Lecture 8

MPI

Outline

- Introduction to HPC computing
- OpenMP
- MPI
  - Introduction
  - Understanding communications
  - Collective communications
  - Communicators
  - Topologies
  - Grouping Data for Communication
  - Input / output
  - Design and Coding of Parallel Programs
  - Parallel libraries

Additional material

- The material in these slides should be sufficient, but if you have questions you can consult this book:

  “Parallel Programming with MPI” by Peter Pacheco

  or any other book on MPI programming.

- Very useful web resource (lists all MPI functions with example codes):

  http://mpi.deino.net/mpi_functions

What is MPI?

Message Passing Interface

Language-independent communications protocol.

Portable, platform independent, de facto standard for parallel computing on distributed memory systems.

Various implementations exist (MPICH, Open MPI, LAM, vendor versions).

Many popular software libraries have parallel MPI versions.

Principal drawback: it is very difficult to design and develop programs using message passing.

“assembly language of parallel computing”

MPI = MPI-1 (1994) - standard “original” version
MPI-2 (1997) - superset of MPI-1, various useful extensions
MPI-3 (2012) – even more extensions
Extensions not part of standard also available

MPI is not a new programming language.

It is a collection of functions and macros, or a library that can be used in C programs (also C++, Fortran etc.).

Most MPI programs are based on SPMD model - Single Program Multiple Data. This means that the same executable in a number of processes, but the input data makes each copy compute different things.

All MPI identifiers begin with MPI_ (C++ bindings are depreciated as of MPI-3)

Each MPI function returns an integer which is an error code, but the default behavior of MPI implementations is to abort execution of the whole program if an error is encountered. This default behavior can be changed.

Preliminaries

A process is an instance of a program.

Processes can be created and destroyed.

MPI assumes statically allocated processes[*] - their number is set at the beginning of program execution, no additional processes created (unlike threads).

Each process is assigned a unique number or rank, which is from *0* to *p-1*, where *p* is the number of processes.

Number of processes is not necessarily number of processors; a processor may execute more than one process.

Generally, to achieve the close-to-ideal parallel speedup (i.e. *n* times faster with *n* processes), each process must have exclusive use of one processor core. Possible on a multicore processor if *p* does not exceed number of cores.

Running MPI programs with one processor core is fine for testing and debugging, but of course will not give parallel speedup.

Blocking communication

Assume that process 0 sends data to process 1.

In a blocking communication, the sending routine returns only after the buffer it uses is ready to be reused.

Similarly, in process 1, the receiving routine returns after the data is completely stored in its buffer.

Blocking send and receive: MPI_Send and MPI_Recv .

MPI_Send: sends data; does not return until the data have been safely stored away so that the sender is free to access and overwrite the send buffer.

The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

MPI Recv: receives data; it returns only after the receive buffer contains the newly received message.

Message structure

Each message consists of two parts:

1. Data transmitted

2. Envelope, which contains:

- rank of the receiver
- rank of the sender
- a tag
- a communicator

Receive does not need to know the exact size of data arriving but it must have enough space in its buffer to store it.

MPI program structure

- Include mpi.h

- Initialize MPI environment (MPI_Init)

- Do computations

- Terminate MPI environment (MPI_Finalize)

MPI program structure

```
#include "mpi.h"
int main(int argc, char* argv[])
{
/* ... */

/* This must be the first MPI call */
 MPI_Init(&argc, &argv);

/* Do computation */

 MPI_Finalize();
/* No MPI calls after this line */

/* ... */

  return 0;
}
```

MPI program structure - quick Fortran example

Note the differences in the way MPI functions called. Almost all Fortran MPI calls have additional ierr argument, which has the same meaning as the return value of the function in C

```
       program main
       include "mpif.h"
       integer ierr

c ...

c This must be the first MPI call
       call MPI_INIT(ierr)
c  Do computation

       call MPI_FINALIZE(ierr)
c No MPI calls after this line

c ...
       stop
       end
```

MPI_Send

int MPI_Send (void * *buf*, int *count*, MPI_Datatype *datatype*, int *dest* , int *tag* , MPI_Comm *comm*)

---

buf - beginning of the buffer containing the data to be sent

count - number of elements to be sent (not bytes)

datatype - type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR

dest - rank of the process, which is the destination for the message

tag - number, which can be used to distinguish among messages

comm - communicator: a collection of processes that can send messages to each other, e.g. MPI_COMM_WORLD (all the processes running when execution begins)

Returns error code.

