Fig. 00.

Parallel I/O for SwissTx

Emin Gabrielyan
Prof. Roger D. Hersch
Peripheral Systems Laboratory
Ecole Polytechnique Fédérale de Lausanne
Switzerland

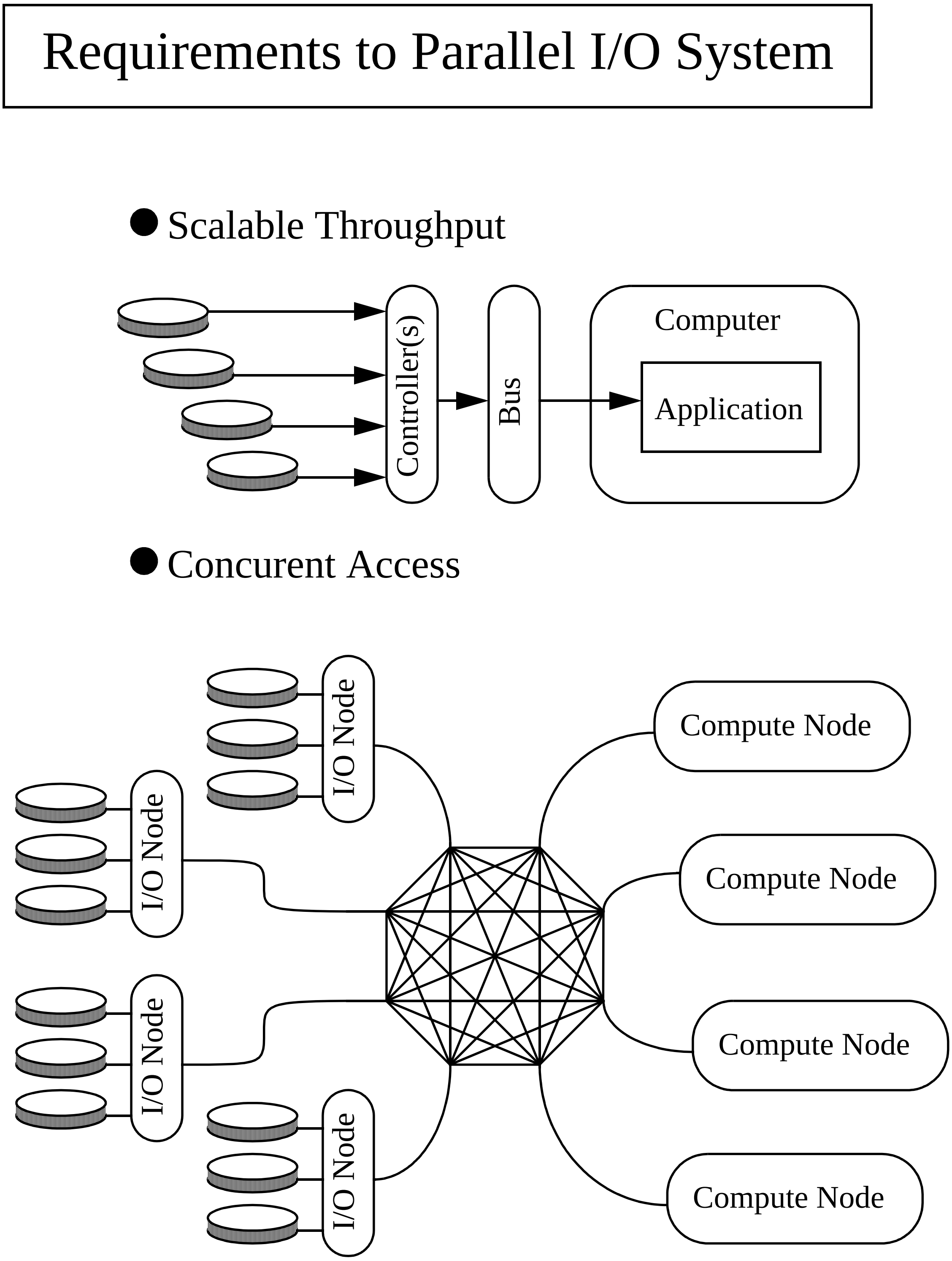
- Introduction
- SFIO (Striped File I/O) software architecture
- SFIO interface
- Performance measurements
- Integration into MPI-II I/O
- Conclusion

Fig. 01.

Parallel I/O for Swiss-Tx (Introduction)

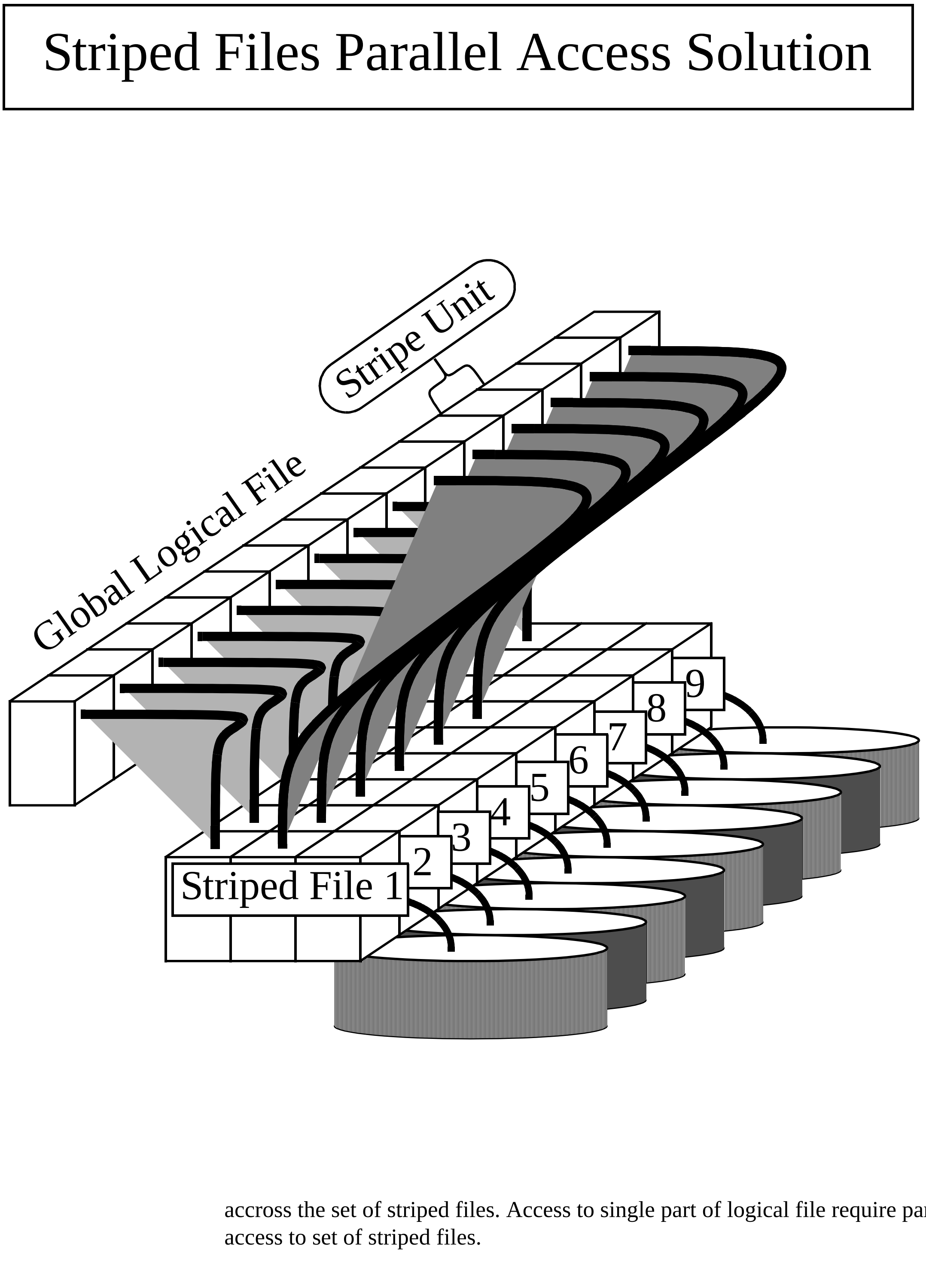
- Requirements for Parallel I/O
- Striped Files Parallel Access Solution
- Interface
- Network Communication and
  Disk Access optimisation

Fig. 02. We are trying to be close to the linear dependance of throughput from the



number of I/O devices. The upper limit of throughput can be the controller(s)/bus bandwidth or the TOTAL network bandwidth.

Fig. 03. The basic idea of our implementation is cyclical distribution of logical file



accross the set of striped files. Access to single part of logical file require parallel access to set of striped files.

Fig. 04. The native interface is SFIO, but we work to provide also MPI-II I/O interface.

Interface

● SFIO

MFILE* **mopen**(char *s, int unitsz); // “t0-p1,/tmp/a;t0-p2,/tmp/a”

void **mclose**(MFILE *f);

void **mchsize**(MFILE *f, long size);

void **mdelete**(char *s);

void **mcreate**(char *s);

void **mread**(MFILE *f, long offset, char *buf, unsigned count);

void **mwrite**(MFILE *f, long offset, char *buf, unsigned count);

void **mreadb**(MFILE *f, unsigned bcount,
    long Offset[], char *Buf[], unsigned Count[]);

void **mwriteb**(MFILE *f, unsigned bcount,
    long Offset[], char *Buf[], unsigned Count[]);

● MPI-II I/O

MPI_File_open	MPI_File_write_all
MPI_File_set_view	MPI_File_read_all
MPI_File_write	MPI_File_write_at_all
MPI_File_read	MPI_File_read_at_all
MPI_File_write_at	MPI_File_close
MPI_File_read_at	MPI_File_delete

Fog. 05.

