Design and Implementation of Parallel File Aggregation Mechanism

Jun Kato* and Yutaka Ishikawa
The University of Tokyo
* Currently affiliated with Fujitsu Laboratories Limited

Agenda

- File organization trend of HPC applications
  - use of millions of small files
- Problem of single shared file approach for reducing the number of files
  - exhibiting low I/O performance through a benchmark program
- PFA (Parallel File Aggregation) Mechanism
  - providing single shared file APIs for high I/O performance
- Evaluation result on a real HPC application
  - 3.8 times faster than the original with reducing the number of files by about 100,000 files
- Conclusion
- Q & A

Agenda

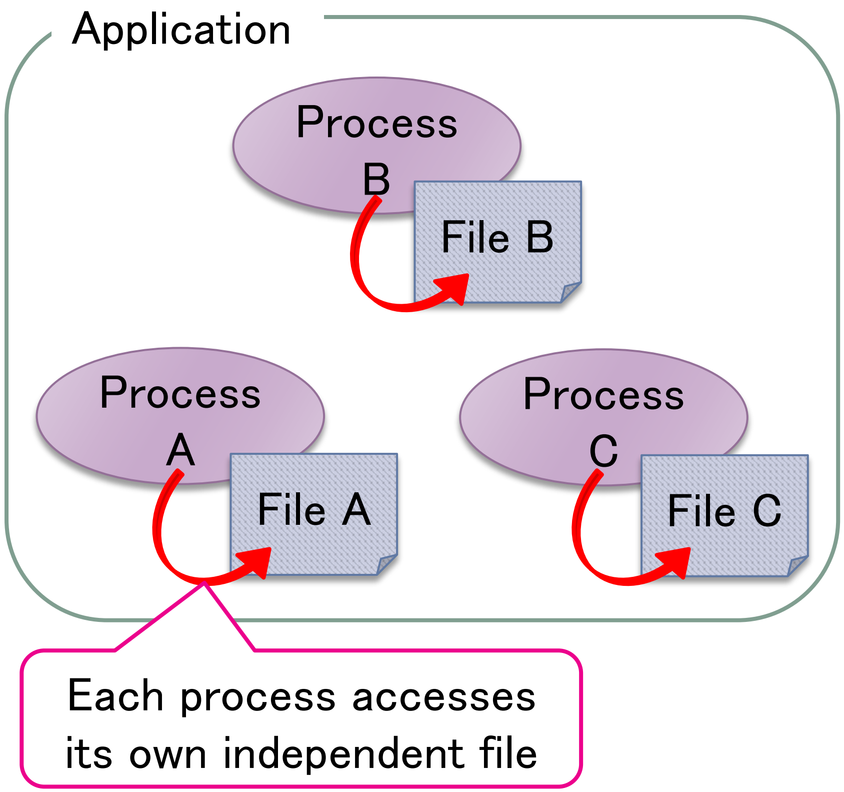
- File organization trend of HPC applications
  - use of millions of small files
- Problem of single shared file approach for reducing the number of files
  - exhibiting low I/O performance through a benchmark program
- PFA (Parallel File Aggregation) Mechanism
  - providing single shared file APIs for high I/O performance
- Evaluation result on a real HPC application
  - 3.8 times faster than the original with reducing the number of files by about 100,000 files
- Conclusion
- Q & A

File Organization Trend of HPC Applications

- Use of millions of several-MB-sized files
  - Examples of real HPC applications
    - Integrated Microbial Genomes System [Rockville 2009]
      - 65 million files
      - Average file size : < 1KB
    - Nearby Supernova Factory [Cecilia 2009]
      - over 100 million files
      - Max file size : 8MB
  - Statistics on HPC file systems [Shobhit 2008]
    - 60% of files : < 1MB
    - 80% of files : < 8MB
    - 99% of files : < 64MB

