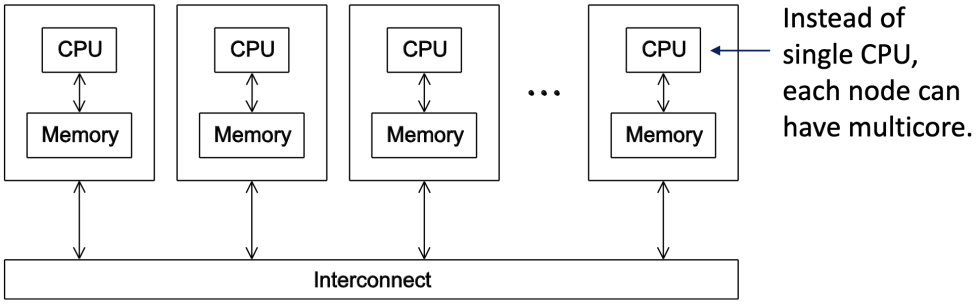
10-14 MPI

Lecture 10 MPI - I

MPI (Message Passing Interface)

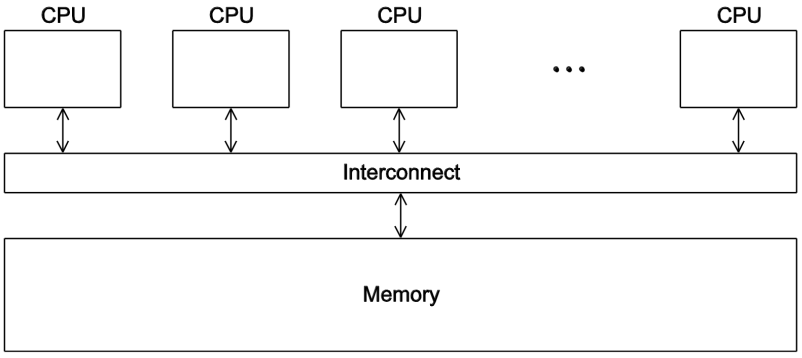


We will talk about processes

Each process can be single threaded or multithreaded.

But for now, we will assume each process is single threaded ... until we learn OpenMP.

We Will Study OpenMP for This



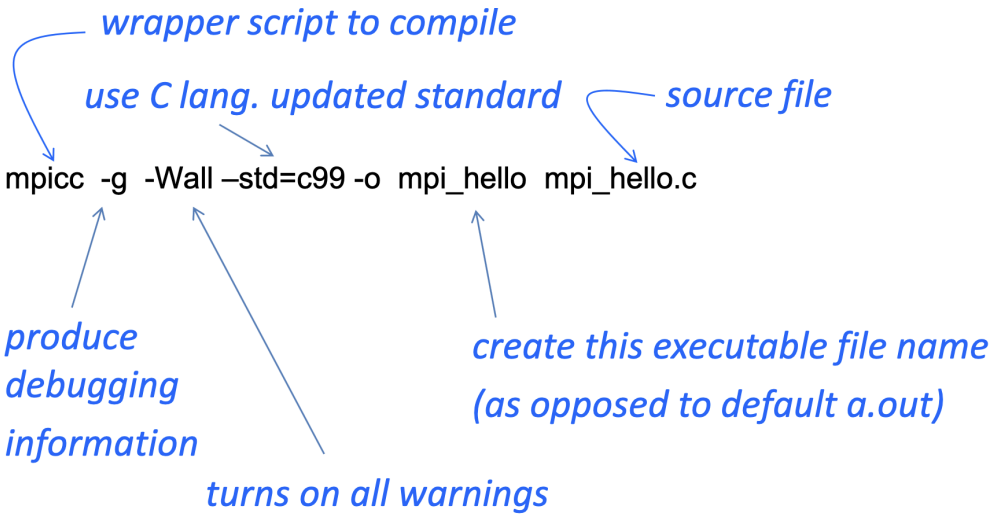
We will talk about Threads

MPI processes

Each process has a unique rank.

p processes are numbered 0, 1, 2, .. p-1

Compilation



MPI is NOT a language.

Just libraries called from C/C++, Fortran, and any language that can call libraries from those.

Need to add mpi.h header file.

