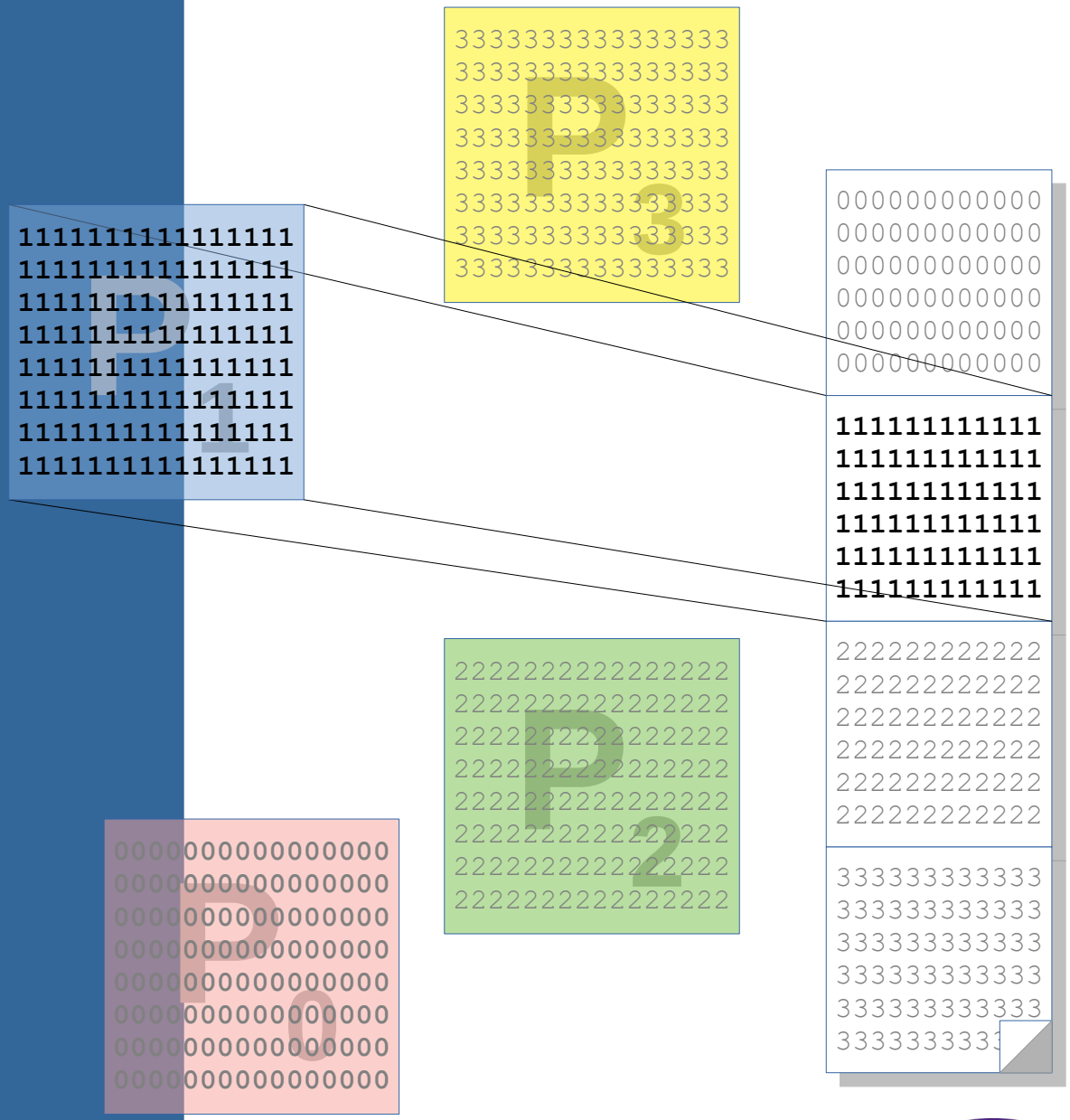
Concurrent I/O by Multiple Processes – Part I

September 12, 2018

Ge Baolai, *Western University*
SHARCNET | Compute Ontario

Outline

- Issues to address in parallel computing
- Reading data chunks from a single file concurrently
- Writing data chunks to a single file concurrently
- Parallel I/O with MPI
- Q&A



SHARCNET™

Western

Issues to Address in Parallel Computing

Consider the scenarios in parallel computing:

- One needs to read a large file.
- Or writes computed data to a file.

Usual workflows look like this:

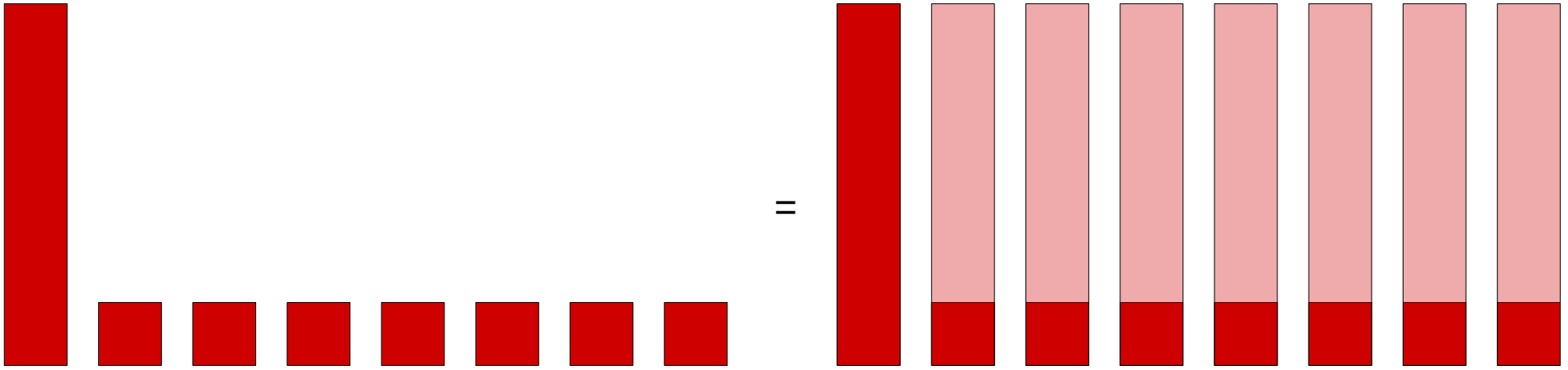
- One process, e.g. rank 0, reads the file;
- Rank 0 distributes data to other processes;
- Everyone gets work done;
- Rank 0 collects the results, assembles received data into the global storage;
- Rank 0 then writes the data to a file.

