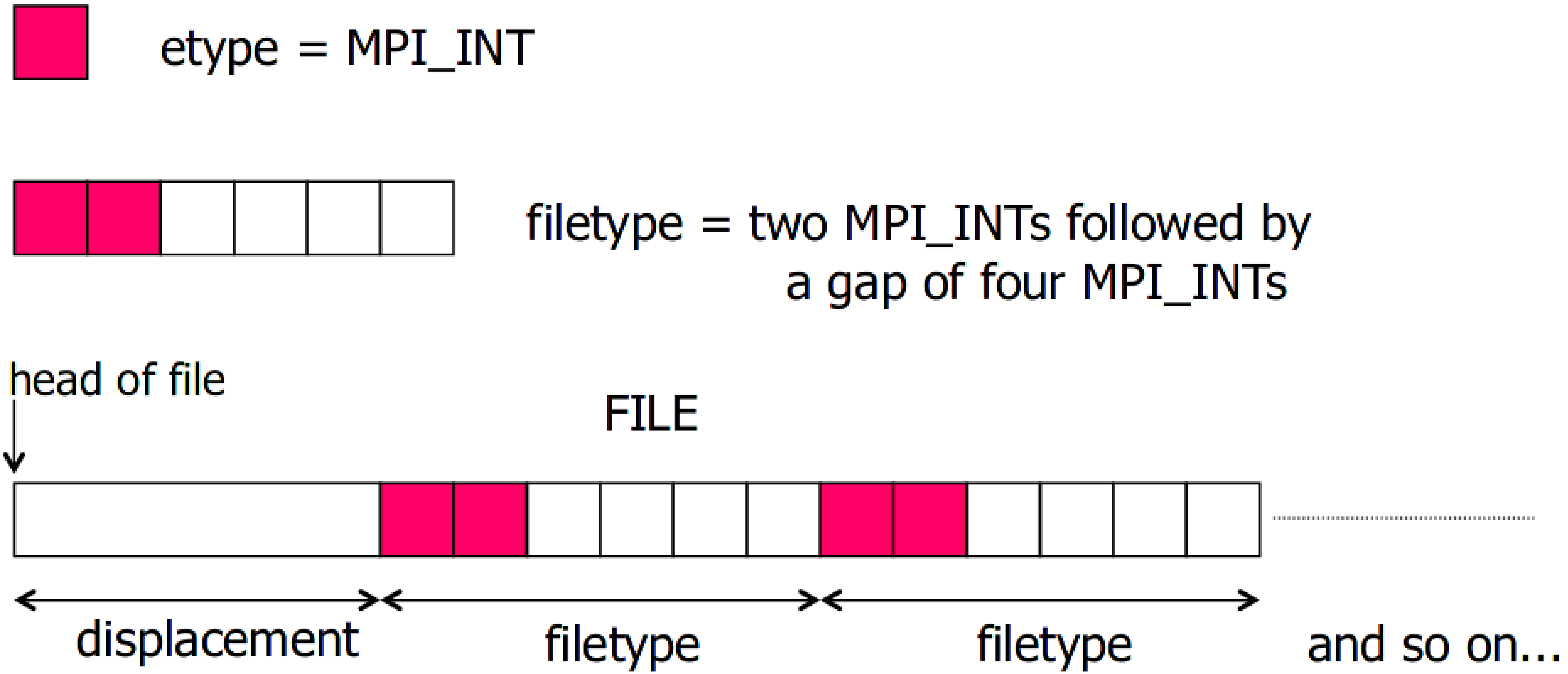
File I/O and Shared Memory

- Base type: MPI_File
- To open:
  - MPI_File_open(comm, fname, mode, info, fh)
    - Fname is a string
    - Mode is an access mode
      - e.g. MPI_MODE_RDWR
    - Info can contain other directives for file access, or MPI_INFO_NULL
    - Fh is the MPI_File to use later

Parallel File I/O

- Can manipulate files across processors similar to send and receive
    - MPI_File_seek, sets writer position
    - MPI_File_read, MPI_File_read_at, etc.
    - MPI_File_write, MPI_File_write_at, etc.
- Can also use MPI_File_set_view to dictate which parts of a file will be used by the processor
    - Can build in displacements, e.g. matrix columns!

