface?
  - When using MPI-I/O?

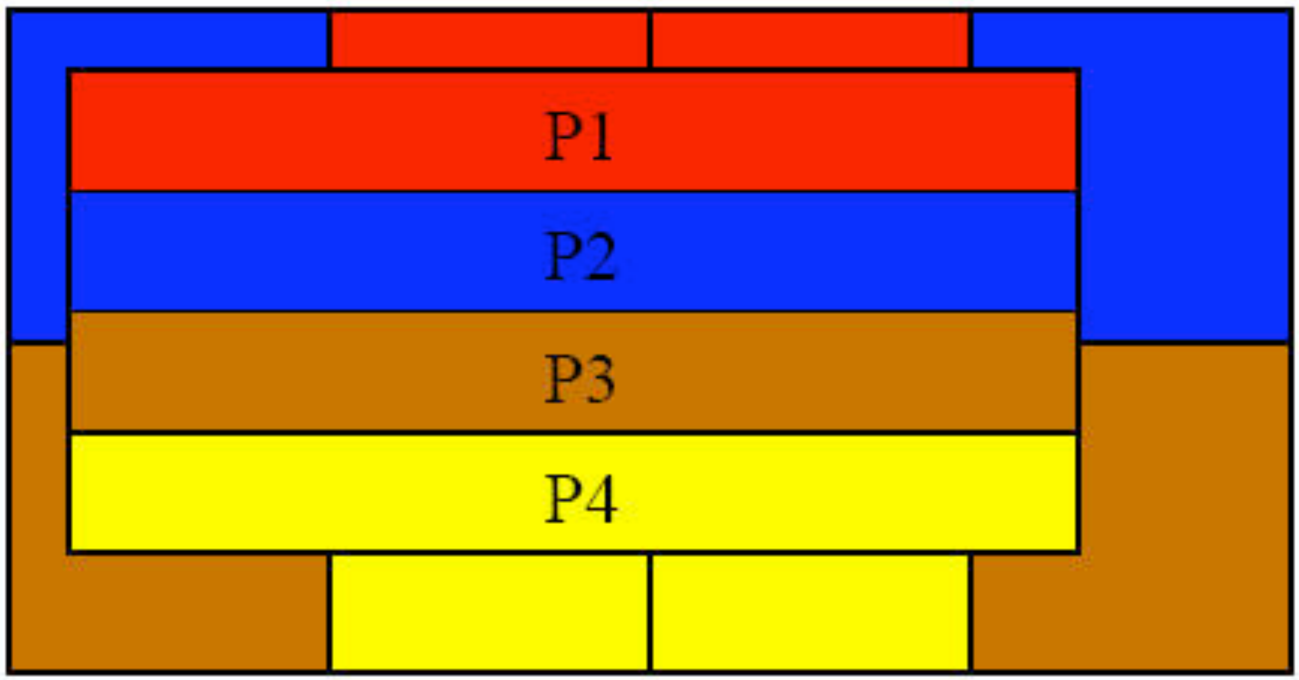
Multi-Collective I/O



Multi-Collective I/O



Multi-Collective I/O



Multi-Collective I/O