The problem with this workflow:

- One process, e.g. rank 0, reads the file. *When the data size is HUGE, this process needs **HUGE memory** to store the data, which might not be available;*
- Rank 0 distributes data to other processes. *This yields a cost of communication;*
- Everyone gets work done;
- Rank 0 collects the results, assembles received data into the global storage. *Again, the assembly needs HUGE memory;*
- Rank 0 then writes the data to a file. This may take quite some time.

Another issue with such unbalanced memory usage

- The job shall request for per core the max memory needed by the I/O process.
- This may result in longer wait time and cost more to run – on graham and cedar, the cost is based on the number of cores used and the amount of memory per core – max RAM requested in this case – consumed



Parallel I/O addresses theses issues with the following benefits:

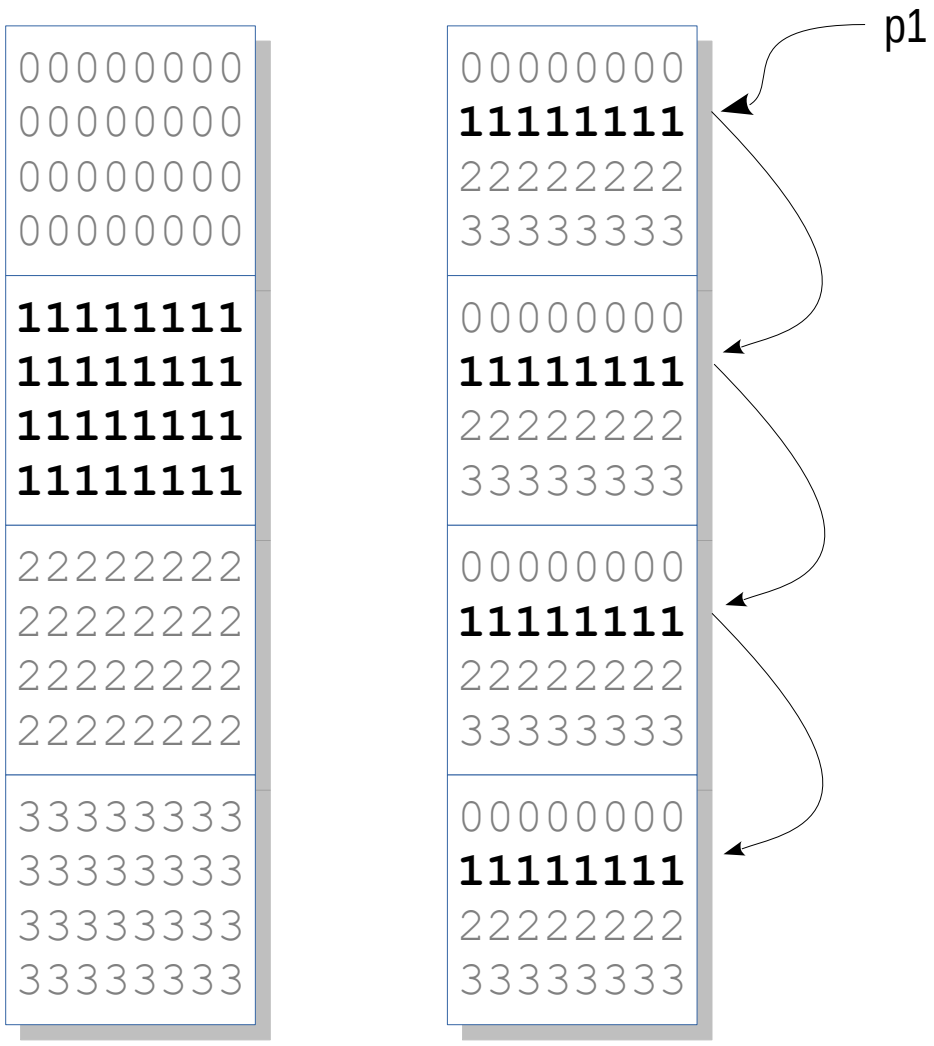
- Each process reads a chunk of the file. *Less memory is required by rank 0;*
- *The cost of communication to distribute data is avoided;*
- Everyone gets work done;
- *Collecting and assembling data is no longer required, again less memory is required by rank 0;*
- Each process writes its own data to a file*, so less amount of data to write, hence takes less time.*

Reading Data from A Single File by Multiple Processes – General View

Task: Have multiple processes read chunks of data from the same file

- Each process reads chunks from predefined locations.
- The last process might need to read a bit more when the total number of records can not be evenly divided.

