UiT

THE ARCTIC UNIVERSITY OF NORWAY

Lecture 3b: Message-Passing Computing - MPI

Parallell Programming (INF-3201)

John Markus Bjørndalen

MPI (Message Passing Interface)

- What?
  - a specification for the developers and users of message passing libraries (not a library).
- Why use MPI?
  - Standardization - MPI is the only message passing library interface which can be considered a standard.
  - Portability - There is no need to modify your source code when you port your application to a different platform.
  - Performance - Vendor implementations can exploit native hardware features to optimize performance.
  - Functionality - Over 115 routines are defined in MPI-1 alone.
  - Availability - A variety of implementations are available.

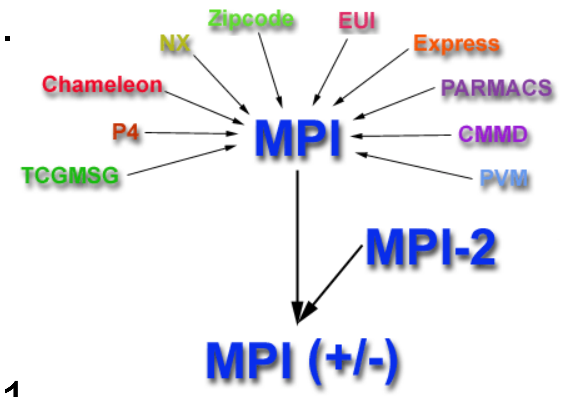


Figure from Blaise Barney, "Message Passing Interface (MPI)", Livermore Computing

MPI Process Creation and Execution

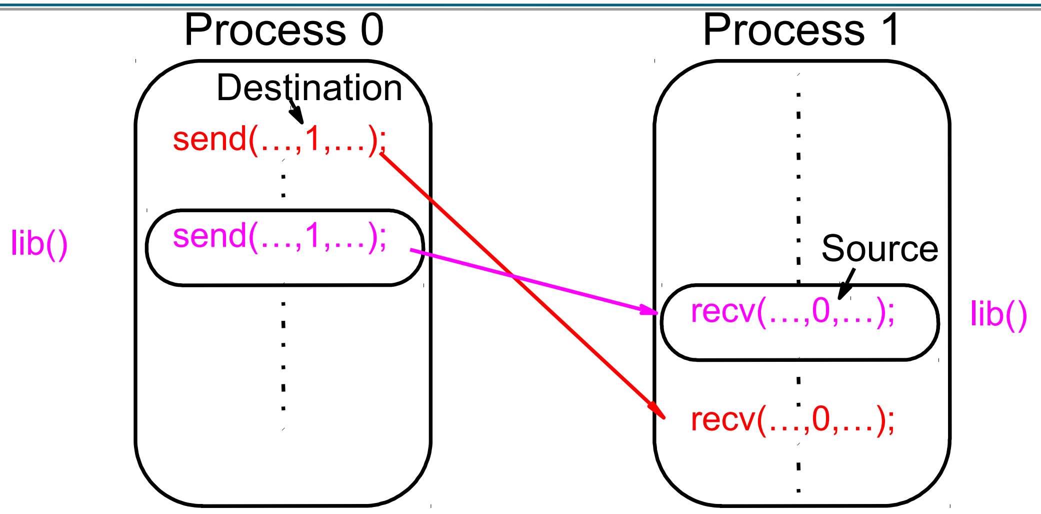
- Purposely not defined - Will depend upon implementation.
- MPI version 1
  - Only static process creation supported.
  - All processes must be defined prior to execution and started together.
- MPI version 2
  - Dynamic process creation supported (MPI_Comm_spawn)
- Originally SPMD model of computation.
  - MPMD also possible with static creation

Using SPMD Computational Model

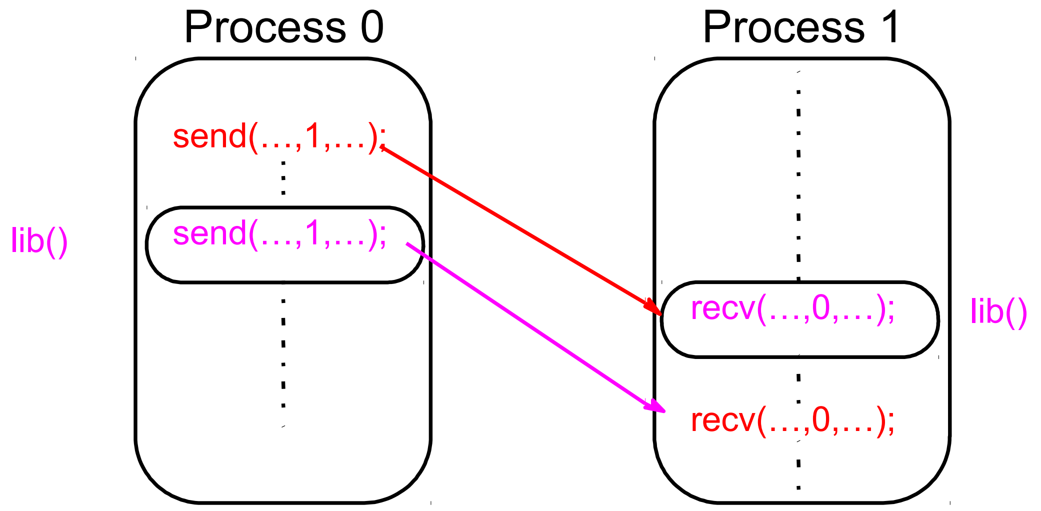
```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
        .
        .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */

    if (myrank == 0)
        master();
    else
        slave();
        .
        .
    MPI_Finalize();
}
```

where master() and slave() are to be executed by master process and slave process, respectively.

