Lecture 11

Alternative: pack/unpack

MPI_Pack - allows one to explicitly store noncontiguous data in contiguous memory locations.

MPI_Unpack - can be used to copy data from a contiguous buffer into noncontiguous memory locations.

MPI_Pack

```
int MPI_Pack ( void *inbuf, int incount, MPI_Datatype datatype,
               void *outbuf, int outcount, int *position, MPI_Comm comm )
```

inbuf - input buffer start (what needs to be packed)

incount - number of input data items

datatype - datatype of each input data item

outbuf - output buffer start (packed message)

outcount - output buffer size, in bytes (size of container for message)

position - current position in buffer, in bytes (integer).
In a typical MPI_Pack call, position references the first free
location in buffer (in that case *position*=0 for first call)
MPI_Pack then returns incremented position indicating the
first free location in buffer after data has been packed.
Convenient since position is automatically calculated for
user.

comm - communicator that will be using *outbuf*

MPI_Unpack

```
int MPI_Unpack ( void *inbuf, int insize, int *position,
                 void *outbuf, int outcount, MPI_Datatype datatype,
                 MPI_Comm comm )
```

inbuf - input buffer, contains packed message made by MPI_Pack

insize - size of input buffer, in bytes

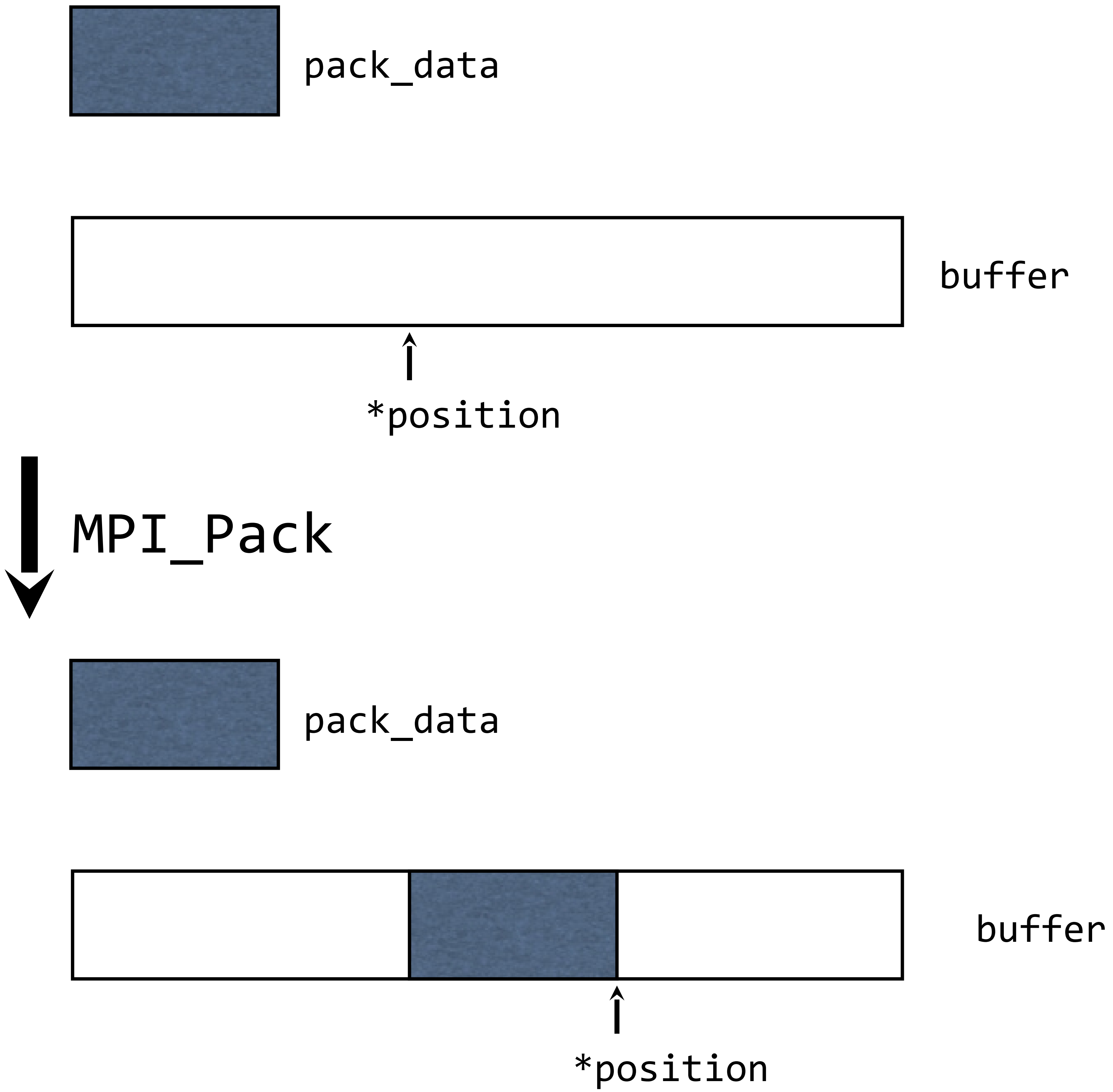
position - current position in buffer, in bytes (integer).
Data unpacked in same order as it was packed.
Position incremented when MPI_Unpack returns.
Convenient since position is automatically calculated for user.