Design of Current HPC Applications

- N-N pattern
  - *N* processes utilize *N* independent files

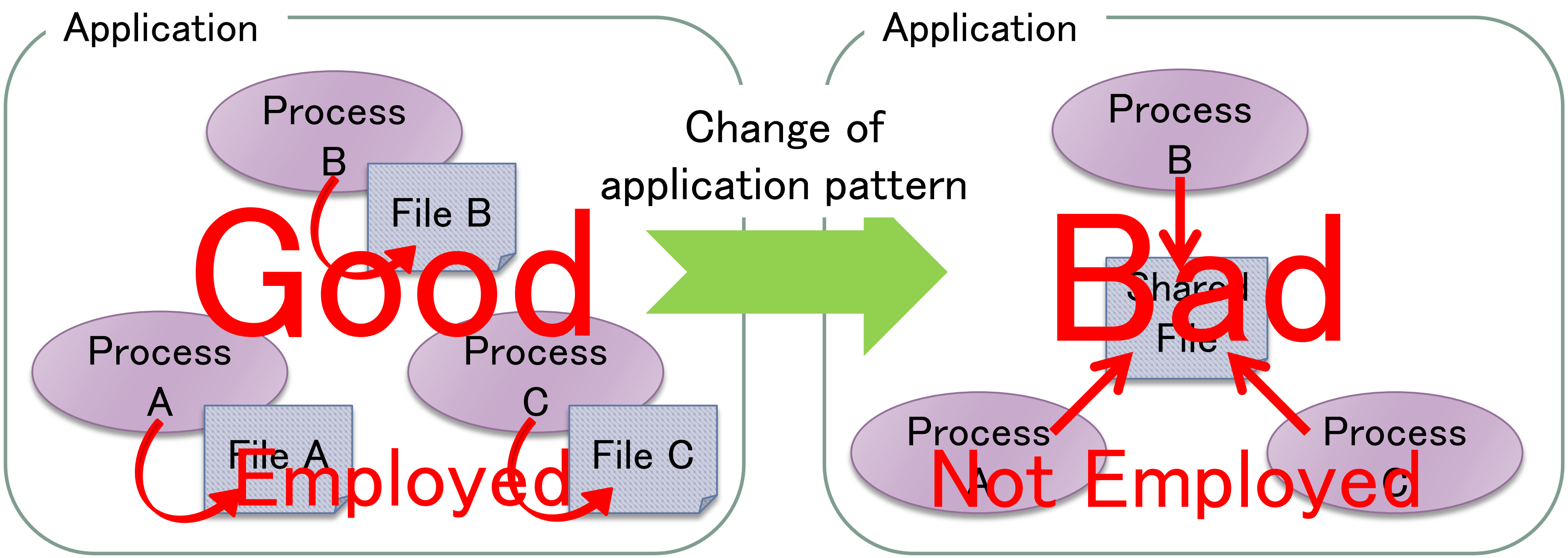


Millions of process utilize
millions of files
on millions of CPU cores

✓ **Hard file management**
✓ **Heavy metadata workload**

Goal of This Research

- N-1 pattern
  - *N* processes utilize *1* shared file

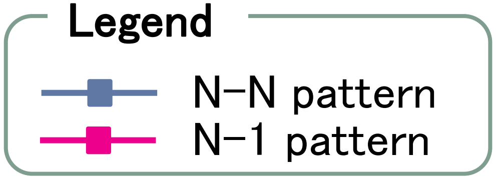


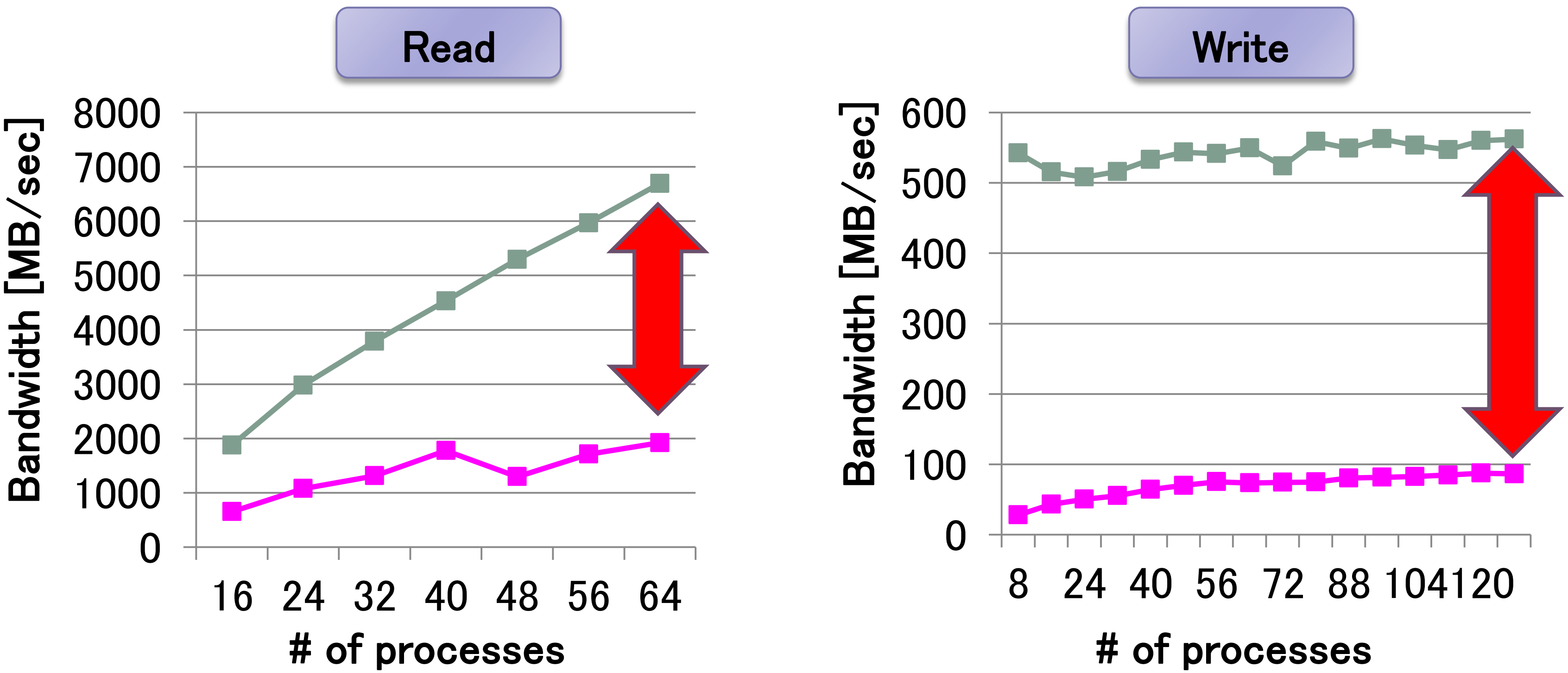
N-N pattern **N-1 pattern**

Why do current HPC applications not employ the N-1 pattern ?

Problem of the N-1 pattern (1/2)

- Low I/O Performance
  - Benchmark Program : MPI-IO Test
  - File System : Lustre Parallel File System



