Introduction to MPI I/O

Stefano Markidis

KTH Royal Institute of Technology

What does Parallel I/O Mean?

- **At the program level:**
  - Concurrent reads or writes from multiple processes to a common file
- **At the system level:**
  - A parallel file system and hardware that support such concurrent access

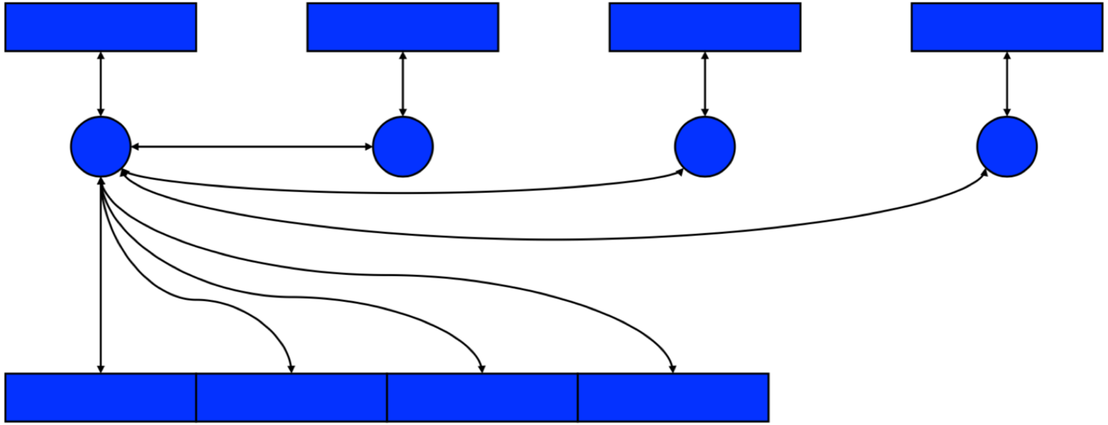
Parallel I/O in MPI

- Why do I/O in MPI? Why not just POSIX?
  - Parallel performance
  - Single file (instead of one file / process)
- MPI has replacement functions for POSIX I/O
  - Provides migration path
- Multiple styles of I/O can all be expressed in MPI
  - Including some that cannot be expressed without MPI

Why MPI is a Good Setting for Parallel I/O

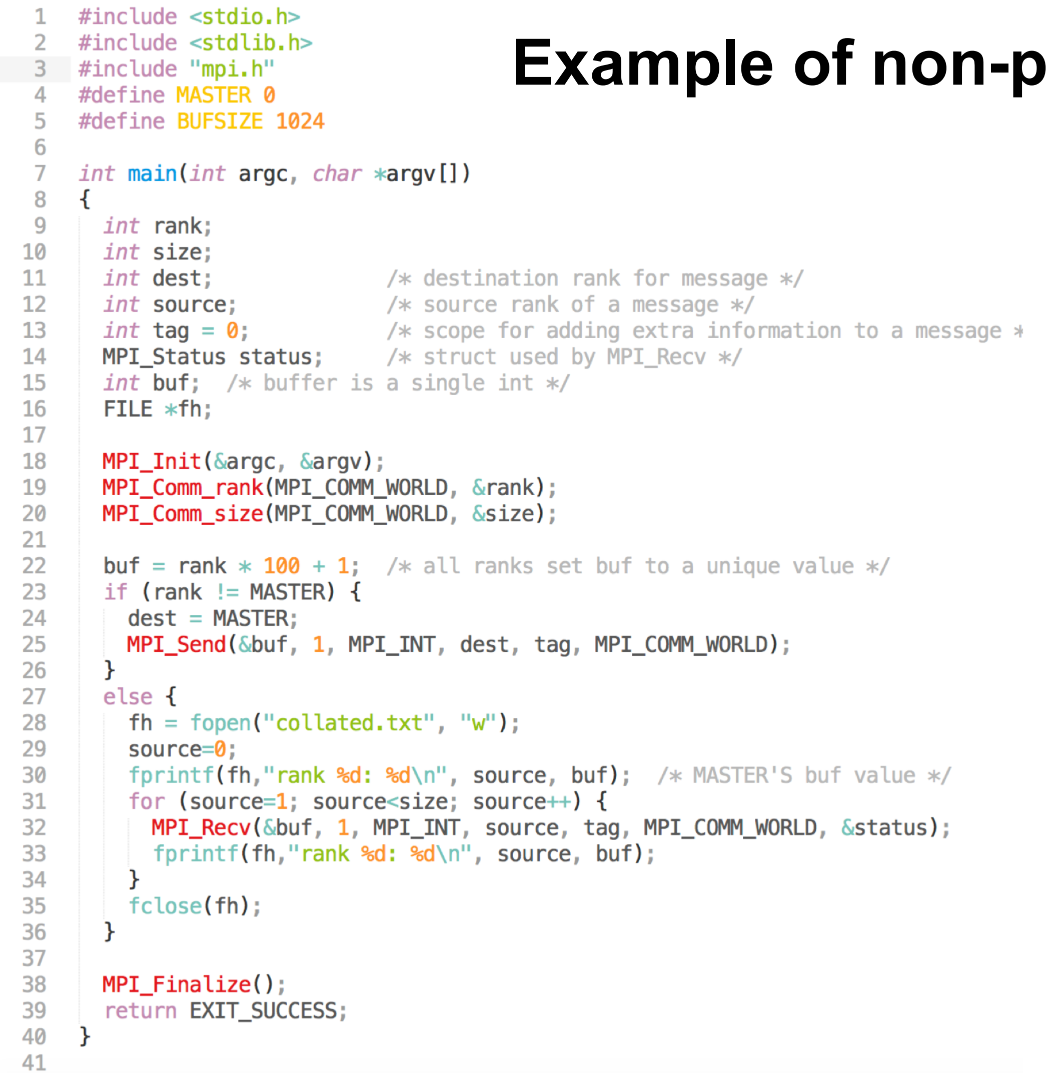
- Writing is like sending and reading is like receiving
- Any parallel I/O system will need:
  - user-defined datatypes to describe both memory and file layout
  - non-blocking operations
  - collective operations