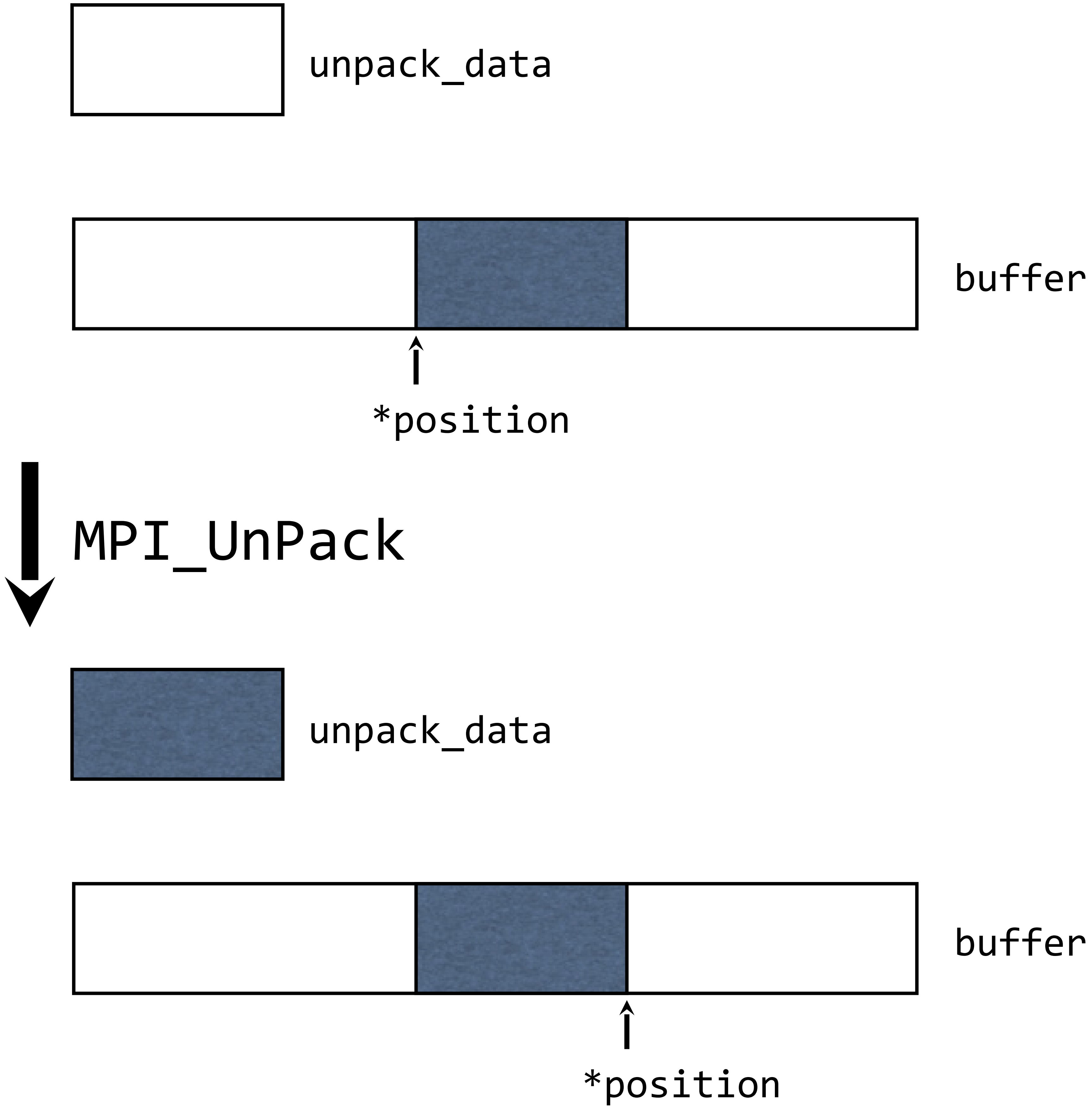
outbuf - output buffer start (where you want to unpack the data to)