Unsafe message passing - Example



(a) Intended behavior



(b) Possible behavior

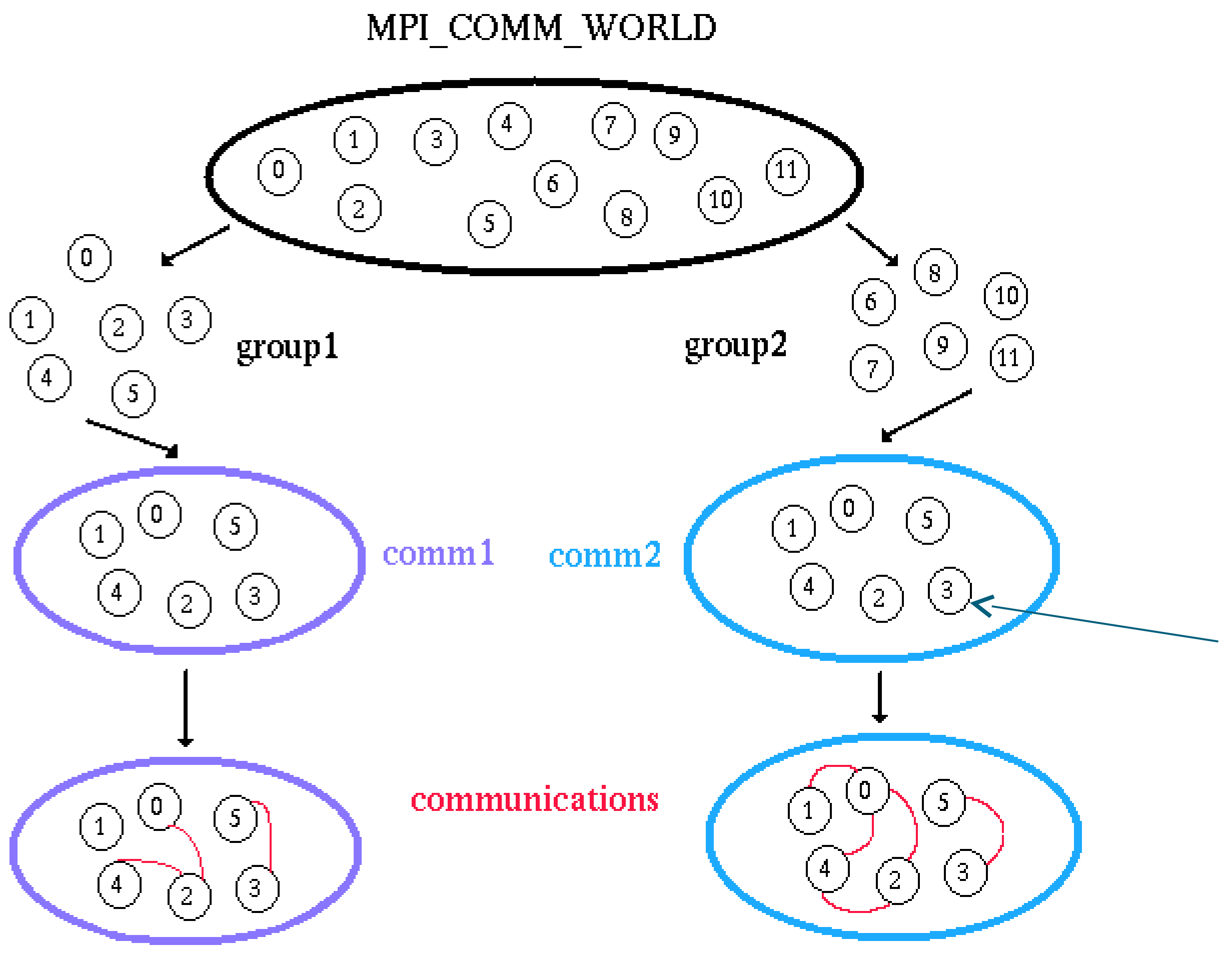
MPI Solution: "Communicators"

- Defines a communication domain - a set of processes that are allowed to communicate between themselves.
- Processes have ranks associated with communicator (0 to n-1)

- Initially, all processes enrolled in a "universe" called MPI_COMM_WORLD
- Other communicators can be established for groups of processes.

- Communication domains of libraries can be separated from that of a user program.
- Used in all point-to-point and collective MPI message-passing communications.