Non Parallel I/O with MPI I/O



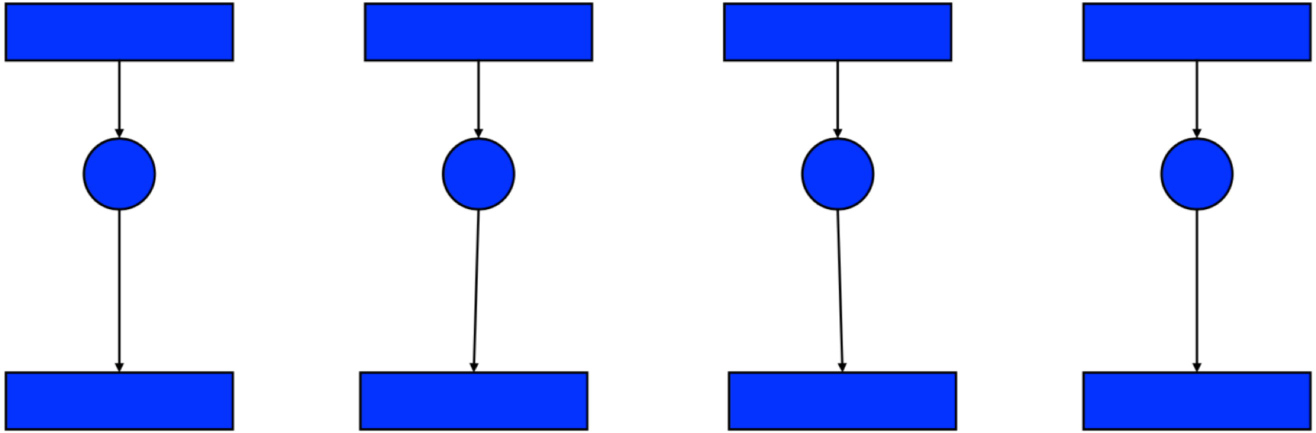
- Non-parallel
- Performance worse than sequential
- Legacy from before application was parallelized

