An Introduction to MPI

Parallel Programming with the Message Passing Interface

- Chapter 8 - PP
- http://www-unix.mcs.anl.gov/mpi/
- Note: Partially based on: W. Gropp, E. Lusk, An Introduction to MPI,Argonne National Laboratory

Outline

- Background
  - The message-passing model
  - Origins of MPI and current status
  - Sources of further MPI information
- Basics of MPI message passing
  - Hello, World!
  - Fundamental concepts
  - Simple examples in Fortran and C
- Extended point-to-point operations
  - non-blocking communication
  - modes

Outline (cont.)

- Advanced MPI topics
  - Collective operations
  - More on MPI datatypes
  - Application topologies
  - The profiling interface
- Toward a portable MPI environment

What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

MPI Sources

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *MPI: The Complete Reference Vol 1 and 2,*MIT Press, 1998(Fall).
- Other information on Web:
  - at http://www-unix.mcs.anl.gov/mpi/

Companion Material

- Online examples and exercises for our Labs, available at http://www-unix.mcs.anl.gov/mpi/tutorials/perf/
- ftp://ftp.mcs.anl.gov/pub/mpi/mpiexmpl.tar.gz contains source code and run scripts that allows you to evaluate your own MPI implementation

The Message-Passing Model

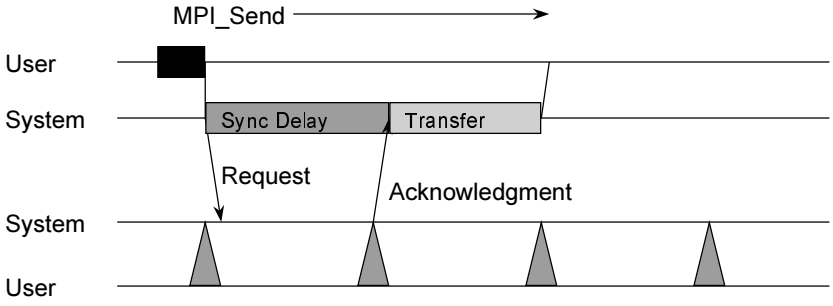
- In the message- passing programming paradigm, we think of a computation as consisting of one or more *processes*.
- Each process has access to some memory (private) and they communicate with each other by sending *messages* through a network. Communication is handled by calls to special MPI subroutines.
- At a given time different processes can be executing either the same or different parts of the program.

The Message-Passing Model (cont.)

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

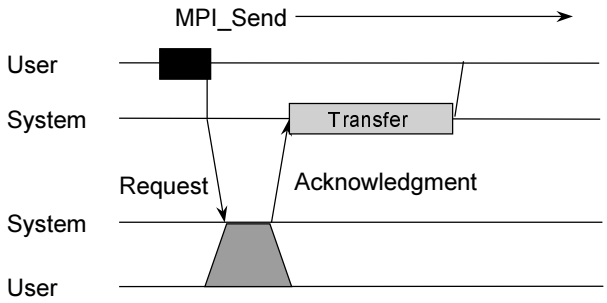
Polling Mode MPI

Message passing is a cooperative method - if the partner doesn't react quickly, a delay results



There is a performance tradeoff caused by reacting quickly - it requires devoting resources to checking for things to do.

Interrupt Mode MPI



- Cost of interrupt higher than polling (usually)

Types of Parallel Computing Models

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (Single Program Multiple Data) is equivalent to MIMD since each MIMD program can be made SPMD for example by a big case statement. (similarly for SIMD, but not in practical sense.)
- Message passing (and MPI) is for MIMD/SPMD parallelism (equal programs for all processes).
- HPF is an example of an SIMD interface.

MPI Programming Model

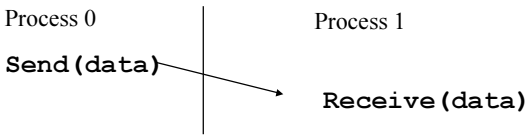
- Most parallel algorithms can be implemented using MPI. Algorithms that create just one task per processor can be implemented directly, with point-to-point or collective communication routines.
- Algorithms that create tasks in a dynamic fashion or that rely on the concurrent execution of several tasks on a processor must be further refined to permit an MPI implementation.
- For example, branch-and-bound search algorithm which creates a tree of ''search'' tasks dynamically must be redesigned, in the way to create a fixed set of worker processes, to be implemented in MPI.

Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.

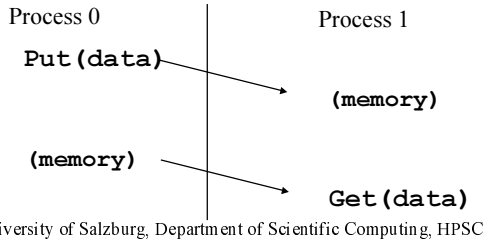
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.



Is MPI Large or Small?

- MPI is large (129 functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions.
- MPI is just right (24 functions)
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

A Minimal MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

- `#include "mpi.h"` provides basic MPI definitions and types
- `MPI_Init` starts MPI, `MPI_Finalize` exits MPI
- Note that all non-MPI routines are local, thus the `printf` run on each process.

A Minimal MPI Program (Fortran)

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Language Bindings

- Much of the discussion about MPI will be language because MPI is a general library specification.
- In concrete program examples some syntax differences will be noticed.
- Different language bindings have slightly different syntax. Sources of syntactic difference include:
- **function names** and mechanism used for **return codes**,
- representation of handles used to access specialised MPI data structures such as `COMMUNICATORS`,
- and the implementation of the `STATUS` datatype returned by `MPI_RECV`. The use of handles hides the internal representation of MPI data structures.
- ***C++*** bindings, and **Fortran-90** issues, are part of MPI-2.