outcount - number of items to be unpacked

datatype - datatype of each output item

comm - communicator that will be using *inbuf*


pack_data
buffer
*position
MPI_Pack
pack_data
buffer
*position


unpack_data
buffer
*position
MPI_UnPack
unpack_data
buffer
*position

MPI_Pack/MPI_Unpack example

from P. Pacheco, Parallel Programming with MPI

```
/* read real a and b, integer n, pack data, MPI_Bcast, then unpack */
void Get_data4(
         float*  a_ptr    /* out */,
         float*  b_ptr    /* out */,
         int*    n_ptr    /* out */,
         int     my_rank  /* in  */) {

    char  buffer[100];  /* Store data in buffer        */
    int   position;     /* Keep track of where data is */
                        /*     in the buffer           */

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        /* Now pack the data into buffer.  Position = 0 */
        /* says start at beginning of buffer.           */
        position = 0;

        /* Position is in/out */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented: it now        */
        /* references the first free location in buffer. */

        MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented again. */

        MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented again. */

        /* Now broadcast contents of buffer */
        MPI_Bcast(buffer, 100, MPI_PACKED, 0,
            MPI_COMM_WORLD);
}
```

```
else {
        MPI_Bcast(buffer, 100, MPI_PACKED, 0,
            MPI_COMM_WORLD);

        /* Now unpack the contents of buffer */
       position = 0;
        MPI_Unpack(buffer, 100, &position, a_ptr, 1,
            MPI_FLOAT, MPI_COMM_WORLD);
        /* Once again position has been incremented: */
        /* it now references the beginning of b.     */

        MPI_Unpack(buffer, 100, &position, b_ptr, 1,
            MPI_FLOAT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &position, n_ptr, 1,
            MPI_INT, MPI_COMM_WORLD);
    }
} /* Get_data4 */
```

MPI_Pack_size

```
int MPI_Pack_size ( int incount, MPI_Datatype datatype, MPI_Comm comm,
                    int *size )
```

incount - count argument to packing call (i.e. number of elements to be packed)

datatype - datatype of elements to be packed

comm - communicator for packing call

size - upper bound on size of packed message, in bytes

Useful for determining the size of the buffer which needs to be allocated for packed message.

Which method to use?

If data stored as consecutive entries of array, always use the *count* and *datatype* parameters in communications functions. Involves no additional overhead.

If there are a large number of elements not in contiguous memory locations, building a derived type will probably involve less overhead than a large number of calls to MPI_Pack/MPI_Unpack .

If the data all have the same type and are stored at regular intervals in memory (e.g. column of matrix), then it will almost certainly be much easier and faster to use a derived datatype rather than MPI_Pack/MPI_Unpack .

Typically, overhead of creating datatype is incurred only once, while the overhead of calling MPI_Pack/MPI_Unpack must be incurred every time data is communicated.

Using MPI_Pack/MPI_Unpack may be appropriate if only sending heterogenous data only a few times.

Ultimately, choice will depend on particular MPI implementation and network hardware.

Design and Coding of Parallel Programs

Example: Sorting

Writing parallel code

Use various strategies you learned for writing serial code.