Example of non-parallel I/O



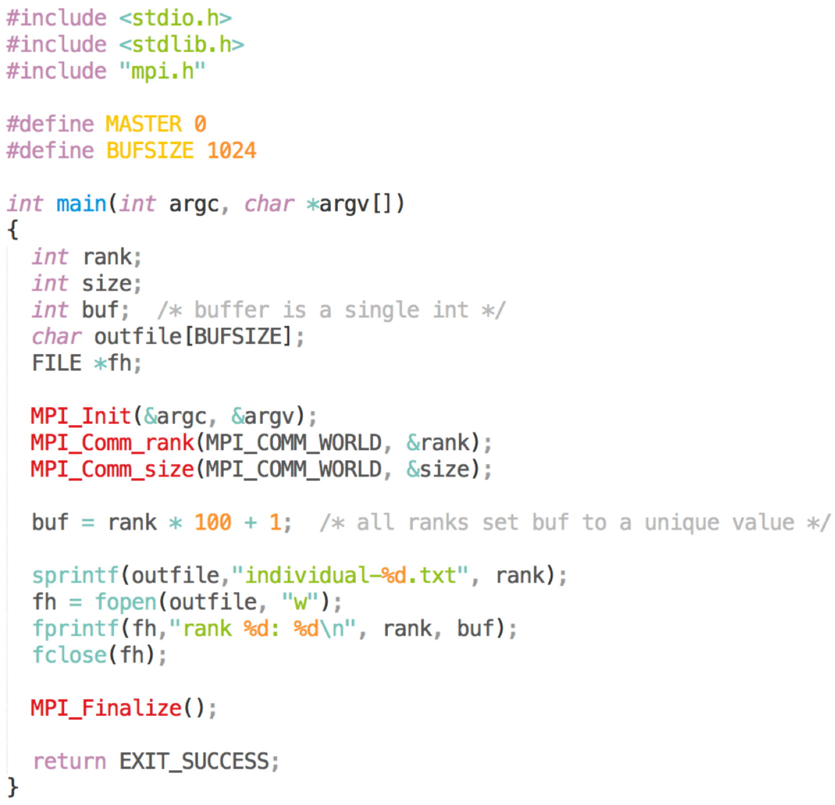
```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define MASTER 0
#define BUFSIZE 1024

int main(int argc, char *argv[])
{
  int rank;
  int size;
  int dest;              /* destination rank for message */
  int source;            /* source rank of a message */
  int tag = 0;           /* scope for adding extra information to a message *
  MPI_Status status;     /* struct used by MPI_Recv */
  int buf;  /* buffer is a single int */
  FILE *fh;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  buf = rank * 100 + 1;  /* all ranks set buf to a unique value */
  if (rank != MASTER) {
    dest = MASTER;
    MPI_Send(&buf, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
  }
  else {
    fh = fopen("collated.txt", "w");
    source=0;
    fprintf(fh,"rank %d: %d\n", source, buf);  /* MASTER'S buf value */
    for (source=1; source<size; source++) {
      MPI_Recv(&buf, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
      fprintf(fh,"rank %d: %d\n", source, buf);
    }
    fclose(fh);
  }

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Independent Parallel I/O



- Pro: parallelism
- Con: lots of small files to manage
- Legacy from before MPI I/O
    - MPI or not

Example of Independent Parallel I/O (No MPI)