Predefined MPI datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
etc.	
MPI_BYTE*	no direct C equivalent
MPI_PACKED*	no direct C equivalent

In addition, user-defined datatypes are possible.

MPI_Recv

int MPI_Recv (void * *buf*, int *count*, MPI_Datatype *datatype*, int *source* , int *tag* , MPI_Comm *comm*, MPI_Status **status*)

---

buf - beginning of the buffer where data is received

count - number of elements to be received (not bytes)

datatype - type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR

source - rank of the process from which to receive message

tag - number, which can be used to distinguish among messages

comm - communicator

status - information about the data received, e.g. rank of source, tag,
error code. Replace with MPI_STATUS_IGNORE if never used.

Returns error code.

Wildcards are possible for source and tag (MPI_ANY_SOURCE, MPI_ANY_TAG).

MPI_Comm_rank

int MPI_Comm_rank (MPI_Comm *comm*, int **rank*)

comm - communicator

rank - of the calling process in group comm

MPI_Comm_size

int MPI_Comm_size (MPI_Comm *comm*, int **size*)

comm - communicator

size - number of processes in group comm

First Program

Adapted from P. Pacheco, *Parallel Programming with MPI*

```
/* greetings.c
 * Send a message from all processes with rank != 0
 * to process 0. * Process 0 prints the messages received.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
  int          my_rank;       /* rank of process      */
  int          p;             /* number of processes  */
  int          source;        /* rank of sender       */
  int          dest;          /* rank of receiver     */
  int          tag = 0;       /* tag for messages     */
  char         message[100];  /* storage for message  */
  MPI_Status   status;        /* status for receive   */
```

/home/syam/ces745/mpi/mpi_basics

```
/* Start up MPI */
MPI_Init(&argc, &argv);

/* Find out process rank  */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &p);


if (my_rank != 0)
  {
    /* Create message */
    sprintf(message, "Greetings from process %d!",
               my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
                    dest, tag, MPI_COMM_WORLD);
  }
  else
    { /* my_rank == 0 */
      for (source = 1; source < p; source++)
        {
          MPI_Recv(message, 100, MPI_CHAR, source, tag,
                          MPI_COMM_WORLD, &status);
          printf("%s\n", message);
        }
    }

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
}
```

Compilation and execution

Ubuntu Linux example:

1. Install necessary packages: mpich-bin, libmpich1.0-dev .

2. Make sure ssh client and server installed.

3. ssh localhost - needs to work without password. To set up keys:

```
$  ssh-keygen -t rsa
$  cd .ssh
$  cp id_rsa.pub authorized_keys
```

4. compile code with:

```
$  mpicc greetings.c -o test.x
```

5. Execute:

```
$  mpirun -np 8 ./test.x
```

-np option specifies number of processes, which will all run on localhost.

Compilation and execution

SHARCNET example:

1. After getting an account, choose a suitable cluster, log in, then reserve a node with salloc command:

```
$  salloc --time=0-03:00 -n 4 -A def-account --mem-per-cpu=1G
```

2. compile code with:

```
$  mpicc greetings.c -o test.x
```

mpicc is a script, use it with -v option to see what is actually executed.

3. On interactive (obtained with salloc) nodes only, you can run mpi code directly (bypassing the scheduler), e.g.:

```
$  mpirun -np 8 ./test.x
```

Submitting a job

- Production runs on our clusters should be submitted as batch (non-interactive) jobs to the scheduler.
- On national systems (graham/cedar/...):

```
  $  sbatch job_mpi.sh
```

where the file job_mpi.sh contains

```
#!/bin/bash
#SBATCH -t 0-01:00
#SBATCH --mem-per-cpu=4G
#SBATCH -n 4
#SBATCH -A def-user

srun  ./mpi_code
```