Notes on C Programming Language

- Function names are as in the MPI definition but with only the MPI prefix and the first letter of the function name in upper case: `MPI_Finalize();`
- Status values are returned as integer return codes. The return code may represent an error code or `MPI_SUCCESS` for successful competition.
- Compile-time constants are all in upper case and are defined in the file mpi.h, which must be included in any program that makes MPI calls by `#include "mpi.h"`
- Function parameters with type IN are passed by value, while parameters with type OUT and INOUT are passed by reference (that is, as pointers): `MPI_Comm_size(comm, &size );`
- A status variable has type `MPI_Status` and is a structure with fields `status.MPI_SOURCE` and `status.MPI_TAG` containing source and tag information.

Notes on Fortran

- Fortran language binding, function names are in upper case `call MPI_INIT( ierr )`
- Function return codes are represented by an additional integer argument `ierr`.
- Status values, return code for successful completion, set of error codes, and compile-time constants are all in upper case and are defined in the file mpif.h, which must be included in any program that makes MPI calls by `use MPI`.
- A status variable is an array of integers of size `MPI_STATUS_SIZE`, with the constants `MPI_SOURCE` and `MPI_TAG` indexing the source and tag fields, respectively.

Data Types - C, Fortran

- An MPI datatype is defined for each C datatype:

  `MPI_CHAR, MPI_INT, MPI_FLOAT,`
  `MPI_UNSIGNED_CHAR, MPI_UNSIGNED,`
  `MPI_UNSIGNED_LONG, MPI_LONG,`
  `MPI_DOUBLE, MPI_LONG_DOUBLE,` etc.

- An MPI datatype is defined for each Fortran datatype:

  `MPI_CHARACTER, MPI_INTEGER, MPI_REAL,`
  `MPI_DOUBLE_PRECISION, MPI_COMPLEX,`
  `MPI_LOGICAL,` etc.

Error Handling

- MPI does not provide mechanisms for dealing with failures in the physical communication system and and for handling processor failures.
- MPI programs may exhibit *program errors* when an MPI call is called with an incorrect argument, or *resource errors* when a program exceeds the amount of available system resources.
- By default, an error causes all processes to abort.
- The user may specify that no error is fatal, and handle error codes returned by MPI calls by custom error handlers.

MPICH a Portable MPI Environment

- MPICH is a high-performance portable implementation of MPI (1).
- It runs on MPP's, clusters, and heterogeneous networks of workstations.
- In a wide variety of environments, one can do:

```
configure
make
mpicc -mpitrace myprog.c
mpirun -np 10 myprog
upshot myprog.log
```

to build, compile, run, and analyze performance of your program.

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- The MPI-2 recommends `mpiexec <args>`, but it is not a requirement.
- An example for MPICH implementation of MPI:
  `mpirun -np 1 a.out -mpiversion` returns MPI version
  `mpirun -np 2 first` run program `first` on two processors
  `mpirun -t` shows the commands that `mpirun` would execute
  `mpirun -help` shows all options to `mpirun`

Special Compilation Commands

- The commands

  ```
  mpicc -o first first.c
  mpif77 -o firstf firstf.f
  ```

  may be used to build simple programs when using MPICH implementation of MPI.
- These provide different options that exploit the profiling features of MPI
  - `mpilog` generate log files of MPI calls
  - `mpitrace` trace execution of MPI calls
  - `mpianim` real-time animation of MPI (not available on all systems)
- Other implementations may provide similar commands.

Using Makefiles (*)

- The file ***Makefile.in*** is a template ***Makefile***.
- The program (script) `mpireconfig` translates this to a Makefile for a particular system. This allows you to use the same Makefile for a network of workstations and a massively parallel computer, even when they use different compilers, libraries, and linker options.
- `mpireconfig` ***Makefile*** Note that you must have `mpireconfig` in your PATH.

Using a Single Computer

- If you are testing your program on a single computer you may define several processes and run and test your code. The configuration file machines.LINUX should only contain "localhost".
- Editing

  `vi` ***first.c***
- Compiling and Linking

  `mpicc -o` ***first first.c*** /*first - executable and first.o - object file*/
- Running

  `mpirun -np 2` ***first*** /*"Hello, world!" is printed two times from a single computer if first is hello.c */

Using a Computer Network

- If you are testing your program on the computer network in computer room you may select several computers to perform defined processes and run and test your code. The configuration file machines.LINUX should contain names of computers to be used (I.e. hydra, yapok etc, each in a separate line).
- Editing `vi first.c`
- Compiling and Linking

  `mpicc -o first first.c` /*first - executable and first.o - object file*/
- Running

  `mpirun -np 3 first` /*three times "Hello, world!" is printed, from three different computers, if first is hello.c */

Using MPP Processor

- If you are testing your program on the MPP, composed of 9 processors connected in a tours topology and accessible for HPSC course on: `mreza.ijs.si` you should use the following procedure:
- Logging on: `Username:` `Password:`
- FTP your program to shome directory:

  `scp first.c username@mreza.ijs.si:shome`
- Edit, compile and link in the same way as before:

  `mpicc -o first first.c` /*first - executable and first.o - object file*/
- Run: `mpirun -np 2 first` /*two times "Hello, world!" is printed, from two processors, if first is hello.c */

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A **group and context** together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator `MPI_COMM_WORLD` whose group contains all initial processes, and whose context is default.

Finding Out About the Environment

- Any program should have some functions to control starting and terminating procedures on all processors,
- Two additional questions arise early in a parallel program:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions.