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

#define MASTER 0
#define BUFSIZE 1024

int main(int argc, char *argv[])
{
  int rank;
  int size;
  int buf;  /* buffer is a single int */
  char outfile[BUFSIZE];
  FILE *fh;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  buf = rank * 100 + 1;  /* all ranks set buf to a unique value */

  sprintf(outfile,"individual-%d.txt", rank);
  fh = fopen(outfile, "w");
  fprintf(fh,"rank %d: %d\n", rank, buf);
  fclose(fh);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

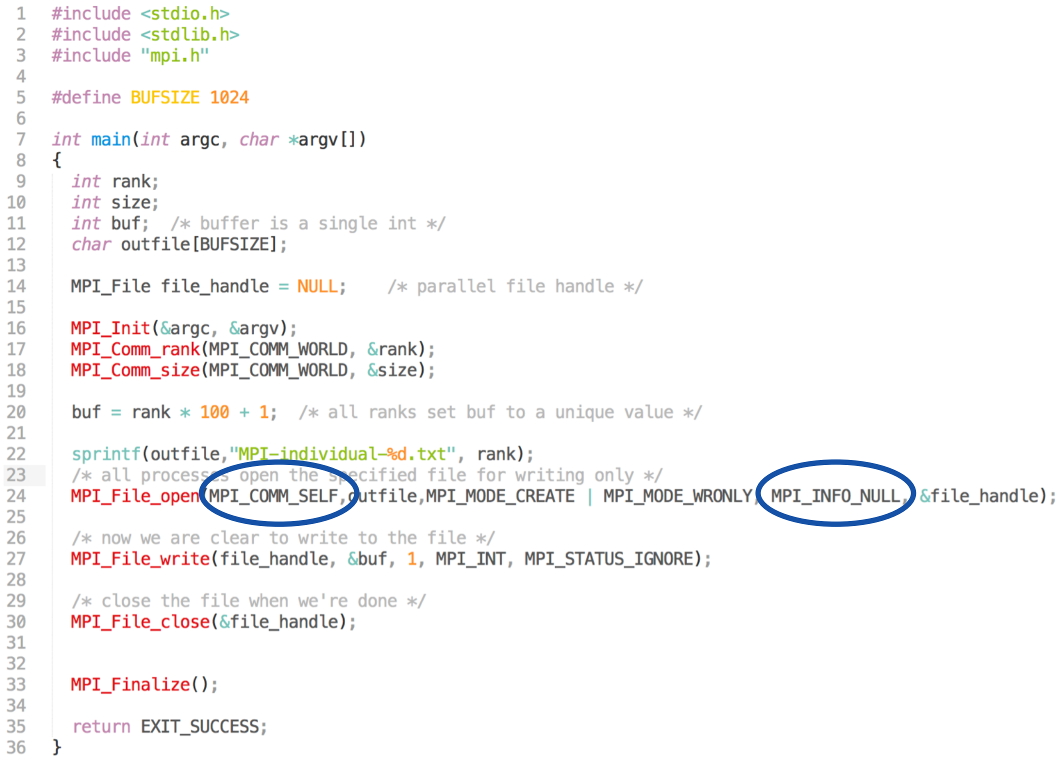
Independent Parallel I/O with MPI

- Just like POSIX I/O, you need to
  - Open the file
  - Read or Write data to the file
  - Close the file
- In MPI, these steps are almost the same:
  - Open the file: MPI_File_open
  - Write to the file: MPI_File_write / MPI_File_read
  - Close the file: MPI_File_close

Writing to a File with MPI I/O

- MPI_File_open: use MPI_MODE_WRONLY or MPI_MODE_RDWR as the flags. If the file doesn’t exist previously, the flag MPI_MODE_CREATE must also be passed to MPI_File_open
  - We can pass multiple flags by using bitwise-or ‘|’ in C, or addition ‘+” in Fortran
- Use MPI_File_write to write to file.

Example of Independent I/O with MPI

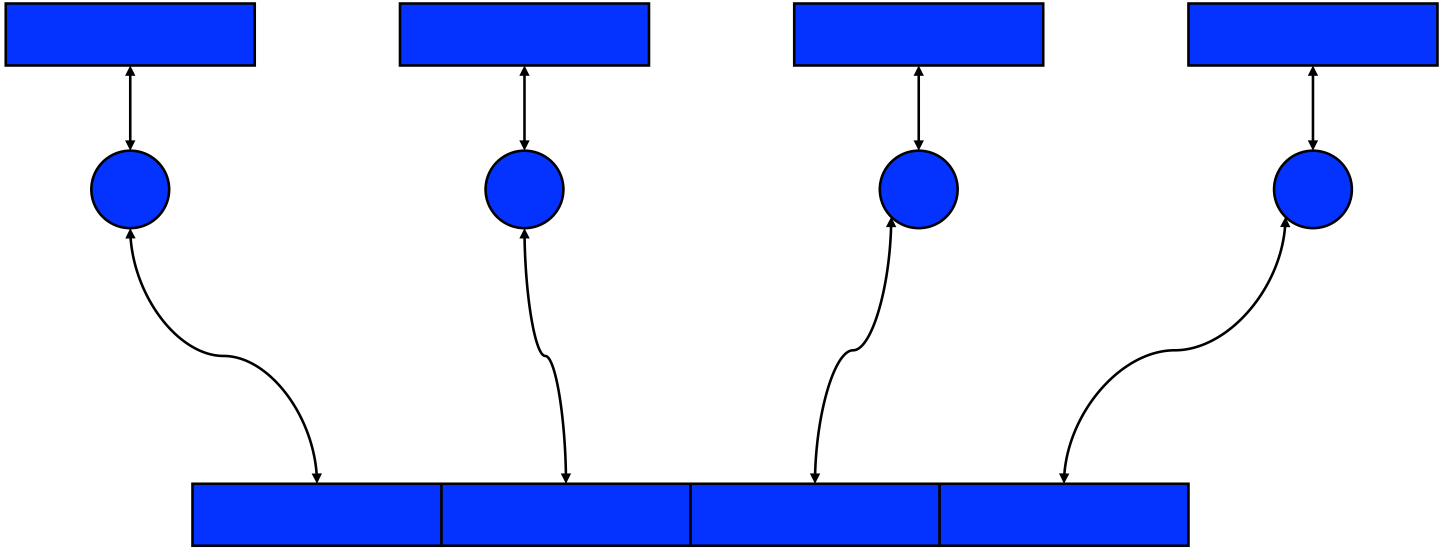