https://docs.alliancecan.ca/wiki/Running_jobs

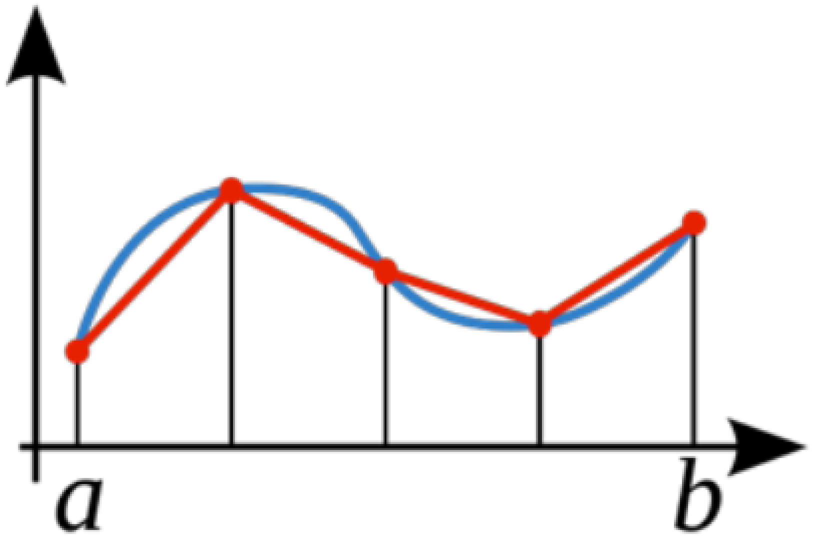
Example: Numerical integration

Trapezoid rule for integrating $\int_a^b f(x)dx$

with $h = (b-a)/n$ is:

$$f(x) \approx \frac{h}{2}\big(f(x_0) + f(x_n)\big) + h\sum_{i=1}^{n-1} f(x_i),$$

where $x_i = a + ih,\ i = 0, 1, \ldots, n$



Given p processes, each process can work on n/p intervals

Note: for simplicity will assume n/p is an integer

process	interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
$\vdots$	
$p-1$	$[a + (p-1)\frac{n}{p}h, b]$

Parallel trapezoid integration

Assume $f(x) = x^2$

Of course could have chosen any desired (integrable) function here.

Write function f(x) in

```
/* func.c */

float f(float x)
{
  return x*x;
}
```

~syam/ces745/mpi/mpi_basics

Trapezoid rule

```c
/* traprule.c */

extern float f(float x); /* function we're integrating */

float Trap(float a, float b, int n, float h)
{
  float integral;   /* Store result in integral  */
  float x;
  int i;

  integral = (f(a) + f(b))/2.0;

  x = a;
  for ( i = 1; i <= n-1; i++ )
    {
      x = x + h;
      integral = integral + f(x);
    }

  return integral*h;
}
```

```c
/* trap.c -- Parallel Trapezoidal Rule
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 *        The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024) */


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

extern float Trap(float a, float b, int n, float h);
```

```
int main(int argc, char** argv)
{
  int          my_rank;   /* My process rank           */
  int          p;         /* The number of processes   */
  float        a = 0.0;   /* Left endpoint             */
  float        b = 1.0;   /* Right endpoint            */
  int          n = 1024;  /* Number of trapezoids      */
  float        h;         /* Trapezoid base length     */
  float        local_a;   /* Left endpoint my process  */
  float        local_b;   /* Right endpoint my process */
  int          local_n;   /* Number of trapezoids for  */
                          /* my calculation            */
  float        integral;  /* Integral over my interval */
  float        total=-1;  /* Total integral            */
  int          source;    /* Process sending integral  */
  int          dest = 0;  /* All messages go to 0      */
  int          tag = 0;
  MPI_Status   status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  h = (b-a)/n;    /* h is the same for all processes */
  local_n = n/p;  /* So is the number of trapezoids */

  /* Length of each process' interval of  integration = 
local_n*h. */
  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  integral = Trap(local_a, local_b, local_n, h);

  /* Add up the integrals calculated by each process */
  if (my_rank == 0)
    {
      total = integral;
      for (source = 1; source < p; source++)
        {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                          MPI_COMM_WORLD, &status);
          printf("PE %d <- %d,   %f\n", my_rank,source,
                    integral);
          total = total + integral;
        }
    }
  else
    {
      printf("PE %d -> %d,   %f\n", my_rank, dest, integral);
      MPI_Send(&integral, 1, MPI_FLOAT, dest,
                      tag, MPI_COMM_WORLD);
    }

  /* Print the result */
  if (my_rank == 0)
    {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %f\n",
                 a, b, total);
    }

  MPI_Finalize();

  return 0;
}
```

I/O

We want to read *a*, *b*, and *n* from the standard input.

Function Get_data reads *a*, *b* and *n.*

Cannot be called in each process.

Process 0 calls Get_data, which sends these data to processes
1,2,...,*p*-1.

The same scheme applies if we read from a file.

Most supercomputing clusters are equipped with parallel storage hardware so writing to or reading from files need not be coded in parallel by the user explicitly.

It is possible for each process to read to or write from file, as long as they don’t interfere with each other (i.e. it’s best to use a separate file for each process).