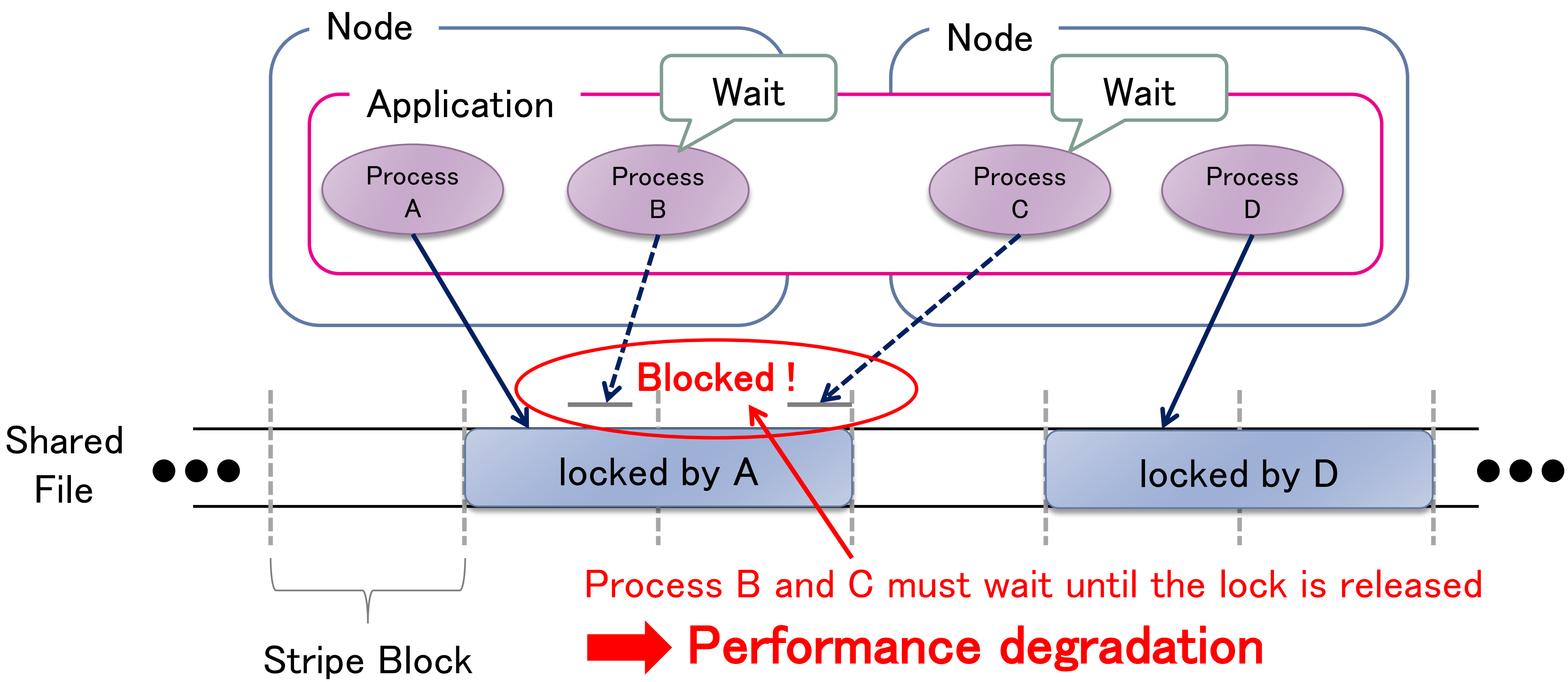


over 3 times lower

over 5 times lower

Problem of the N-1 pattern (2/2)

- File lock contention [Richard 2005]
  - Each process must acquire file lock every stripe block before data access for consistency



Agenda

- File organization trend of HPC applications
  - use of millions of small files
- Problem of single shared file approach for reducing the number of files
  - exhibiting low I/O performance through a benchmark program
- PFA (Parallel File Aggregation) Mechanism
  - providing single shared file APIs for high I/O performance
- Evaluation result on a real HPC application
  - 3.8 times faster than the original with reducing the number of files by about 100,000 files
- Conclusion
- Q & A

Proposed Mechanism

- PFA (Parallel File Aggregation) Mechanism
  - provides N-1 pattern APIs based on memory-map
  - reduces I/O contention by aggregating I/Os
  - does not need file lock
  - reduces amount of data by incremental logging feature

✓**improves the write bandwidth of the N-1 pattern**
✓**reduces the # of files with the use of the N-1 pattern**

APIs of the PFA Mechanism

- Data are read and written sequentially through the APIs based on memory-map

✓ Write data

```
const size_t buf_size = 272,383;
/* allocate a memory region for write */
char* buf
    = pfa_mmap( “foo.txt”, buf_size, rank,… );

while ( condition ) {
    buf[ … ] = …; /* edit data */

    pfa_append( buf, … ); /* append data */
}
/* free the memory region */
pfa_munmap( buf );
```

✓ Read data

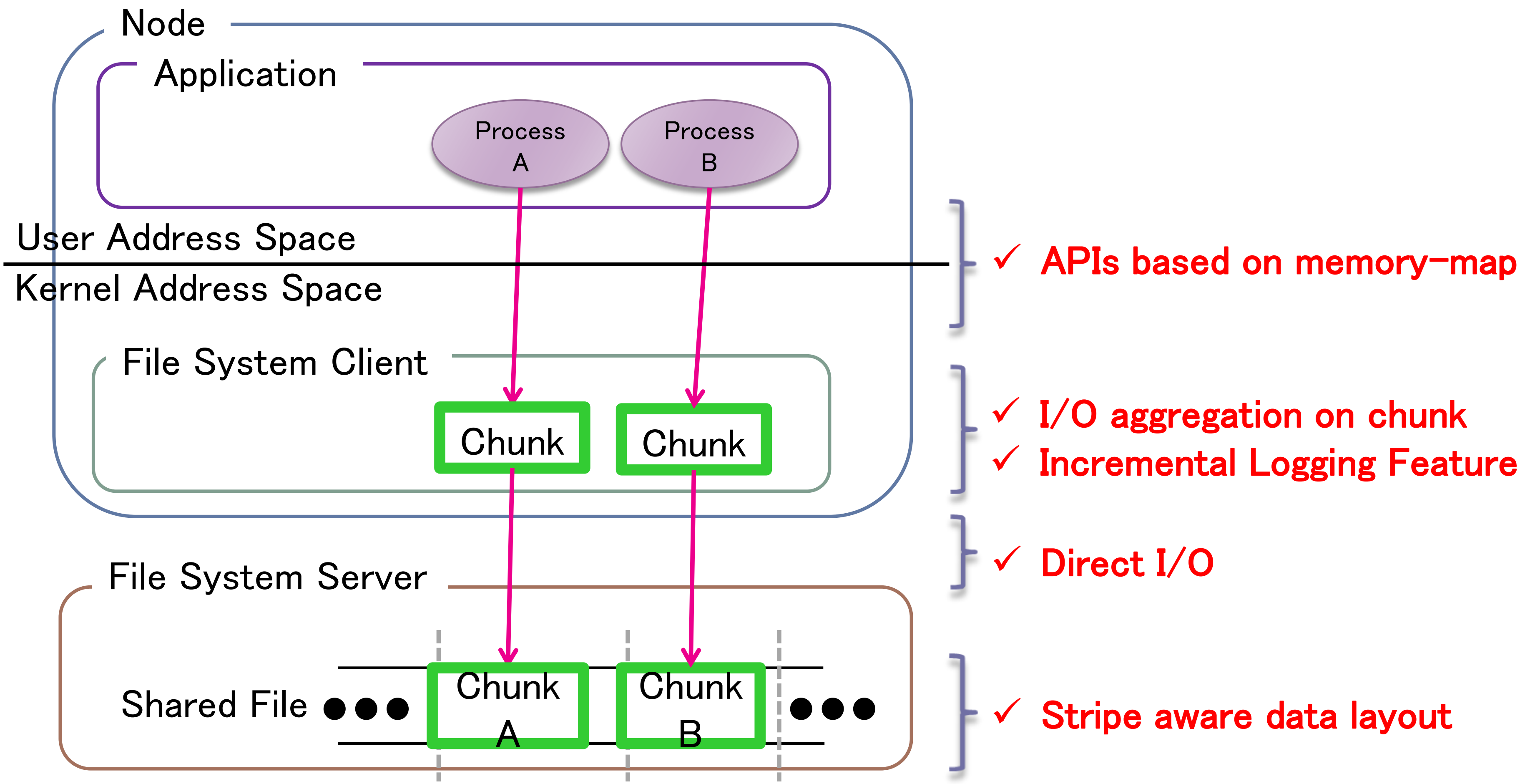
```
const size_t buf_size = 272,383;
/* allocate a memory region for read */
char* buf
    = pfa_mmap( “foo.txt”, buf_size, rank, … );

while ( condition && ! pfa_eof( buf ) ) {
    … = buf[ … ]; /* read data */

    pfa_seek( buf, … ); /* read data */
}
/* free the memory region */
pfa_munmap( buf );
```