```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
  int rank;
  int size;
  int buf;   /* buffer is a single int */
  char outfile[BUFSIZE];

  MPI_File file_handle = NULL;    /* parallel file handle */

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  buf = rank * 100 + 1;  /* all ranks set buf to a unique value */

  sprintf(outfile,"MPI-individual-%d.txt", rank);
  /* all processes open the specified file for writing only */
  MPI_File_open(MPI_COMM_SELF,outfile,MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &file_handle);

  /* now we are clear to write to the file */
  MPI_File_write(file_handle, &buf, 1, MPI_INT, MPI_STATUS_IGNORE);

  /* close the file when we're done */
  MPI_File_close(&file_handle);


  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

- File Open is collective over the communicator
  Modes similar to Unix open
  MPI_Info provides additional hints for performance
- File Write is independent
- Many important variations covered in later slides
- File close is collective; similar in style to MPI_Comm_free

Why MPI I/O for Independent I/O?

Cooperative Parallel I/O – Single File

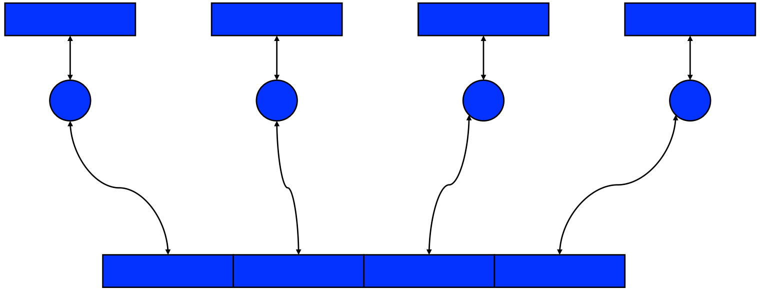


- Parallelism
  - Can only be expressed in MPI

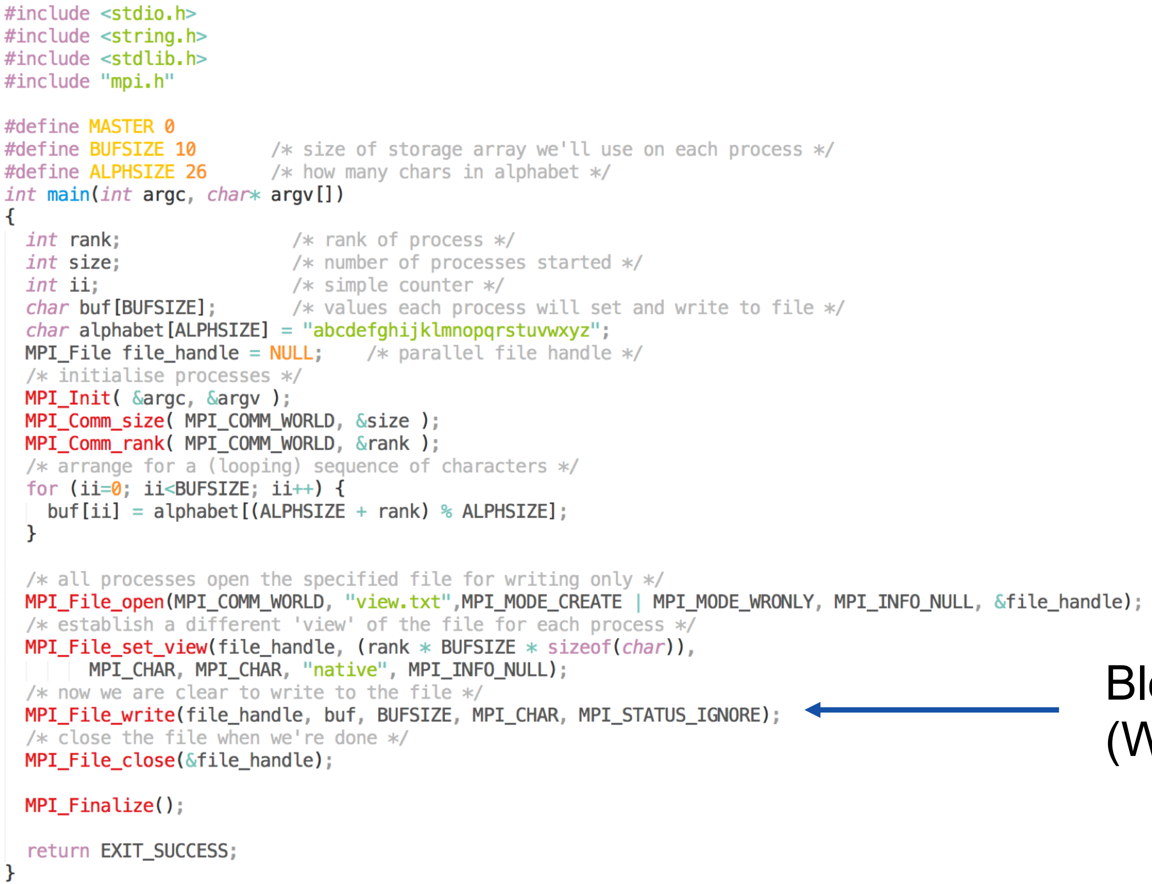
File Views

- Use MPI File View to tell each process which part of the file is allowed to write to or read from
  - Described in MPI with an offset and an MPI_Datatype