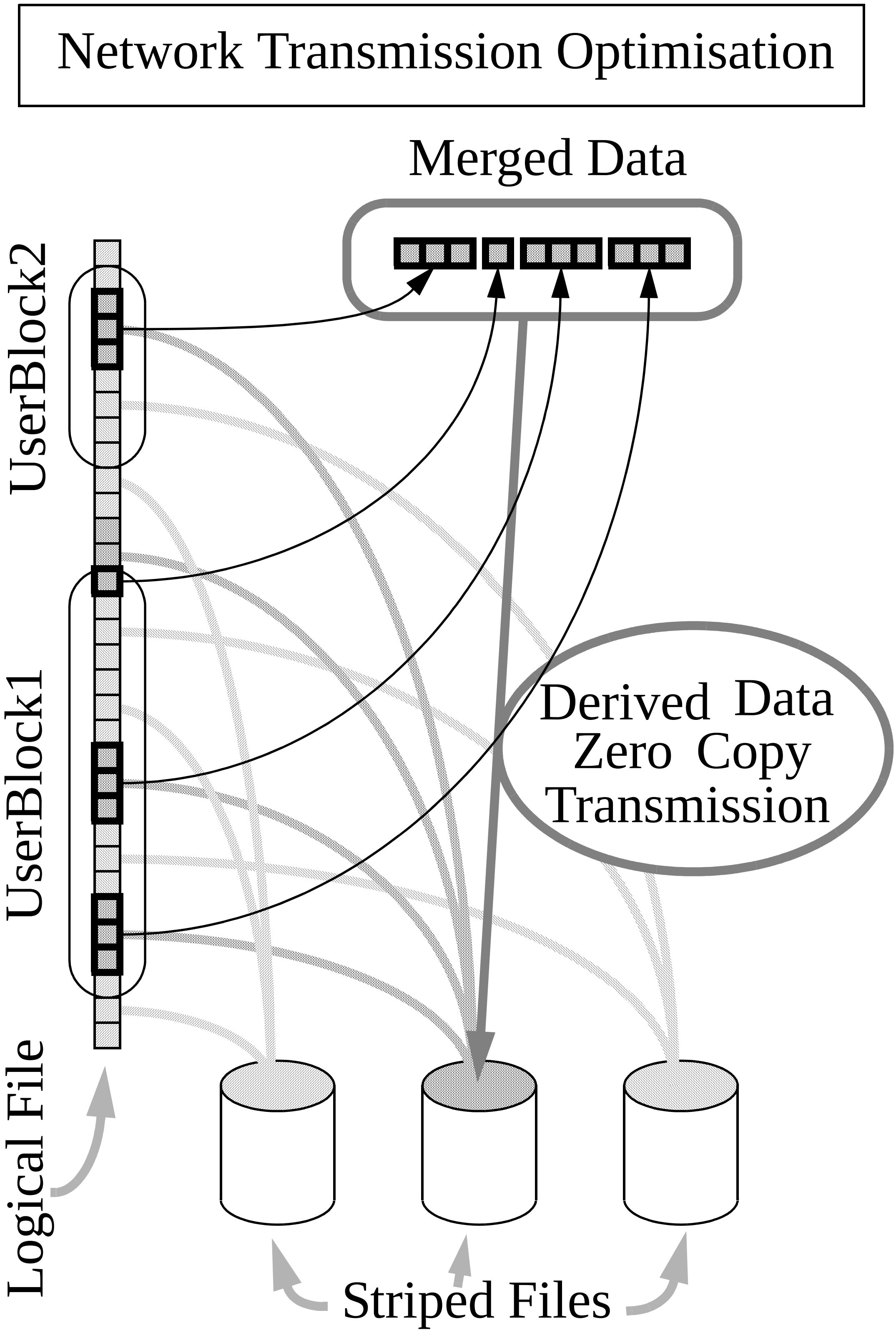
Network and Disk Optimisation

- Network optimisation for noncollective access

- Disk optimisation for noncollective access

- Collective access optimisation

Pg. 06. Fragmented data optimisation for communication



Disk Access Optimisation

Fig. 07. Data fragmentation is resolved for network communication

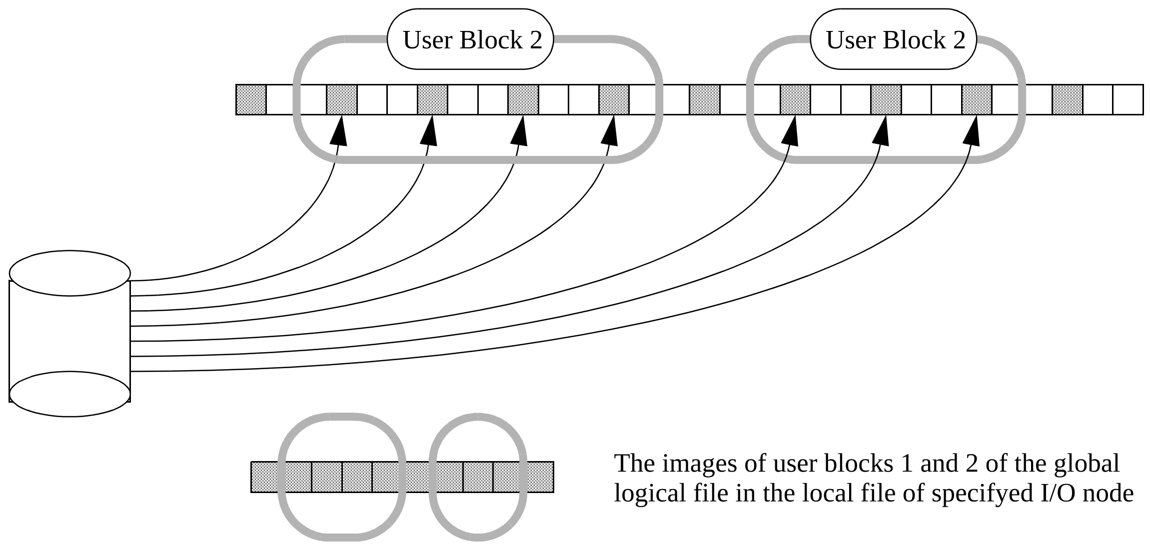


Fig. 08. Data fragmentation still exist at disk access

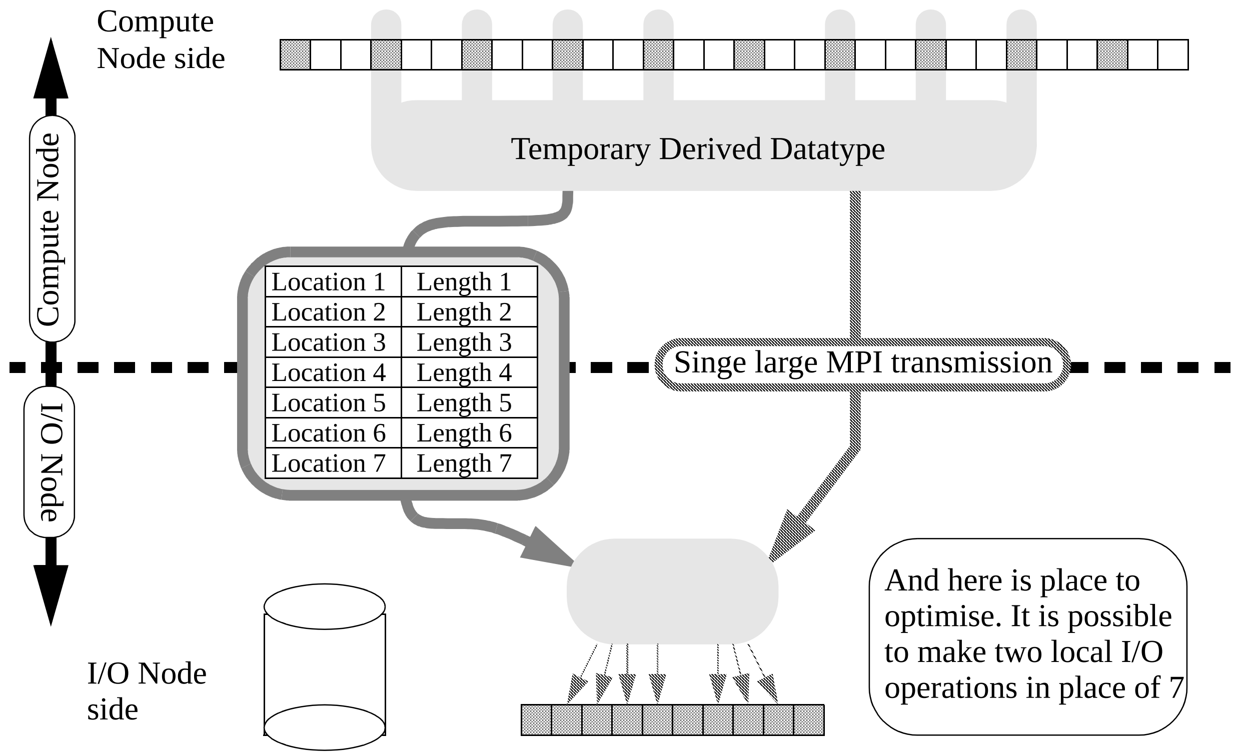


Fig. 09. Data fragmentation is resolved on disk level also

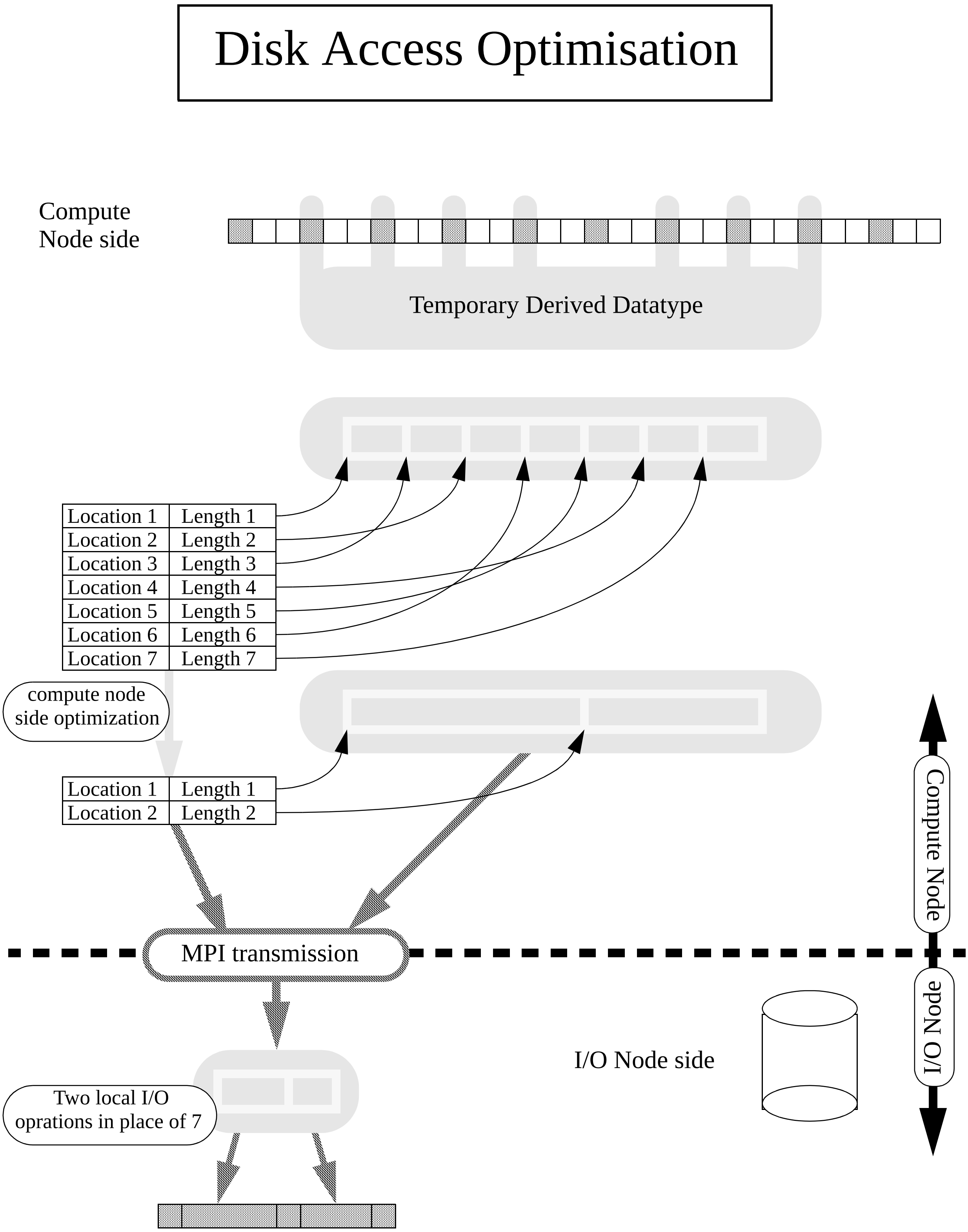


Fig. 10. Disk Access Optiisation of Collective Operations

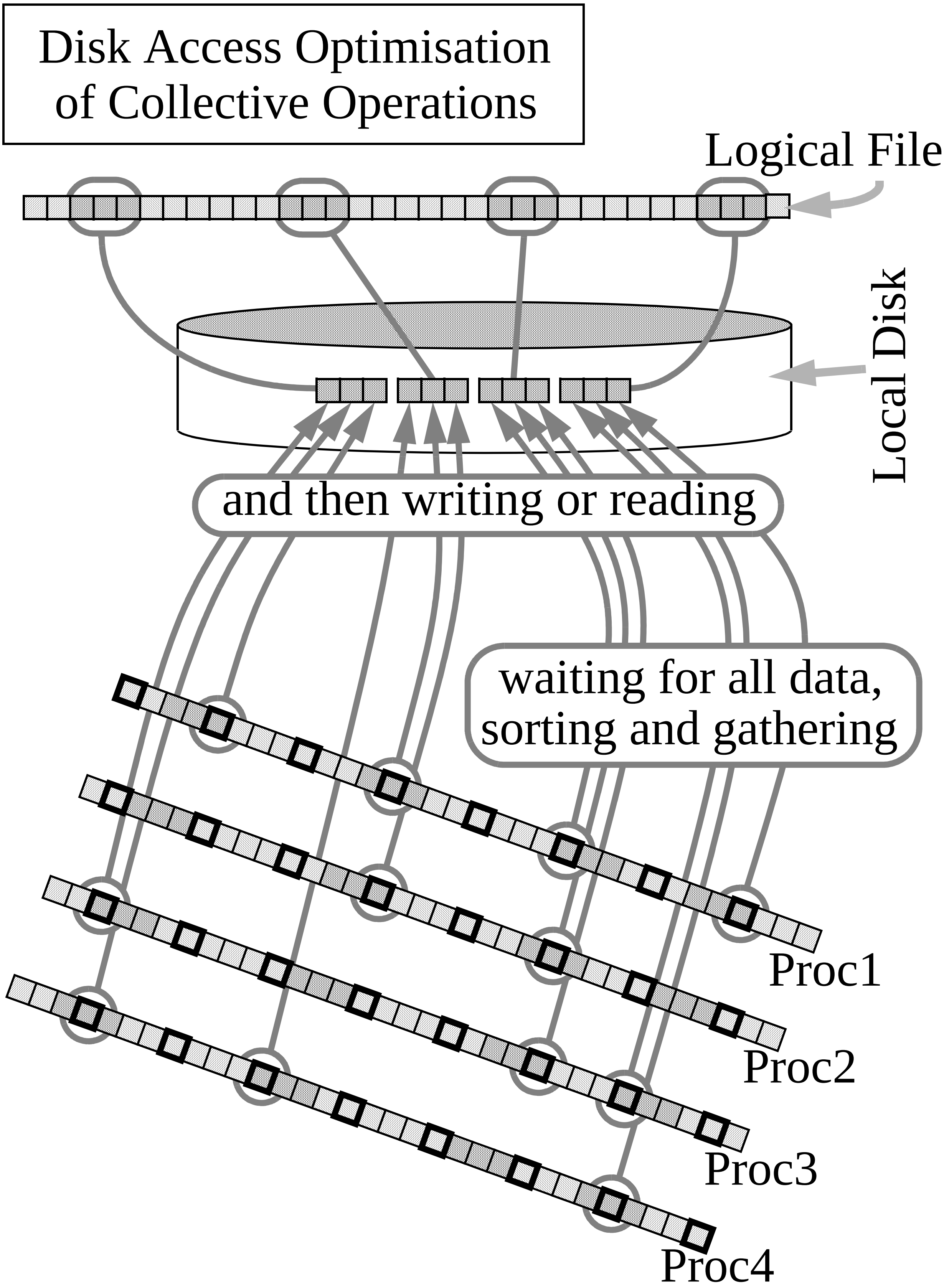