All processes can write to standard output as already illustrated.

```
/* getdata.c


 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes.
 */

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

```
void Get_data( float* a_ptr, float* b_ptr, int* n_ptr,
               int my_rank, int p )
{
  int source = 0, dest, tag;
  MPI_Status status;

  if (my_rank == 0)
    {
      printf("Rank %d: Enter a, b, and n\n", my_rank);
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

      for (dest = 1; dest < p; dest++)
        {
          Tag = 0;
          MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                   MPI_COMM_WORLD);
          Tag = 1;
          MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                   MPI_COMM_WORLD);
          Tag = 2;
          MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
                   MPI_COMM_WORLD);
        }
    }
  else
    {
      Tag = 0;
      MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
               MPI_COMM_WORLD, &status);
      Tag = 1;
      MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
               MPI_COMM_WORLD, &status);
      Tag = 2;
      MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
               MPI_COMM_WORLD, &status);
    }
}
```

Parallel program with input

```
/* get_data.c -- Parallel Trapezoidal Rule,
    uses basic Get_data function for input.
*/
#include <stdio.h>
#include "mpi.h"

extern void Get_data(float* a_ptr, float* b_ptr,
                     int* n_ptr, int my_rank, int p);
extern float Trap(float a, float b, int n, float h);

int main(int argc, char** argv)
{
  int         my_rank, p;
  float       a, b, h;
  int         n;
  float       local_a, local_b;
  int         local_n;
  float       integral;
  float       total=-1;
  int         source, dest = 0, tag = 0;
  MPI_Status  status;
```

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

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data(&a, &b, &n, my_rank, p);

h = (b-a)/n;    /* h is the same for all processes */
local_n = n/p;  /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

```
 /* Add up the integrals calculated by each process */
  if (my_rank == 0)
    {
      total = integral;
      for (source = 1; source < p; source++)
        {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                   MPI_COMM_WORLD, &status);
          total = total + integral;
        }
    }
  else
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
             tag, MPI_COMM_WORLD);
  /* Print the result */
  if (my_rank == 0)
    {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %f\n",
             a, b, total);
    }
  MPI_Finalize();
  return 0;
}
```

Example makefile

```
MPICC = mpicc
CFLAGS = -Wall -O2 -g

OBJECTS1 = trap.o func.o traprule.o
OBJECTS2 = func.o traprule.o iotrap.o getdata.o

all: partrap iopartrap

partrap: $(OBJECTS1)
        $(MPICC) -o partrap $(OBJECTS1)

iopartrap: $(OBJECTS2)
        $(MPICC) -o iopartrap $(OBJECTS2)

trap.o: trap.c
        $(MPICC) $(CFLAGS) -c trap.c

traprule.o: traprule.c
        $(MPICC) $(CFLAGS) -c traprule.c

func.o: func.c
        $(MPICC) $(CFLAGS) -c func.c

getdata.o: getdata.c
        $(MPICC) $(CFLAGS) -c getdata.c

iotrap.o: iotrap.c
        $(MPICC) $(CFLAGS) -c iotrap.c

clean:
        rm $(OBJECTS1) $(OBJECTS2)
```

Summary

To write many MPI parallel programs you only need:

MPI_Init

MPI_Comm_rank

MPI_Comm_size

MPI_Send

MPI_Recv

MPI_Finalize

Understanding Communications

Buffering

Safe programs

Non-blocking communications

Buffering

Suppose we have