Communicator Example



MPI_Group_incl

MPI_Comm_create

MPI_Comm_rank

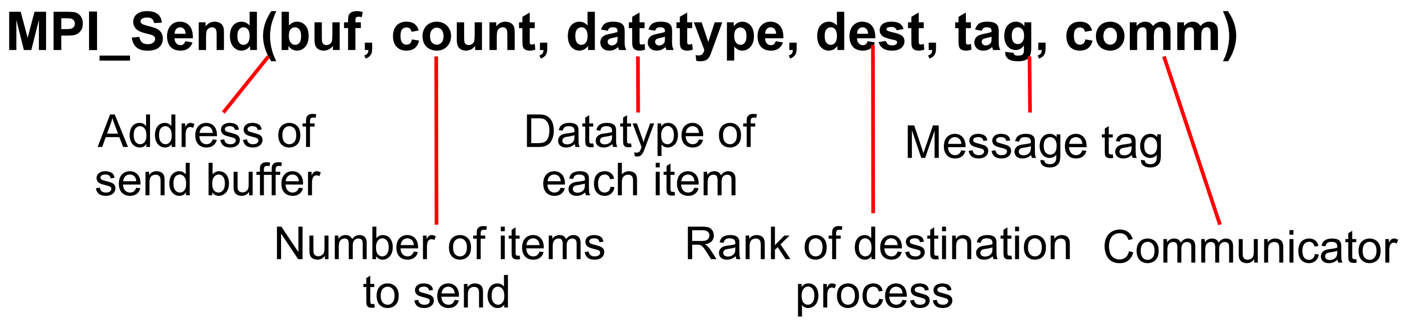
MPI Point-to-Point Communication

- Involves one source process and one destination process
- Uses send and receive routines with message tags (and communicator).
- Wild card message tags available

MPI Blocking Routines

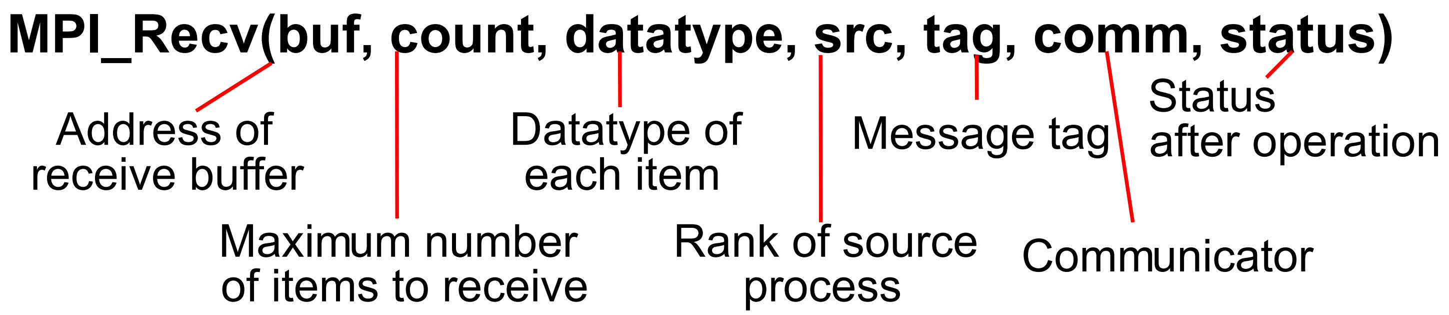
- Return when locally complete
  - location used to hold message can be used again or altered without affecting message being sent.
- When blocking send returns
  - process free to move on without adversely affecting message.
  - does not mean that message has been received,

Parameters of blocking send



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

Parameters of blocking receive



MPI_Recv(buf, count, datatype, src, tag, comm, status)

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */

if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

MPI Nonblocking Routines

- Nonblocking send - MPI_Isend() - will return immediately even before source location is safe to be altered.

- Nonblocking receive - MPI_Irecv() - will return even if no message to accept.

Nonblocking Routine Formats

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request);