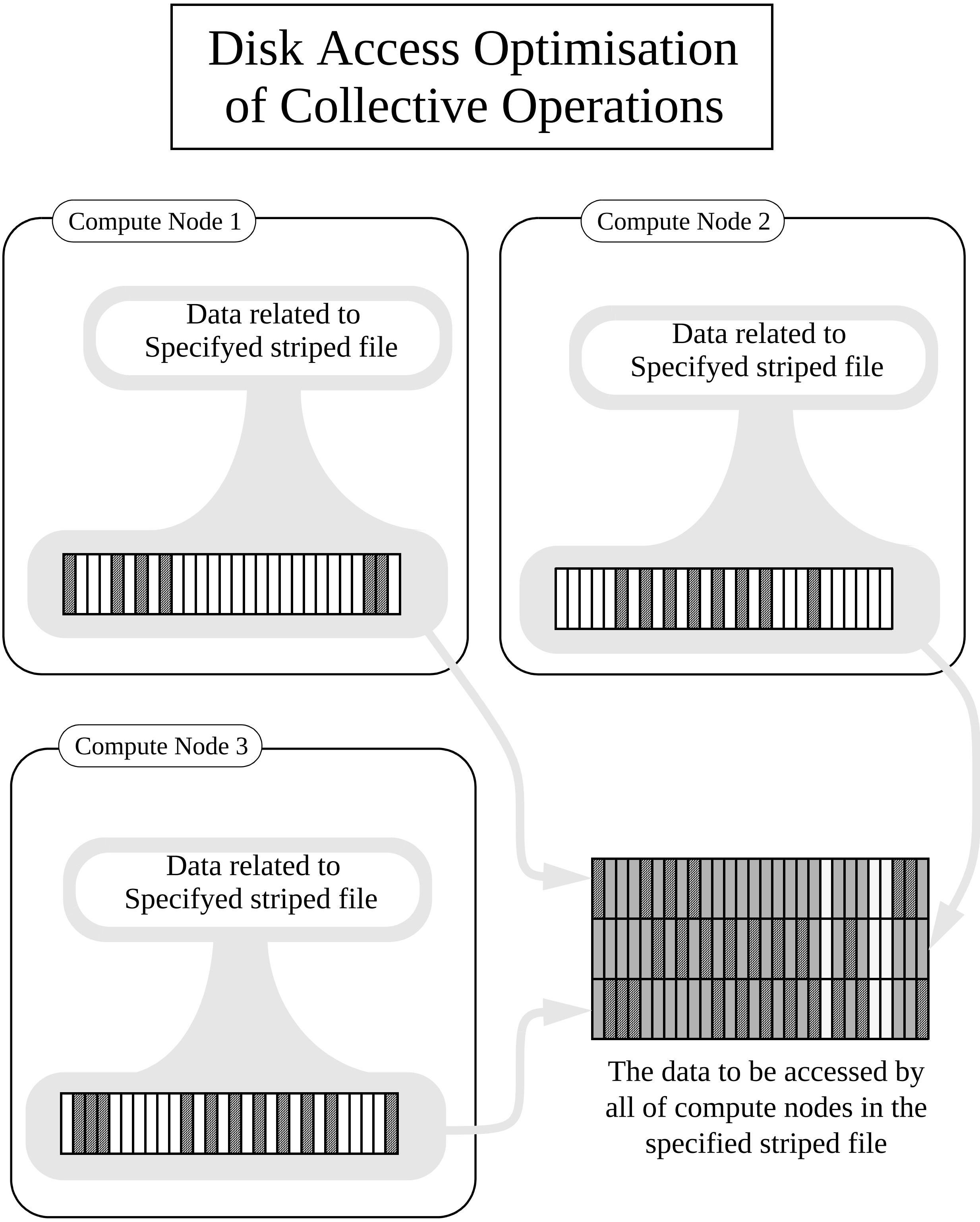


Fig. 11. Another example when disk access optimisation is required at collective



operations. Here each compute node tryes to access some part of logical file, but in the picture is shown the translation of this part to the view of some specifyed striped file. For each of compute node the striped file is same.

Fig. 12.

SFIO Software Architecture

- Encapsulation over MPI

- Functionality

- Data organisation and optimisation

Fig. 13. Encapsulation over MPI

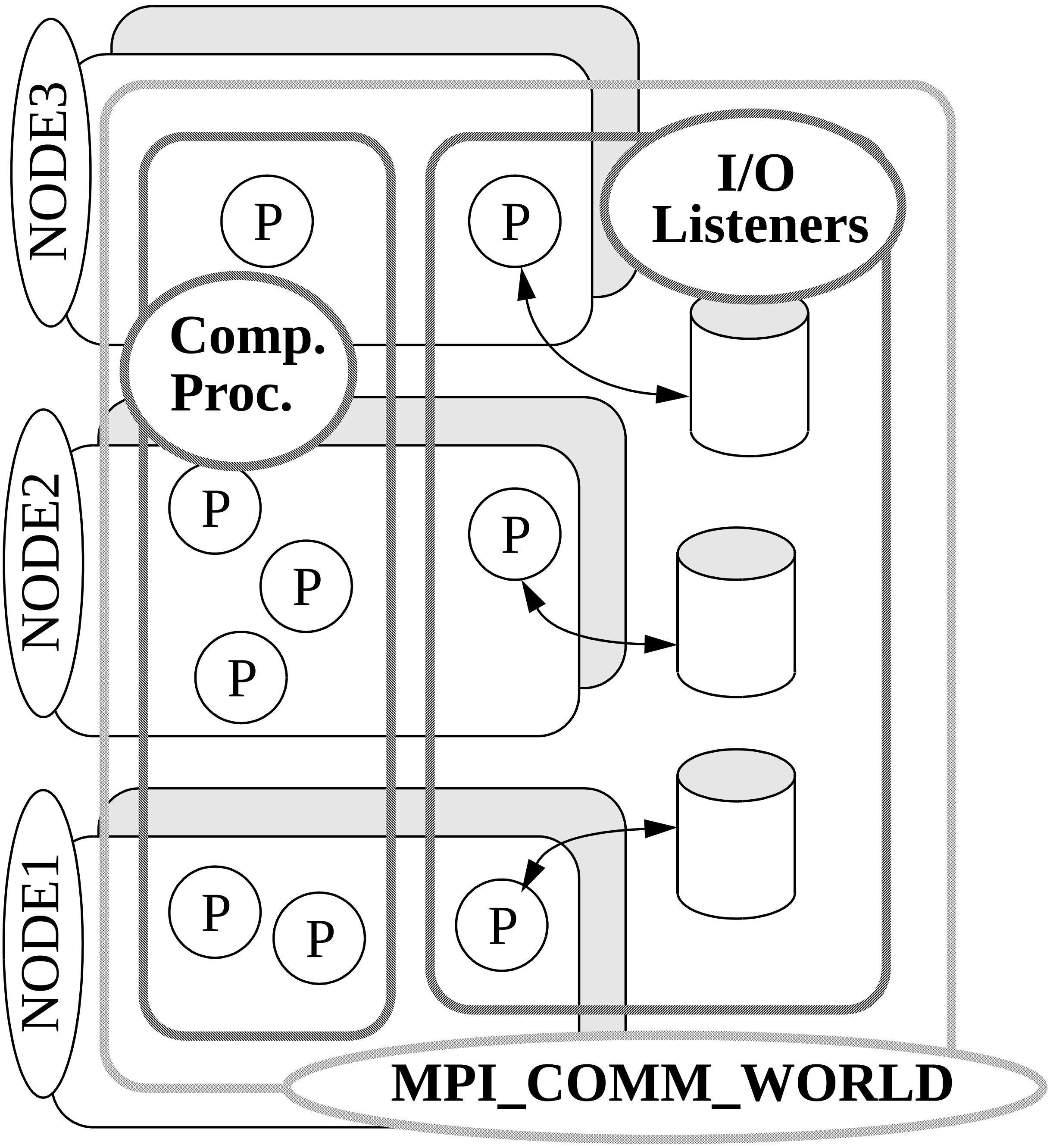
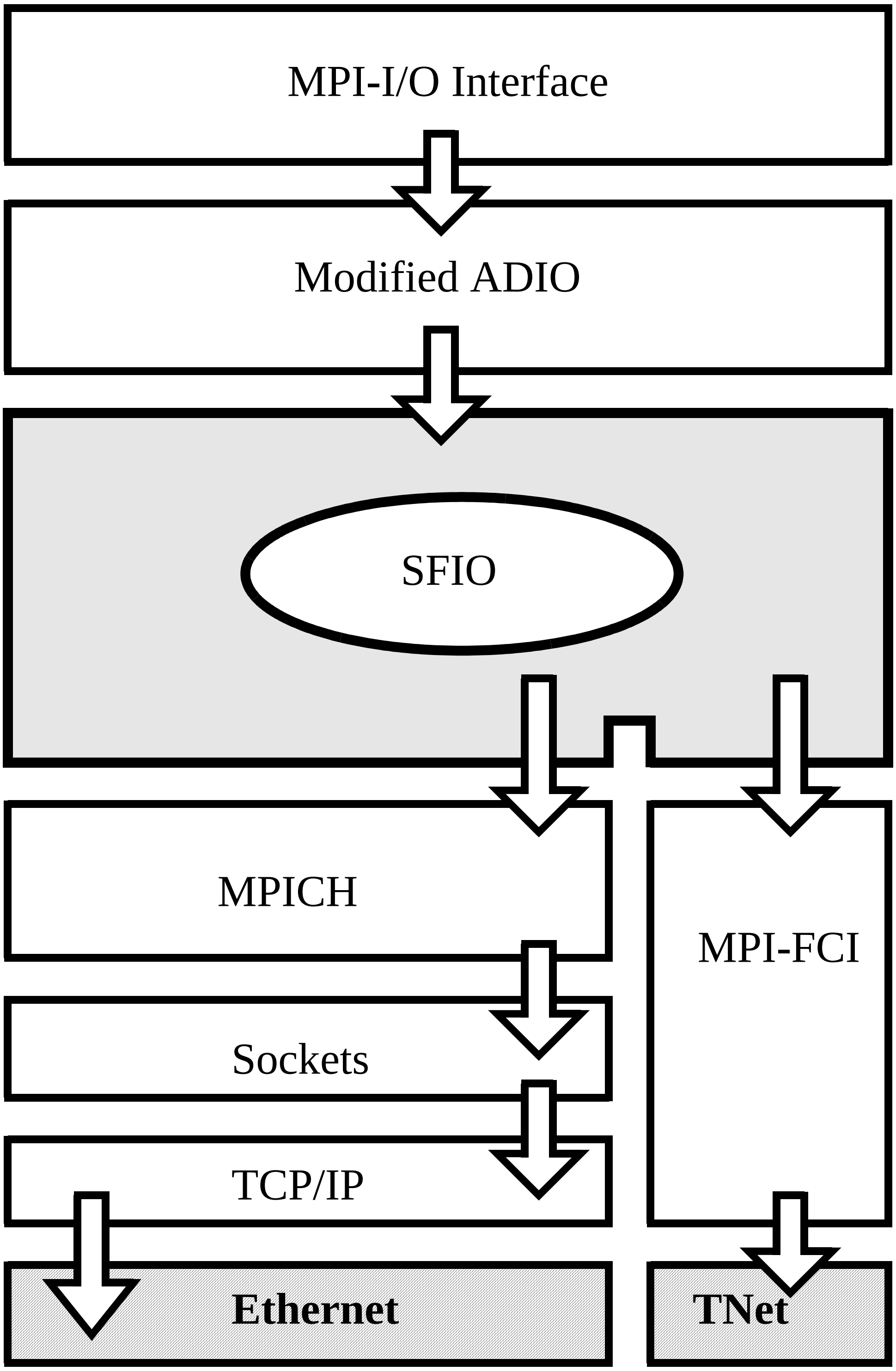


Fig. 14. Functionality. SFIO is implemented and tested with Digital Unix



(MPICH, FCI) and Intel Windos NT (MPICH).