A binary file containing “records” of same length



Scenario A

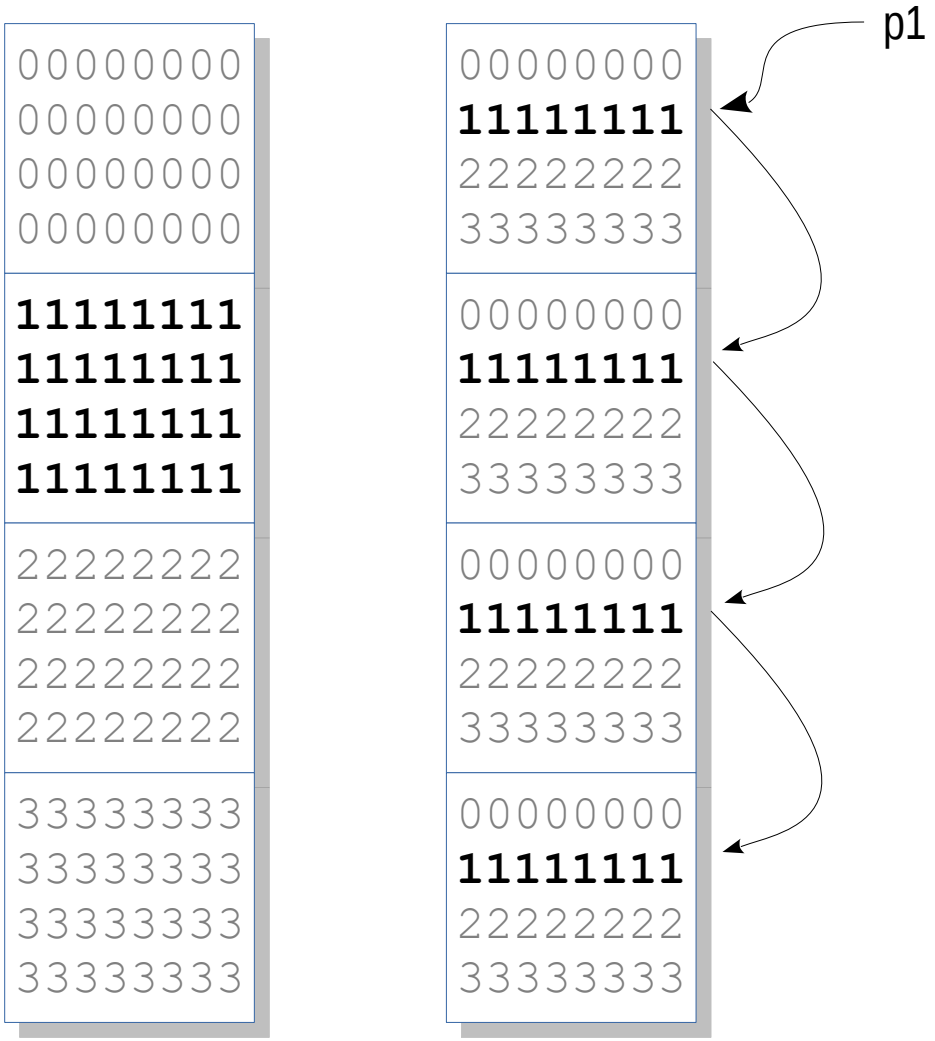
Scenario B

Concurrent reads

We need

- Each process opens the file.
- Each can seek the position to read the data off.
- Each reads, completes and closes the file independent of others.

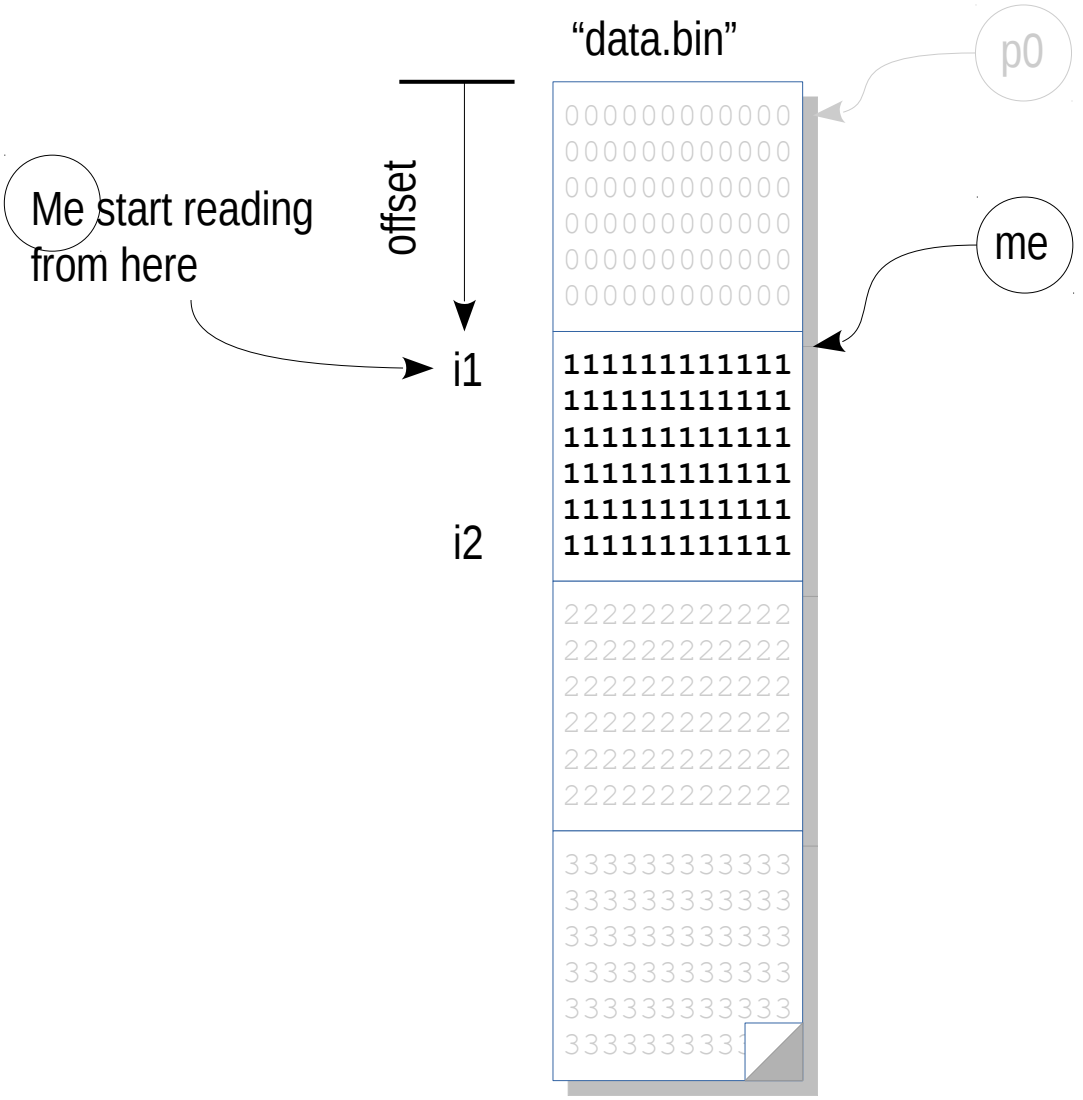
A binary file containing “records” of same length