It’s hard to write a code in one go, so proceed in stages.

First, decide how your data and computational work will be distributed between processes.

Once that is clear, use a mixture of “top down” and “bottom up” approach to gradually build the code.

Proceeding “top down”, write the main function first. In the first rough draft, use dummy type definitions and define subprograms as stubs initially.

When the main program is done, start working on subprograms (“bottom up”).

Delay defining data structures as long as possible, to maintain flexibility.

If something looks hard, procrastinate i.e. leave writing it until later if possible, replace it with a stub function.

Test your code frequently as you proceed along the writing stages.

Test with various numbers of processors. Code which works on 1 or 2 may fail on 3 or more etc.

Use a “driver” to test your subprogram i.e. a highly simplified main function that has only sufficient data to correctly launch the subprogram.

Add informative output statements to your code so you can see what is happening. Put those in #DEBUG sections if you like.

Since I/O can be troublesome, you can use hardwired input values initially.

Take advantage of a debugger (DDT on SHARCNET) as much as possible.

If compiling on SHARCNET, keep all your files there and use a graphical editor to edit your files remotely.

Example : Sorting

Let's write a program which sorts a list of keys (e.g. numbers or words) into process-ascending order: keys on process *q* are to be sorted into increasing order, with all keys on *q* being greater than all keys on *q-1* and less then all keys on *q+1*.

In other words, you can think of a linear array distributed in block fashion across processes, and we want to sort the elements of the array in increasing order.

General sorting on parallel processors is a difficult problem and an active area of research.

We will consider the simplifying case where the keys are distributed randomly and uniformly. This will allow for easily dividing the keys almost uniformly between processes.

In this example, will assume keys are non-negative integers uniformly distributed in some range. This allows for easy data distribution.

For example, assume range is 1 and 100, and assume there are 4 processes. In that case, process 0 should receive keys 1 to 25, process 1 keys 26 to 50 etc.

Initial state before sort

Process	Keys (in range 1 to 100)
0	71 , 4 , 63 , 42 , 64, 82
1	27 , 96 , 38 , 7 , 47 , 76
2	53 , 75 , 10 , 13, 2 , 58
3	49 , 79 , 22 , 85 , 33 , 99

Final state after sort

Process	Keys (in range 1 to 100)
0	2 , 4 , 7 , 10 , 13, 22
1	27 , 33 , 38 , 42 , 47 , 49
2	53 , 58 , 63 , 64, 71 , 75
3	76 , 79 , 82 , 85 , 96 , 99

Algorithm

1. Get local keys - set up initial state.

2. Use the uniform distribution to determine which process should get each key.

3. Send keys to appropriate processes. Receive keys from processes.

4. Sort local keys.

5. Print results.

Let’s write an outline main function with stubs to reflect this algorithm.

Keep things simple with dummy type definition.

```
typedef int LOCAL_LIST_T;

typedef int KEY_T;
```

```
/* sort_1.c -- level 1 version of sort program
 *
 * Input: none
 * Output:  messages indicating flow of control through program
 *
 * See Chap 10, pp. 226 & ff in PPMPI.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include "cio.h"
#include "sort_1.h"

int       p;
int       my_rank;
MPI_Comm  io_comm;

main(int argc, char* argv[]) {
    LOCAL_LIST_T  local_keys;
    int           list_size;
    int           error;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);
    list_size = Get_list_size();

    /* Return negative if Allocate failed */
    error = Allocate_list(list_size, &local_keys);

    Get_local_keys(&local_keys);
    Print_list(&local_keys);
    Redistribute_keys(&local_keys);
    Local_sort(&local_keys);
    Print_list(&local_keys);

    MPI_Finalize();}
 /* main */

int Get_list_size() {
    Cprintf(io_comm,"","%s", "In Get_list_size");
    return 0;
} /* Get_list_size */
```

```
/* Return value negative indicates failure */
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys) {

    Cprintf(io_comm,"","%s", "In Allocate_key_list");
    return 0;
} /* Allocate_list */

void Get_local_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s", "In Get_local_keys");
} /* Get_local_keys */

void Redistribute_keys(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s", "In Redistribute_keys");
} /* Redistribute_keys */

void Local_sort(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s","In Local_sort");
} /* Local_sort */

void Print_list(LOCAL_LIST_T* local_keys) {
    Cprintf(io_comm,"","%s","In Print_list");
} /* Print_list */
```

```c
/* sort_1.h -- header file for sort_1.c */
#ifndef SORT_H
#define SORT_H

#define KEY_MIN 0
#define KEY_MAX 32767
#define KEY_MOD 32768

typedef int KEY_T;
typedef int LOCAL_LIST_T;

#define List_size(list) (0)
#define List_allocated_size(list) (0)

int Get_list_size(void);
int Allocate_list(int list_size,
    LOCAL_LIST_T* local_keys);
void Get_local_keys(LOCAL_LIST_T* local_keys);
void Redistribute_keys(LOCAL_LIST_T* local_keys);
void Local_sort(LOCAL_LIST_T* local_keys) ;
void Print_list(LOCAL_LIST_T* local_keys) ;
#endif
```