Fig. 15. Functionality, data organisation and optimisation

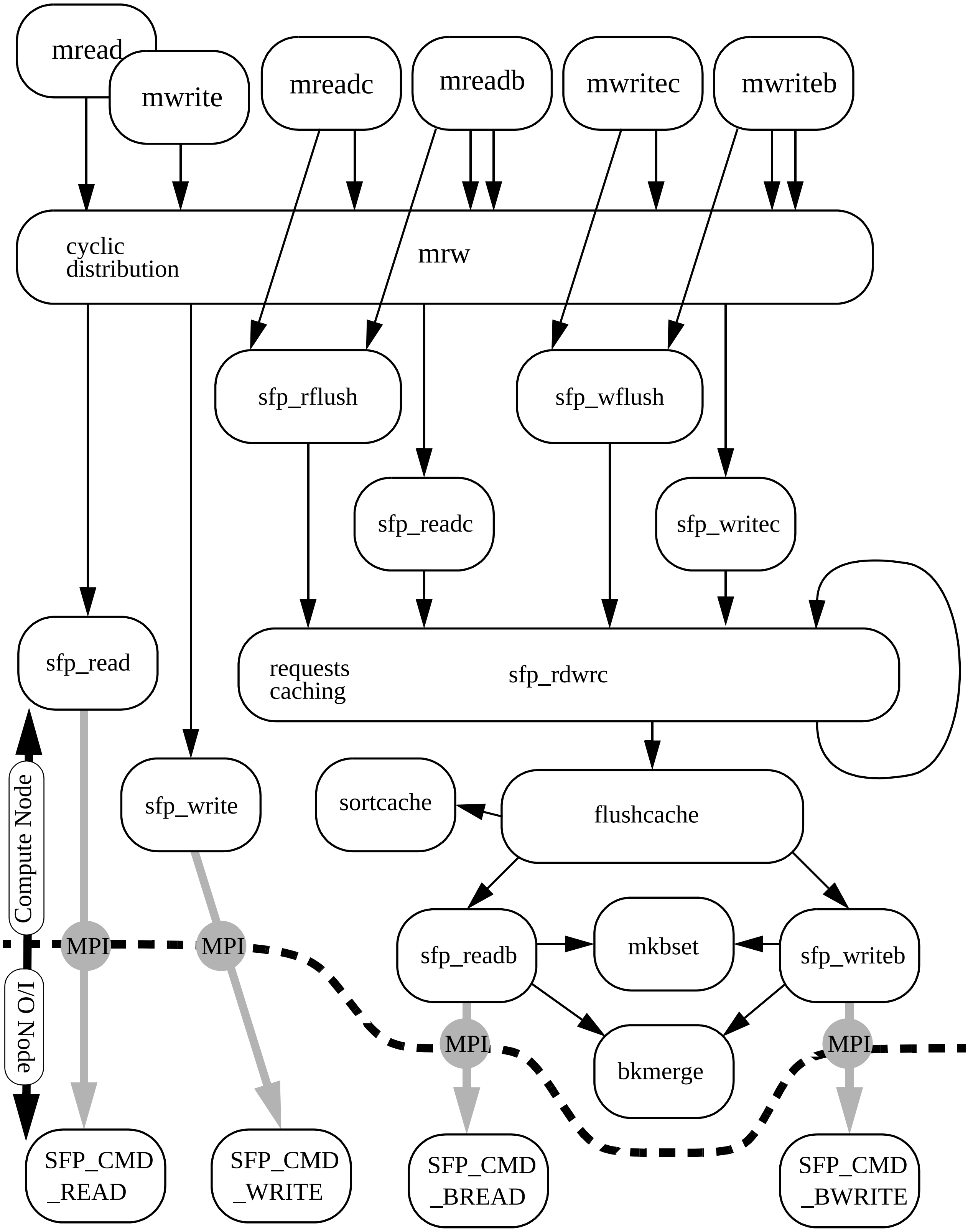


Fig. 16.

SFIO Interface

- Single Block Access Interface

- Multiple Block Access Interface

Fig. 17. If there is one compute node

Single Block Access Interface

```
#include "/usr/p5/gabriely/mpi/lib/mio.h"

int _main(int argc, char *argv[])
{
   char buf[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   MFILE *f;

   f=mopen( "t0-p1.epfl.ch,/tmp/aa;t0-p2.epfl.ch,/tmp/aa" , 10 );

   mwrite( f , 0 , buf , 26 );

   mclose(f);
   return 0;
}
```

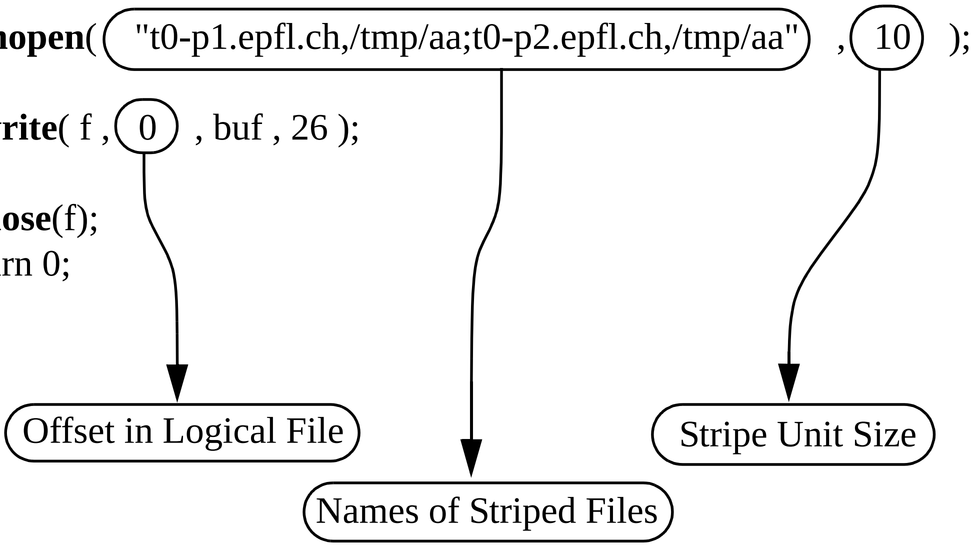


Fig. 18. If there are more than one compute nodes

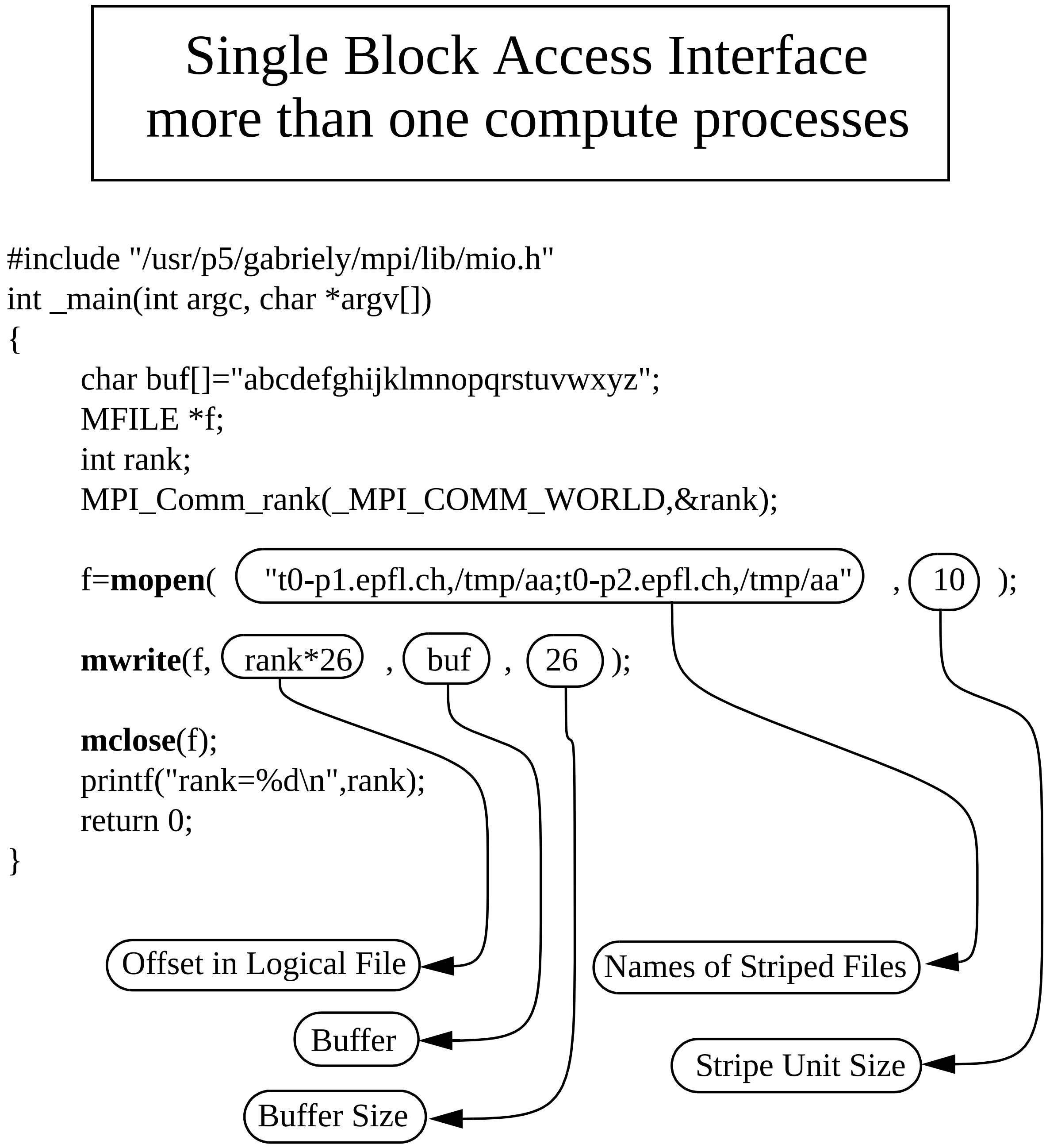


Fig. 19.

Multiple Block Access Interface

```
#include "/usr/p5/gabriely/mpi/lib/mio.h"
int _main(int argc, char *argv[])
{
        char buf1[]="abcdefghijklmnopqrstuvwxyz";
        char buf2[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        char buf3[]="0123456789";
        char* Buf[]={buf1,buf2,buf3};
        long Offset[]={0,26,52};
        unsigned Count[]={26,26,10};
        MFILE *f;

        f=mopen("t0-p1.epfl.ch,/tmp/aa;t0-p2.epfl.ch,/tmp/aa",10);

        mwriteb(f, 3 ,Offset,Buf,Count);

        mclose(f);
        return 0;
}
```

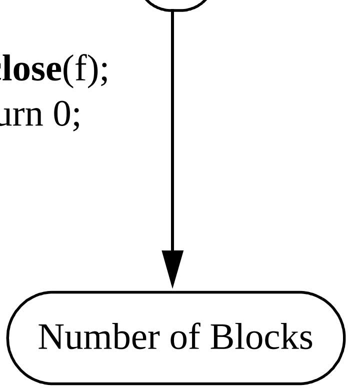


Fig. 20.

Performance Measurements

Data Access Performance on Windows NT cluster (4 I/O nodes). Effect of Optimisation

Data Access Performance on Swiss-Tx SFIO over MPICH with 100Mbps ethernet HUB (7 I/O nodes). Effect of Optimisation

Data Access throughput dependance from stripe unit size (susz) and user block size (bksz) on Windows NT cluster (4 I/O nodes)

Data Access throughput dependance from stripe unit size (susz) and user block size (bksz) on Swiss-Tx SFIO over MPICH with 100Mbps ethernet HUB (7 I/O nodes).

Performance measurements on Swiss-Tx SFIO over MPICH with GigaEthernet Switch (4 I/O nodes) for different number of concurently reading/writing compute processes.

Performance measurements on Swiss-Tx SFIO over MPI-FCI with TNET Crossbar Switch (4 I/O nodes). Effect of Optimisation

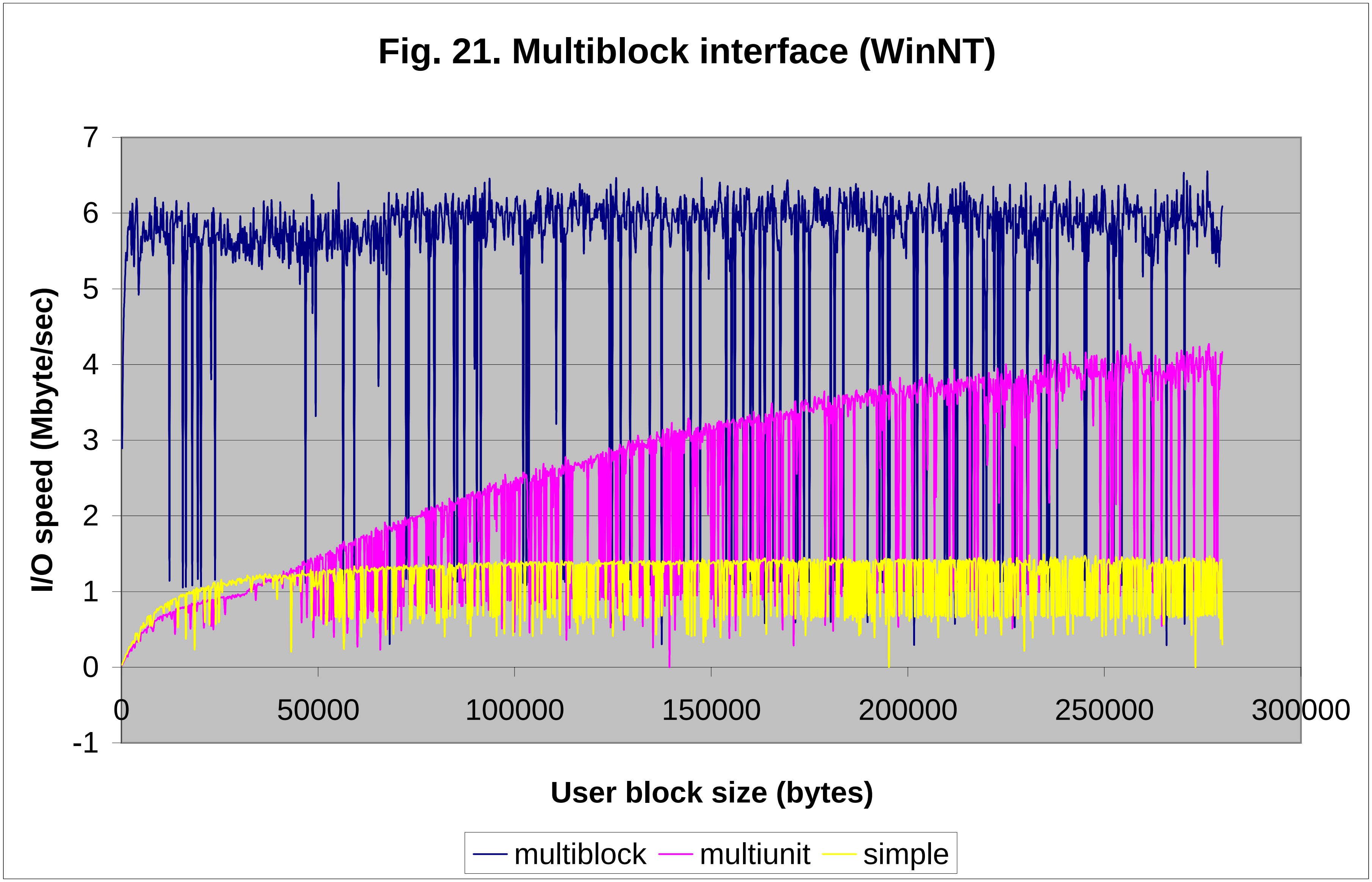


Fig. 21. Multiblock interface (WinNT)
I/O speed (Mbyte/sec)
7
6
5
4
3
2
1
0
-1
0
50000
100000
150000
200000
250000
300000
User block size (bytes)
multiblock
multiunit
simple