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **MPI_File_set_view**
  - *displacement* = number of bytes to be skipped from the start of the file
  - *etype* = basic unit of data access (can be any basic or derived datatype)
  - *filetype* = etype if contiguous access or MPI derived data type for non-contiguous access



Example of Writing with File View

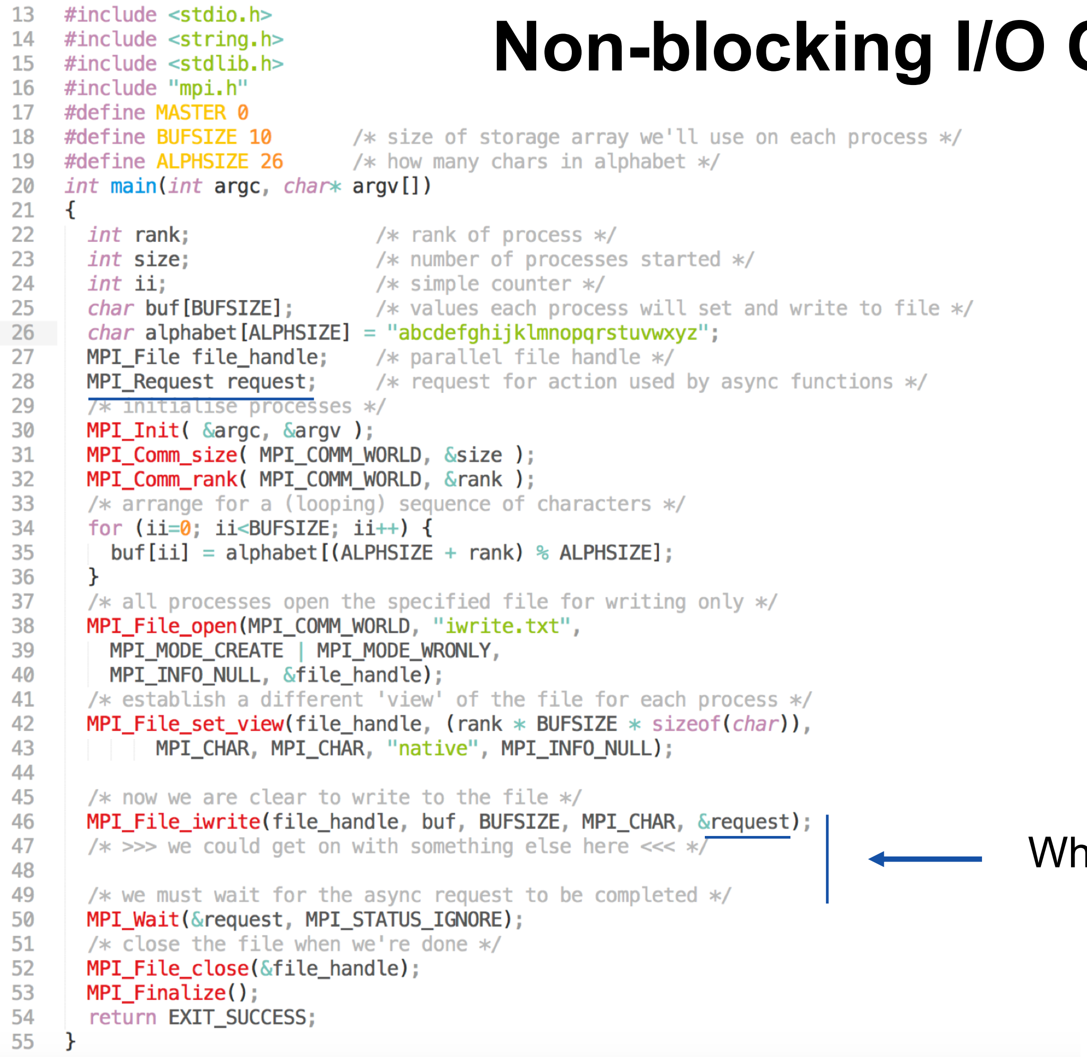


```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"

#define MASTER 0
#define BUFSIZE 10       /* size of storage array we'll use on each process */
#define ALPHSIZE 26      /* how many chars in alphabet */
int main(int argc, char* argv[])
{
  int rank;                /* rank of process */
  int size;                /* number of processes started */
  int ii;                  /* simple counter */
  char buf[BUFSIZE];       /* values each process will set and write to file */
  char alphabet[ALPHSIZE] = "abcdefghijklmnopqrstuvwxyz";
  MPI_File file_handle = NULL;    /* parallel file handle */
  /* initialise processes */
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  /* arrange for a (looping) sequence of characters */
  for (ii=0; ii<BUFSIZE; ii++) {
    buf[ii] = alphabet[(ALPHSIZE + rank) % ALPHSIZE];
  }

  /* all processes open the specified file for writing only */
  MPI_File_open(MPI_COMM_WORLD, "view.txt",MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &file_handle);
  /* establish a different 'view' of the file for each process */
  MPI_File_set_view(file_handle, (rank * BUFSIZE * sizeof(char)),
        MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);
  /* now we are clear to write to the file */
  MPI_File_write(file_handle, buf, BUFSIZE, MPI_CHAR, MPI_STATUS_IGNORE);
  /* close the file when we're done */
  MPI_File_close(&file_handle);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

←

Blocking Write
(What does it mean?)

Non-blocking I/O Operations

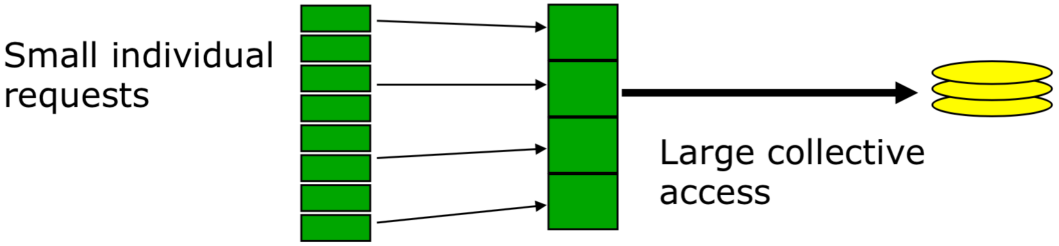