Execution

mpiexec -n <number of processes> <executable>

Run with 1 processes: mpiexec -n 1 ./mpi_hello

Run with 4 processes: mpiexec -n 4 ./mpi_hello

You can use mpirun instead of mpiexec and -np instead of -n.

Our first MPI program

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

MPI Components

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

Pointers to the two arguments of main()

Tells MPI to do all the necessary setup.

No MPI functions should be called before this.

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

```
int MPI_Finalize(void);
```

Tells MPI we're done, so clean up anything allocated for this program.

No MPI function should be called after this.

Communicators

A collection of processes that can send messages to each other.

MPI_Init defines a communicator that consists of all the processes created when the program started.

Called MPI_COMM_WORLD.

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);
```

number of processes in the communicator

MPI_COMM_WORLD for now

```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

my rank
(rank of the process making this call)

Communication

```
int MPI_Send(
      void*          msg_buf_p     /* in */,
      int            msg_size      /* in */,
      MPI_Datatype   msg_type      /* in */,
      int            dest          /* in */,
      int            tag           /* in */,
      MPI_Comm       communicator  /* in */);
```

num of elements in msg_buf

type of each element in msg_buf

To distinguish messages

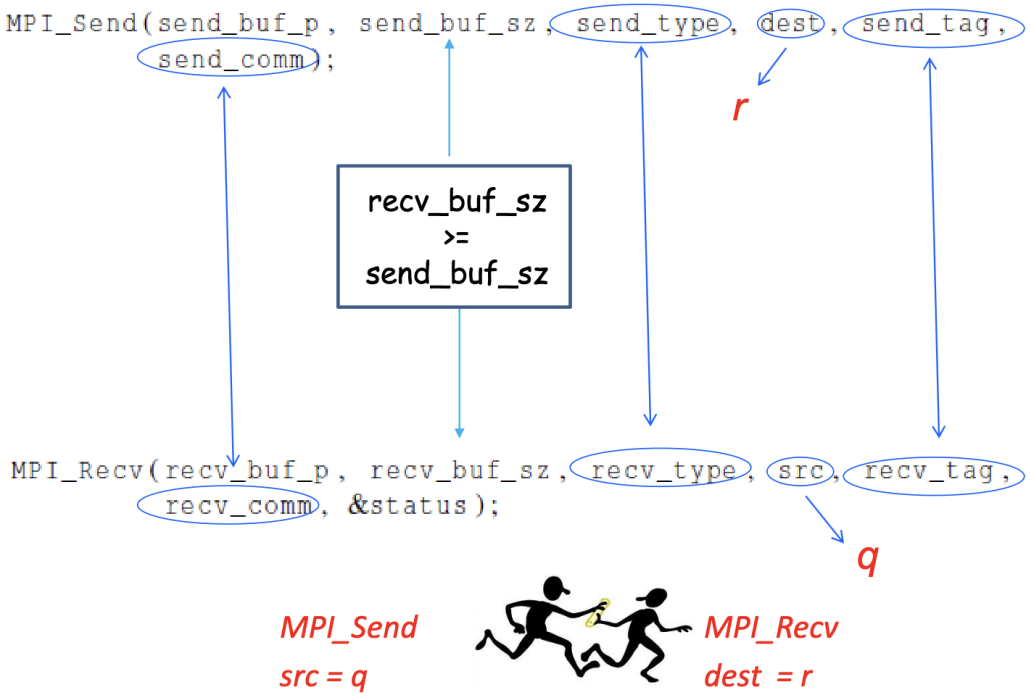
rank of the receiving process

msg_buf_p: pointer to the message

Message sent by a process using one communicator cannot be received by a process in another communicator.

```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

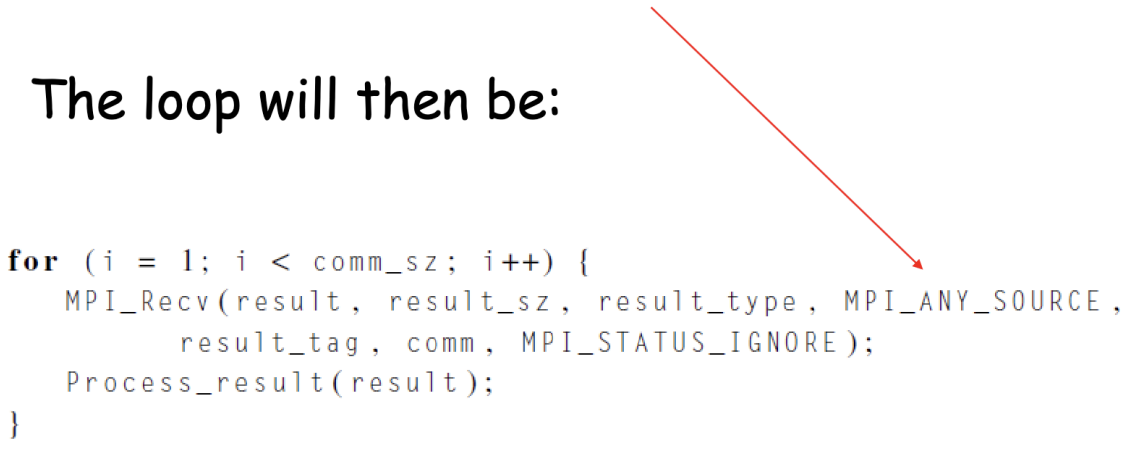
Message matching



MPI wildcard: in order → any order

Wildcard: MPI_ANY_SOURCE

The loop will then be:



```
for (i = 1; i < comm_sz; i++) {
    MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
          result_tag, comm, MPI_STATUS_IGNORE);
    Process_result(result);
}
```

Wildcard: MPI_ANY_TAG

The loop will then be:

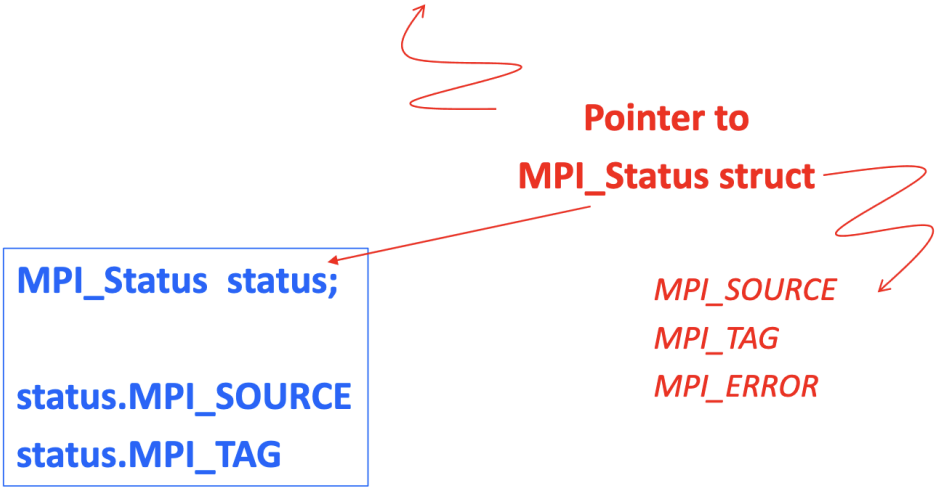
```
for(q = 1; q < comm_sz; q++) {
   MPI_Recv(result, result_sz, result_type,
   q,
   MPI_ANY_TAG, comm,
   MPI_STATUS_IGNORE);
}
```

By using these wildcards, a receiver can get a message without knowing:

- the amount of data in the message,
- the sender of the message,
- or the tag of the message.

Status argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```



How much data am I receiving?