Basic MPI Functions

- `MPI_INIT(int *argc, char ***argv)`

  Initiate a computation. `argc` and `argv` are required only in C language binding, where they are the main program's arguments.
- `MPI_FINALIZE()`

  Shut down a computation. No MPI routines can be called before `MPI_INIT` or after `MPI_FINALIZE`. The one exception is `MPI_INITIALIZED(flag)` which queries if `MPI_INIT` has been called.
- Parameters used by function but not modified (IN-not underlined), not used but may be updated (OUT-underline), or both used and possibly updated by a function (INOUT-underline and italic).

Basic MPI Functions

- `MPI_COMM_SIZE(comm, size )`

  Determine the number of processes in a computation. `comm` communicator(handle); `size` number of processes in the group (if `comm` is `MPI_COMM_WORLD` then number of all processes).

- `MPI_COMM_RANK(comm, pid )`

  Determine the identifier of the current process. `comm` communicator(handle); `pid` process ID in the group of `comm` (it could be `0` to `size-1`).

Better Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

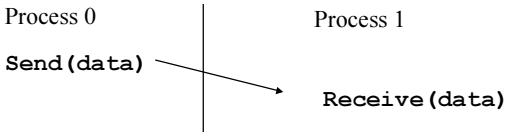
- If all processes can do output we can expect `size` lines.

Better Hello (Fortran)

```
program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

MPI Basic Send/Receive

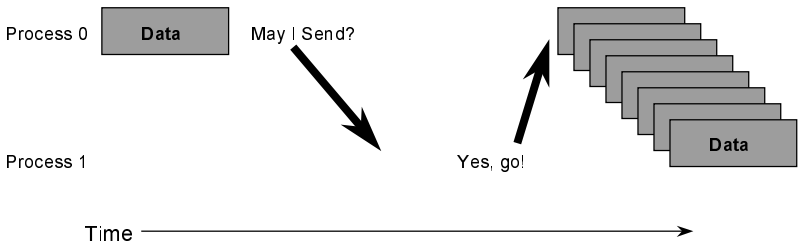
- We need to go a bit more in detail of process-process communication.



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

MPI Basic (Blocking) Send

`MPI_SEND (buf, count, datatype, dest, tag, comm)`

- The message buffer is described by (`buf, count, datatype`).
- The target process is specified by `dest`, which is the rank of the target process in the communicator specified by `comm` and by message `tag` (envelope).
- This is a blocking call which won't complete until there is a matching receive that will empty the buffer.
- When this function returns, the data has been delivered to the system and the buffer can be reused.

MPI Basic (Blocking) Receive

`MPI_RECV(buf,count,datatype,source,tag,comm, status)`

- Waits until a matching message (on `source`, `tag` and `comm`) is received from the system, and the buffer `buf` can be used.
- `source` equals to the rank in communicator specified by `comm`, or it could be `MPI_ANY_SOURCE`.
- Receiving fewer than `count>=0` occurrences of `datatype` is OK, but receiving more is an error.
- `status` contains further information.

MPI Send-receive

- `MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)`

  This function combine in one call the sending of a message to one destination `dest` and the receiving of another message, from another process `source`.
- A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used, then one needs to order them correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock.
- By `MPI_SENDRECV` the communication subsystem takes care of these issues (variant: `MPI_SENDRECV_REPLACE`).

Retrieving Further Information

- `status` is a data *structure* allocated in the user's program.
- In *C*:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status; /* allocate space for current status of receive */
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG,...,&status)
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
/*primarily of use when MPI_ANY_TAG and/or MPI_ANY_SOURCE in receive*/
MPI_Get_count( &status, datatype, &recvd_count);
/* to determine how much data of a particular type was received*/
```

MPI Datatypes

- As MPI does not require communicating processes to use the same representation of a *datatype*, it needs to keep track of possible datatypes.
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., `MPI_INT`, `MPI_DOUBLE_PRECISION`),
  - a contiguous array of MPI datatypes,
  - a strided block of datatypes,
  - an indexed array of blocks of datatypes,
  - an arbitrary structure of datatypes.

Custom Datatypes

- There are MPI functions to construct custom datatypes, such as array of (int, float) pairs, or a row of a matrix stored columnwise.
- As long as you are sure of the size and representation of your data, you can transmit raw data using the datatype `MPI_BYTE`.
- MPI communication never entails type conversions, e.g. from INTEGER to REAL.

MPI or Custom Datatypes?

- Since all data is labeled by type,
  - An MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication),
  - further, porting of parallel programs between machines using different representations of basic datatypes is possible.
- On the other hand, specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

MPI Tags

- Message *tags* provide a further mechanism (beside *source*) for distinguishing between different messages.
- Tag is an integer in the range [0, UB] where UB can be found by querying the predefined constant `MPI_TAG_UB`.
- Messages are sent with an accompanying user-defined tag, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive.

MPI Tags (cont.)

- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.
- When a message posted by a send has been collected by a receive, the message is said to be completed.
- The entire envelope (dest/source, datatype, tag and communicator) must match between the send and receive for the message to complete.
- If a message from any source is acceptable to a receiver, the wildcard `MPI_ANY_SOURCE` can be used in a call to `MPI_RECV`. Similarly, the receive can specify the wildcard `MPI_ANY_TAG` to match any tag.

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "wild card" tags.
- *Contexts* are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

Determinism

- Message-passing programming models are by default nondeterministic: the arrival order of messages sent from two processes, A and B, to a third process, C, is not defined.
- However, MPI does guarantee that two messages sent from one process, A, to another process, B, will arrive in the order sent.
- It is the programmer's responsibility to ensure that a computation is deterministic when (as is usually the case) this is required.