```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"
#define MASTER 0
#define BUFSIZE 10        /* size of storage array we'll use on each process */
#define ALPHSIZE 26       /* how many chars in alphabet */
int main(int argc, char* argv[])
{
  int rank;                 /* rank of process */
  int size;                 /* number of processes started */
  int ii;                   /* simple counter */
  char buf[BUFSIZE];        /* values each process will set and write to file */
  char alphabet[ALPHSIZE] = "abcdefghijklmnopqrstuvwxyz";
  MPI_File file_handle;     /* parallel file handle */
  MPI_Request request;      /* request for action used by async functions */
  /* initialise processes */
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  /* arrange for a (looping) sequence of characters */
  for (ii=0; ii<BUFSIZE; ii++) {
    buf[ii] = alphabet[(ALPHSIZE + rank) % ALPHSIZE];
  }
  /* all processes open the specified file for writing only */
  MPI_File_open(MPI_COMM_WORLD, "iwrite.txt",
    MPI_MODE_CREATE | MPI_MODE_WRONLY,
    MPI_INFO_NULL, &file_handle);
  /* establish a different 'view' of the file for each process */
  MPI_File_set_view(file_handle, (rank * BUFSIZE * sizeof(char)),
        MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);

  /* now we are clear to write to the file */
  MPI_File_iwrite(file_handle, buf, BUFSIZE, MPI_CHAR, &request);
  /* >>> we could get on with something else here <<< */

  /* we must wait for the async request to be completed */
  MPI_Wait(&request, MPI_STATUS_IGNORE);
  /* close the file when we're done */
  MPI_File_close(&file_handle);
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

What is this equivalent to?