Example: Multiple processes read data chunks from a data file

```
// Calculate the chunk size
chunk = n / num_procs;
i1 = (myid-1)*chunk;
if (myid == num_procs)
    chunk += n % num_procs;
i2 = i1 + chunk;

// Read my chunk of elements
fp = fopen("data.bin", "rb");
for (i = i1; i < i2; i++)
{
    offset = i*sizeof(float);
    fseek(fp,offset,SEEK_SET);
    fread(&v,sizeof(float),1,fp);
}
fclose(fp);
```



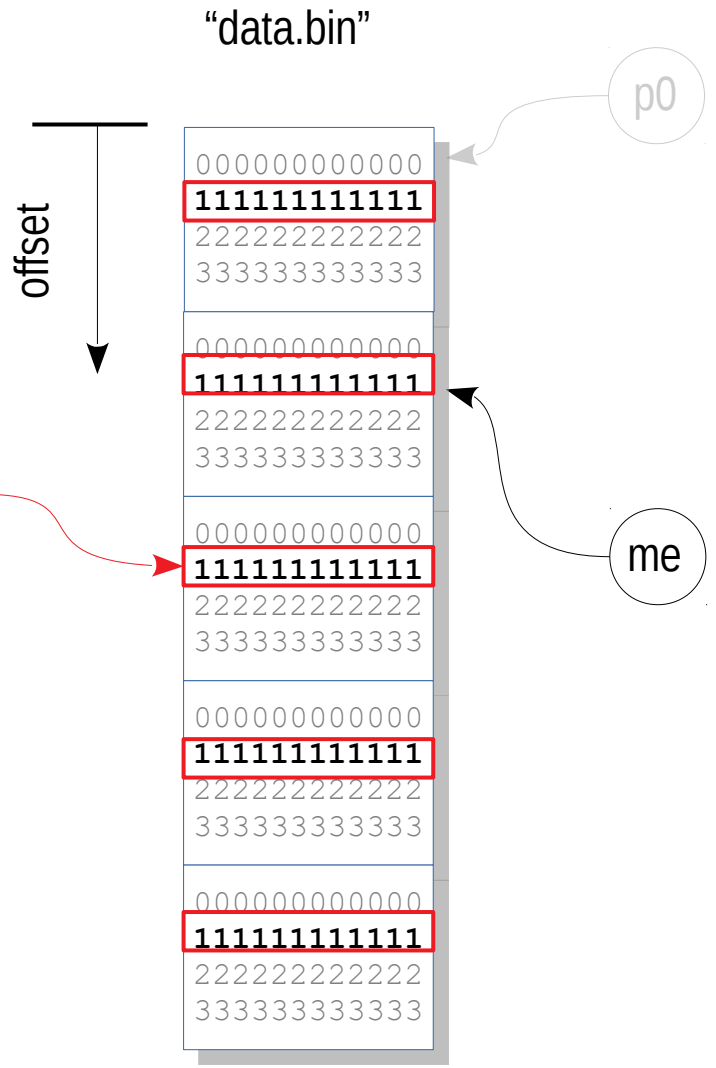
The total size of data, e.g. *n* – in terms of bytes – is Is known in advance.

Example: Multiple processes read data chunks from a data file (cont'd)

```
fp = fopen("data.bin", "rb");

for (... ...) // Loop over interlaced chunks
{
    ... ...
    i1 = ...
    i2 = ...
    ... ...
    for (i = i1; i < i2; i++) // Loop over records within a chunk
    {
        offset = i*sizeof(float);
        fseek(fp,offset,SEEK_SET);
        fread(&v,sizeof(float),1,fp);
    }
}

fclose(fp);
```