```
int MPI_Get_count(
      MPI_Status*   status_p  /* in  */,
      MPI_Datatype  type      /* in  */,
      int*          count_p   /* out */);
```

Conclusions

MPI is the choice when we have distributed memory organization.

MPI is built around messages

Lecture 11-12 MPI - II

Dealing with I/O

In all MPI implementations, all processes in MPI_COMM_WORLD have access to stdout and sterr.

BUT .. In most of them there is no scheduling of access to output devices. Processes are competing for stdout!

Result: nondeterminism!

How About Input?

Most MPI implementations only allow process 0 in MPI_COMM_WORLD to access to stdin.

If there is some input needed, process 0 must read the data and send to the other processes.

All the processes get the input from the command line though (the arguments of main()).

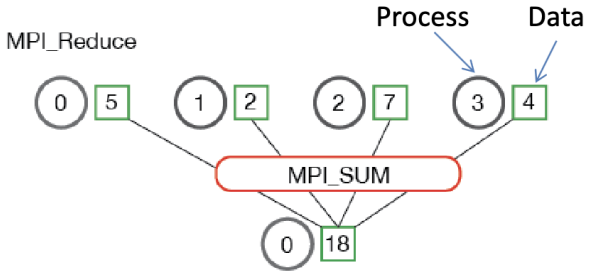
Reduction

Reducing a set of numbers into a smaller set of numbers via a function.

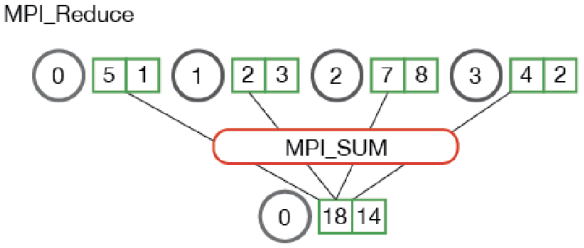
- Example: reducing the group [1, 2, 3, 4, 5] with the sum function → 15

MPI provides a handy function that handles almost all of the common reductions that a programmer needs to do in a parallel application.

You can specify which process receives the result.



Every process has an element



MPI_Reduce

has size:
sizeof(datatype) * count

```
int MPI_Reduce(
      void*          input_data_p    /* in  */,
      void*          output_data_p   /* out */,
      int            count           /* in  */,
      MPI_Datatype   datatype        /* in  */,
      MPI_Op         operator        /* in  */,
      int            dest_process    /* in  */,
      MPI_Comm       comm            /* in  */);
```

only relevant
to dest_process

Examples:

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

MPI_Reduce is called by all processes involved.
This is why it is called **collective call.**

Example of MPI_MAXLOC

```
/* each process has an array of 30 int: num[30] */
   int num[30];
   int  result[30], index[30];
   struct {
      int val;
      int   rank;
   } in[30], out[30];

  for (i=0; i<30; ++i) {
      in[i].val = num[i];
      in[i].rank = myrank;
   }

   MPI_Reduce( in, out, 30, MPI_2INT, MPI_MAXLOC, 0, MPI_COMM_WORLD );

  if (myrank == 0) {
      for (i=0; i<30; ++i) {
         result[i] = out[i].val;
         index[i] = out[i].rank;
      }
   }
```

Collective vs. Point-to-Point Communications

All the processes in the communicator must call the same collective function.

- For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous.

The arguments passed by each process to an MPI collective communication must be "compatible."

- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous.

The output_data_p argument is only used on dest_process. However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it's just NULL.

All collective communication calls are blocking.

Point-to-point communications are matched on the basis of tags and communicators.

Collective communications don't use tags. They're matched solely on the basis of the communicator and the order in which they're called (since they are blocking).

MPI_Allreduce

Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

No destination argument!

```
int MPI_Allreduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      MPI_Comm      comm           /* in  */);
```

Broadcast

Data belonging to a single process is sent to all of the processes in the communicator.

ALL processes in the communicator must call MPI_Bcast()

```
int MPI_Bcast(
      void*         data_p         /* in/out */,
      int           count          /* in     */,
      MPI_Datatype  datatype       /* in     */,
      int           source_proc    /* in     */,
      MPI_Comm      comm           /* in     */);
```

A version of Get_input that uses MPI_Bcast

```
void Get_input(
      int       my_rank  /* in  */,
      int       comm_sz  /* in  */,
      double*   a_p      /* out */,
      double*   b_p      /* out */,
      int*      n_p      /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

Conclusions

A communicator is a collection of processes that can send messages to each other.

Collective communications involve all the processes in a communicator.

When studying MPI be careful of the caveats (i.e. usage that leads to crash, nondeterministic behavior, ...).

Lecture 13 MPI - III

Different partitions of a 12-component vector among 3 processes

Block: Assign blocks of consecutive components to each process.

Cyclic: Assign components in a round robin fashion.