Collective I/O and MPI

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
  - Allows communication of “big picture” to file system
- Framework for I/O optimizations at the MPI-IO layer
  - Basic idea: build large blocks, so that reads/writes in I/O
- **Requests from different processes may be merged together**

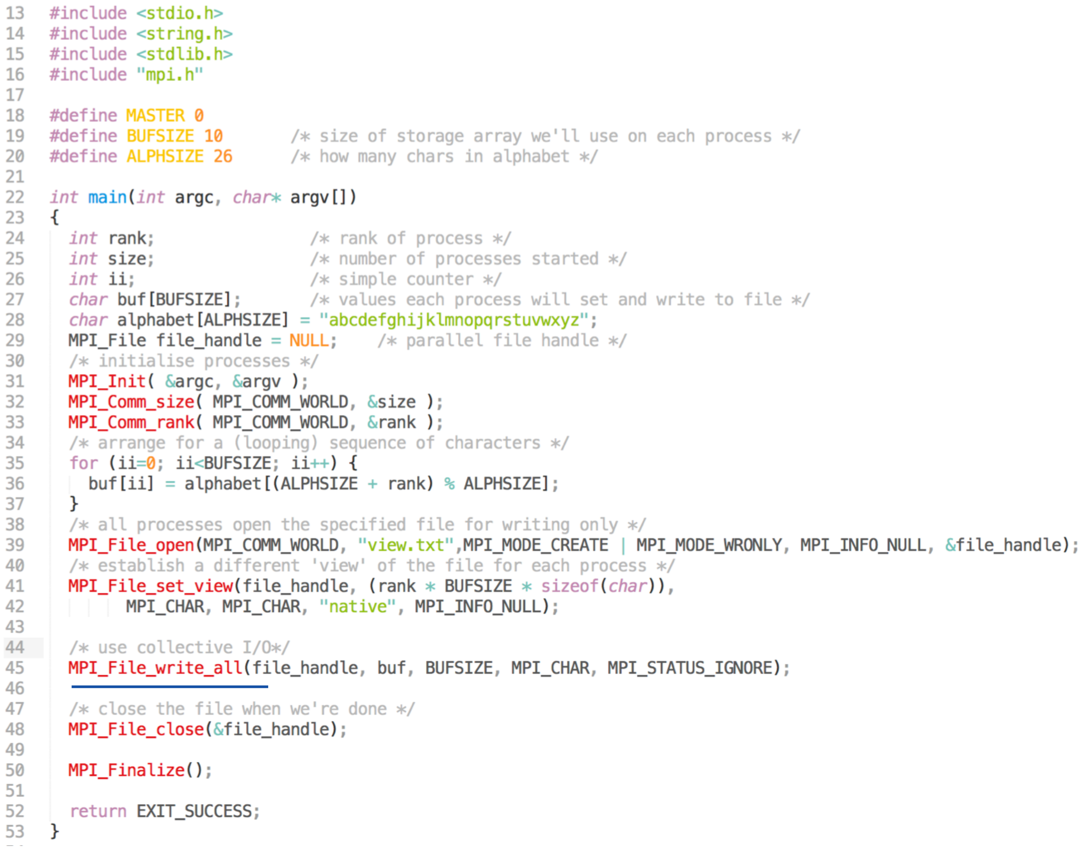


Collective I/O Functions

- **MPI_File_write_all**
  - **_all** indicates that all processes in the group specified by the communicator passed to **MPI_File_open** will call this function
- Each process specifies only its own **access information** - the argument list is the same as for the non-collective functions

How do we specify *access information*?

Example Collective MPI I/O



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"

#define MASTER 0
#define BUFSIZE 10       /* size of storage array we'll use on each process */
#define ALPHSIZE 26      /* how many chars in alphabet */

int main(int argc, char* argv[])
{
  int rank;                /* rank of process */
  int size;                /* number of processes started */
  int ii;                  /* simple counter */
  char buf[BUFSIZE];       /* values each process will set and write to file */
  char alphabet[ALPHSIZE] = "abcdefghijklmnopqrstuvwxyz";
  MPI_File file_handle = NULL;    /* parallel file handle */
  /* initialise processes */
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  /* arrange for a (looping) sequence of characters */
  for (ii=0; ii<BUFSIZE; ii++) {
    buf[ii] = alphabet[(ALPHSIZE + rank) % ALPHSIZE];
  }
  /* all processes open the specified file for writing only */
  MPI_File_open(MPI_COMM_WORLD, "view.txt",MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &file_handle);
  /* establish a different 'view' of the file for each process */
  MPI_File_set_view(file_handle, (rank * BUFSIZE * sizeof(char)),
        MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);

  /* use collective I/O*/
  MPI_File_write_all(file_handle, buf, BUFSIZE, MPI_CHAR, MPI_STATUS_IGNORE);

  /* close the file when we're done */
  MPI_File_close(&file_handle);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

Summary

- MPI I/O is a convenient way to express both independent (many files) and cooperative (shared file) parallel I/O
- MPI cooperative parallel I/O might use MPI File Views and supports different types of I/O as different communication:
  - Blocking
  - Non-blocking
  - Collective