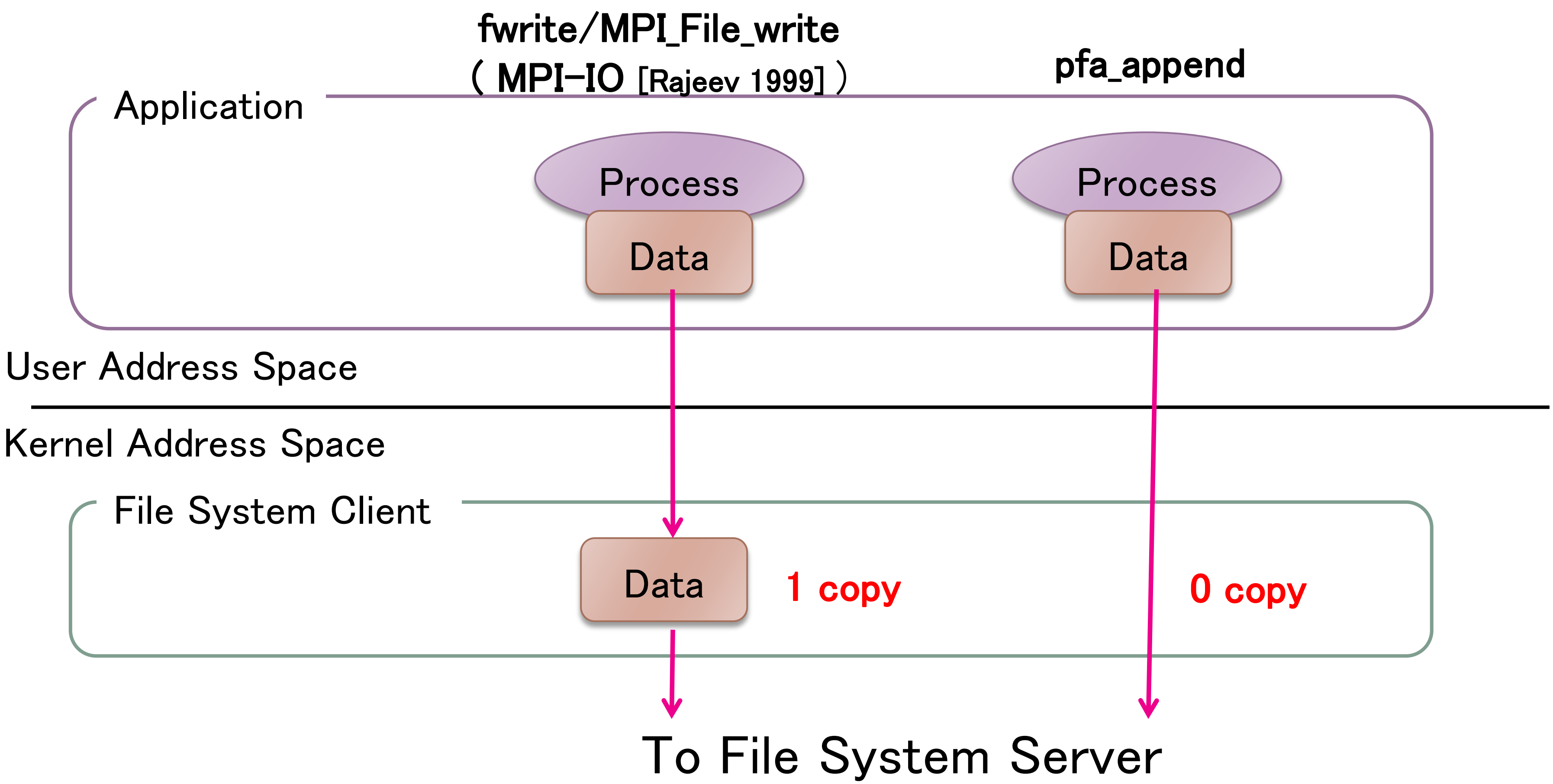
Overview of the PFA mechanism

- The PFA mechanism works on file system client
  - It does not need to modify file system server