Six basic MPI functions

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`
- Point-to-point (send/recv) isn't the only way...

Simple Fortran Example - 1

```
      program main
      use MPI

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(10)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src  = 0
```


Simple Fortran Example - 2

```
       if (rank .eq. 0) then
         do 10, i=1, 10
           data(i) = i
10       continue

         call MPI_SEND( data, 10, MPI_DOUBLE_PRECISION,
     +                  dest, 2001, MPI_COMM_WORLD, ierr)
       else if (rank .eq. dest) then
         tag = MPI_ANY_TAG
         source = MPI_ANY_SOURCE
         call MPI_RECV( data, 10, MPI_DOUBLE_PRECISION,
     +                  source, tag, MPI_COMM_WORLD,
     +                  status, ierr)
```

Simple Fortran Example - 3

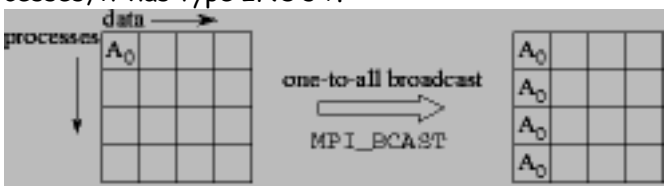
```
call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
+                     st_count, ierr )
   st_source = status( MPI_SOURCE )
   st_tag    = status( MPI_TAG )
   print *, 'status info: source = ', st_source,
+           ' tag = ', st_tag, 'count = ', st_count
 endif

 call MPI_FINALIZE( ierr )
 end
```


Basic Collective Operations in MPI - MPI_ BARRIER

- Collective operations have to be called by all processes in a communicator.
- `MPI_BARRIER (comm)`
  - This function is used to synchronize execution of a group of processes in `comm`. No process returns from this function until all processes have called it.
  - It is the programmer's responsibility to make sure that all processes make a call to `MPI_BARRIER` .
  - A barrier is a simple way of separating two phases of a computation to ensure that messages generated in the two phases do not interfere.
  - In many cases the need for an explicit barrier can be avoided by the appropriate use of tags, source specifiers, and/or contexts.

Basic Collective Operations in MPI - MPI_BCAST

- `MPI_BCAST(`*`inbuf`*`, incnt, intype, root, comm)`
- Implements a one-to-all broadcast operation whereby a single named process (root) sends the same data to all other processes.
- Each process receives this data from the root process. At the time of call, the data are located in *`inbuf`* in process root and consists of `incnt` data items of a specified `intype`.
- After the call, the data are replicated in *`inbuf`* in all processes.
- As *`inbuf`* is used for input at the root and for output in other processes, it has type INOUT.

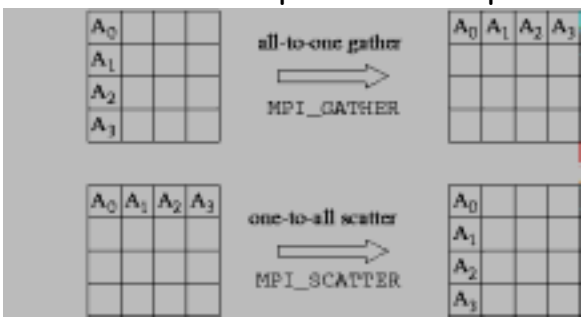


Basic Collective Operations in MPI - MPI_GATHER, MPI_SCATTER

- `MPI_GATHER(inbuf, incnt, intype,` *`outbuf`*`, outcnt, outtype, root, comm)`

  gathers data from all processes to one process
- `MPI_SCATTER(inbuf, incnt, intype,` *`outbuf`*`, outcnt, outtype, root, comm)`

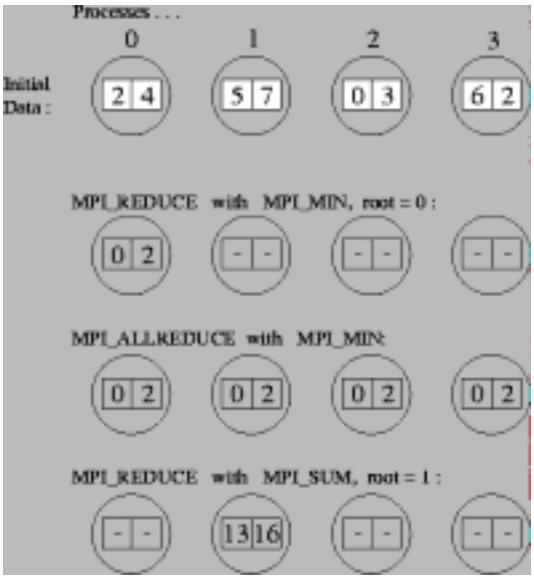
  scatters data from one process to all processes.



Basic Collective Operations in MPI - MPI_REDUCE

- `MPI_REDUCE(inbuf,outbuf,count,type,op, root,comm)`
- combines data from all processes in communicator by operation `op` and returns the result to one process `root`.
- `MPI_ALLREDUCE(inbuf, outbuf, count, type, op)`
- all processes do `MPI_REDUCE` in parallel.
- Valid operations include maximum and minimum (`MPI_MAX` and `MPI_MIN`); sum and product (`MPI_SUM` and `MPI_PROD`); logical and, or, and exclusive or (`MPI_LAND`, `MPI_LOR`, and `MPI_LXOR`); and bitwise and, or, and exclusive or (`MPI_BAND`, `MPI_BOR`, and `MPI_BXOR`).
- In many numerical algorithms, `SEND/RECV` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency.