File Views



etype = MPI_INT

filetype = two MPI_INTs followed by
a gap of four MPI_INTs

head of file

FILE

displacement

filetype

filetype

and so on...

File Views

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Custom Datatypes

- We can create derived datatypes, similar to structured datatypes in other languages

  ```
  MPI_Datatype newtype;
  ```

  ```
  MPI_Type_<sometype>( <old specs>,
  &newtype );
  MPI_Type_commit( &newtype );
  ```

  ```
  /* code that uses your new type */
  ```

  ```
  MPI_Type_free( &newtype );
  ```

Datatype creation

- MPI_Type_contiguous – contiguous blocks of data
- MPI_Type_vector – for strided data (e.g. matrix columns)
- MPI_Type_create_subarray – subsets of higher dimensional block
- MPI_Type_struct – for irregular data
- MPI_Type_indexed – for irregularly strided data

Datatype Creation

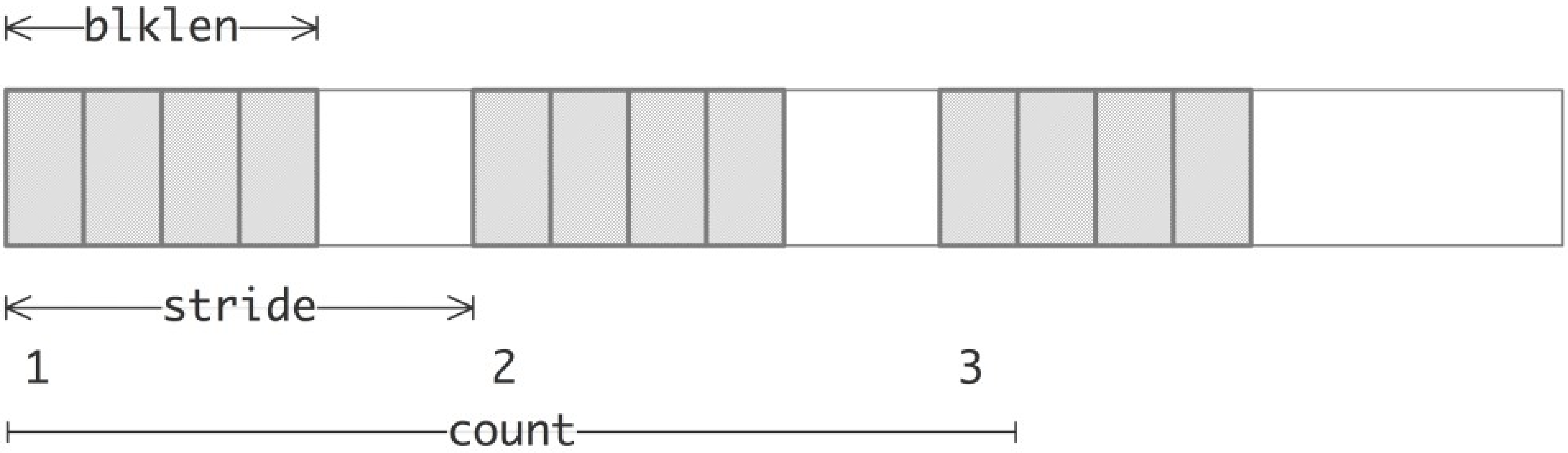
- MPI_Type_commit – tells MPI to do the indexing calculations for the type (which bytes go where)
- MPI_Type_free – declares the type no longer needed
  - The definition will be MPI_DATATYPE_NULL.
  - Communication using this datatype, that was already started, will be completed.
  - Datatypes that are defined in terms of this data type will still be usable.