MPI_Irecv(buf,count,datatype,source,tag,comm,request);
```

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait(request,…)`

waits until operation completed and returns then.

`MPI_Test(request,…)`

returns with flag (set) indicating whether operation completed at that time.

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  /* find rank */
if (myrank == 0) {
     int x;
     MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
     compute();
     MPI_Wait(req1, status);
} else if (myrank == 1) {
     int x;
     MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

Send Communication Modes

- Buffered Mode - Send may start and return before a matching receive. Necessary to specify buffer space via routine MPI_Buffer_attach().
- Synchronous Mode - Send and receive can start before each other but can only complete together.
- Standard Mode - Not assumed that corresponding receive routine has started. If buffering provided, send could complete before receive reached.
- Ready Mode - Send can only start if matching receive already reached, otherwise error. Use with care.

- Each of the four modes can be applied to both blocking and nonblocking send routines.
- Any type of send routine can be used with any type of receive routine.

Collective Communication

- All or None
  - Must involves all processes in the scope of an intra-communicator.
  - It is the programmer's responsibility to insure that all processes within a communicator participate in any collective routines.
- Programming Considerations and Restrictions
  - Routines are blocking.
  - Routines do not take message tag arguments.
  - Can only be used with MPI predefined datatypes. Solution?
    - Convert user-defined datatypes to an array of integer (cf. cast operator in C)
- Principal collective routines:
  - `MPI_Bcast()` - Broadcast from root to all other processes
  - `MPI_Gather()` - Gather values for group of processes
  - `MPI_Scatter()` - Scatters buffer in parts to group of processes
  - `MPI_Alltoall()` - Sends data from all processes to all processes
  - `MPI_Reduce()` - Combine values on all processes to single value
  - `MPI_Reduce_scatter()` - Combine values and scatter results
  - `MPI_Scan()` - Compute prefix reductions of data on processes

MPI_Bcast

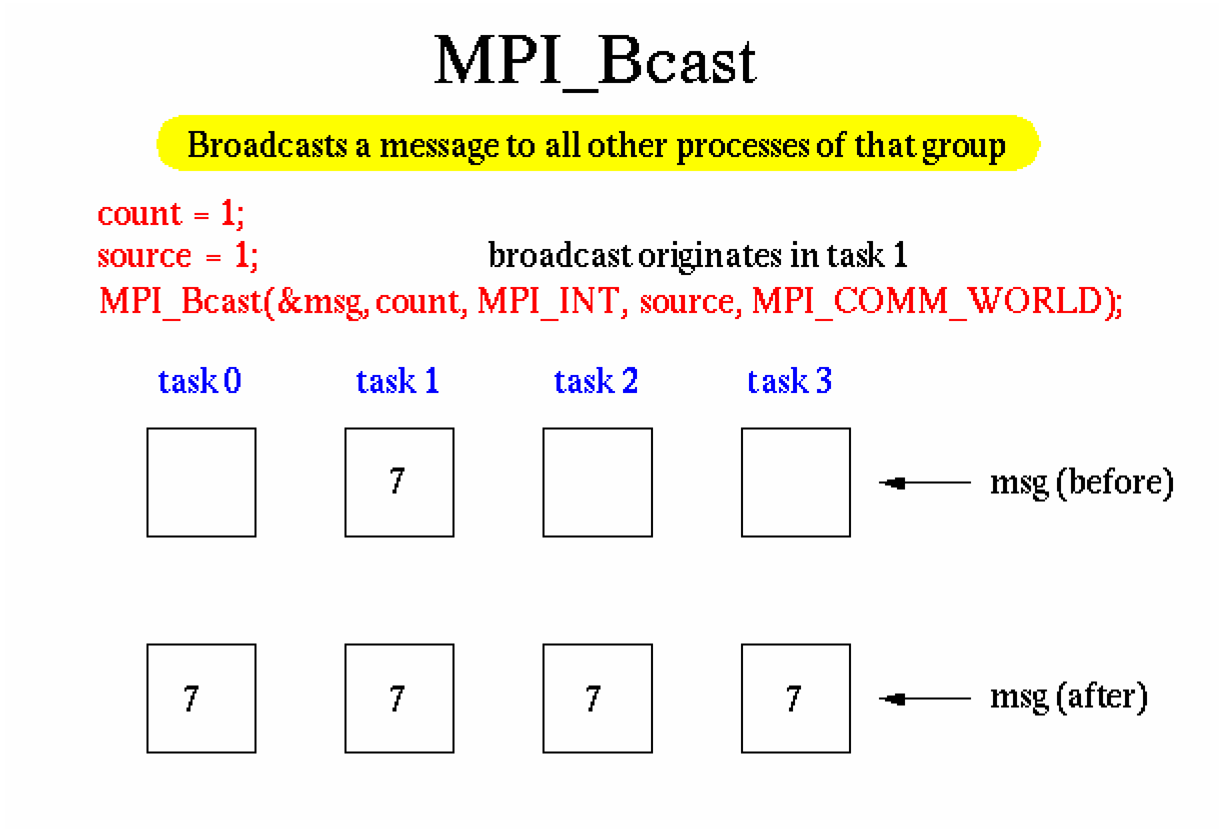
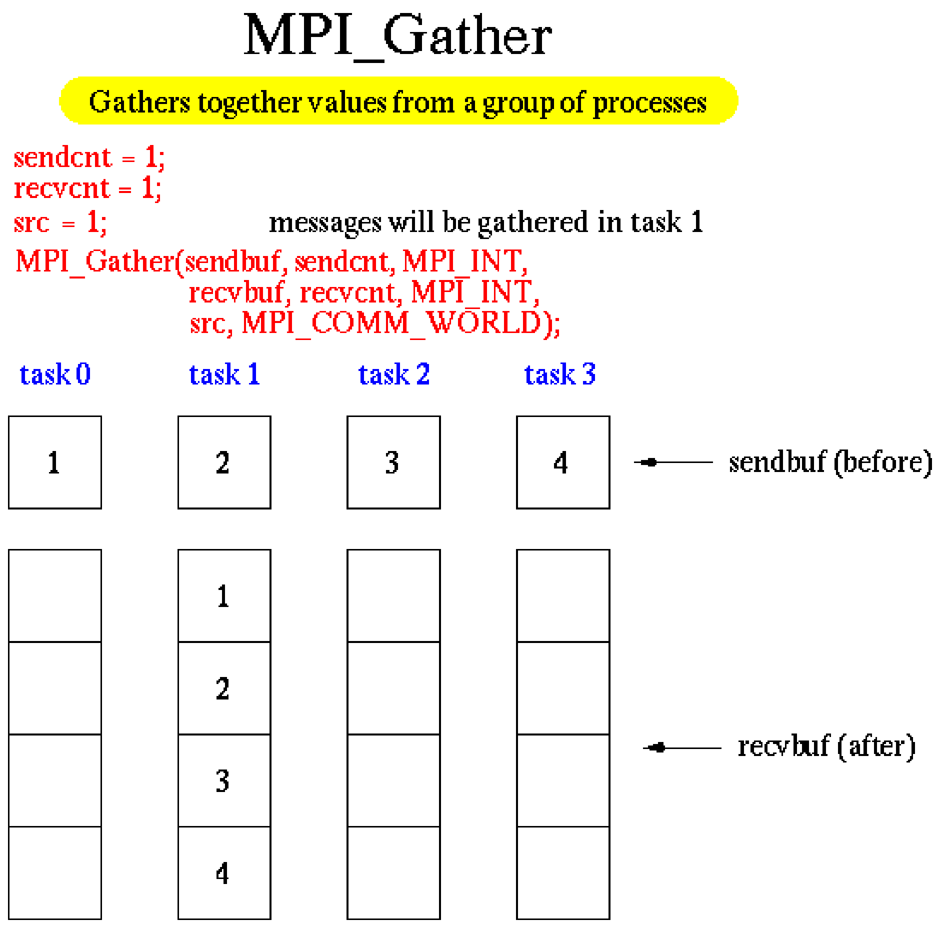

MPI_Bcast
Broadcasts a message to all other processes of that group
count = 1;
source = 1;
broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
7
msg (before)
7
7
7
7
msg (after)

Figure from Blaise Barney, "Message Passing Interface (MPI)", Livermore Computing

MPI_Gather

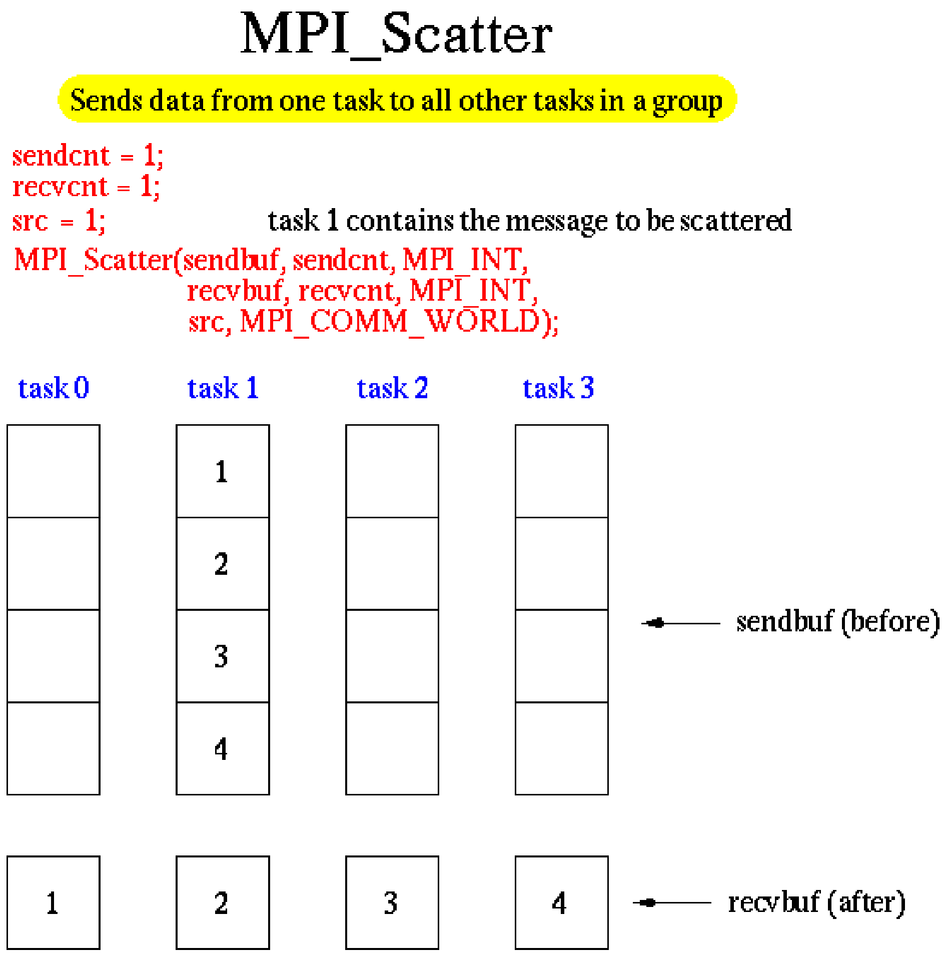


MPI_Gather

Gathers together values from a group of processes