MPI_REDUCE, MPI_ALLREDUCE



Example: PI in C -1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```


Example: PI in C - 2

```
   h   = 1.0 / (double) n;
     sum = 0.0;
     for (i = myid + 1; i <= n; i += numprocs) {
       x = h * ((double)i - 0.5);
       sum += 4.0 / (1.0 + x*x);
     }
     mypi = h * sum;
     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
     if (myid == 0)
       printf("pi is approximately %.16f, Error is %.16f\n",
              pi, fabs(pi - PI25DT));
   }
   MPI_Finalize();
   return 0;
}
```

MPI Timer

- The elapsed (wall-clock) time between two points in an MPI program can be computed using `MPI_WTIME()`

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "Elapsed time is %f\n", t2 - t1 );
```

- The times are local; the attribute `MPI_WTIME_IS_GLOBAL` may be used to determine if the times are also synchronized with each other for all processes in `MPI_COMM_WORLD`.

Current 16 Functions of Simplified MPI

Basic functions:

- `MPI_INIT, MPI_FINALIZE,`
- `MPI_COMM_SIZE, MPI_COMM_RANK,`
- `MPI_SEND, MPI_RECV,`

Collective communication:

- `MPI_BARRIER,`
- `MPI_BCAST, MPI_GATHER, MPI_SCATTER`
- `MPI_REDUCE, MPI_ALLREDUCE`

Control functions:

- `MPI_WTIME, MPI_STATUS, MPI_INITIALIZED`
- `MPI_GET_COUNT`

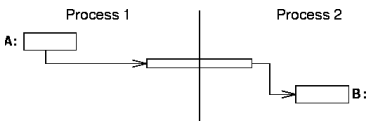
- What else is needed (and why)?

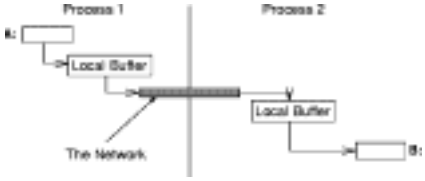
Communication Semantics

- Non- blocking: may return before the operation completes, and before the user is allowed to re-use resources (such as buffers) specified in the call.
- Blocking: return from call indicates that resources can safely be re- used.
- Local: completion depends only on local process.
- Non-local: may require execution of an MPI procedure on another process, or communication with another process.
- Collective: all processes in a group need to invoke the procedure.

Buffering of Messages

- Communication on the left picture uses two buffers. `MPI_SEND` blocks the program and will not return until the message data and envelope have been safely stored away, so that the sender may reuse the send buffer (**blocking mode**).
- Method right implements `MPI_SEND` in the **standard mode**. In this case MPI may buffer outgoing messages and the send call may complete before a matching receive is invoked (additional buffers are needed in the exchange processes,
- So far we have used blocking communication:
  `MPI_SEND` does not complete until buffer is empty (available for reuse).
  `MPI_RECV` does not complete until buffer is full (available for use).





Sources of Deadlocks

- When a process makes a call to `MPI_RECV`, it will wait patiently until a matching *send* is posted.
- If the matching send is never posted, the receive will wait forever.
- In practice, until the system crashes or some time-limit on the job is exceeded.
- If two `MPI_RECV` are issued in the same time on two different processes, they can never finish.

Process 0	Process 1
`Recv(1)`	`Recv(0)`
`Send(1)`	`Send(0)`

Sources of Deadlocks

- If two `MPI_SEND` are issued in the same time on two different processes, they will not finish if `MPI_SENDs` are implemented without buffers.
- If `MPI_SENDs` are implemented with buffering they will finish only if there is enough space for the messages in buffers.

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called "unsafe" because it depends on the `MPI_SEND` implementation and the availability of system buffers.

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:
- Supply receive buffer at same time as send, with `MPI_SENDRECV`
- Use non-blocking operations `MPI_ISEND, MPI_IRECV` with later testing.
- Use explicit `MPI_BSEND` (more buffers!)

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

Communication Modes

- Synchronous mode `MPI_SSEND` does not complete until a matching receive has begun. ("Unsafe" programs become incorrect and usually deadlock within an `MPI_SSEND` .)
- Buffered mode `MPI_BSEND` supplies enough memory to make "unsafe" program safe.
- Ready mode `MPI_RSEND` user guarantees that matching receive has been posted.
- Non-blocking versions: `MPI_ISEND`, `MPI_IBSEND`, `MPI_IRSEND`
- Note that an `MPI_RECV` may receive messages sent with any send mode.

MPI's Non-Blocking Communication

- Non-blocking communication return **immediately** "request handles" that can be waited on and queried:

```
MPI_ISEND(start, count, datatype, dest, tag, comm,
        request)
MPI_IRECV(start, count, datatype, dest, tag, comm,
        request)
```

- Can wait or test for completion with the request handle returned from the non-blocking call.

```
MPI_WAIT(request, status)
MPI_TEST( request, flag, status)
```

- A non- blocking send `MPI_ISEND` immediately followed by wait `MPI_WAIT` is functionally equivalent to a blocking send `MPI_SEND`.

MPI's Non-Blocking Communication (cont.)

- Wait on multiple requests (master/slave program, where the master waits for more slaves' messages

```
MPI_WAITALL(count, array_of_requests,
          array_of_statuses)
MPI_WAITSOME( count, requests, ndone, indices,
           statuses)
```

- Unless using *buffered send*, computation must wait until communication completed – could result in much wasted time.
- If processor can perform useful work while some long communication is in progress, overall time may be reduced - Hiding latency.

MPI's Non-Blocking Communication (exam.)

- For example, in a 2-D finite difference mesh, moving data needed for the boundaries can be done at the same time as computation on the interior.