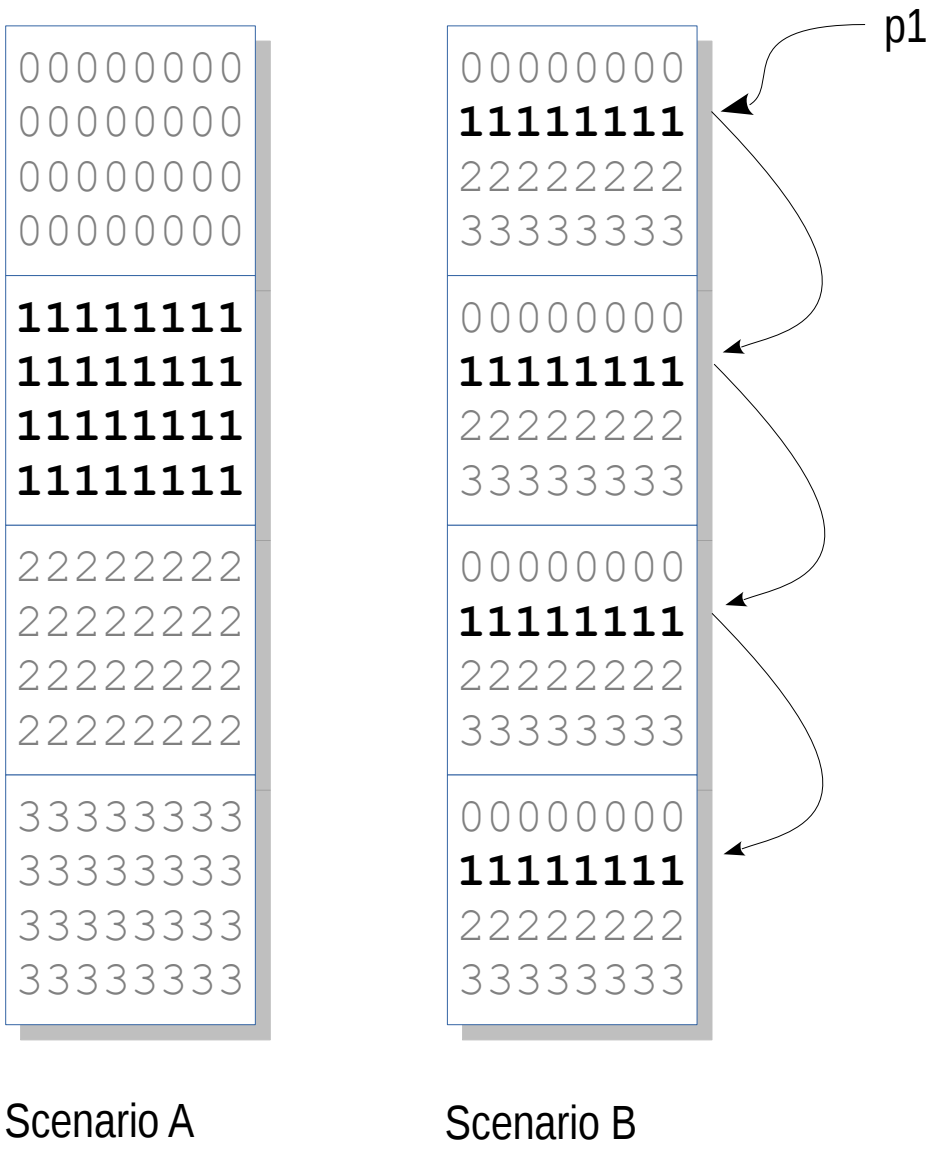
Chunks interlaced.

Writing Data to A Single File by Multiple Processes – General View

Task: Have multiple processes write chunks of data to the same file

- One process creates a file
- Each process opens the same file and writes chunks to predefined locations.
- There are no overlaps.
- Each process closes the file.

