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
- 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
- Other information on Web:
  - at http://www-unix.mcs.anl.gov/mpi/
- 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

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, `stdio.h` is needed because of `printf`.
- `MPI_Init` starts MPI, `MPI_Finalize` exits MPI
- Note that all non-MPI routines are local, thus the `printf` run on each process.

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 (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.

Data Types - C, Fortran

- MPI datatypes are defined for every C datatype:
  `MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_UNSIGNED_CHAR, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG, MPI_DOUBLE, MPI_LONG_DOUBLE`, etc.
- MPI datatypes are defined also for every 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. (Other implementations exists: LAM, ...)
- 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
```

  to build, compile, and run your program.
- To all above MPI commands `".mpich"` should be added, because LAM is default MPI.

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program.
- In general, starting an MPI program is dependent on the implementation of MPI, and might require various scripts, 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.mpich -np 1 a.out -mpiversion` returns MPI version
  `mpirun.mpich -np 2 first` run program `first` on two processess
  `mpirun.mpich -t` shows the commands that `mpirun` would execute
  `mpirun.mpich -help` shows all options to `mpirun`

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 `/etc/mpich/machines.local` should have the following structure:

```
localhost
localhost
....
localhost
```

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.
- Editing

  `vi first.c`
- Compiling and Linking

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

  `mpirun.mpich -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 you may select several computers to perform defined processes and run and test your code.
- The configuration file on babuin `/etc/mpich/machines.LINUX`
- should contain names of computers to be used (i.e.:

```
babuin
banteng
barbe
barsoi
...
```

etc., each in a separate line, and with the first name belonging to the name of the local host).

Using a Computer Network

- Now, you can test your program on listed computers in the network that will run defined processes.
- Editing

  `vi first.c`
- Compiling and Linking on agutis:

  `mpicc.mpich -o first first.c` /*first - executable and first.o - object file saved on the shared diskspace.*/
- Running

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

  `mpirun.mpich -machinefile /etc/mpich/machine.LINUX -p 2 hello`

Using MPP Processor

- You can run your program on the MPP, composed of 32 processors connected in a tours topology and accessible for HPSC course on: kocka**.ijs.si** you should use the following procedure:
- Logging on: `Username:` `Password:`
- FTP your program to shome directory:

  `scp first.c username@kocka.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 9 first` /*nine times "Hello, world!" is printed, from nine 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)`
  Initiates a computation. `argc` and `argv` are required only in C language binding, where they are the main program's arguments.
- `MPI_FINALIZE()`
  Shuts 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;
}
```

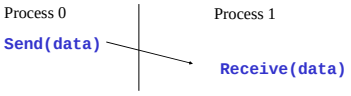
- If all processes can do output we can expect `size` lines.

The Message-Passing Model

- *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.

MPI Basic Send/Receive

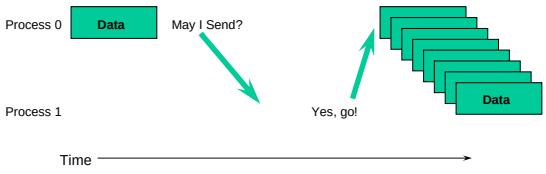
- We need to go a bit more in detail of process-process communication.



- 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.
- 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 or Custom Datatypes?

- Since all data is labeled by MPI types,
  - An MPI program 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 datatypes
  - 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.)

- 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.

Six basic MPI functions

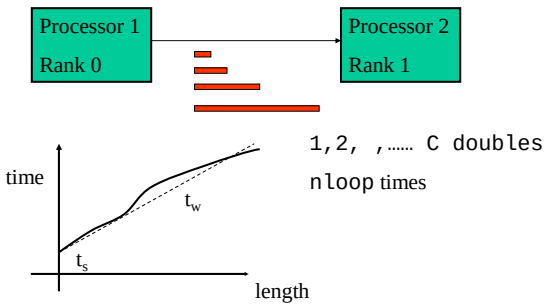
- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`
- Point-to-point (send/recv) isn't the only way...

MPI Timer

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

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


Measuring Bandwidth - 0



Measuring Bandwidth - 1

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

#define NUMBER_OF_TESTS 5  /* Number of tests for more
   reliable average.*/

int main( argc, argv )
int argc;
char **argv;
{
    double       *buf;
    int          rank, numprocs;
    int          n;
    double       t1, t2, tmin;
    int          j, k, nloop;
    MPI_Status   status;

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

Measuring Bandwidth - 2

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 if (numprocs != 2) {
    printf("The number of processes has to be two!\n");
    return(0);
 }
 MPI_Comm_rank( MPI_COMM_WORLD, &rank );
 if (rank == 0)
     printf( "Kind\t\tn\ttime (sec)\tRate (Mb/sec)\n");
 for (n=1; n<110000; n*=2) { /* Message lengths doubles
                                each time */
     if (n == 0) nloop = 100;
     else        nloop  = 100/n;
     if (nloop < 1) nloop = 1;
    buf = (double *) malloc( n * sizeof(double) );
    if (!buf) {
      fprintf( stderr,
        "Could not allocate send/recv buffer of size
          %d\n", n );
      MPI_Abort( MPI_COMM_WORLD, 1 );
    }
```


Measuring Bandwidth - 3