```
sendcnt = 1;
recvcnt = 1;
src = 1;                messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```

MPI_Scatter



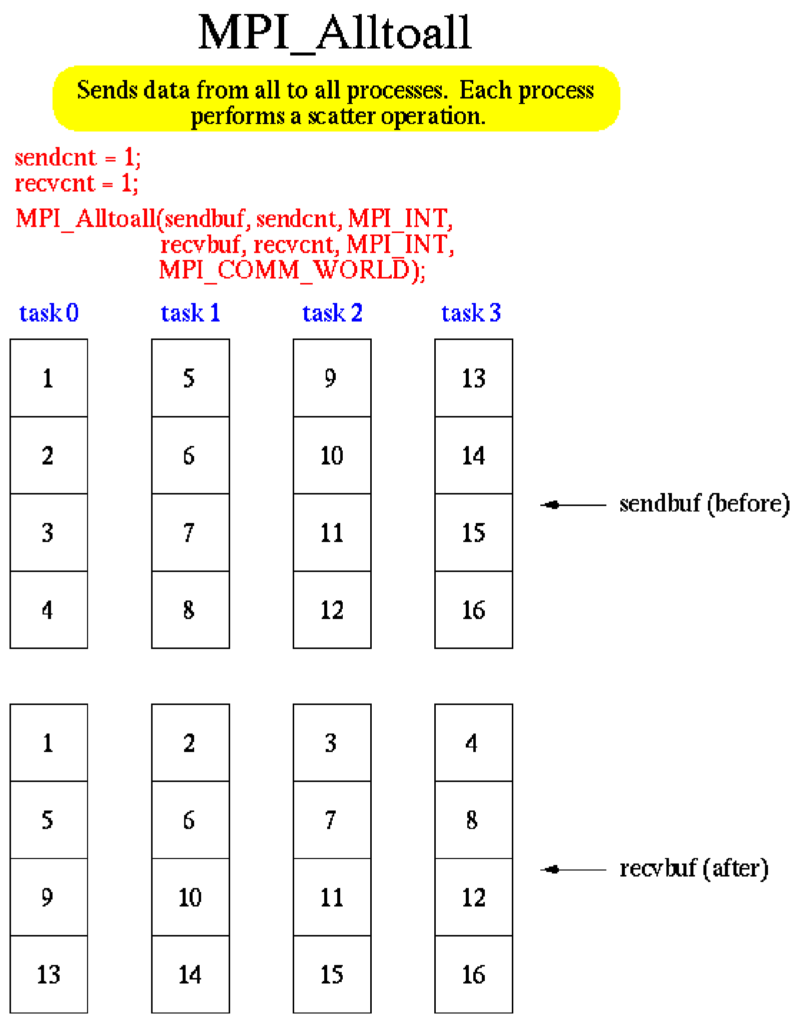
MPI_Scatter

Sends data from one task to all other tasks in a group

```
sendcnt = 1;
recvcnt = 1;
src = 1;                task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

Figure from Blaise Barney, “Message Passing Interface (MPI)”, Livermore Computing

MPI_Alltoall



MPI_Alltoall

Sends data from all to all processes. Each process performs a scatter operation.

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```

MPI_Reduce