A binary file to be written by multiple processes

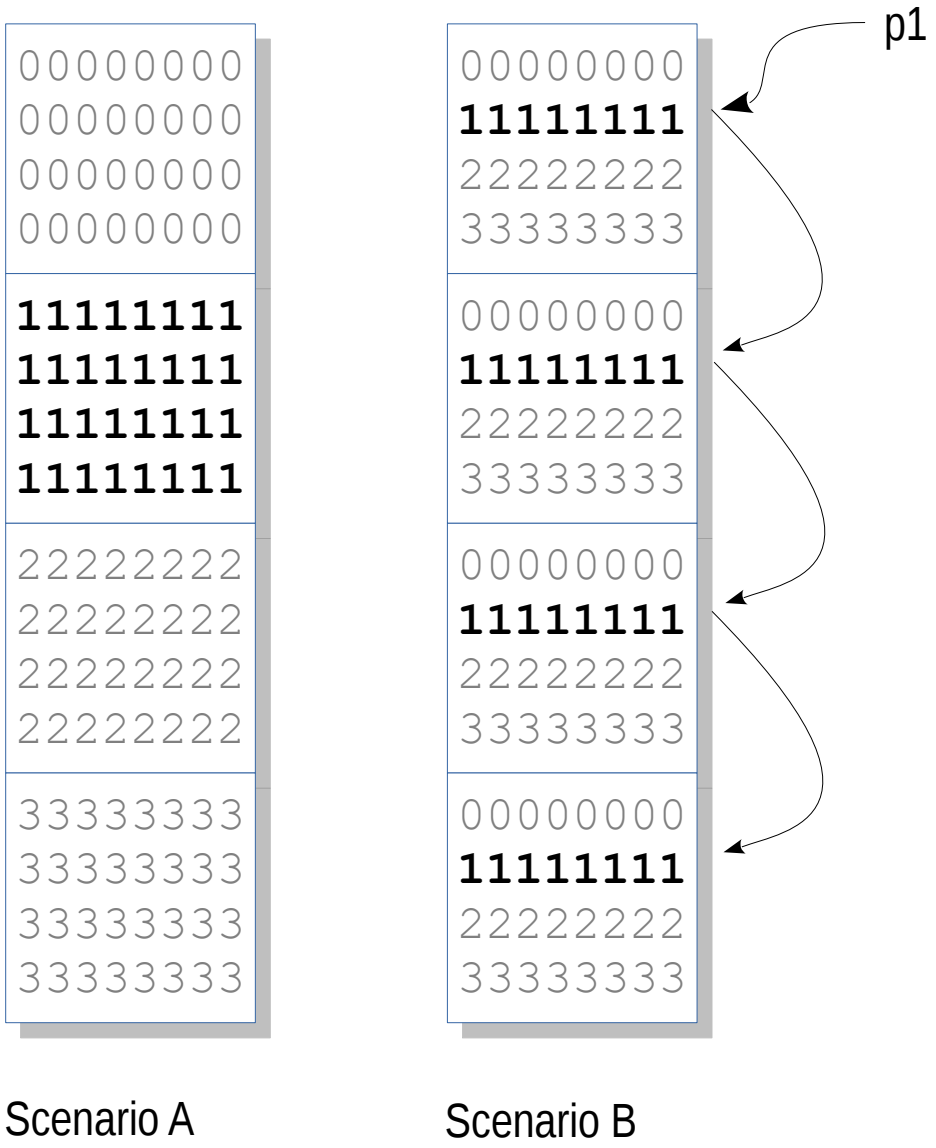


Concurrent writes

We need

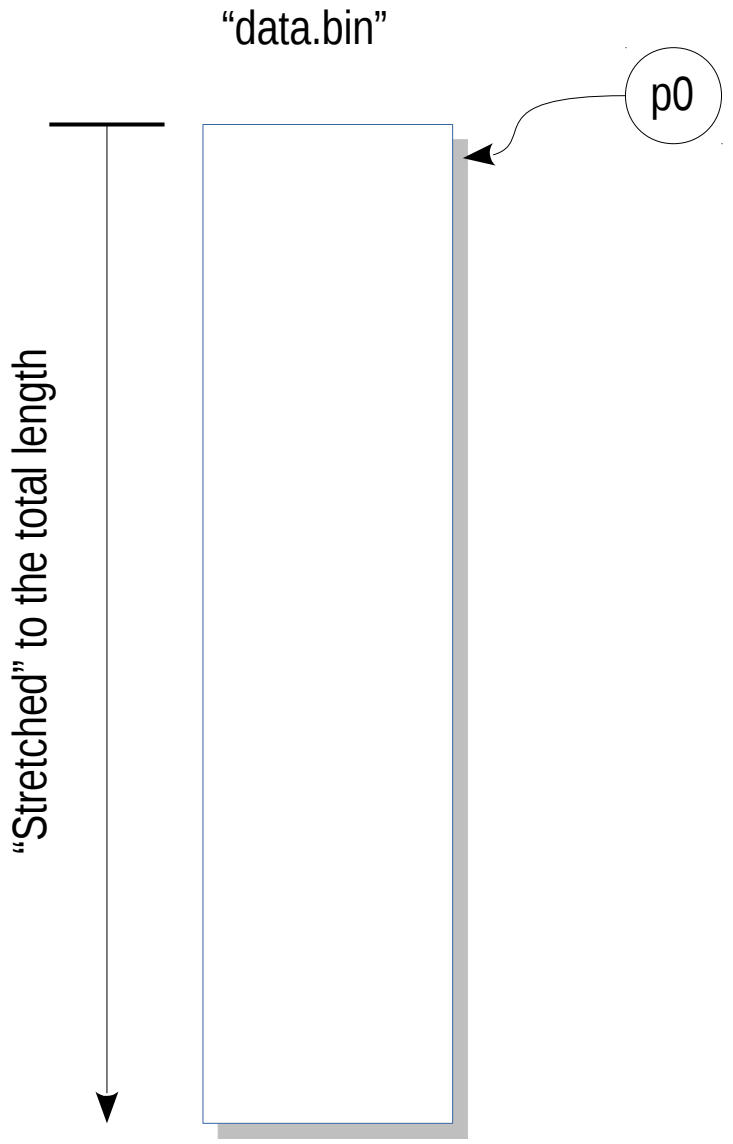
- One process creates a file
- It stretches the file to the desired total length
- Each process opens the same file and writes chunks to predefined locations.
- Each process completes the writes and closes the file independent of others.

A binary file to be written by multiple processes



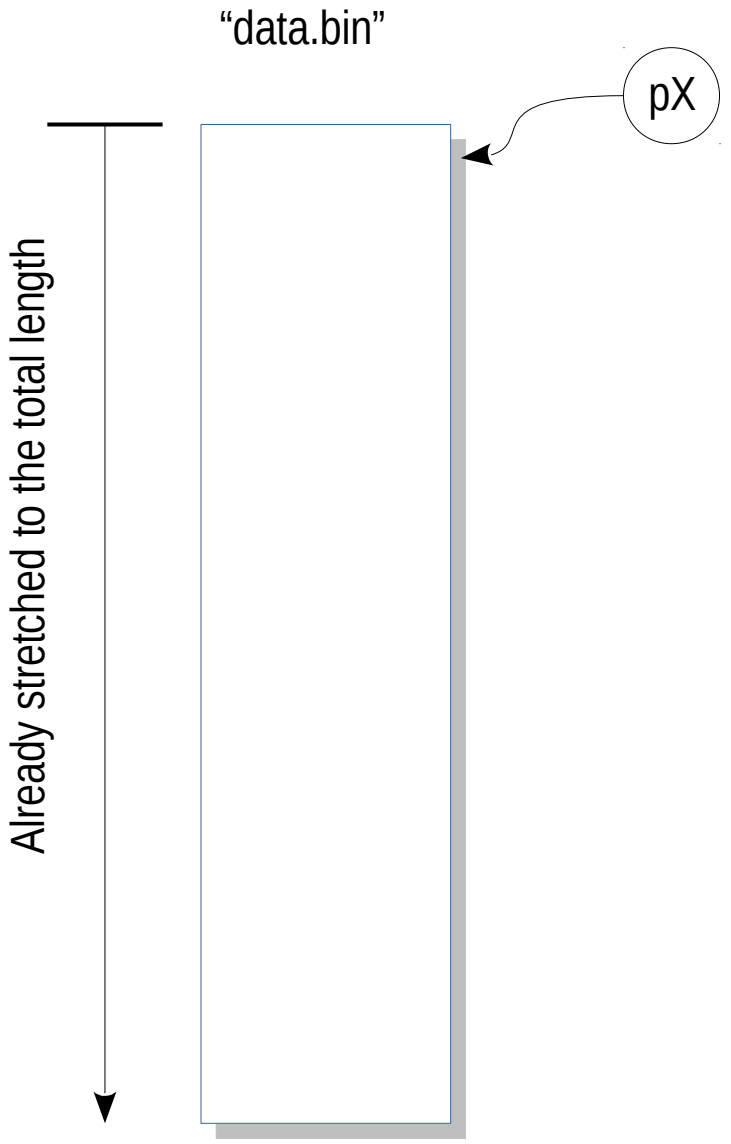
Example: Multiple processes write data chunks to a data file

```
if (myid == 0) {
    fd = open("data.bin", O_CREAT|O_TRUNC|O_RDWR|O_EXCL, 0644);
    ftruncate(fd, sizeof(float)*n); // Stretch it to the total length
    fp = fdopen(fd, "w"); // Associated the fd to file handler
}
else {
    do {
        if (-1 == access("data.bin", F_OK)) { // Check if file exists
            if (ENOENT == errno) {
                printf("Waiting on shared file to be created...\n");
                sleep(2);
            }
            else {
                perror("access");
                exit(EXIT_FAILURE);
            }
        }
        else { // The file exists, open it for writing
            fp = fopen("data.bin","r+b");
            found = 1;
        }
    } while (!found);
    printf("Process %d detected file exists\n", myid);
}
```