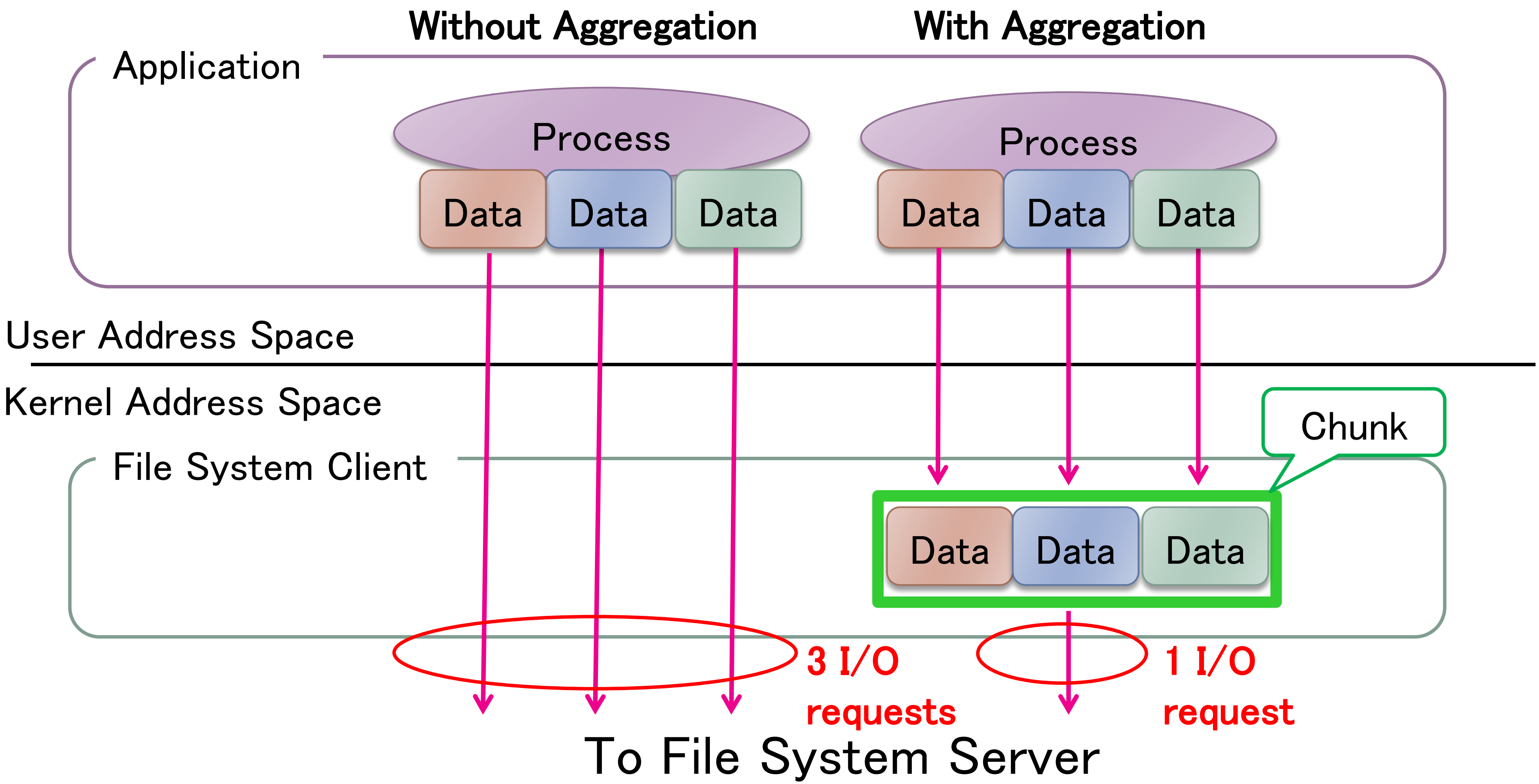
Memory-map

- APIs based on memory-map transfer data from the user address space directly



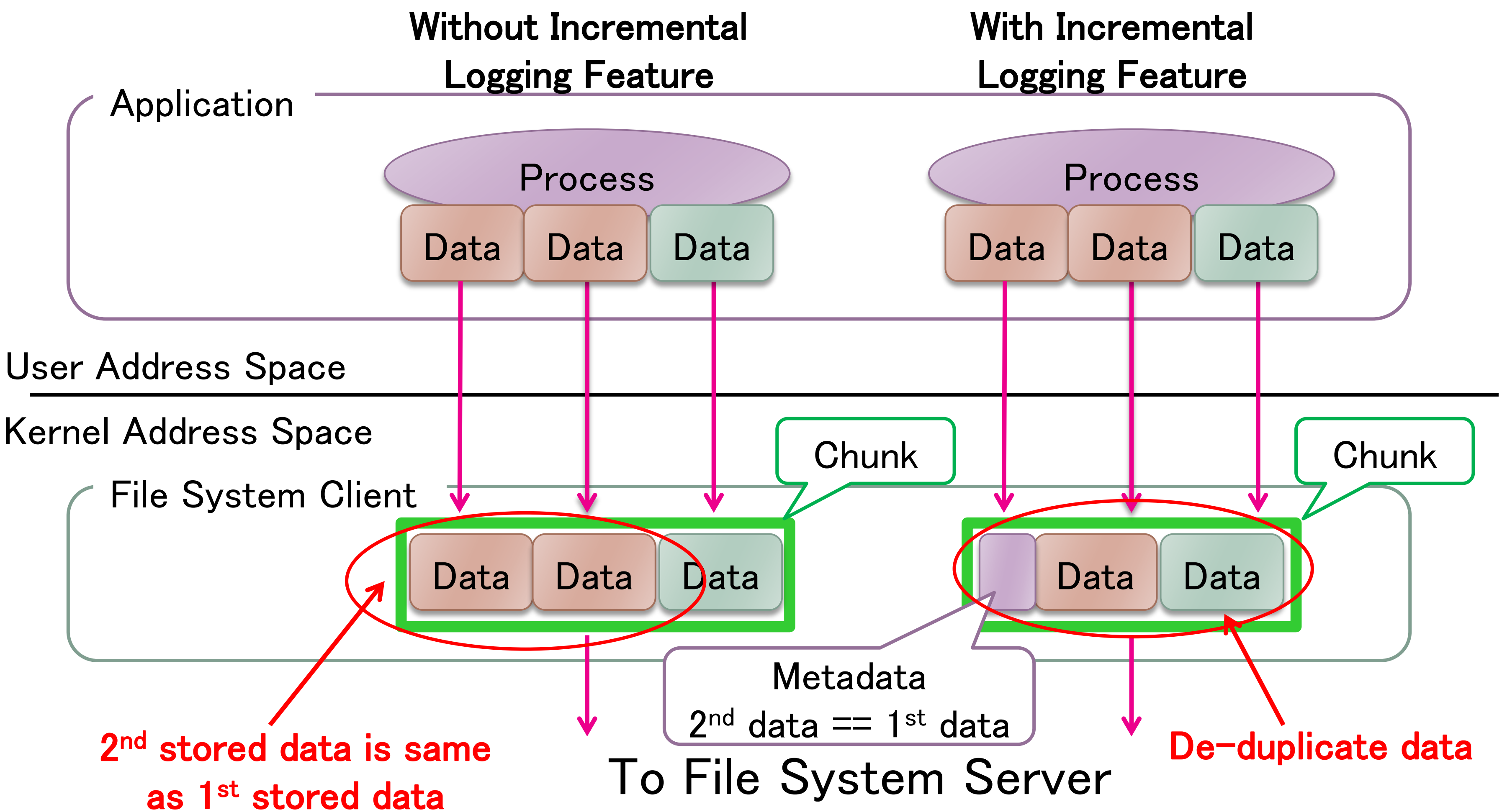
I/O Aggregation

- Data are aggregated into chunk on file system client



Incremental Logging Feature - Overview

- Unmodified data from the previous store are not stored again



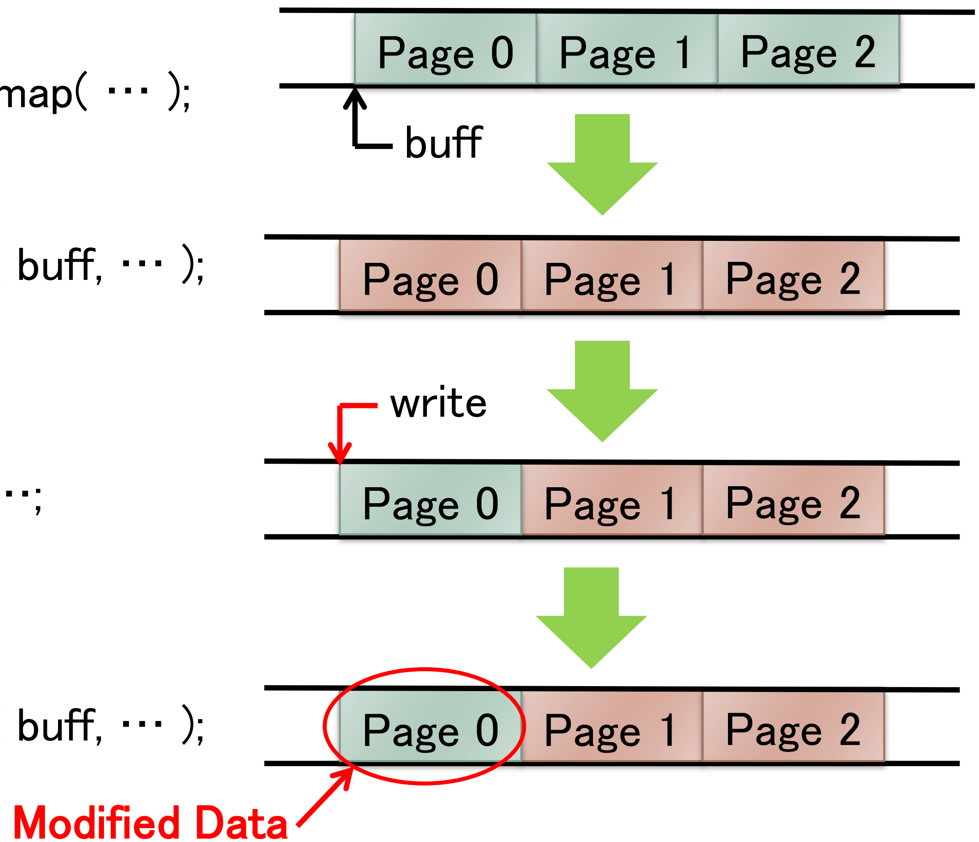
Incremental Logging Feature
- Detection of Modified Data

- Page protection fault is used to detect modified data

Sample Code