Block-cyclic: Use a cyclic distribution of blocks of components.

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Parallel implementation of vector addition

How will you distribute parts of x[] and y[] to processes?

Scatter

Read an entire vector on process 0

MPI_Scatter sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

data items going to each process

Important:

- All arguments are important for the source process (process 0 in our example)
- For all other processes, only recv_buf_p, recv_count, recv_type, src_proc, and comm are important

send_buf_p

- is not used except by the sender.
- However, it must be defined or NULL on others to make the code correct.
- Must have at least communicator size * send_count elements

All processes must call MPI_Scatter, not only the sender.

send_count the number of data items sent to each process.

recv_buf_p must have at least send_count elements

MPI_Scatter uses block distribution

Gather

MPI_Gather collects all of the components of the vector onto process dest process, ordered in rank order.

```
int MPI_Gather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            dest_proc   /* in  */,
      MPI_Comm       comm        /* in  */);
```

number of elements in send_buf_p

number of elements for any single receive

Important:

- All arguments are important for the destination process.
- For all other processes, only send_buf_p, send_count, send_type, dest_proc, and comm are important

Allgather

Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.

As usual, recv_count is the amount of data being received from each process.

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

Keep in mind ...

In distributed memory systems, communication is more expensive than computation.

Distributing a fixed amount of data among several messages is more expensive than sending a single big message.

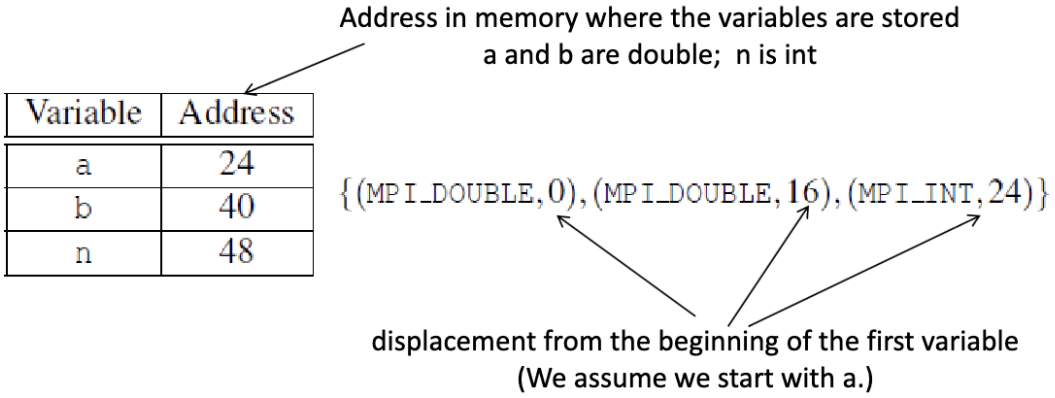
Derived datatypes

Used to represent any collection of data items.

If a function that sends data knows this information about the collection of data items, it can collect the items from memory before they are sent.

A function that receives data can distribute the items into their correct destinations in memory when they're received.

A sequence of basic MPI data types together with a displacement for each of the data types.



MPI_Type create_struct

Builds a derived datatype that consists of individual elements that have different basic types.

```
int MPI_Type_create_struct(
      int              count                          /* in  */,
      int              array_of_blocklengths[]        /* in  */,
      MPI_Aint         array_of_displacements[]       /* in  */,
      MPI_Datatype     array_of_types[]               /* in  */,
      MPI_Datatype*    new_type_p                     /* out */);
```

Number of elements in the type

an integer type that is big enough to store an address on the system.

From the address of item 0

```
int MPI_Get_address(
      void*       location_p   /* in  */,
      MPI_Aint*   address_p    /* out */);
```

Before you start using your new data type

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

When you are finished with your new type

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

This frees any additional storage used.

Example

```
void Build_mpi_type(
      double*         a_p              /* in  */,
      double*         b_p              /* in  */,
      int*            n_p              /* in  */,
      MPI_Datatype*   input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */
```

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

The receiving end can use the received complex data item as if it is a structure.

Measuring time in MPI

We have seen in the past ...

- time in Linux
- clock() inside your code

Elapsed parallel time

Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);

double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

Elapsed time for the calling process

MPI_Wtime() returns wall clock time.