Example: Multiple processes write data chunks to a data file

```
if (myid == 0) {
    fd = open("data.bin", O_CREAT|O_TRUNC|O_RDWR|O_EXCL, 0644);
    ftruncate(fd, sizeof(float)*n); // Stretch it to the total length
    fp = fdopen(fd, "w"); // Associated the fd to file handler
}
else {
    do {
        if (-1 == access("data.bin", F_OK)) { // Check if file exists
            if (ENOENT == errno) {
                printf("Waiting on shared file to be created...\n");
                sleep(2);
            }
            else {
                perror("access");
                exit(EXIT_FAILURE);
            }
        }
        else { // The file exists, open it for writing
            fp = fopen("data.bin","r+b");
            found = 1;
        }
    } while (!found);
    printf("Process %d detected file exists\n", myid);
}
```



Example: Multiple processes write data chunks to a data file (cont'd)

```
if (myid == 0) {
    fd = open("data.bin", O_CREAT|O_TRUNC|O_RDWR|O_EXCL, 0644);
    ftruncate(fd, sizeof(float)*n); // Stretch it to the total length
    fp = fdopen(fd, "w"); // Associated the fd to file handler
}
else {
    do {
        if (-1 == access("data.bin", F_OK)) { // Check if file exists
            if (ENOENT == errno) {
                printf("Waiting on shared file to be created...\n");
                sleep(2);
            }
            else {
                perror("access");
                exit(EXIT_FAILURE);
            }
        }
        else { // The file exists, open it for writing
            fp = fopen("data.bin","r+b");
            found = 1;
        }
    } while (!found);
    printf("Process %d detected file exists\n", myid);
}
```

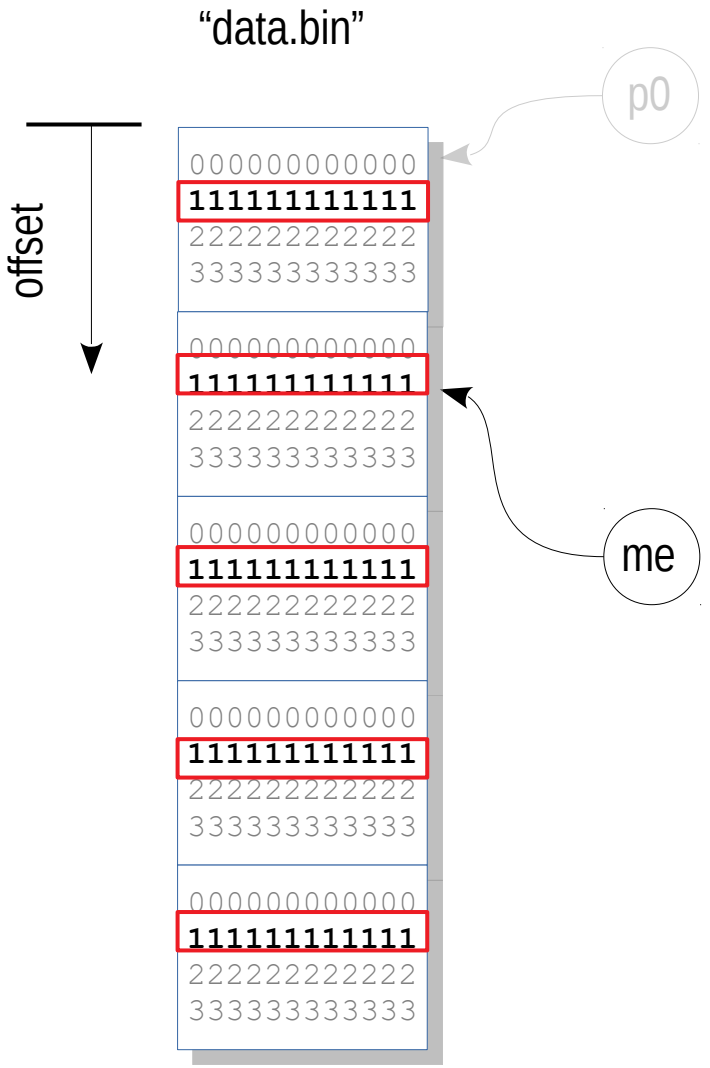
Script launching processes in parallel

```
#!/bin/bash
rm -f mrecs.bin
n=4
nap=0
pids=`shuf -i 1-$n`
for i in $pids
do
    echo "Starting process $i in $nap seconds..."
    sleep $nap
    ./bin_pwrite_c $i $n &
    nap=$((RANDOM % 5))
done
wait
```

Example: Multiple processes write data chunks to a data file (cont'd)

```
for (... ...)
{
    ... ...
    i1 = ...
    i2 = ...
    ... ...
    for (i = i1; i < i2; i++)
    {
        ...
        offset = i*sizeof(float);
        fseek(fp,offset,SEEK_SET);
        fwrite(&v,sizeof(float),1,fp);
    }
}

fclose(fp);
```



Parallel I/O with MPI

MPI has a set of functions that facilitate what we want

- Each process reads a chunk of the same file.
- Each process writes a chunk of global data into a file in desired order.