```
char* buff
    = pfa_mmap( ··· );
        .
        .
pfa_append( buff, ··· );
        .
        .
        .
buff[ 0 ] = ···;
        .
        .
        .
pfa_append( buff, ··· );
```

On Memory



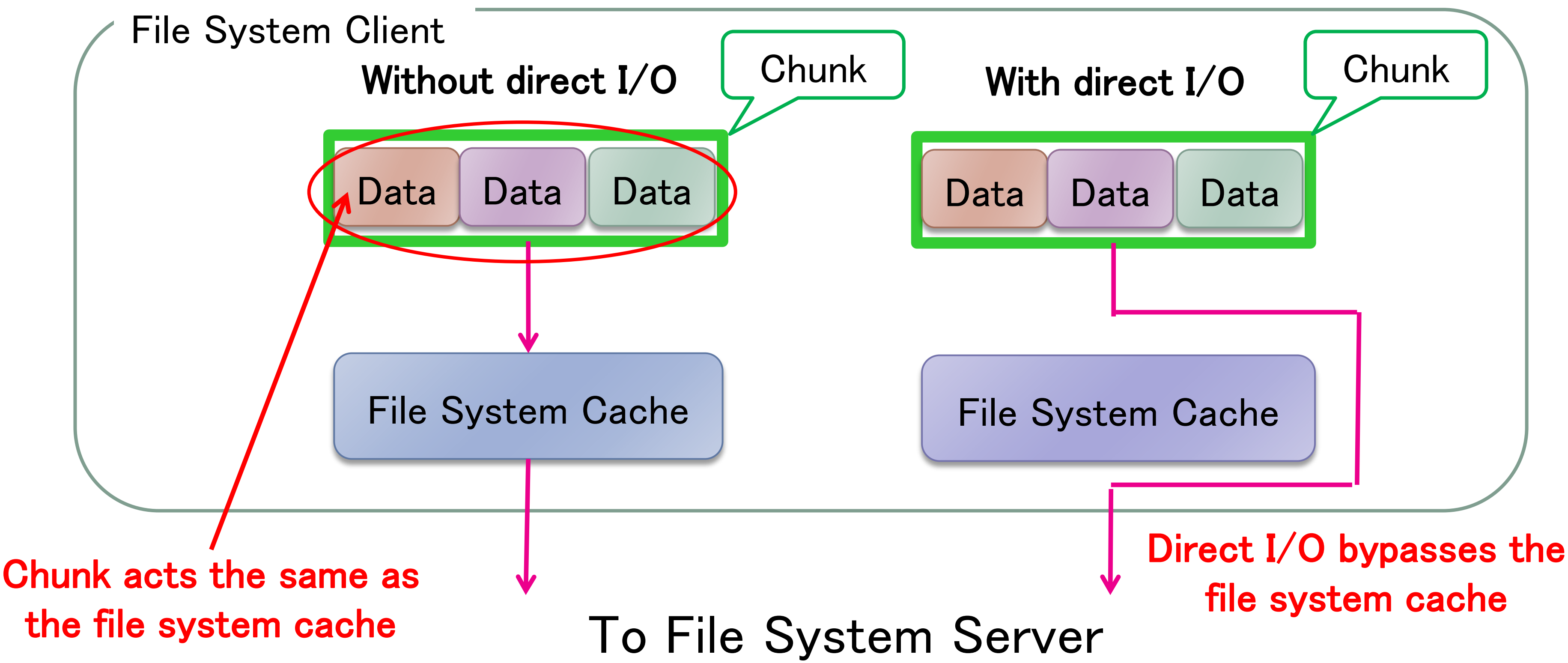
Page: Writable Page

Page: Write Protected Page

- ✓ Allocating pages for buff
- ✓ Storing data
- ✓ Turning off the write bit of the all pages
- ✓ Handling page protection fault on Page 0
- ✓ Turning on the write bit of the Page 0
- ✓ Storing data only on modified pages (= Page 0)