```
MPI_Irecv( ... each ghost edge ... );
MPI_Isend( ... data for each ghost edge ... );
         ... compute on interior
while (still some uncompleted requests) {
   MPI_Waitany( ... requests ... )
   if (request is a receive)
     ... compute on that edge ... }
```

- Note that we call `MPI_Waitany` several times. After a request is satisfied for some edge, computation can be proceeded on that edge.

Fairness in Message-Passing

A communication is:

- **Non-overtaking**:
  If a sender posts two messages to the same receiver, and a receive operation matches both messages, the message first posted will be chosen.
- **Unfair**:
  No matter how long a send has been pending, it can always be overtaken by a message sent from another process.

Fairness in Message-Passing (exam.)

A parallel algorithm is fair if no process is effectively ignored. In the fragment program,

```
if (rank == 0) { /*process 0 receiving messages from all others
   cnt = 100*(size-1); /*number of all messages */
    for (i=0; i < cnt; i++) {
        MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                 MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        printf( "Msg from %d with tag %d\n",
                status.MPI_SOURCE, status.MPI_TAG );
    }
} else { /*all others send 100 messages to process 0 */
    for (i=0; i<100; i++)
        MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
}
```

processes with lower rank might have an advantage in process 0 by receiving their messages.

Fairness in Message-Passing (cont.)

- MPI makes no guarantees about fairness, however, it provides tools to write efficient, fair programs. First part of the previous program should be replaced by:

```
#define large 128
MPI_Request requests[large];
MPI_Status  statuses[large];
int         indices[large];
int         buf[large]; /*buffer is structured*/
for (i=1;i<size;i++)/*trigger the communication, 1 mess. from each */
   MPI_Irecv( buf+i, 1, MPI_INT, i,
              MPI_ANY_TAG, MPI_COMM_WORLD, &requests[i-1]);
while(cnt > 0) { /*wait for all messages */
 MPI_Waitsome( size-1, requests, &ndone, indices, statuses);
    for (i=0; i<ndone; i++) { /*receive a group of ndone messages*/
        j = indices[i];
        printf( "Msg from %d with tag %d\n",
            statuses[i].MPI_SOURCE, statuses[i].MPI_TAG);
        MPI_Irecv( buf+j, 1, MPI_INT, j+1,
                MPI_ANY_TAG, MPI_COMM_WORLD, &requests[j]);
        cnt = cnt - ndone;
}    }
```

Asynchronous Communication

- The need for asynchronous communication can arise when a computation must access elements of a shared data structure in an unstructured manner.
- One implementation approach is to encapsulate the data structure in a set of specialized data tasks to which read and write requests can be directed.
- This approach is not typically efficient in MPI, however, because of its MPMD programming model.

Asynchronous Communication (cont.)

- An alternative implementation approach is to distribute the shared data structure among the computational processes, which must then poll periodically for pending read and write requests.
- This technique is supported by the `MPI_IPROBE(source, tag, comm, flag, status)` that checks for the existence of pending messages without receiving them, thereby allowing us to write programs that interleave local computation with the processing of incoming messages. If flag is true, the message can then be received by using `MPI_RECV`.

Asynchronous Communication (cont.)

- There are two related functions

  `MPI_PROBE(source, tag, comm, status)`

  blocks until a message of the specified source, tag, and communicator is available, then returns and sets its status argument. The function is used to receive messages for which we have incomplete information.
- The inquiry function, already known from previous slides

  `MPI_GET_COUNT(status, datatype, count)`

  yields the length and status of a message just received.

Asynchronous Communication (Exa.)

```
MPI_COMM_RANK(comm, rank, ierr)
 IF (rank.EQ.0) THEN /* integer */
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
  ELSE IF(rank.EQ.1) THEN /* real */
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
  ELSE   ! rank.EQ.2
    DO i=1, 2
     CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                                comm, status, ierr)
     IF (status(MPI_SOURCE) = 0) THEN /*source 0*/
100  CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status,
  ierr) ELSE /*source 1*/
200  CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status,
  ierr) END IF END DO END IF
```

- Each message is received with the right type.

MPI Communicators

- MPI supports *modular programming* via its communicator mechanism, which provides the information hiding and local name space, needed when building modular programs
- Communicators can be used to implement various forms of *sequential and parallel composition*.
- An MPI communication operation always specifies a communicator which identifies the process *group* that is engaged in the communication operation and the *context* in which the communication occurs.

Communicator = process group + context

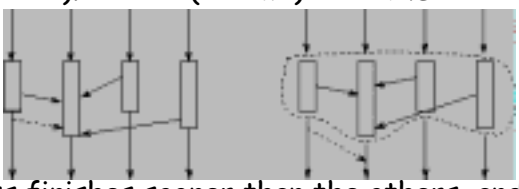
- Different communicators can have same group but not the same context (tag space).

Groups and Context

- Process groups allow a subset of processes to communicate among themselves using local names and to perform collective communication operations without involving other processes.
- The context forms part of the envelope associated with a message (tag space). A receive operation can receive a message only if the message was sent in the same context. Hence, if two routines use different contexts for their internal communication, there can be no danger of their communications being confused.
- Till now, all communication operations have used the default communicator `MPI_COMM_WORLD`, which incorporates *all processes involved* and defines a *default context*.

MPI Context - Figure

- Figure on the left shows a sequential composition of parallel program with an error because two components use the same message tags. Each of the four vertical lines represents a single thread of control (process) in an SPMD program (boxes). All call (arrows) an SPMD library (second box).



- One process finishes sooner than the others, and a message that this process generates during subsequent computation (the dashed arrow) is intercepted by the library. Figure right shows how this problem is avoided by using *contexts*. The library communicates using a distinct tag space, which cannot be penetrated by other messages.

Functions Supporting Modularity

- `MPI_COMM_DUP (comm, newcomm)`

  creates a new communicator comprising the same process group but a new context. Communications performed for different purposes are not confused (see previous slide). This mechanism supports sequential composition.

- `MPI_COMM_SPLIT (comm, color, key, newcomm)`

  creates new communicators comprising subsets of processes of the same `color`. New ranks are assigned according to `key`. This mechanism supports parallel composition. Processes a, b, c, d, oldrank: 0 1 2 3, color=oldrank%2: 0 1 0 1, if `key: 0 0 0 0`, newgroups sorted by newranks: {a, c}, {b, d}, if `key: 7 1 0 3`, newgroups sorted by newranks : {c, a}, {b, d}

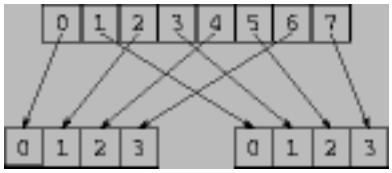
Example-MPI_COMM_SPLIT