Test stage 1 programs.

Even though it’s very simple, testing it will reveal typos, simple mistakes etc.

After this is done, the rest is just a matter of filling in the subprograms.

Since input is a major problem, first write a version with hardwired input.

In this initial version, also restrict yourself to small list size, since you will want to print the list often as you write and debug your program. For example, consider only lists of length 5**p*, where *p* is number of processes.

```
/*********************************************************************/
int Get_list_size(void) {
    Cprintf(io_comm,"","%s","In Get_list_size");
    return 5*p;
} /* Get_list_size */
```

We can't actually allocate the list yet, as we have not decided on the actual structure.

We do want members of the structure to record number of keys and space allocated. So, we provisionally modify LOCAL_LIST_T .

In sort.h, we now have:

```
typedef struct {
    int allocated_size;
    int local_list_size;
    int keys;  /* dummy member */
} LOCAL_LIST_T;

/* Assume list is a pointer to a struct of type
 * LOCAL_LIST_T */
#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
```

We will use a random number generator to generate the keys on each process. For now define empty function Insert_key .

```
/*********************************************************************/
void Get_local_keys(LOCAL_LIST_T* local_keys) {
    int i;

    /* Seed the generator */
    srand(my_rank);

    for (i = 0; i < List_size(local_keys);  i++)
        Insert_key(rand() % KEY_MOD, i, local_keys);
} /* Get_local_keys */

/*********************************************************************/
void Insert_key(KEY_T key, int i,
    LOCAL_LIST_T* local_keys) {
} /* Insert_key */
```

Stage 2 complete. Very minor changes from stage 1.

Now is the time to decide on how to send the original local keys from each process to the appropriate address.

This exchange of data will be an example of an:

all-to-all scatter/gather or total exchange.

MPI provides dedicated functions for this, which we will use:

MPI_Alltoall and MPI_Alltoallv

Say each process determines how many elements it must send to the other process.

This information must then be exchanged among processes so that we can then follow with the actual sending of keys.

After keys are locally sorted, each process knows how many have to be send to the other processes. But that information must be passed to the recipients. For example, information in shaded column must go to process 2 etc.

Source process	# of Keys for 0	# of Keys for 1	# of Keys for 2	# of Keys for 3
0	5	7	6	7
1	4	6	8	7
2	7	4	9	3
3	10	7	3	5

MPI_Alltoall

```
int MPI_Alltoall ( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm )
```

sendbuf - start address of send buffer.

sendcount - number of elements to send to each process.