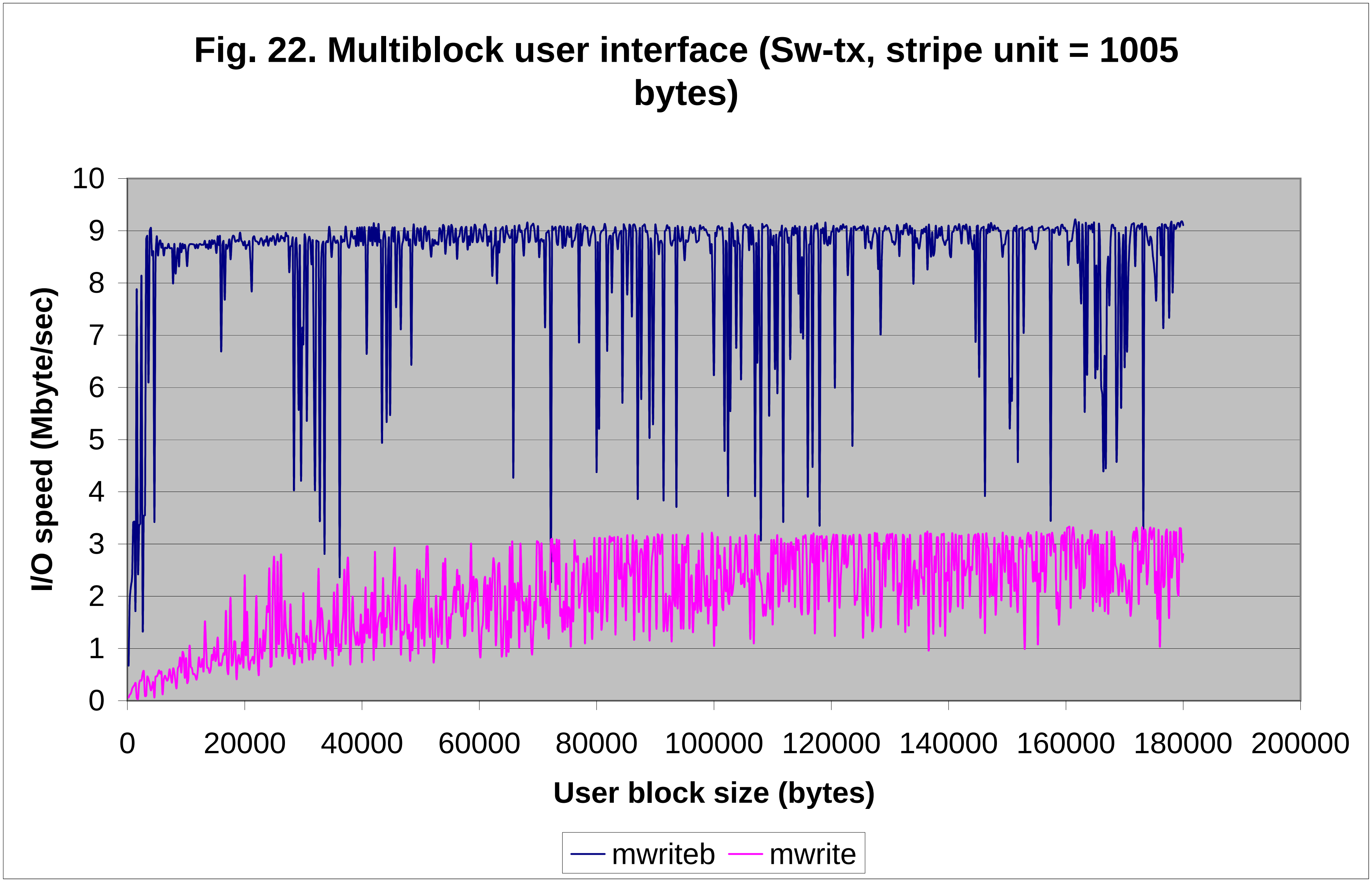


Fig. 22. Multiblock user interface (Sw-tx, stripe unit = 1005 bytes)
I/O speed (Mbyte/sec)
0
1
2
3
4
5
6
7
8
9
10
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
User block size (bytes)
mwriteb
mwrite

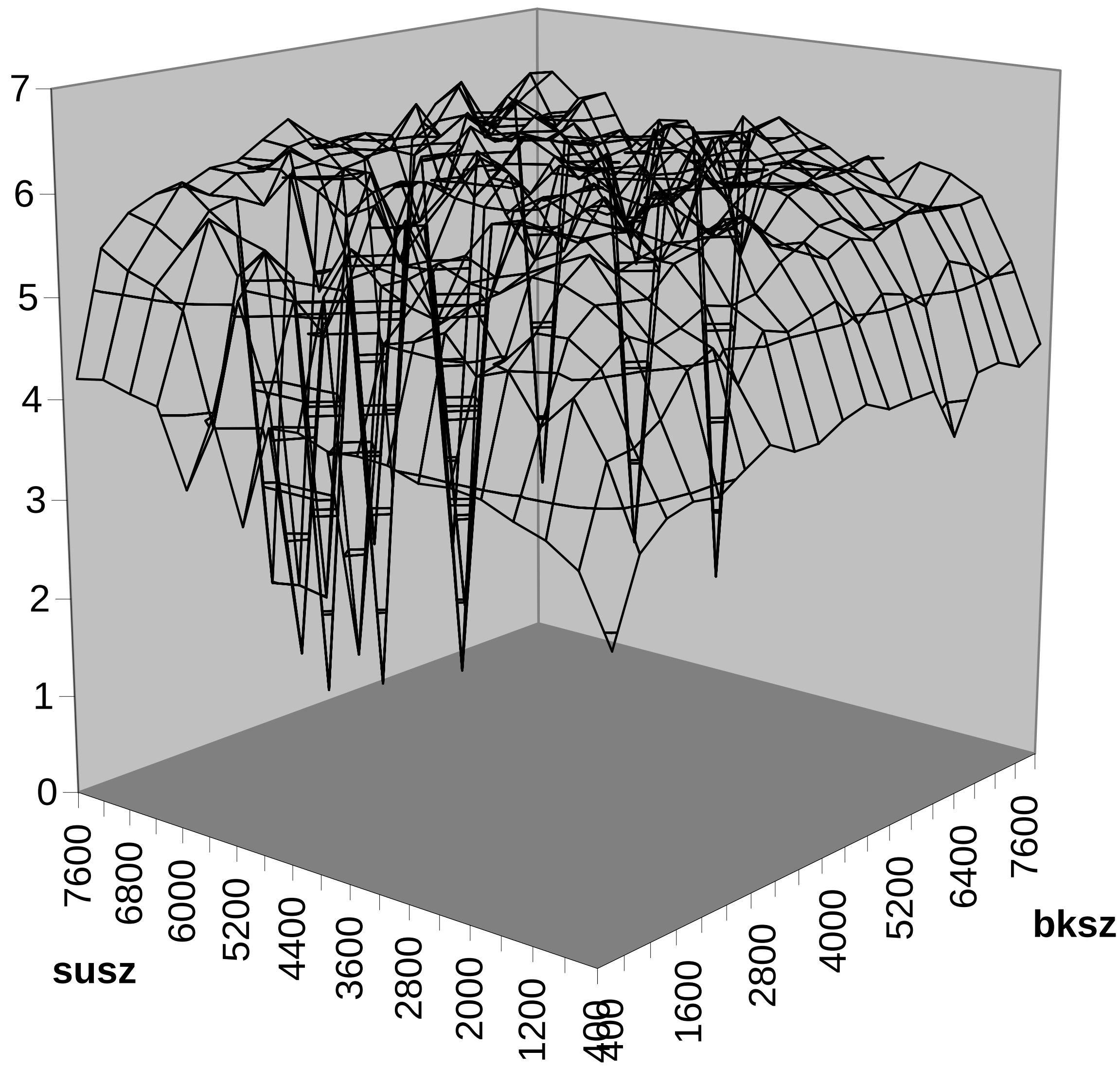

7
6
5
4
3
2
1
0
7600
6800
6000
5200
4400
3600
2800
2000
1200
400
susz
400
1600
2800
4000
5200
6400
7600
bksz

Fig. 23.
wnt

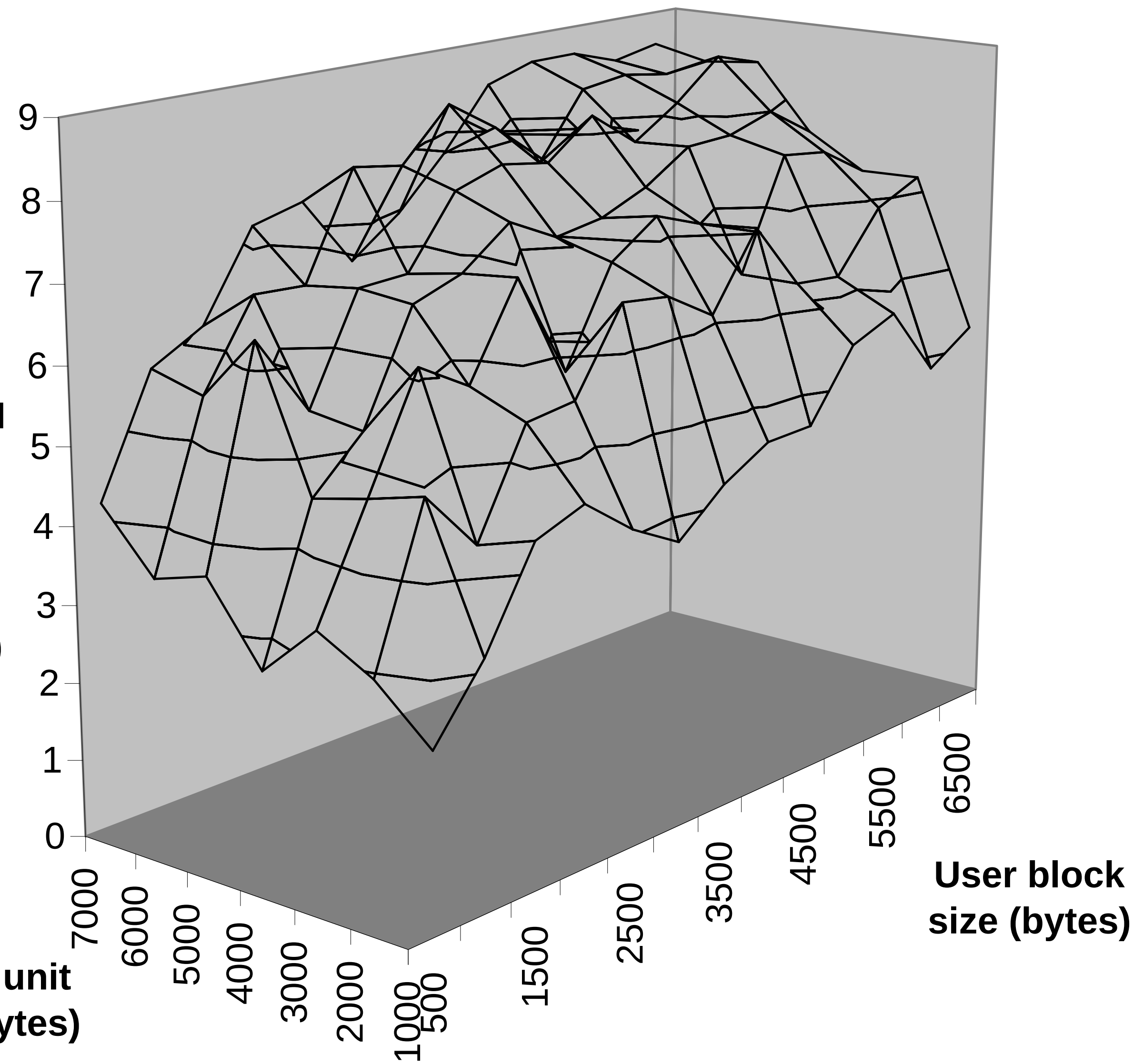


Fig. 24.
Swiss-Tx 100Mbps MPICH 10MB data, average of 11 measur.