```
int main( int argc, char **argv ) {
  ...
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_split( MPI_COMM_WORLD, rank%2, 0, &new_comm );

  MPI_Comm_rank( new_comm, &new_rank );
  printf("Proc %d in MPI_COMM_WORLD has rank %d\
          in new_comm.\n", rank, new_rank );
}
/* output */
Proc 0 in MPI_COMM_WORLD has rank 0 in new_comm.
Proc 1 in MPI_COMM_WORLD has rank 0 in new_comm.
Proc 2 in MPI_COMM_WORLD has rank 1 in new_comm.
Proc 3 in MPI_COMM_WORLD has rank 1 in new_comm.
```


Example - Splitting Processes

- `MPI_COMM_SPLIT` is a collective communication operation, meaning that it must be executed by each process in the process group associated with comm. The following code creates two new communicators:

```
MPI_Comm comm, newcomm; int myid, color;
MPI_Comm_rank(comm, &myid);
color = myid%2;
MPI_Comm_split(comm, color, myid, &newcomm);
```



- Initial communicator of 8 processes is partitioned in two communicators `newcomm` with processes 0, 1, 2, 3.

Master/Slave

- Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a few processes (often just one process) managing the tasks. The manager is called the "master" and the others the "workers" or the "slaves".
- This can be accomplished by dividing the processes in MPI_COMM_WORLD into two sets - the master (who will receive messages and do all of the I/O) and the slaves (who will do some calculation and communication and send messages to the master).
- Eventual communication between slaves, can be implemented in a simple way within slave_comm.

Example-Master/Slave

```
int main( int argc, char **argv ) {
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_split( MPI_COMM_WORLD, rank == 0, 0, &new_comm );
   if (rank == 0) master_io( MPI_COMM_WORLD, new_comm );
    else slave_io( MPI_COMM_WORLD, new_comm );}
/* Master will communicate through MPI_COMM_WORLD. */
master_io( master_comm, MPI_Comm comm )
{ ...
   MPI_Recv( buf, 1, MPI_CHAR, i, 0, master_comm, &status );
   fputs( buf, stdout );}
/* Slaves will communicate through new_comm among themselves
   and through MPI_COMM_WORLD by master. */
slave_io( master_comm, MPI_Comm comm )
{...
  MPI_Send( buf, strlen(buf)+1, MPI_CHAR, 0, 0, comm );
  MPI_Send( buf, 1, MPI_INT, 0, 0, master_comm );}
```

Functions Supporting Modularity (cont.)

- `MPI_INTERCOMM_CREATE(comm, leader, peer, rleader, tag, inter)`

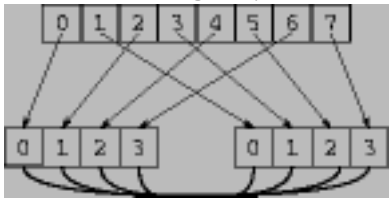
  constructs an inter-communicator `inter`, which links processes of two groups leading by local `leader` and remote `rleader` by `peer` intra-communicator. This mechanism supports parallel composition.
- `MPI_COMM_FREE (comm)`

  releases a communicator created using the preceding three functions.

Example - Intercommunication

- A communicator `newcomm` returned by `MPI_COMM_SPLIT` can be used to communicate within a group of processes - intracommunicator.
- An intercommunicator `intercomm` can created by a collective call executed in the two groups that are to be connected. Processes in the two groups must each supply a local `comm` intracommunicator that identifies the processes involved in their group. They must also agree on the identifier of a ''leader'' process in each group and a parent communicator `peercomm` that contains all the processes in both groups.

```
MPI_INTERCOMM_CREATE (comm,
     local_leader, peercomm,
      remote_leader, tag,
         intercomm)
```



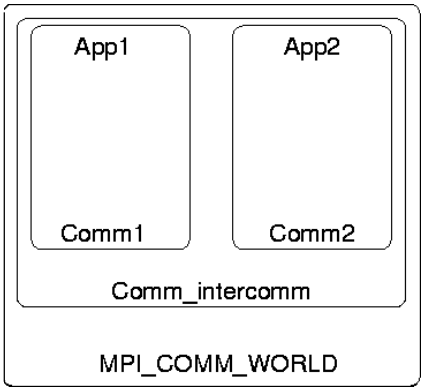
Sending Messages Between Different Programs (exam.)

- Programs share `comm = MPI_COMM_WORLD` and have separate communicators.
- App1 /*create and send*/