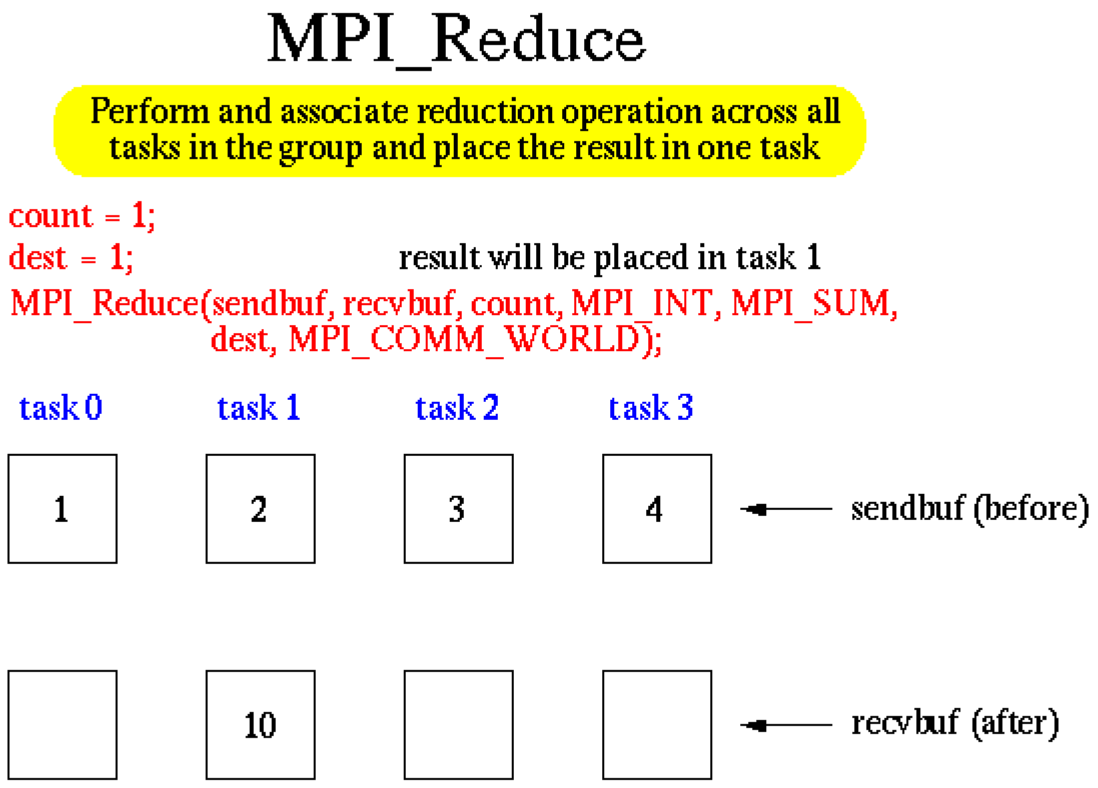
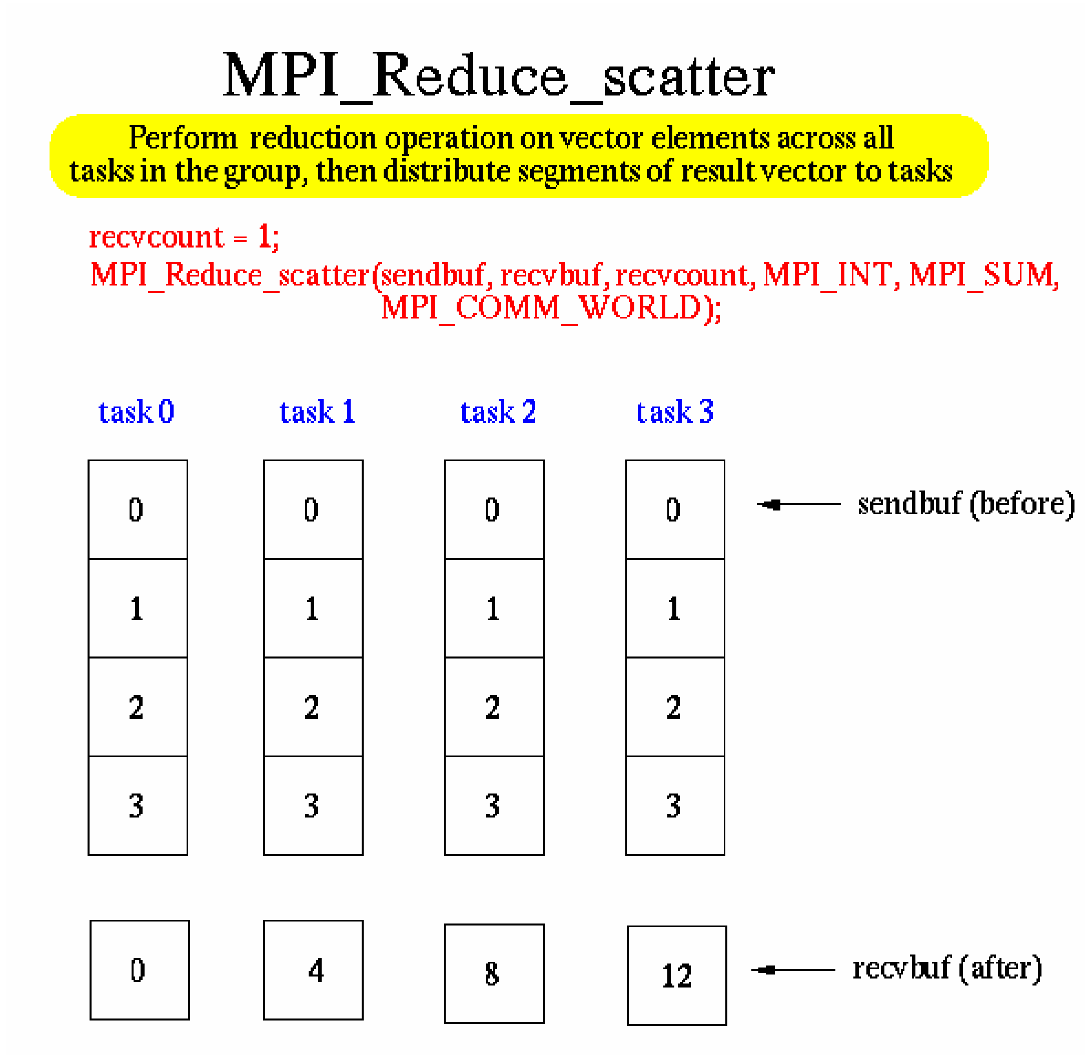

MPI_Reduce
Perform and associate reduction operation across all tasks in the group and place the result in one task
count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

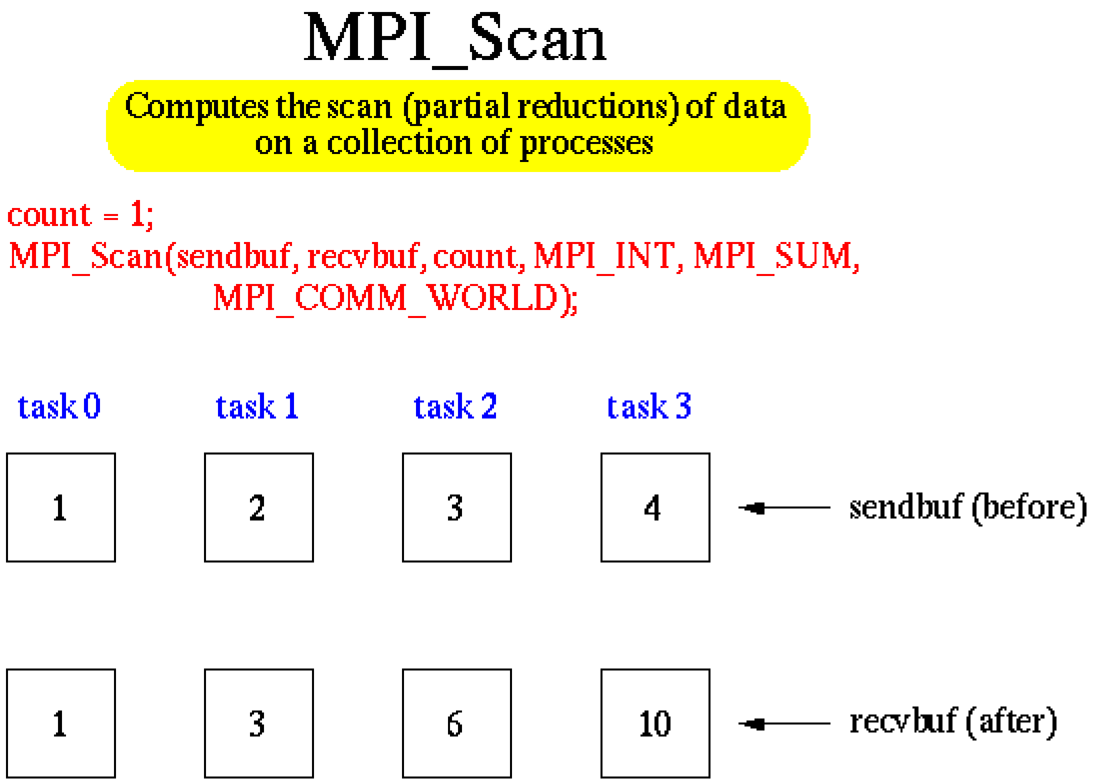
Figure from Blaise Barney, “Message Passing Interface (MPI)”, Livermore Computing

MPI_Reduce_scatter


MPI_Reduce_scatter
Perform reduction operation on vector elements across all tasks in the group, then distribute segments of result vector to tasks
task 0
task 1
task 2
task 3
0
1
2
3
sendbuf (before)
0
4
8
12
recvbuf (after)