speed (mbyte/s)

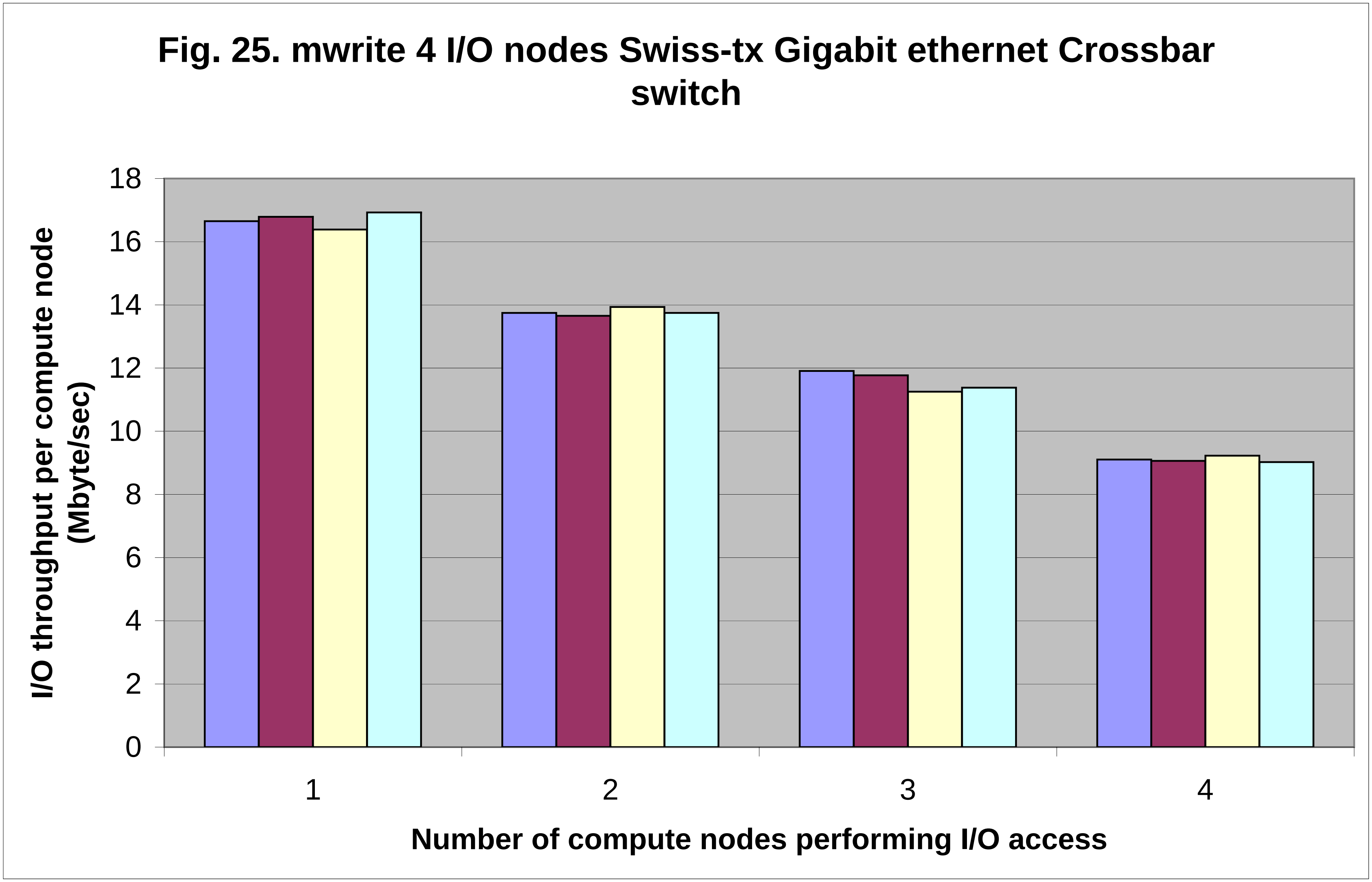


Fig. 25. mwrite 4 I/O nodes Swiss-tx Gigabit ethernet Crossbar switch
I/O throughput per compute node (Mbyte/sec)
18
16
14
12
10
8
6
4
2
0
1
2
3
4
Number of compute nodes performing I/O access

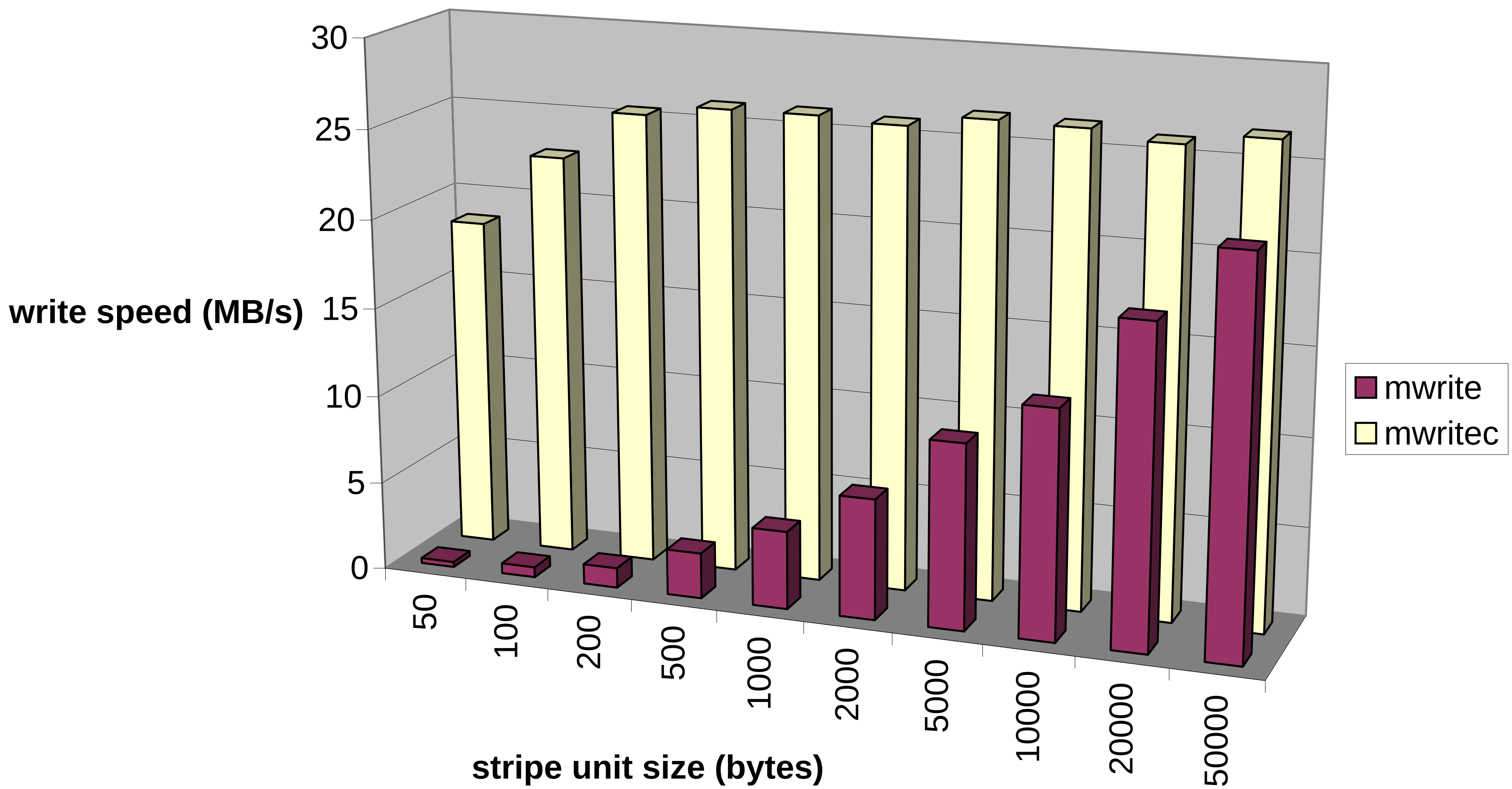


Fig. 26. TNET 3 I/O nodes 1 compute node 660Mbyte

Fig. 27.

Integration into MPI-II I/O

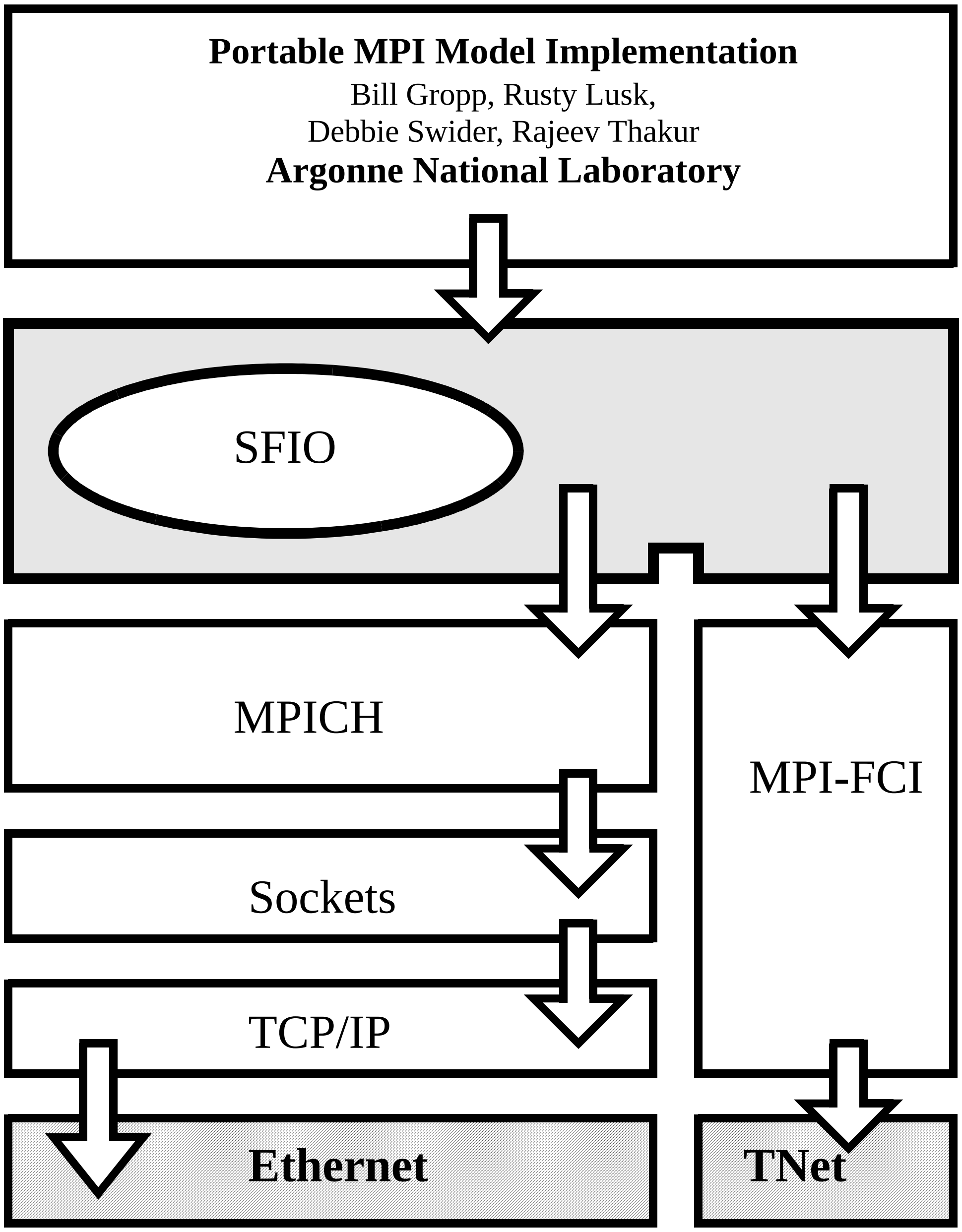


Fig. 28.