```
if (rank==0)
   MPI_Send(sendbuf,...,1,...)
if (rank==1)
   MPI_Recv(recvbuf,...,0,...)
```

These are blocking communications, which means they will not return until the arguments to the functions can be safely modified by subsequent statements in the program.

Assume that process 1 is not ready to receive.

There are 3 possibilities for process 0:

(a) stops and waits until process 1 is ready to receive,

(b) copies the message at sendbuf into a system buffer (can be on process 0, process 1 or somewhere else) and returns from MPI_Send, or

(c) fails.

As long as buffer space is available, (b) is a reasonable alternative.

An MPI implementation is permitted to copy the message to be sent into internal storage, but it is not required to do so.

What if not enough space is available?

- In applications communicating large amounts of data, there may not be enough memory (left) in buffers.
- Until receive starts, no place to store the send message.
- Practically, (a) results in a serial execution.

A programmer should not assume that the system provides adequate buffering.

Example

Consider a program executing

Process 0	Process 1
MPI_Send to process 1 MPI_Recv from process 1	MPI_Send to process 0 MPI_Recv from process 0

Such a program may work in many cases, but it is certain to fail for message of some size that is large enough.

Possible solutions

Ordered send and receive - make sure each receive is matched with send in execution order across processes.

This matched pairing can be difficult in complex applications. An alternative is to use MPI_Sendrecv. It performs both send and receive such that if no buffering is available, no deadlock will occur.

Buffered sends. MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in buffer) via MPI_Bsend .

Nonblocking communication. With buffering, a send may return before a matching receives is posted. With no buffering, communication is deferred until a place for receiving is provided. Important: in this case you must make certain that you do not modify (or use) the data until you are certain communication has completed.

MPI_Sendrecv

int MPI_Sendrecv (void **sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, int *dest*, int *sendtag*, void **recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status **status*)

Combines:
MPI_Send - send data to process with rank=*dest*
MPI_Recv - receive data from process with rank=*source*

Source and dest may be the same.

MPI_Sendrecv may be matched by ordinary MPI_Send or MPI_Recv .

Performs Send and Recv, and organizes them in such a way that even in systems with no buffering program won’t deadlock.

MPI_Sendrecv_replace

int MPI_Sendrecv_replace (void **buf*, int *count*, MPI_Datatype *datatype*, int *dest*,
int *sendtag*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status **status*)

Sends and receives using a single buffer.

Process sends contents of *buf* to *dest*, then replaces them with data from *source.*

If *source*=*dest*, then function swaps data between process which calls it and process source.

Safe programs

A program is safe if it will produce correct results even if the system provides no buffering.

Need safe programs for portability.

Most programmers expect the system to provide some buffering, hence many unsafe MPI programs are around.

Write safe programs using matching send with receive, MPI_Sendrecv, allocating own buffers, nonblocking operations.

Nonblocking communications

Nonblocking communications are useful for overlapping communication with computation, and ensuring safe programs.

That is, compute while communicating data.

A nonblocking operation requests the MPI library to perform an operation (when it can).

Nonblocking operations do not wait for any communication events to complete.

Nonblocking send and receive: return almost immediately.

The user can modify a send (receive) buffer only after send (receive) is completed.

There are “wait” routines to figure out when a nonblocking operation is done.

MPI_Isend

Performs nonblocking send

---

int MPI_Isend (void * *buf* , int *count*, MPI_Datatype *datatype* , int *dest*, int *tag* ,
MPI_Comm *comm*, MPI_Request * *request*)

---

buf - starting address of buffer

count - number of entries in buffer

datatype - data type of buffer

dest - rank of destination

tag - message tag

comm - communicator

request - communication request (out)

MPI_Irecv

Performs nonblocking receive

---

int MPI_Irecv (void * *buf* , int *count* , MPI_Datatype *datatype* , int *source* , int *tag* , MPI_Comm *comm* , MPI_Request * *request*)

---

buf - starting address of buffer (out)

count - number of entries in buffer

datatype - data type of buffer

source - rank of source

tag - message tag

comm - communicator

request - communication request (out)

Wait routines

---

int MPI_Wait (MPI_Request $*$ *request* , MPI_Status $*$ *status*)

Waits for MPI_Isend or MPI_Irecv to complete.

request - request (in), which is out parameter in MPI_Isend and MPI_Irecv

status - status output, replace with MPI_STATUS_IGNORE if not used

---

Other routines include

MPI_Waitall waits for all given communications to complete.

MPI_Waitany waits for any of given communications to complete.

MPI_Test* tests for completion of send or receive, i.e returns true if completed, false otherwise.

MPI_Testany tests for completion of any previously initiated communication in the input list.

MPI_Waitall

---

int MPI_Waitall (int *count*, MPI_Request *array_of_requests[]* , MPI_Status *array_of_statuses[]*)

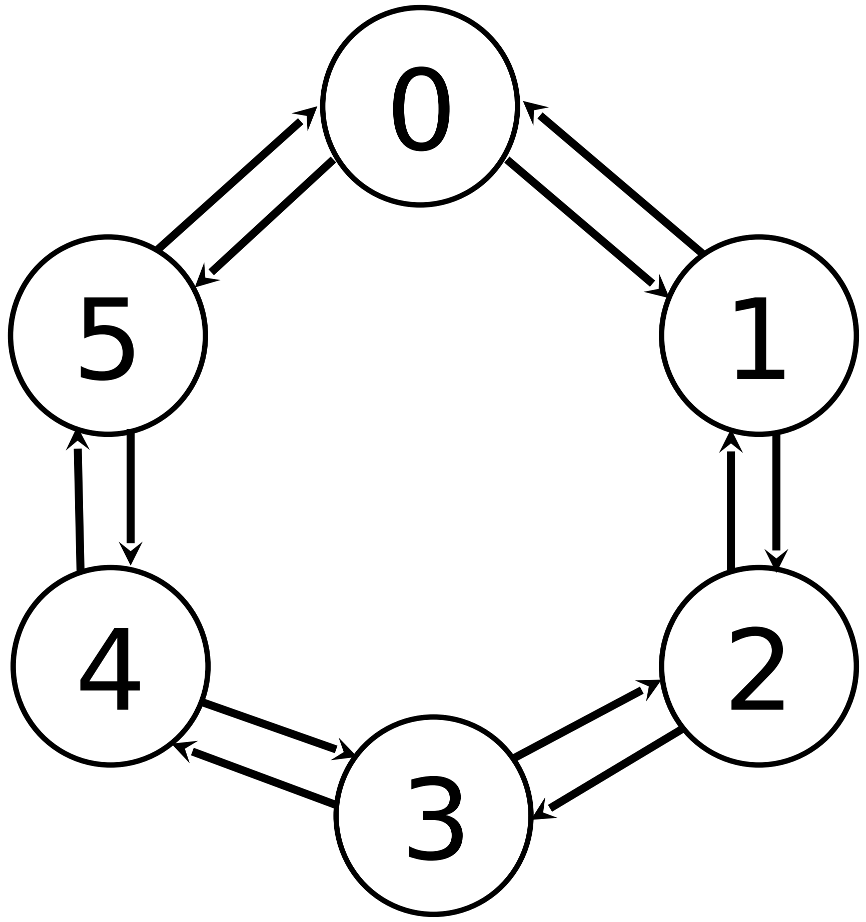
Waits for all given communications to complete

count - list length

array_of_requests - each request is an output parameter in MPI_Isend and MPI_Irecv

array_of_statuses - array of status objects, replace with MPI_STATUSES_IGNORE if never used

Example: Communication between processes in ring topology



With blocking communications it is not possible to write a simple code to accomplish this data exchange. For example, if we have MPI_Send first in all processes, program will get stuck as there will be no matching MPI_Recv to send data to.

Nonblocking communication avoids this problem.

Ring topology example

From https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_ringtopo.c