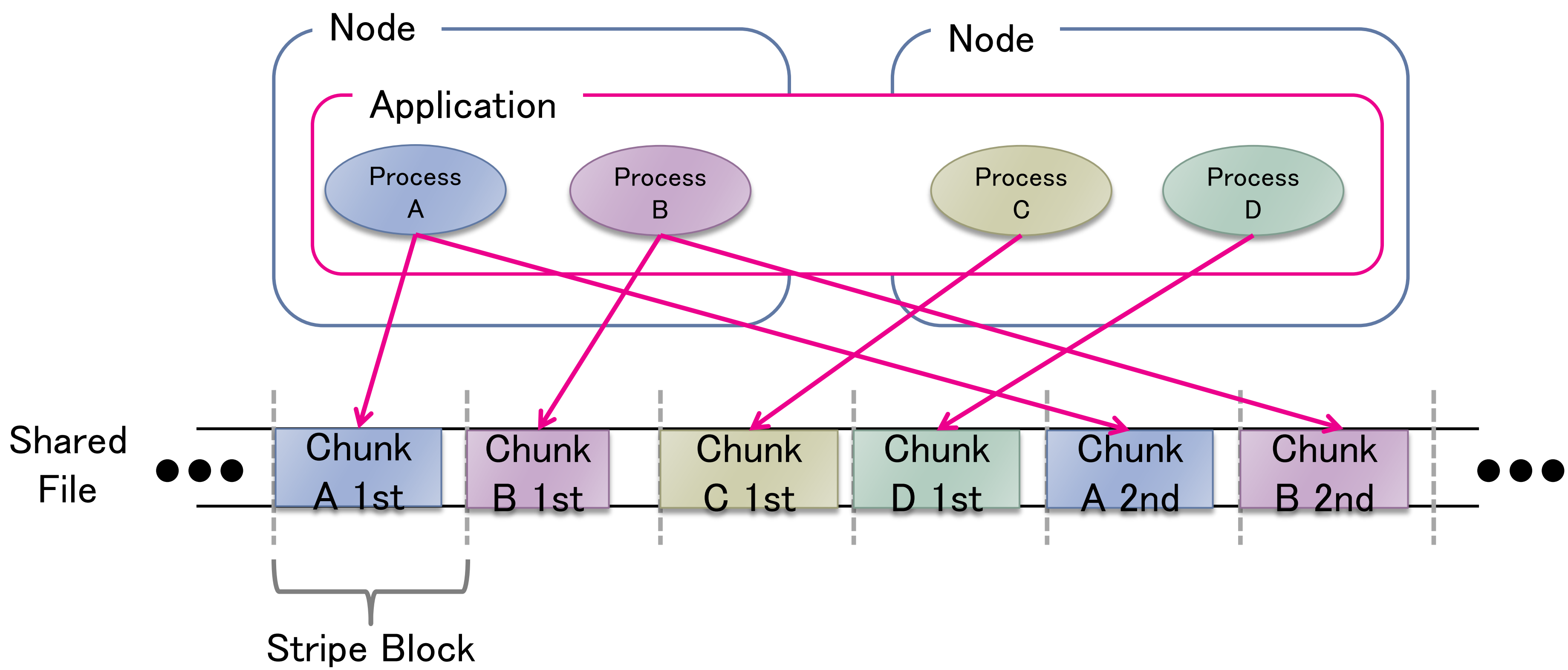
Direct I/O

- Direct I/O avoids cache duplication between file system cache and chunk of the PFA mechanism



Data Layout on Shared File

- Each chunk is aligned on stripe block



Each process does not need to acquire file lock

Agenda

- File organization trend of HPC applications
  - use of millions of small files
- Problem of single shared file approach for reducing the number of files
  - exhibiting low I/O performance through a benchmark program
- PFA (Parallel File Aggregation) Mechanism
  - providing single shared file APIs for high I/O performance
- Evaluation result on a real HPC application
  - 3.8 times faster than the original with reducing the number of files by about 100,000 files
- Conclusion
- Q & A

Evaluation Environment

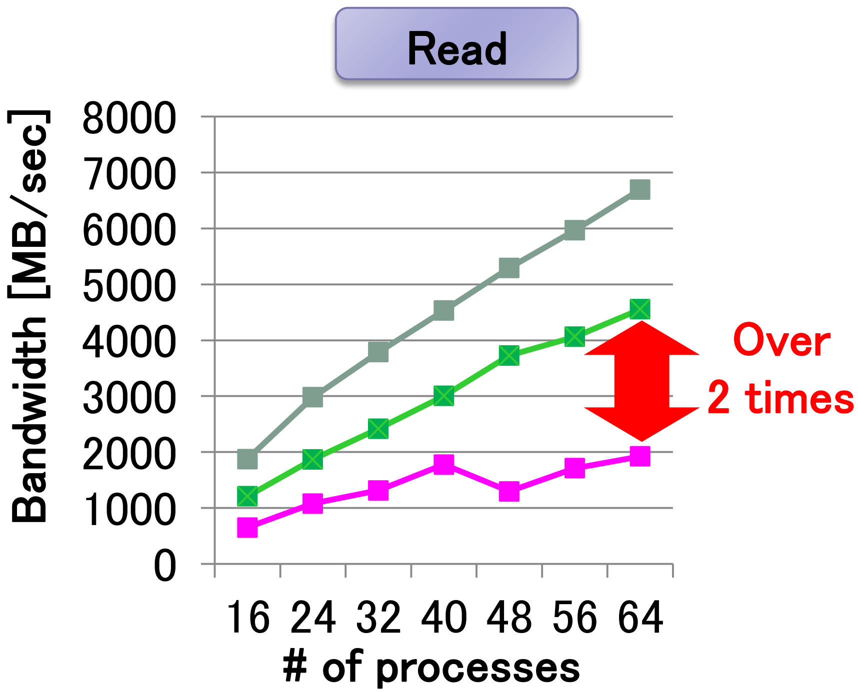
- Evaluated on Lustre Parallel File System
  - Lustre Client
    - 128 cores (= 4 cores * 2 sockets * 16 nodes)
  - Lustre Server
    - 1 MDS (Meta Data Server) on VMWare vSphere 4
    - 4 OSS (Object Storage Server) + 6 OST (Object Storage Target)