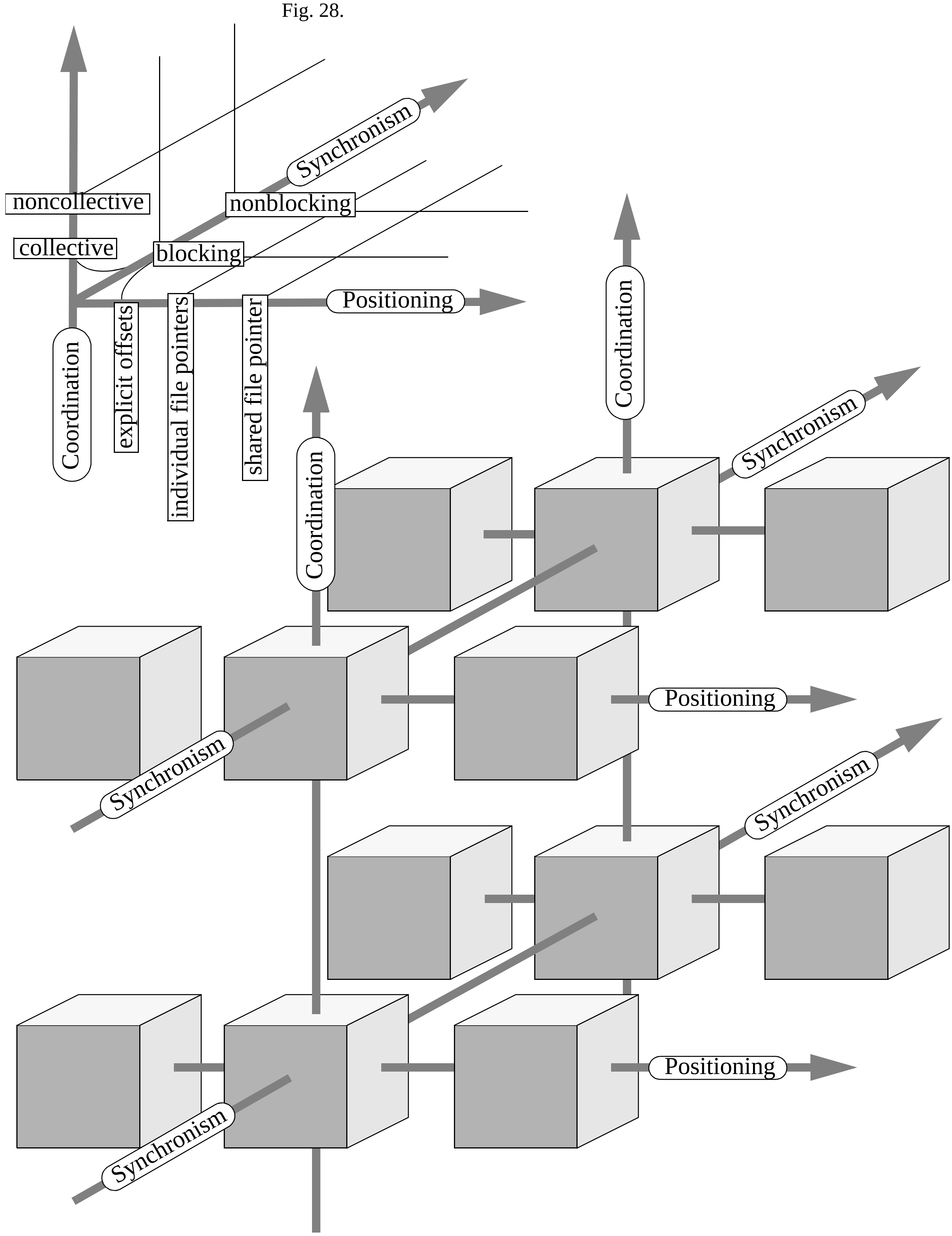

noncollective
collective
nonblocking
blocking
Synchronism
Positioning
Coordination
explicit offsets
individual file pointers
shared file pointer
Coordination
Synchronism
Coordination
Positioning
Synchronism
Synchronism
Positioning
Synchronism

Fig. 29.

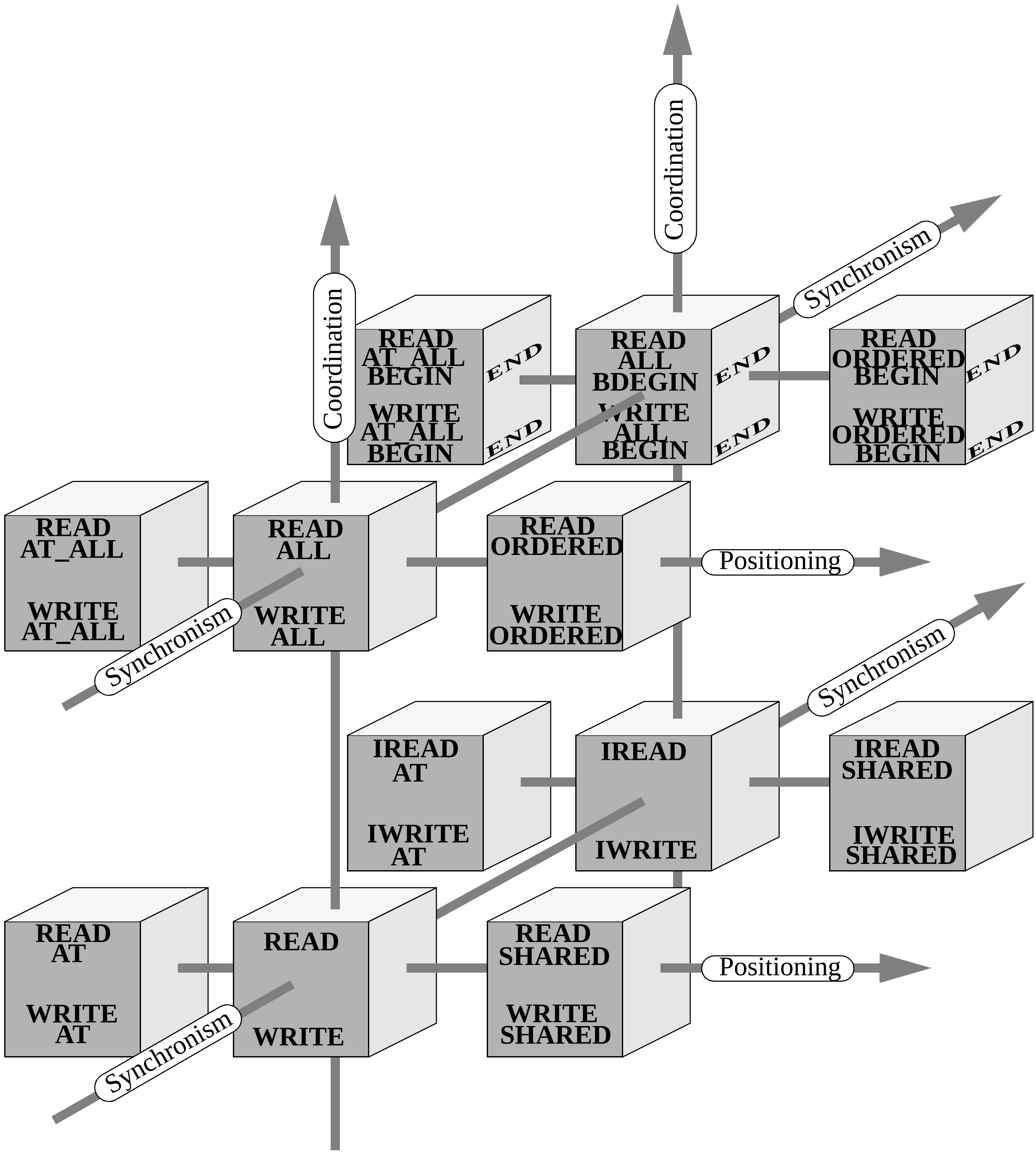
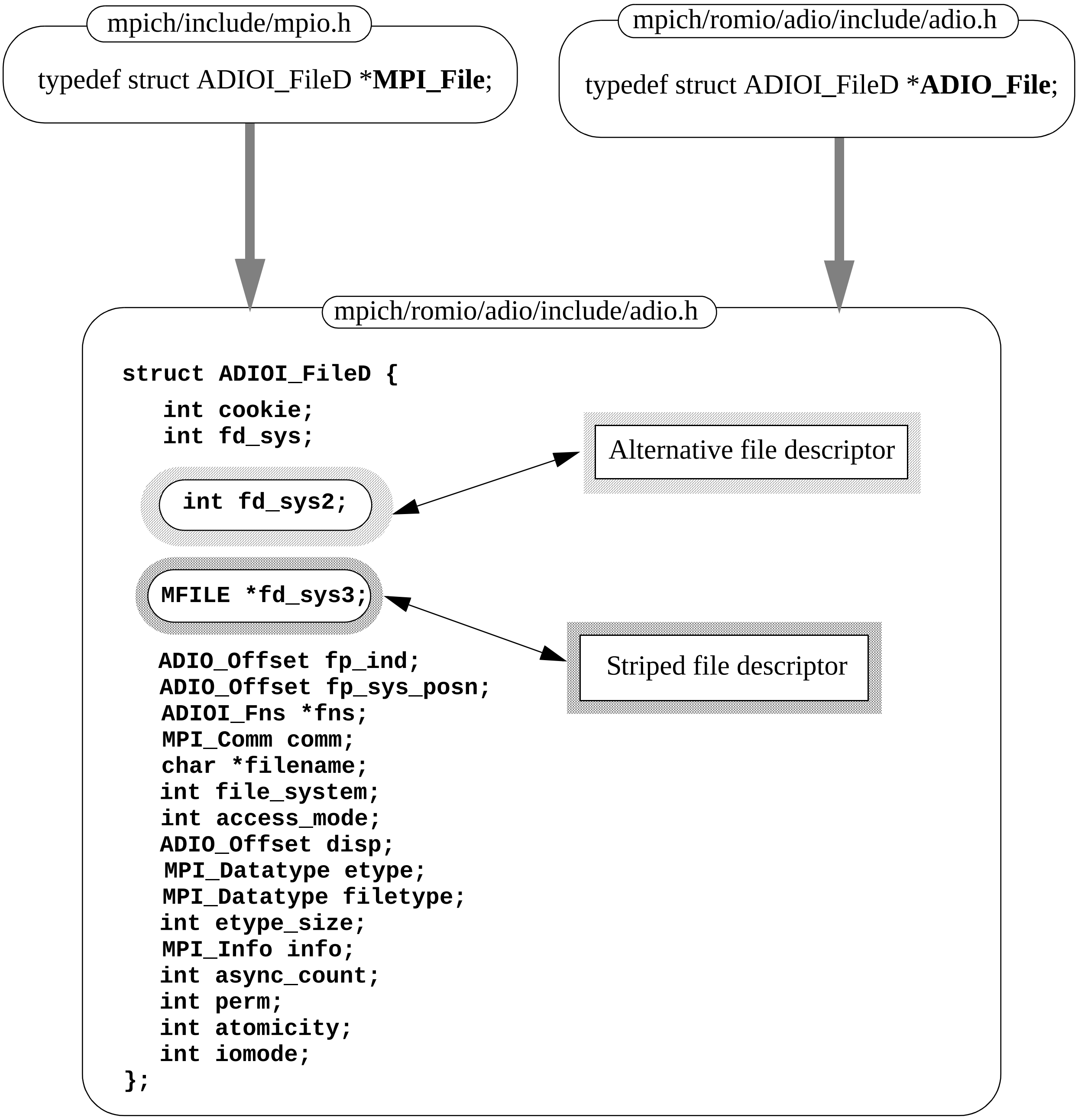

Coordination
Coordination
Synchronism
READ AT_ALL BEGIN
WRITE AT_ALL BEGIN
END
END
READ ALL BDEGIN
WRITE ALL BEGIN
END
END
READ ORDERED BEGIN
WRITE ORDERED BEGIN
END
END
READ AT_ALL
WRITE AT_ALL
READ ALL
WRITE ALL
READ ORDERED
WRITE ORDERED
Positioning
Synchronism
Synchronism
IREAD AT
IWRITE AT
IREAD
IWRITE
IREAD SHARED
IWRITE SHARED
READ AT
WRITE AT
READ
WRITE
READ SHARED
WRITE SHARED
Positioning
Synchronism

Fig. 30.



- We modify MPI_File_open operation
- We modify MPI_File_close operation

Fig. 31.