Contiguous Data

- MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - e.g. you could create a matrix row as its own type, consisting of “m” MPI_FLOAT elements

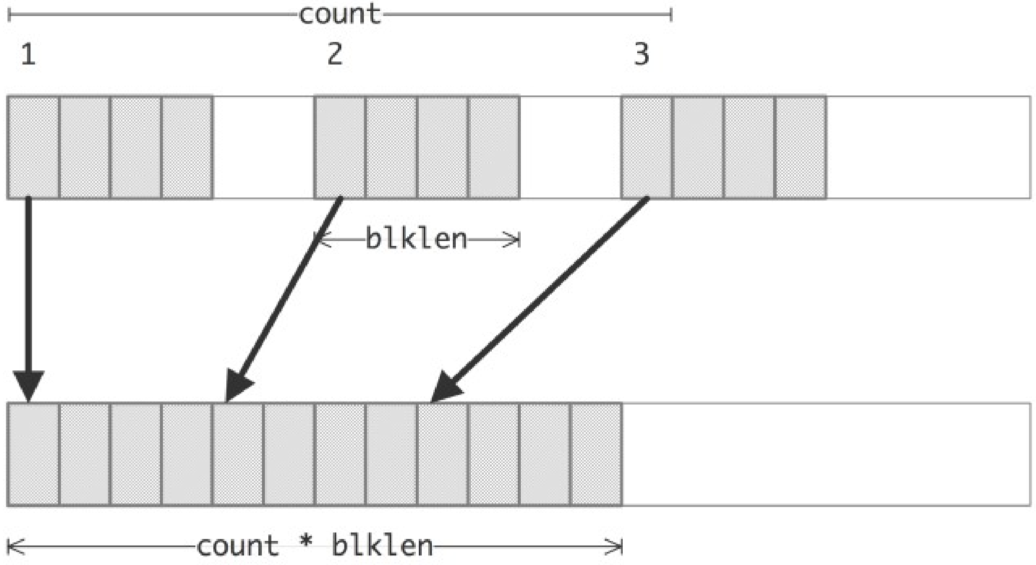
Non-contiguous Data

- MPI_Type_vector(count, blocklength, stride, oldtype, newtype)