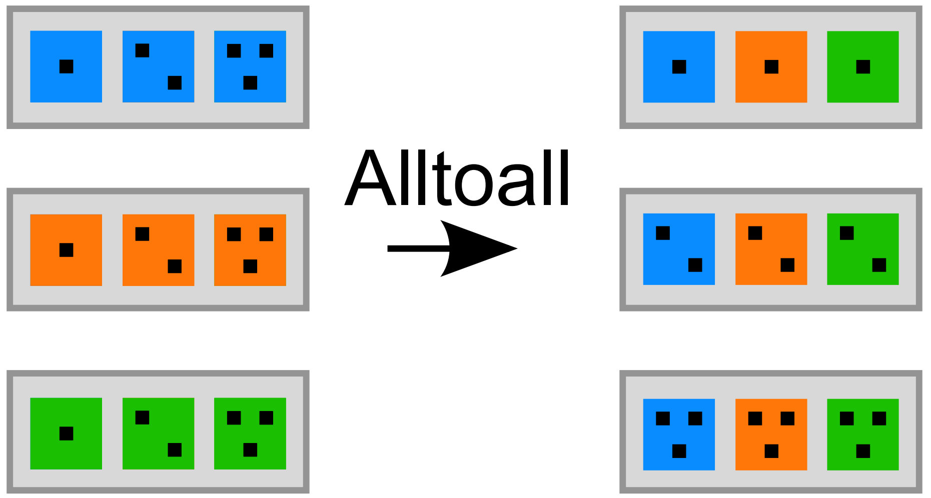
sendtype - datatype of send elements.

recvbuf - address of receive buffer.

recvcount - number of elements received from any process.

recvtype - datatype of recv elements.

comm – communicator.



In a more general case *sendcount* and *recvcount* will not be just single constants for the whole communication.

We want to allow for variable amounts of data to be communicated between each process pair. Therefore must have an array describing datatype counts.

sendcount -> sendcounts[]
recvcount -> recvcounts[]

As these are now variable, must also have an array describing displacements, as these cannot just be assumed to be multiples of some constant:

sdispls[] - send data displacements,
rdispls[] - receive data displacements.

MPI_Alltoallv

```
int MPI_Alltoallv (
    void *sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype,
    MPI_Comm comm )
```

sendbuf - starting address of send buffer.

sendcounts - array of send counts.

sdispls - array of send displacements.

sendtype - send datatype.

recvbuf - starting address of receive buffer.

recvcounts - array of receive counts.

rdispls - array of receive displacements.

recvtype - receive datatype.

comm – communicator.

To distribute the keys:

1. Determine what and how much data is to be sent to each process.

2. Carry out a total exchange on the amount of data to be sent/received by each process (MPI_Alltoall).

3. Compute the total amount of space needed for the data to be received and allocate storage.

4. Find the displacements of the data to be received.

5. Carry out a total exchange of the actual keys (MPI_Alltoallv).

6. Free old storage.

Modify structure, add pointer which will eventually point to list array. We also add some useful macros.

```
typedef int KEY_T;
typedef struct {
    int allocated_size;
    int local_list_size;
    KEY_T* keys;
} LOCAL_LIST_T;

#define List_size(list) ((list)->local_list_size)
#define List_allocated_size(list) ((list)->allocated_size)
#define List(list) ((list)->keys)
#define List_free(list) {free List(list);}
#define List_key(list,i) (*((list)->keys + i))

#define key_mpi_t MPI_INT
```

```
void Redistribute_keys(
         LOCAL_LIST_T* local_keys  /* in/out */) {
    int new_list_size, i, error = 0;
    int* send_counts;
    int* send_displacements;
    int* recv_counts;
    int* recv_displacements;
    KEY_T* new_keys;

    /* Allocate space for the counts and displacements */
    send_counts = (int*) malloc(p*sizeof(int));
    send_displacements = (int*) malloc(p*sizeof(int));
    recv_counts = (int*) malloc(p*sizeof(int));
    recv_displacements = (int*) malloc(p*sizeof(int));

    Local_sort(local_keys);
    Find_alltoall_send_params(local_keys,
        send_counts, send_displacements);

    /* Distribute the counts */
    MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts,
        1, MPI_INT, MPI_COMM_WORLD);
    /* Allocate space for new list */
    new_list_size = recv_counts[0];
    for (i = 1; i < p; i++)
        new_list_size += recv_counts[i];
    new_keys = (KEY_T*)
        malloc(new_list_size*sizeof(KEY_T));

    Find_recv_displacements(recv_counts, recv_displacements);

    /* Exchange the keys */
    MPI_Alltoallv(List(local_keys), send_counts,
        send_displacements, key_mpi_t, new_keys,
        recv_counts, recv_displacements, key_mpi_t,
        MPI_COMM_WORLD);

    /* Replace old list with new list */
    List_free(local_keys);
    List_allocated_size(local_keys) = new_list_size;
    List_size(local_keys) = new_list_size;
    List(local_keys) = new_keys;

    /* Free temporary storage */
    free(send_counts);
    free(send_displacements);
    free(recv_counts);
    free(recv_displacements);
} /* Redistribute_keys */
```

At this stage, Find_alltoall_send_params and Find_recv_displacements are defined by us as stub functions.