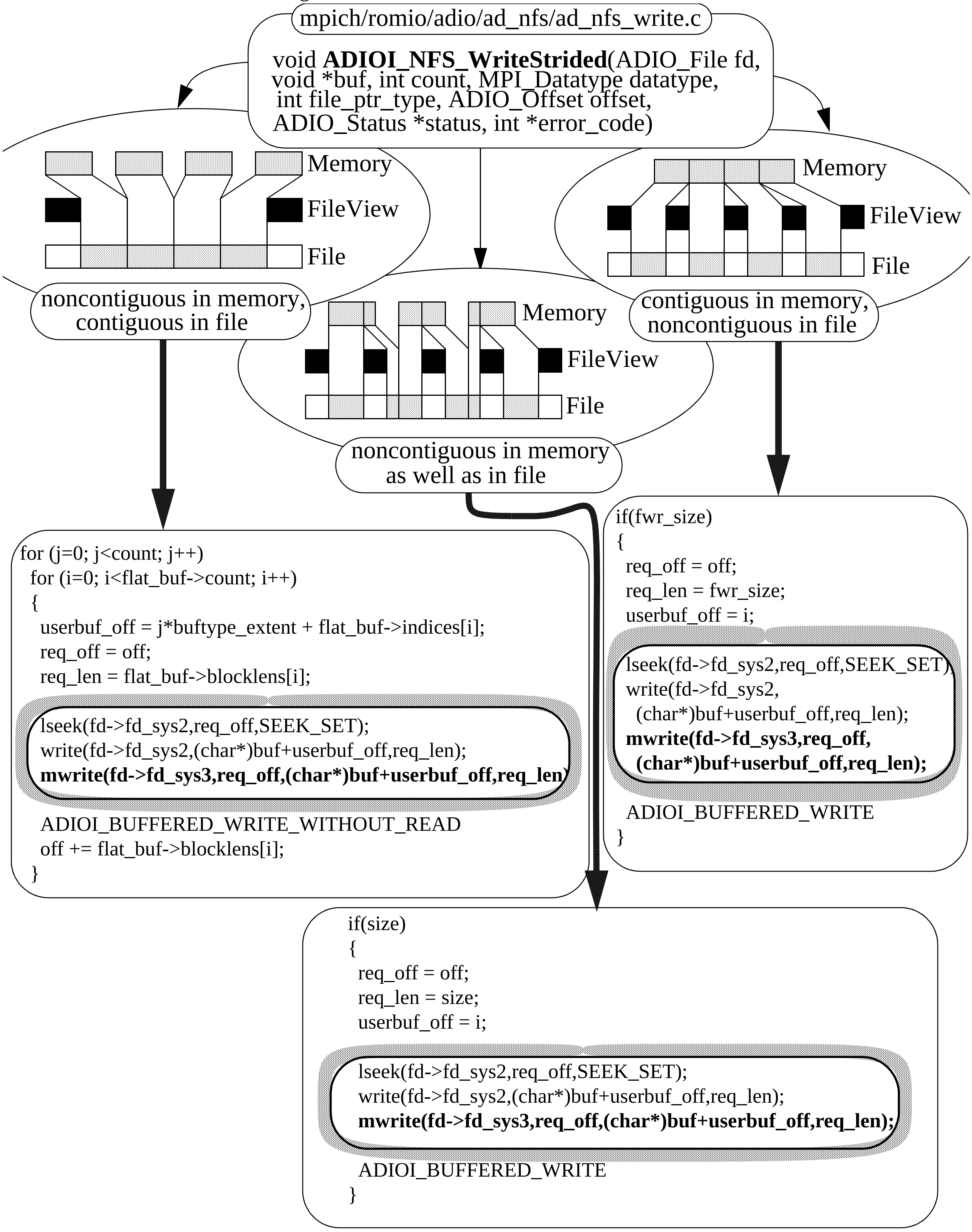

mpich/romio/adio/ad_nfs/ad_nfs_write.c
void **ADIOI_NFS_WriteStrided**(ADIO_File fd,
void *buf, int count, MPI_Datatype datatype,
int file_ptr_type, ADIO_Offset offset,
ADIO_Status *status, int *error_code)
Memory
FileView
File
noncontiguous in memory,
contiguous in file
Memory
FileView
File
noncontiguous in memory
as well as in file
Memory
FileView
File
contiguous in memory,
noncontiguous in file
for (j=0; j<count; j++)
for (i=0; i<flat_buf->count; i++)
{
userbuf_off = j*buftype_extent + flat_buf->indices[i];
req_off = off;
req_len = flat_buf->blocklens[i];
lseek(fd->fd_sys2,req_off,SEEK_SET);
write(fd->fd_sys2,(char*)buf+userbuf_off,req_len);
mwrite(fd->fd_sys3,req_off,(char*)buf+userbuf_off,req_len);
ADIOI_BUFFERED_WRITE_WITHOUT_READ
off += flat_buf->blocklens[i];
}
if(fwr_size)
{
req_off = off;
req_len = fwr_size;
userbuf_off = i;
lseek(fd->fd_sys2,req_off,SEEK_SET);
write(fd->fd_sys2,
(char*)buf+userbuf_off,req_len);
mwrite(fd->fd_sys3,req_off,
(char*)buf+userbuf_off,req_len);
ADIOI_BUFFERED_WRITE
}
if(size)
{
req_off = off;
req_len = size;
userbuf_off = i;
lseek(fd->fd_sys2,req_off,SEEK_SET);
write(fd->fd_sys2,(char*)buf+userbuf_off,req_len);
mwrite(fd->fd_sys3,req_off,(char*)buf+userbuf_off,req_len);
ADIOI_BUFFERED_WRITE
}

Fig. 32.

Conclusion

- This is a cheap way of obtaining high performance I/O

Fig. 33.

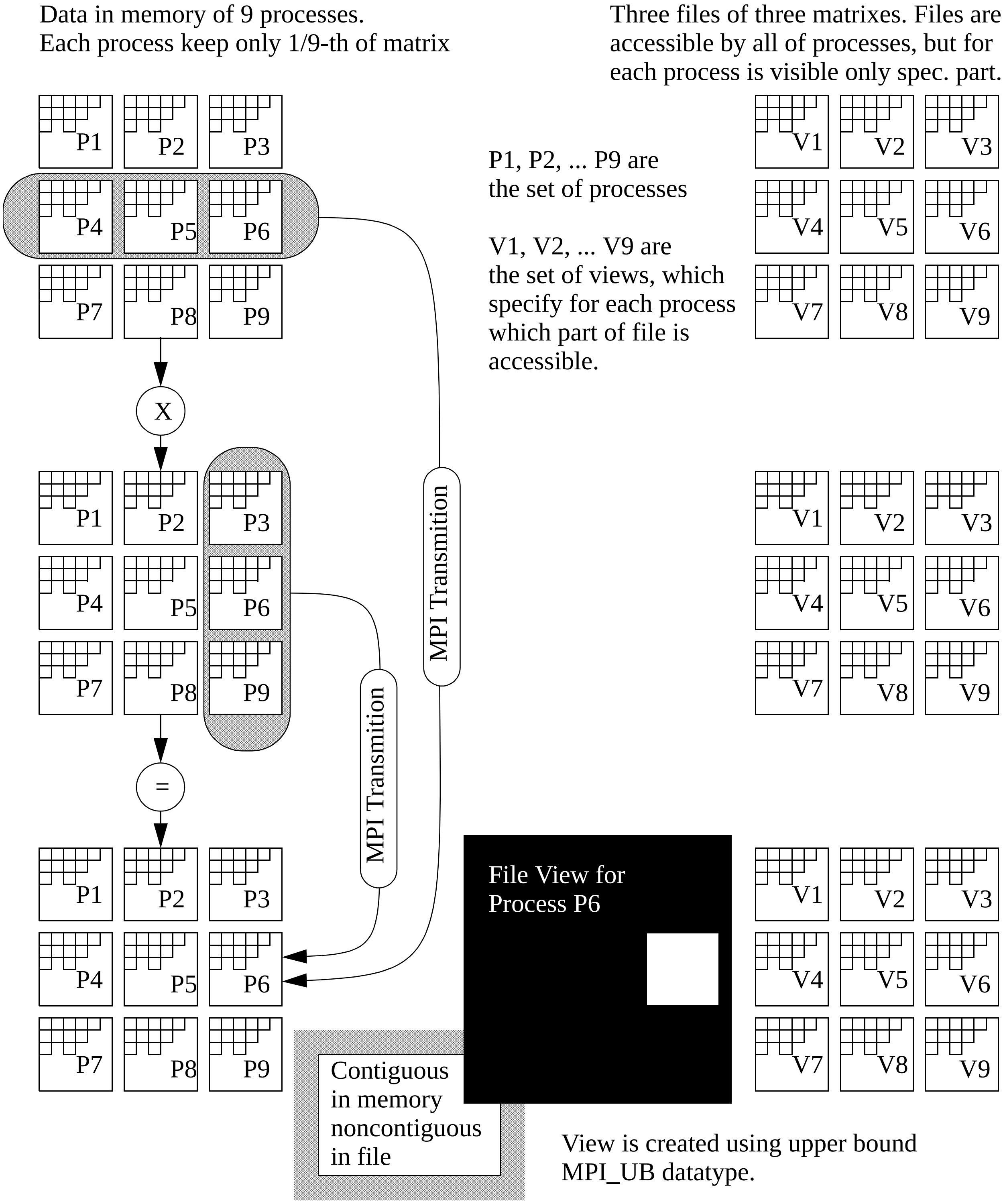
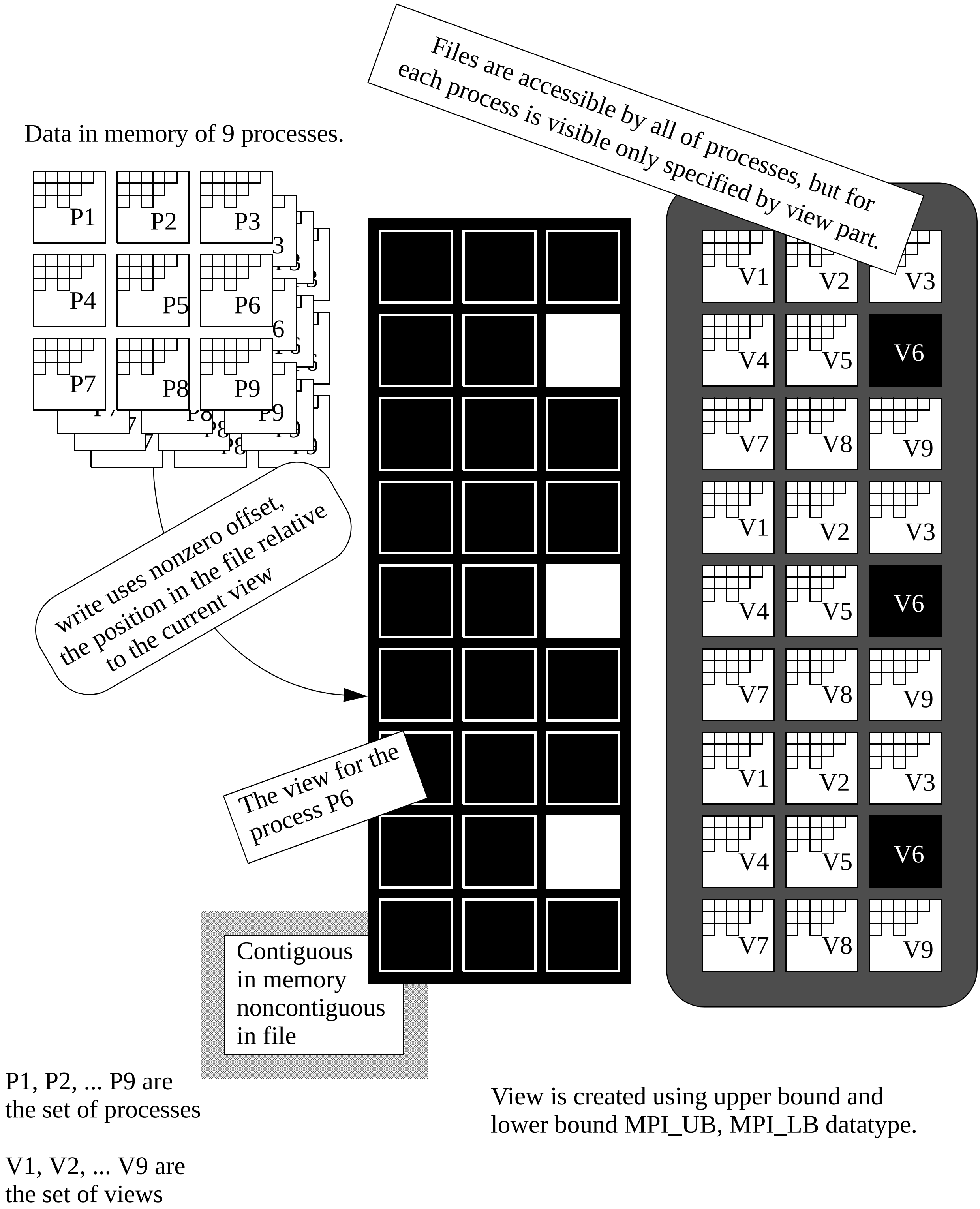

Data in memory of 9 processes.
Each process keep only 1/9-th of matrix
Three files of three matrixes. Files are accessible by all of processes, but for each process is visible only spec. part.
P1, P2, ... P9 are the set of processes
V1, V2, ... V9 are the set of views, which specify for each process which part of file is accessible.
P1
P2
P3
P4
P5
P6
P7
P8
P9
V1
V2
V3
V4
V5
V6
V7
V8
V9
X
=
MPI Transmition
MPI Transmition
File View for Process P6
Contiguous in memory noncontiguous in file
View is created using upper bound MPI_UB datatype.

Fig. 34.


Data in memory of 9 processes.
P1
P2
P3
P4
P5
P6
P7
P8
P9
Files are accessible by all of processes, but for each process is visible only specified by view part.
V1
V2
V3
V4
V5
V6
V7
V8
V9
write uses nonzero offset, the position in the file relative to the current view
The view for the process P6
Contiguous in memory noncontiguous in file
P1, P2, ... P9 are the set of processes
V1, V2, ... V9 are the set of views
View is created using upper bound and lower bound MPI_UB, MPI_LB datatype.