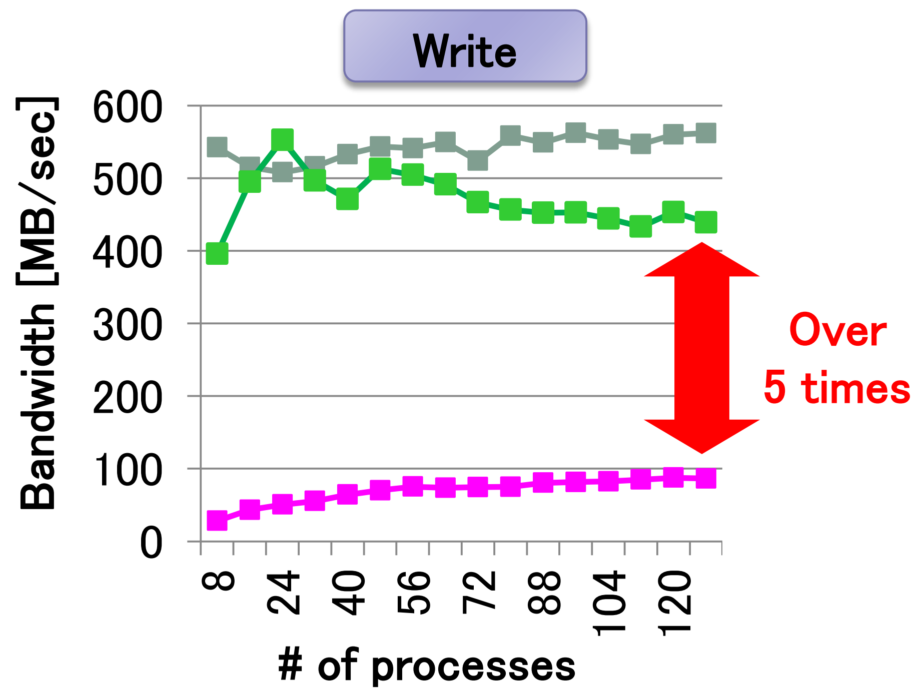
	Client	MDS	OSS
CPU	Intel Xeon X5550 2.67GHz, 8cores	Intel Xeon L5640 2.26GHz, 4 cores in 12 cores	Intel Xeon L5640 2.26GHz, 12 cores
Memory	DDR3 24GB	DDR3 16008MB in 48GB	DDR3 48GB
Disk	160GB SATA	6Gbps 7,200 rpm SAS 500GB x 4	6Gbps 7,200 rpm SAS 500GB x 2
Interconnect	Infiniband 4x QDR	Infiniband 4x QDR	Infiniband 4x QDR
OS	RHEL5(2.6.18-194)	RHEL5(2.6.18-164)	RHEL5(2.6.18-164)
Lustre	1.8.4	1.8.3	1.8.3

MPI-IO Test Benchmark

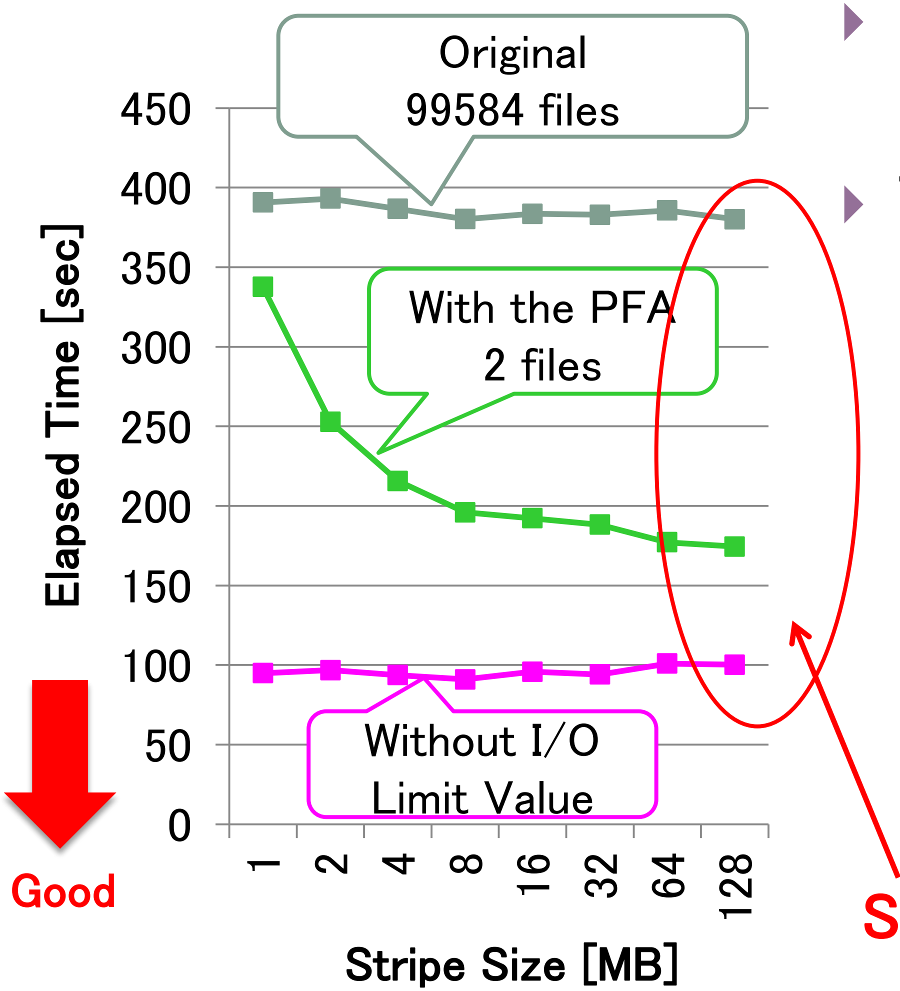
- Test Configuration
  1. Write 272,383 bytes for a minute
  2. Read written data
- Result
  - N-N > N-1 with the PFA > N-1
  - N-N pattern generates **128 files at most … too low**

Legend
- N-N pattern
- N-1 pattern
- N-1 pattern with the PFA





Athena Application [Stone 2008]



- Simulating Rayleigh–Taylor instability with 128 processes
- Total 99584 files in original
  - 49792 simulation data files
    - Average file size : 737534 byte
    - with incremental logging
      - Saving 30.8% data
  - 49792 checkpoint data files
    - Average file size : 272383 byte
    - without incremental logging

Speeding up 3.8 times faster than the original in I/O part

Related Work & Comparison

- MPI-IO [Rajeev 1999]
  - provides N-1 pattern APIs based on file
  - requires copy between the user and the kernel address spaces
- SIONlib [Frings 2009]
  - converts the N-N pattern into N-1 pattern on the library
  - incurs performance degradation due to the file lock contention
- PLFS [Bent 2009]
  - provides virtual view of the shared file on the file system server
  - incurs metadata stress due to actually employing the N-N pattern
- The PFA mechanism
  - provides N-1 pattern APIs based on memory-map
  - works on the file system client

Conclusion

- The N-1 pattern exhibits poor I/O performance
  - Most applications employ the N-N pattern and generate millions of small files
- PFA (Parallel File Aggregation) Mechanism
  - It improves I/O performance of the N-1 pattern
    - providing N-1 pattern APIs based on memory-map
    - reducing I/O contention by aggregating I/Os
    - no file lock
    - reducing amount of data by incremental logging feature
- The Athena application speeds up 3.8 times than the original with reducing the number of files by about 100,000 files