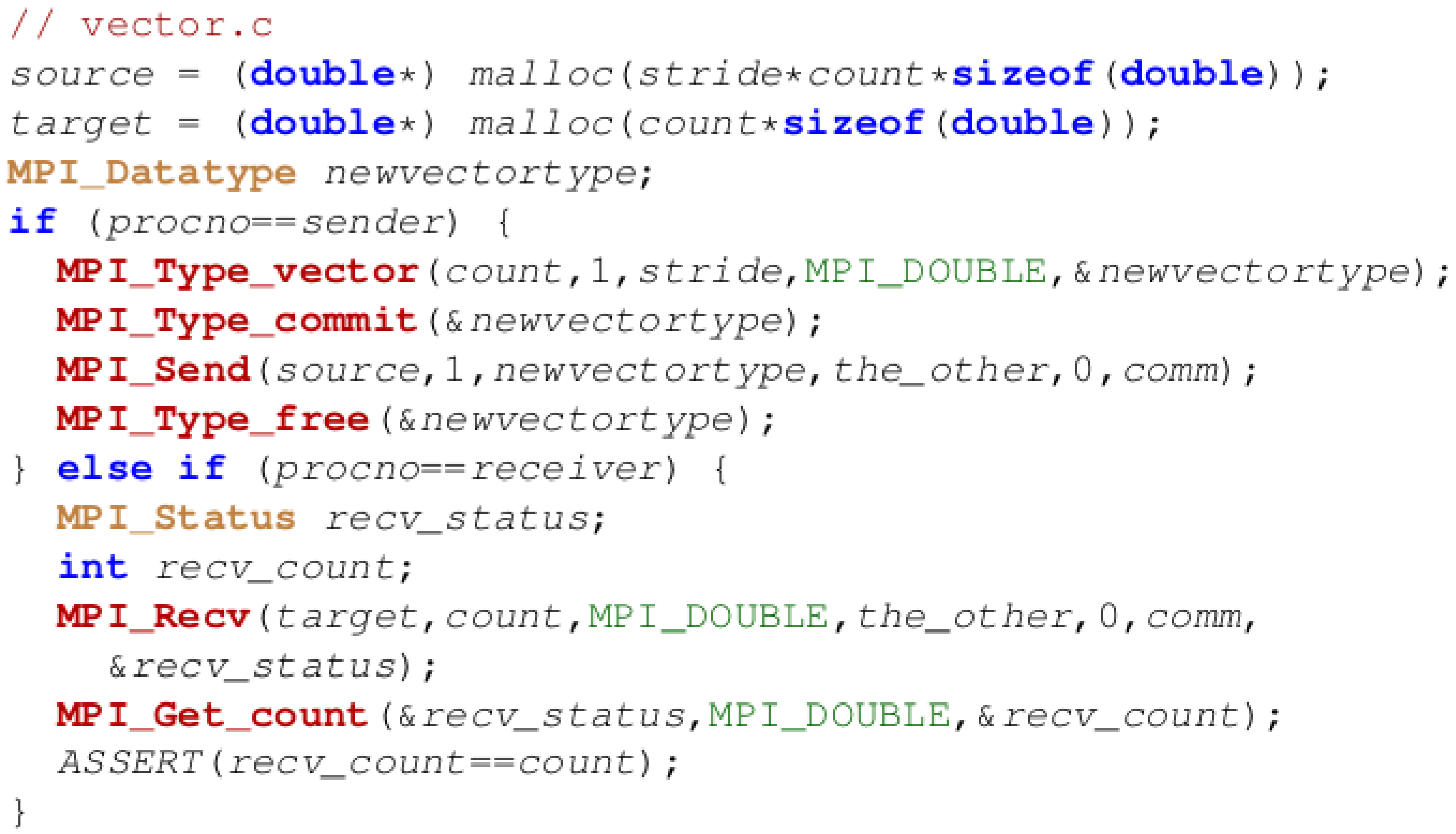


Communicating Datatypes

- Datatypes can differ on the sender and receiver!
- E.g. the sender may create a vector, sending the results to a contiguous datatype on the receiver



Example



```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(recv_count==count);
}
```

Shared Memory

- Two (or more) processors can access each other’s memory through pointers
- Bad use case: remote update
  - Same issues with traditional shared memory
- Good use case: read-only from large dataset
  - Think of your matrix labs!
- MPI can optimize shared memory access, giving you faster code “for free” over message-passing

Shared Memory Tools

- Use “MPI_Comm_split_type” to find processes on the same shared memory
- Use “MPI_Win_allocate_shared” to create a window between processes
- Use “MPI_Win_shared_query” to get a pointer to another process’ data
- Can use “memcpy” instead of MPI_Put

Invoking Shared Memory

```
MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,
    MPI_COMM_TYPE_SHARED,procno,
    info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procno);
ASSERT(new_procno<CORES_PER_NODE);
```

Application

- Quicksort in shared memory:
  - Use the parallel prefix method to partition in log(n) time:
    - Compute $X_i = \#\{a_j \mid j < i \text{ and } a_j < p\}$ for a given pivot p
    - Do the same for the bigger elements
    - Now we know where each one goes!
  - After the O(log(n)) steps, total is $O(\log^2(n))$ time!

Extra Useful Tool: Wallclock

- MPI Offers a tool to try calculating time since an event in the past: MPI_Wtime
- For parallel processes, the clock is usually not global :(
- Typical usage:

```
MPI_Barrier(comm);
t = MPI_Wtime();
// do stuff
MPI_Barrier(comm);
```

  - `t = MPI_Wtime() - t; // offset by first time`

Submitting to the cluster

- (See course webpage for extra instructions)
- Create a bash script with “SBATCH” parameters to control the MPI environment
- Queue the job with “sbatch <filename>”
- The output will be stored in the file indicated by the script parameters

Example:

- Read a large matrix from a shared MPI file
  - https://www.kaggle.com/bradklassen/pga-tour-20102018-data
- (Try a small csv file first)
- Put the data into distributed matrices
- Will adapt Gauss-Jordan to now work with this large matrix!

Coming up: Benchmarking

- Using the “TAU” software package
- Requires compiling software with the TAU compiler wrapper script
- Runs like normal, output goes into “profile” files
- Use the TAU “pprof” utility to see the aggregate summary