```
/* nonb.c */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[])

{
  int numtasks, rank, next, prev,
    buf[2], tag1=1, tag2=2;

  tag1=tag2=0;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

~syam/ces745/mpi/nonblocking/nonb.c

```
  prev = rank-1;
  next = rank+1;

  if (rank == 0)            prev = numtasks – 1;
  if (rank == numtasks - 1) next = 0;

  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
            MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
            MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag2,
            MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1,
            MPI_COMM_WORLD, &reqs[3]);

  MPI_Waitall(4, reqs, stats);
  printf("Task %d communicated with tasks %d & %d\n",
         rank,prev,next);

  MPI_Finalize();
  Return 0;
}
```

MPI_Test

---

int MPI_Test (MPI_Request **request*, int **flag*, MPI_Status **status*)

---

request - (input) communication handle, which is output parameter in MPI_Isend and MPI_Irecv

flag - true if operation completed (logical)

status - status output, replace with MPI_STATUS_IGNORE if not used

MPI_Test can be used to test if communication completed, can be called multiple times, in combination with nonblocking send/receive, to control execution flow between processes.

```
if (my_rank == 0){
(... do computation ...)

/* send signal to other processes */
     for (proc = 1; proc < nproc; proc++){
       MPI_Send(&buf, 1, MPI_INT, proc, tag , MPI_COMM_WORLD)
     }
}
else{
/* initiate nonblocking receive */
   MPI_Irecv(&buf, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &reqs);

   for(i = 0; i <= Nlarge; i++){
/* test if Irecv completed */
     MPI_Test(&reqs, &flag, &status);

     if(flag){
        break;    /* terminate loop */
     }
     else{
        (... do computation ...)
     }
   }
}
```

Some details

Nonblocking send can be posted whether a matching receive has been posted or not.

Send is completed when data has been copied out of send buffer.

Nonblocking send can be matched with blocking receive and vice versa.

Communications are initiated by sender.

A communication will generally have lower overhead if a receive buffer is already posted when a sender initiates a communication.