```
recvcount = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount, MPI_INT, MPI_SUM,
                   MPI_COMM_WORLD);
```

MPI_Scan



MPI_Scan
Computes the scan (partial reductions) of data on a collection of processes
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
3
6
10
recvbuf (after)

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
         MPI_COMM_WORLD);
```

Figure from Blaise Barney, “Message Passing Interface (MPI)”, Livermore Computing

Example

To gather items from group of processes into process 0, using dynamically allocated memory in root process:

```
int data[10];        /*data to be gathered from processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
    buf = (int*)malloc(grp_size * 10 * sizeof(int)); /*allocate memory*/
}
MPI_Gather(data, 10, MPI_INT, buf, grp_size*10, MPI_INT, 0, MPI_COMM_WORLD);
```

gathers from all processes, including root.

Barrier routine

- A means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.
  - **MPI_Barrier (comm)**

Sample MPI program

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result;
    char fn[255];
    char *fp;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {  /* Open input file and initialize data */
       strcpy(fn,getenv(“HOME”));
       strcat(fn,”/MPI/rand_data.txt”);
       if ((fp = fopen(fn,”r”)) == NULL) {
           printf(“Can’t open the input file: %s\n\n”, fn);
           exit(1);
       }
       for(i = 0; i < MAXSIZE; i++) fscanf(fp,”%d”, &data[i]);
    }
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
    x = n/nproc; /* Add my portion Of data */
    low = myid * x;
    high = low + x;
    for(i = low; i < high; i++)
        myresult += data[i];
    printf(“I got %d from %d\n”, myresult, myid); /* Compute global sum */
    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf(“The sum is %d.\n”, result);
    MPI_Finalize();
}
```

Estimating speedup

Sequential execution time, $t_{s:}$ Estimate by counting computational steps of best sequential algorithm.

Parallel execution time, $t_{p:}$ In addition to number of computational steps, t_{comp}, need to estimate communication overhead, $t_{\text{comm:}}$

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

Computation Time

- Count number of computational steps.
- When more than one process executed simultaneously, count computational steps of most complex process.
- Generally, function of number of data elements *n* and number of processors *p*, i.e.

$$t_{\text{comp}} = f(n, p)$$

Often break down computation time into parts. Then

$$t_{\text{comp}} = t_{\text{comp1}} + t_{\text{comp2}} + t_{\text{comp3}} + \ldots$$

Analysis usually done assuming that all processors are same and operating at same speed.

Communication Time

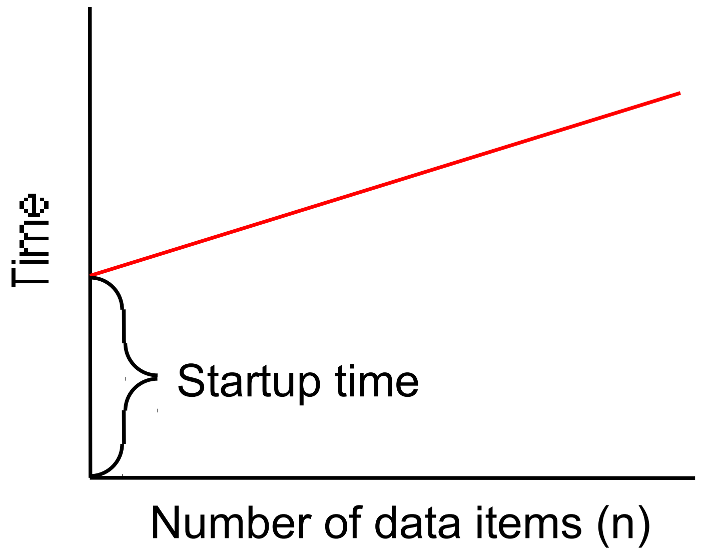
Many factors, including network structure and network contention. As a first approximation, use

$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

t_{startup} is startup time, essentially time to send a message with no data. Assumed to be constant.

t_{data} is transmission time to send one data word, also assumed constant, and there are *n* data words.

Idealized Communication Time



Final communication time, t_{comm}

Sum of communication times of all sequential messages from a process, i.e.

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + \ldots$$

– Assumption: Communication patterns of all processes are the same and take place together

$\Rightarrow$ only one process need be considered.

Both $t_{startup}$ and t_{data}, measured in units of one computational step, so that can add t_{comp} and t_{comm} together to obtain parallel execution time, t_p.

Benchmark Factors

With $t_{s,}$ $t_{comp,}$ and t_{comm}, can establish speedup factor and computation/communication ratio for a particular algorithm/implementation.

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp}+t_{comm}}$$

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}}$$

Both functions of number of processors, *p*, and number of data elements, *n*.

Factors give indication of scalability of parallel solution with increasing number of processors and problem size.

Computation/communication ratio will highlight effect of communication with increasing problem size and system size.

References

- Barry Wilkinson & Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.
- Blaise Barney, “Message Passing Interface (MPI)”, Livermore Computing.