- So, includes any idle time.

clock() returns CPU time.

How to Sync Processes?

MPI_Barrier: Ensures that no process will return from calling it until every process in the communicator has started calling it.

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

Conclusions

Reducing messages sent is a good performance strategy!

- Collective vs point-to-point

Distributing a fixed amount of data among several messages is more expensive than sending a single big message.

Lecture 14 MPI - IV

A parallel sorting algorithm

n keys and p = comm size processes

n/p keys assigned to each process

When the algorithm terminates:

- The keys assigned to each process should be sorted in increasing order.
- If $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

Odd-even transposition sort

A sequence of phases

Even phases, compare swaps:

(a[0], a[1]), (a[2], a[3]), ...

Odd phases, compare swaps:

(a[1], a[2]), (a[3], a[4]), ...

Parallel odd-even transposition sort

Assume P processors (=4) and list n (=16) numbers

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

phase 0 and phase 2

- Processes (0,1) exchange their elements
- Processes (2, 3) exchange their elements
- Processes 0 and 2 keep the smallest 4
- Processes 1 and 3 keep the largest 4

phase 1 and phase 3

- Processes (1, 2) exchange their elements
- Process 1 keeps smallest 4 and process 2 keeps largest 4

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank - 1;
   else                         /* Even rank */
      partner = my_rank + 1;
else                         /* Odd phase */
   if (my_rank % 2 != 0)        /* Odd rank */
      partner = my_rank + 1;
   else                         /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

- Constant defined by MPI
- When used as source/destination in point-to-point comm, no comm will take place.

MPI_Ssend

The extra "s" stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts.

What do we do if we find out that our program is not safe?

- Original:

MPI_Send

MPI_Recv

- Updated:

```
if (my_rank % 2 == 0) {
   MPI_Send
   MPI_Recv
} else {
   MPI_Recv
   MPI_Send
}
```

MPI_Sendrecv

- Carries out a blocking send and a receive in a single call

MPI_Sendrecv_replace

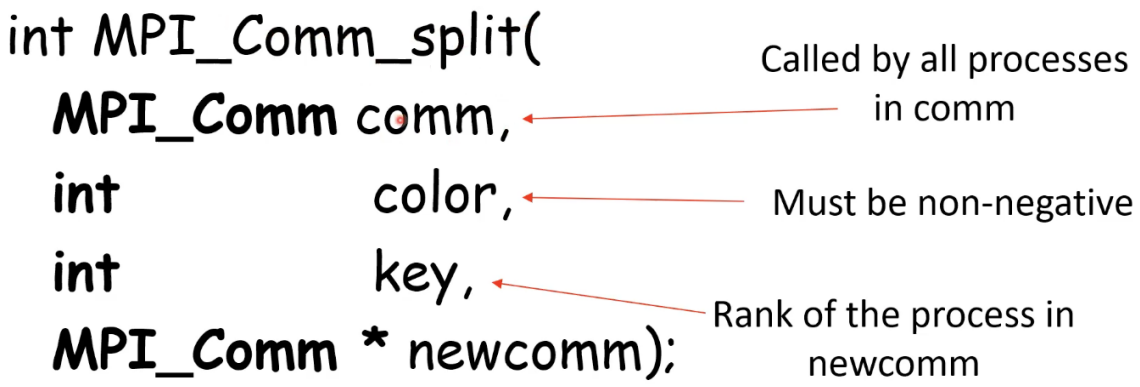
- In this case, what is in buf_p will be replaced by what is received

The communicators

We are familiar with the communicator MPI_COMM_WORLD

Groups allow collective operations to work on a subset of processes

MPI_Comm_split



Partitions the group associated with comm into disjoint subgroups

Processes with the same color will be in the same group

Within each subgroup, the processes are ranked in the order defined by the value of the "key"

- with ties broken according to their rank in the old group

The original communicator does not go away!

If a process uses the color MPI_UNDEFINED, it won't be included in the new communicator.

MPI_Comm_free

```
int MPI_Comm_free(
  MPI_Comm * newcomm);
```

Deallocation of created communicator
Better do it if you are not using the comm again.