```
MPI_INTERCOMM_CREATE(Comm1,
  app1_leader,comm, app2_leader,
  MPI_ANY_TAG,Comm_intercomm);
MPI_Send(buf,...,0,intercomm);
```

- App2 /*create receive and local broadcast*/

```
MPI_INTERCOMM_CREATE(Comm2,
  app2_leader,comm, app1_leader,
  MPI_ANY_TAG,Comm_intercomm);
MPI_Recv(buf,...,0,intercomm,..);
MPI_Bcast(buf, ..., Comm2);
```



MPI Datatypes

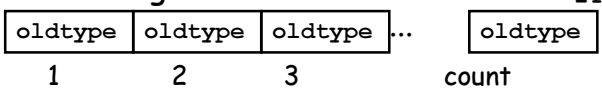
- *Elementary*:
  Language-defined types (e.g., `MPI_INT` or `MPI_COMPLEX`)
- *Vector*:
  Separated by constant ''stride''
  - Contiguous: Vector with stride of one element
  - Hvector: Vector, with stride in bytes
- *Indexed*:
  Array of indices (for scatter/gather)
- *Hindexed*:
  Indexed, with indices in bytes
- *Struct*:
  General mixed types (for C structs etc.)

Derived Datatypes

- Till now, MPI routines have been used to communicate elementary datatypes, such as integers and reals, or arrays of these types.
- MPI user can create derived types at run-time, a mechanism allowing noncontiguous data elements (structures, vectors with non-unit stride, etc.) to be grouped together in a message.
- This mechanism permits us to avoid data copy operations. Without it, the sending of a row of a two-dimensional array stored by columns would require that these noncontiguous elements be copied into a buffer before being sent.

Derived Datatypes Constructors

- `MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)`
  constructor is used to define a `newtype` comprising `count` contiguous data elements of `oldtype`.



- `MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)`
  defines a `newtype` comprising `count` blocks containing `blocklen` data elements of `oldtype` separated by a constant `stride` in an array.
- A call to `MPI_TYPE_CONTIGUOUS(count, oldtype,newtype)` is equivalent to a call to `MPI_TYPE_VECTOR(count,1,1, oldtype, newtype)` or to a call to `MPI_TYPE_VECTOR(1,count,n,oldtype,newtype)`, n arbitrary.

Vector Example

Suppose we have an array of 35 integers arranged as shown below.

To specify a row we can use:

```
MPI_Type_contiguous( 7, MPI_INT, &newtype);
```

To specify a column we can use:

```
MPI_Type_vector( 5, 1, 7, MPI_INT, &newtype);
```

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

Derived Datatypes (cont.)

- `MPI_TYPE_COMMIT (type)` is the commit function that must be applied to a derived `type` before it can be used in a communication operation.
- `MPI_TYPE_FREE (type)` should be applied to a derived `type` after use, to reclaim storage.
- Using previous example for specifying columns, we should use in our program the following statements:

```
MPI_Type_vector( 5,1,7, MPI_DOUBLE, &column);
MPI_Type_commit ( &column );
....
MPI_Typr_Free (&column);
```

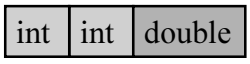
More General Derived Datatypes

- `MPI_TYPE_STRUCT(count, blocklens, indices, oldtypes, newtype)`
  defines a `newtype` comprising `count` blocks of length `blocklens` separated by relative displacements in `indices`.
- Is the most general type constructor. It further generalises the previous one in that it allows each block to consist of replications of different datatypes.

Example - Sharing a Structure

- Have your program read 2 integers and 1 double-precision value from standard input (from process 0), and communicate this to all of the other processes.

```
int rank;
struct { int a; int b; double c } value;
```



We will pack these data into MPI_Datatype `mystruct` composed of first block of integers and second block of doubles.

```
MPI_Datatype mystruct;  /* newtype */
int blocklens[2];/*lengths of a block in structure */
MPI_Aint indices[2]; /* relative addresses */
MPI_Datatype old_types[2]; /* old data types */
```

Example - Sharing a Structure (cont.)

```
/* Two values of first oldtype and one value second*/
  blocklens[0] = 2; blocklens[1] = 1;
/* The base types */
  old_types[0] = MPI_INT; old_types[1] = MPI_DOUBLE;
/* The locations of each element */
  MPI_Address( &value.a, &indices[0] );
  MPI_Address( &value.c, &indices[1] );
/* Make relative */
    indices[1] = indices[1] - indices[0];
    indices[0] = 0;
  MPI_Type_struct( 2, blocklens, indices, old_types,
                   &mystruct );
  MPI_Type_commit( &mystruct ); ... /* later comm. */
  MPI_Bcast( &value, 1, mystruct, 0, MPI_COMM_WORLD);
```


Extending futures in the MPI-2

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/ Fortran-90; interlanguage issues

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

When *not* to use MPI

- Regular computation matches HPF
- Solution (e.g., library) already exists
  http://www.mcs.anl.gov/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.

Summary

- The parallel computing community has cooperated on the development of a standard for message-passing library - MPI.
- There are many implementations of MPI, on nearly all platforms.
- MPI subsets are easy to learn and use.
- Lots of MPI material is available:

  http://www-unix.mcs.anl.gov/mpi/

  Exercises:

  http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/

Labs Exercises

- Study, download, compile and run exercises 1.-6.:

  http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/

1. Getting started with "Hello World"
2. Sharing data; Using MPI datatypes to share data
3. Sending in a ring (broadcast by ring)
4. Finding PI using MPI collective operations
5. Fairness in message passing and related exercise: Implementing Fairness using Waitsome
6. A simple Jacobi iteration
7. Master/slave programs and related exercise: Simple I/O server in MPI.