Thus the code at this stage will compile, but you cannot yet actually run Redistribute_keys with these functions empty so comment it out.

```
void Find_alltoall_send_params(LOCAL_LIST_T* local_keys,
        int send_counts[], int send_displacements[]) {

} /* Find_alltoall_send_params */

void Find_recv_displacements(int recv_counts[],
         int recv_displacements[]) {
}
```

Write function to sort the list at this stage, use intrinsic qsort.

```
/*********************************************************************/
void Local_sort(LOCAL_LIST_T* local_keys) {

    qsort(List(local_keys), List_size(local_keys), sizeof(KEY_T),
        (int(*)(const void*, const void*))(Key_compare));
} /* Local_sort */

/*********************************************************************/
int Key_compare(const KEY_T* p, const KEY_T* q) {
    if (*p < *q)
        return -1;
    else if (*p == *q)
        return 0;
    else /* *p > *q */
        return 1;

}  /* Key_compare */
```

Fill out allocate list function and insert key function.

```
/* Return value negative indicates failure */
int Allocate_list(
        int             list_size  /* in  */,
        LOCAL_LIST_T* local_keys /* out */) {

    List_allocated_size(local_keys) = list_size/p;
    List_size(local_keys) = list_size/p;
    List(local_keys) = (KEY_T*)
        malloc(List_allocated_size(local_keys)*sizeof(KEY_T));
    if (List(local_keys) == (KEY_T*) NULL)
        return -1;
    else
       return 0;
} /* Allocate_list */

void Insert_key(KEY_T key, int i,
    LOCAL_LIST_T* local_keys) {
    List_key(local_keys, i) = key;
} /* Insert_key */
```

Fill out Print_list function as well.

```
/*********************************************************************/
void Print_list(MPI_Comm io_comm, LOCAL_LIST_T* local_keys) {
    char list_string[LIST_BUF_SIZE];
    char key_string[MAX_KEY_STRING];
    int  i;

    list_string[0] = '\0';
    for (i = 0; i < List_size(local_keys); i++) {
        sprintf(key_string,"%d ", List_key(local_keys,i));
        strcat(list_string, key_string);
    }
    Cprintf(io_comm,"Contents of the list", "%s", list_string);
} /* Print_list */
```

This completes stage 3, one last stage left.

We need to add code to compute send_counts and send_displacements, and we are almost done.

To write the functions necessary, we can make use of the fact that our local list will be already sorted before they are called. This means we have the relation:

```
send_displacements[q] = send_displacements[q-1] + send_counts[q-1]
```

Must have a separate case for *q*=0.

Also need a cutoff value for each process i.e. given process will not be given keys higher than cutoff. As keys are uniformly distributed between 0 and KEY_MAX,

```
cutoff = (q+1)*(KEY_MAX+1)/p
```

is the cutoff for process *p* .

```c
void Find_alltoall_send_params(
         LOCAL_LIST_T* local_keys         /* in  */,
         int*          send_counts        /* out */,
         int*          send_displacements /* out */) {
    KEY_T cutoff;
    int i, j;

    /* Take care of process 0 */
    j = 0;
    send_displacements[0] = 0;
    send_counts[0] = 0;
    cutoff = Find_cutoff(0);
    /* Key_compare > 0 if cutoff > key */
    while ((j < List_size(local_keys)) &&
           (Key_compare(&cutoff,&List_key(local_keys,j))
                > 0)) {
        send_counts[0]++;
        j++;
    }
    /* Now deal with the remaining processes */
    for (i = 1; i < p; i++) {
        send_displacements[i] =
            send_displacements[i-1] + send_counts[i-1];
        send_counts[i] = 0;
        cutoff = Find_cutoff(i);
        /* Key_compare > 0 if cutoff > key */
        while ((j < List_size(local_keys)) &&
               (Key_compare(&cutoff,&List_key(local_keys,j))
                   > 0)) {
            send_counts[i]++;
            j++;
        }
    }
} /* Find_alltoall_send_params */

/*defined as separate function for easy changing later if needed*/
int Find_cutoff(int i) {
    return (i+1)*(KEY_MAX + 1)/p;
}  /* Find_cutoff */
```