```
tmin = 1000;
for (k=0; k<NUMBER_OF_TESTS; k++) {
  if (rank == 0) {
    /* Make sure both processes are ready */
    MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
        MPI_BOTTOM,0,MPI_INT,1,14,MPI_COMM_WORLD,&status);
    t1 = MPI_Wtime();
    for (j=0; j<nloop; j++) {/*Send message nloop times.*/
     MPI_Send( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
     MPI_Recv( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD,
                    &status );
    }
    t2 = (MPI_Wtime() - t1) / nloop;
    if (t2 < tmin) tmin = t2;
  }
  else if (rank == 1) {
       /* Make sure both processes are ready */
```


Measuring Bandwidth - 4

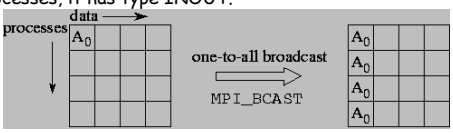
```
   MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
      MPI_BOTTOM, 0,MPI_INT, 0,14, MPI_COMM_WORLD,&status);
   for (j=0; j<nloop; j++) {
      MPI_Recv(buf,n,MPI_DOUBLE,0,k, MPI_COMM_WORLD, &status );
      MPI_Send( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
  } } }
  /* Convert to half the round-trip time */
  tmin = tmin / 2.0;
  if (rank == 0) {  double rate;
    if(tmin>0) rate=n*sizeof(double)*1.0e-6*8/tmin;/*in Mb/sec*/
    else  rate = 0.0;
    printf( "Send/Recv\t%d\t%f\t%f\n", n, tmin, rate );
  }
  free( buf );}
MPI_Finalize( );
return 0;}
```


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.
  - 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.

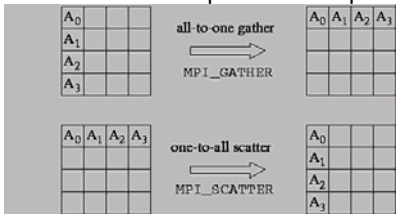


Basic Collective Operations in MPI - MPI_GATHER, MPI_SCATTER

- `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)`

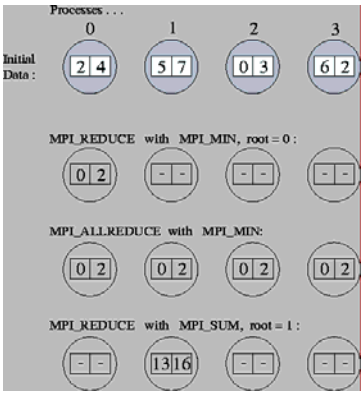
  scatters data from one process to all processes.



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



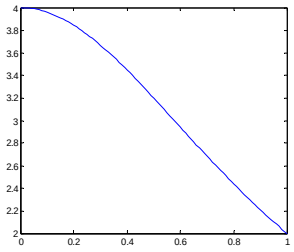
Example: Calculating PI -0

$$\int_0^1 4/(1+x^2) = \pi$$

If n is the number of intervals, and p the number of processors, each processor calculates its own sum

$(1/(n/p))^* [4/(1+x^2)]$

The global sum is then approximately equal to PI.



Example: Calculating PI -1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264643;
   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: Calculating PI -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;
}
```


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.

Communication Modes

- Synchronous mode `MPI_SSEND` does not complete until a matching receive has begun.
- 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.

Fairness in Message-Passing

- 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.
- The MPI 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.
- It is the programmer's responsibility to ensure that a computation is deterministic when (as is usually the case) this is required.

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_SEND`s are implemented without buffers.
- If `MPI_SEND`s 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:

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

- Supply receive buffer at same time as send, with `MPI_SENDRECV`

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

- Use non-blocking operations `MPI_ISEND, MPI_IRECV` with later testing.
- Use explicit `MPI_BSEND` (more buffers!)

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

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

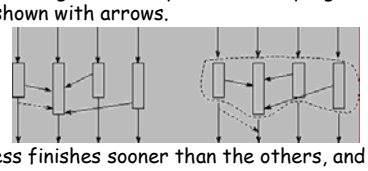
- Different communicators can have same group but not the same context (tag space).

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 is 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 (process) of a program (boxes). Calls are shown with arrows.



- 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 color: {a, c}, {b, d}, if `key:7 1 0 3,` newgroups-color, sorted as keys : {c, a}, {b, d}

Example-MPI_COMM_SPLIT

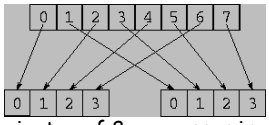
```
int main( int argc, char **argv ) {
  ...
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_split( MPI_COMM_WORLD, rank%2, 0, &new_comm );

  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.