Example: “mpi_pio_hello.c” - “Hello world!” written to a file by all ranks

```
...
// Prepre the message for this process
sprintf(msg, "Hello world from rank %2d\n", rank);
msgsize = strlen(msg);

// Calculate the offset from where the message is written
offset = msgsize*rank;


// Every process opens the same file and writes, then closes it
MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
   MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, offset, MPI_SEEK_SET);
MPI_File_write(fh, msg, msgsize, MPI_CHAR, &status);
MPI_File_close(&fh);
...
```

Example: "mpi_pio_hello_write.c" (cont'd)

```
int MPI_File_open(
    MPI_Comm comm,      // MPI communicator
    const char *filename, // File name
    int amode,          // File access mode
    MPI_Info info,      // Default to MPI_INFO_NULL. See man page
    MPI_File *fh        // On success, the file handle
)
```

Access modes:

- **MPI_MODE_CREATE, MPI_MODE_APPEND, MPI_MODE_DELETE_ON_CLOSE**
- **MPI_MODE_APPEND**
- **MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_RDWR**
- **MPI_MODE_SEQUENTIAL**
- **MPI_MODE_EXCL, MPI_MODE_UNIQUE_OPEN**

Example: “mpi_pio_hello_write.c” (cont’d)

```
int open(
    const char *filename,  // File name
    int oflag,             // Flag for file access mode. See man page
    ...
)
```

Access modes:

- **O_CREATE**
- **O_APPEND**
- **O_RDONLY, O_WRONLY**
- **O_RDWR**
- **O_EXCL**
- **... ...**

Example: “mpi_pio_hello_write.c” (cont’d)

```
int MPI_File_seek(
    MPI_File fh,          // File handle
    MPI_Offset offset,    // Offset
    int whence            // Default tp MPI_SEEK_SET. See man page
)
```

Example: “mpi_pio_hello_write.c” (cont’d)

```
int MPI_File_write(
    MPI_File fh,                // File handle
    const void *buf,            // Buffer holding the data
    int count,                  // Number of elements in the buffer
    MPI_Datatype datatype,      // Data type
    MPI_Status *status          // Status
)
```

Example: “mpi_pio_hello_write.c” (cont’d)

```
int MPI_File_close(
    MPI_File *fh              // File handle
)
```

Example: “mpi_pio_hello_read.c” - Read from file. We have almost identical source code, except that MPI_File_write() is replaced by MPI_File_read(), and the file open mode in the call to MPI_File_open() is slightly different:

```
… …
// Every process opens file and writes, then closes it
MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
   MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, offset, MPI_SEEK_SET);
MPI_File_read(fh, msg, msgsize, MPI_CHAR, &status);
MPI_File_close(&fh);
… …
```

Example: Writing an 8x8 array distributed on 4 processes to a file

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	53	53	54	55	56
57	58	59	60	61	62	63	64

Layout in local memory (if stored in *column major*)

```
 1  9 17 25  2 10 18 26  3 11 19 27  4 12 20 28
33 41 49 57 34 42 50 58 35 43 51 59 36 44 53 60
 5 13 21 29  6 14 22 30  7 15 23 31  8 16 24 32
37 45 53 61 38 46 54 62 39 47 55 63 40 48 56 64
```

When writing local data to the file, we need to map the local array to chunks globally in the file.

If in global memory, the layout would look this (if stored in *column major*)

```
1 9 17 25 33 41 49 57 2 10 18 26 34 42 50 58 … … 8 16 24 32 40 48 56 64]
```

Example: Writing an m-by-n distributed array to a file (cont'd)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    … …
    MPI_File   file;
    MPI_Status status;
    MPI_Datatype viewtype;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Example: Writing an m-by-n distributed array to a file (cont'd)

```
// Calculate the local dimensions
col_rank = rank % gridsize[1]; row_rank = rank / gridsize[1];

locncols = gsize[1] / gridsize[1];
startc = col_rank * locncols;
endc   = startc + locncols - 1;
if (mycol == gridsize[1] - 1) {
    endc = datasize; locncols = endc - startc + 1;
}

locnrows = gsize[0] / gridsize[0];
startr = row_rank * locnrows;
endr   = startr + locnrows - 1;
if (myrow == gridsize[0] - 1) {
    endr = datasize - 1; locnrows = endr - startr + 1;
}
```

Example: Writing an m-by-n distributed array to a file (cont'd)

```
// Set the values of local array
lsize[0] = locnrows;
lsize[1] = locncols;
start_indices[0] = startr;
start_indices[1] = startc;

// Create a "view" of subarray (see man pages)
MPI_Type_create_subarray(2, gsize, lsize, start_indices, MPI_ORDER_C,
    MPI_FLOAT, &viewtype);
MPI_Type_commit(&viewtype);
```

Example: Writing an m-by-n distributed array to a file (cont'd)

```
    // Open the file to be shared by MPI processes (see man pages)
    MPI_File_open(MPI_COMM_WORLD, "a.txt",
        MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, 0,  MPI_FLOAT, viewtype, "native", MPI_INFO_NULL);
    MPI_File_write_all(fh, &(mydata[0][0]), locnrows*locncols, MPI_FLOAT, &status);
    MPI_File_close(&fh);

    MPI_Type_free(&viewtype);
    … …

    MPI_Finalize();
    return 0;
}
```

References

[1] William Gropp, Ewing Lusk, Anthony Skjellum, "***Using MPI***", The MIT Press, 1999.

[2] William Gropp, Ewing Lusk, Rajeev Thakur, "***Using MPI-2***", The MIT Press, 1999.

[3] OpenMPI, https://www.open-mpi.org/

[4] SHARNET general interest seminars on MPI, Youtube, SHARNCET channel.