Fill in Find_recv_displacements function:

```
void Find_recv_displacements(int recv_counts[],
    int recv_displacements[]){
    int i;

    recv_displacements[0] = 0;
    for (i = 1; i < p; i++)
        recv_displacements[i] =
            recv_displacements[i-1]+recv_counts[i-1];
} /* Find_recv_displacements */
```

```
Final piece is just a function which will read the only input parameter
necessary at runtime, which specifies how big the list is.

Assuming you have working access to standard input:

/*********************************************************************/
int Get_list_size(void) {
    int size;

    Cscanf(io_comm,"How big is the list?","%d",&size);
    return size;
} /* Get_list_size */
```

At this point you might consider writing a serial driver for Find_alltoall_send_params to test it out.

This driver would read in *p*, *my_rank*, and a list of keys. It would then calculate the send counts and displacements and print them out.

It is worthwhile to write your code explicitly in stages.

There are tools to help you managing your code as you modify it. A good one is called GIT.

Parallel Libraries

Using Libraries: Pro and Con

Pro:
- Libraries can provide high quality, high performance code.
- User does not have to write the parallel program, which is difficult.
- MPI has good support for parallel libraries.
- Use of communicators allows library to isolate its communications universe from rest of program, avoiding conflicts.

Con:
- Writing and documenting a library is difficult and time-consuming, hence some libraries may be deficient in some respects and difficult to use.
- Examples of well-designed and well-documented libraries: ScaLAPACK, PETSc, FFTW .

Parallel Libraries: FFTW with MPI

FFTW - Fastest Fourier Transform in the West

Most widely used implementation of FFT (Fast Fourier Transform).

Forward discrete Fourier transform of 1d complex array *X* of size *n*, computes array *Y* of size *n* via:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n}$$

Algorithm scales as *N logN* .

Higher dimensional transforms are straightforward extensions of 1D transform.

Using FFTW with MPI support

```
$ ssh graham
$ salloc  --time=0-01:00 -n 16  -A def-user –mem-per-cpu=4G
$ module spider fftw-mpi
$ module spider fftw-mpi/3.3.8    -> To know which modules to load
$ module load nixpkgs/16.09 intel/2018.3 openmpi/3.1.2 fftw-mpi/3.3.8
$ mpicc -I $EBROOTFFTW/include -L $EBROOTFFTW/lib -lfftw3_mpi -lfftw3
test.c
```

Example: 2D complex forward FFT in parallel.

Data distribution: 1D block along first dimension.

FFTW will decide how to distribute the data, and this information must be extracted from it and used in the program.

Example code from:
http://www.fftw.org/doc/2d-MPI-example.html

```
#include <complex.h>
#include <fftw3-mpi.h>

fftw_complex my_function(ptrdiff_t i_in, ptrdiff_t j_in){
 return 3*i_in+4*j_in*I;  // simple example function
}
```

~syam/ces745/mpi/parallel_fftw/parallel_fftw.c

```c
int main(int argc, char **argv)
{
    const ptrdiff_t N0 = 200, N1 = 200;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(N0, N1, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    data = fftw_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */
    plan = fftw_mpi_plan_dft_2d(N0, N1, data, data, MPI_COMM_WORLD,
                                FFTW_FORWARD, FFTW_ESTIMATE);
    /* initialize data to some function my_function(x,y) */

    for (i = 0; i < local_n0; ++i)
         {
           for (j = 0; j < N1; ++j)
           {
                 data[i*N1 + j] = my_function(local_0_start + i, j);
           }
         }

    /* compute transforms, in-place, as many times as desired */
    fftw_execute(plan);

    fftw_destroy_plan(plan);

    MPI_Finalize();
}
```

The end