Week 4 Lecture Notes

Distributed Memory Programming
Using Advanced MPI (Message Passing Interface)

Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

MPI_Bcast

MPI_Bcast(void *message, int count, MPI_Datatype dtype, int source, MPI_Comm comm)

- **Collective communication**
- **Allows a process to broadcast a message to all other processes**

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while(1)
 {
  if (myid == 0)
   {
    printf("Enter the number of intervals: (0 quits) \n");
    fflush(stdout);
    scanf("%d",&n);
   } // if myid == 0
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

MPI_Reduce

MPI_Reduce(void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op int root, MPI_Comm comm)

- **Collective communication**
- **Processes perform the specified "reduction"**
- **The "root" has the results**

```
if (myid == 0)
 {
  printf("Enter the number of intervals: (0 quits) \n");
  fflush(stdout);
  scanf("%d",&n);
 } // if myid == 0
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
if (n == 0) break;
else
 {
  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));
   } // for
  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_Allreduce

MPI_Allreduce(void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, MPI_Comm comm)

- **Collective communication**
- **Processes perform the specified "reduction"**
- **All processes have the results**

```
    start = MPI_Wtime();
    for (i=0; i<100; i++)
     {
      a[i] = i;
      b[i] = i * 10;
      c[i] = i + 7;
      a[i] = b[i] * c[i];
     }
    end = MPI_Wtime();
    printf("Our timers precision is %.20f seconds\n",MPI_Wtick());
    printf("This silly loop took %.5f seconds\n",end-start);
   }
  else
   {
    sprintf(sig,"Hello from id %d, %d or %d processes\n",myid,myid
+1,numprocs);
    MPI_Send(sig,sizeof(sig),MPI_CHAR,0,0,MPI_COMM_WORLD);
   }
  MPI_Allreduce(&myid,&sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
  printf("Sum of all process ids = %d\n",sum);
  MPI_Finalize();
  return 0;
 }
```

MPI Reduction Operators

• MPI_BAND	bitwise and
• MPI_BOR	bitwise or
• MPI_BXOR	bitwise exclusive or
• MPI_LAND	logical and
• MPI_LOR	logical or
• MPI_LXOR	logical exclusive or
• MPI_MAX	maximum
• MPI_MAXLOC	maximum and location of maximum
• MPI_MIN	minimum
• MPI_MINLOC	minimum and location of minimum
• MPI_PROD	product
• MPI_SUM	sum

MPI_Gather (example 1)

MPI_Gather (sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm)

- **Collective communication**
- **Root gathers data from every process including itself**

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

int main(int argc, char **argv )
 {
  int i,myid, numprocs;
  int *ids;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0)
   ids = (int *) malloc(numprocs * sizeof(int));
  MPI_Gather(&myid,1,MPI_INT,ids,1,MPI_INT,0,MPI_COMM_WORLD);
  if (myid == 0)
   for (i=0;i<numprocs;i++)
    printf("%d\n",ids[i]);
  MPI_Finalize();
  return 0;
 }
```

MPI_Gather (example 2)

MPI_Gather (sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm)

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

int main(int argc, char **argv )
 {
  int i,myid, numprocs;
  char sig[80];
  char *signatures;
  char **sigs;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  sprintf(sig,"Hello from id %d\n",myid);
  if (myid == 0)
   signatures = (char *) malloc(numprocs * sizeof(sig));
  MPI_Gather(&sig,sizeof(sig),MPI_CHAR,signatures,sizeof(sig),MPI_CHAR,
    0,MPI_COMM_WORLD);
  if (myid == 0)
   {
    sigs=(char **) malloc(numprocs * sizeof(char *));
    for(i=0;i<numprocs;i++)
     {
      sigs[i]=&signatures[i*sizeof(sig)];
      printf("%s",sigs[i]);
     }
   }
  MPI_Finalize();
  return 0;
 }
```

MPI_Alltoall

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm)

- **Collective communication**
- **Each process sends & receives the same amount of data to every process including itself**

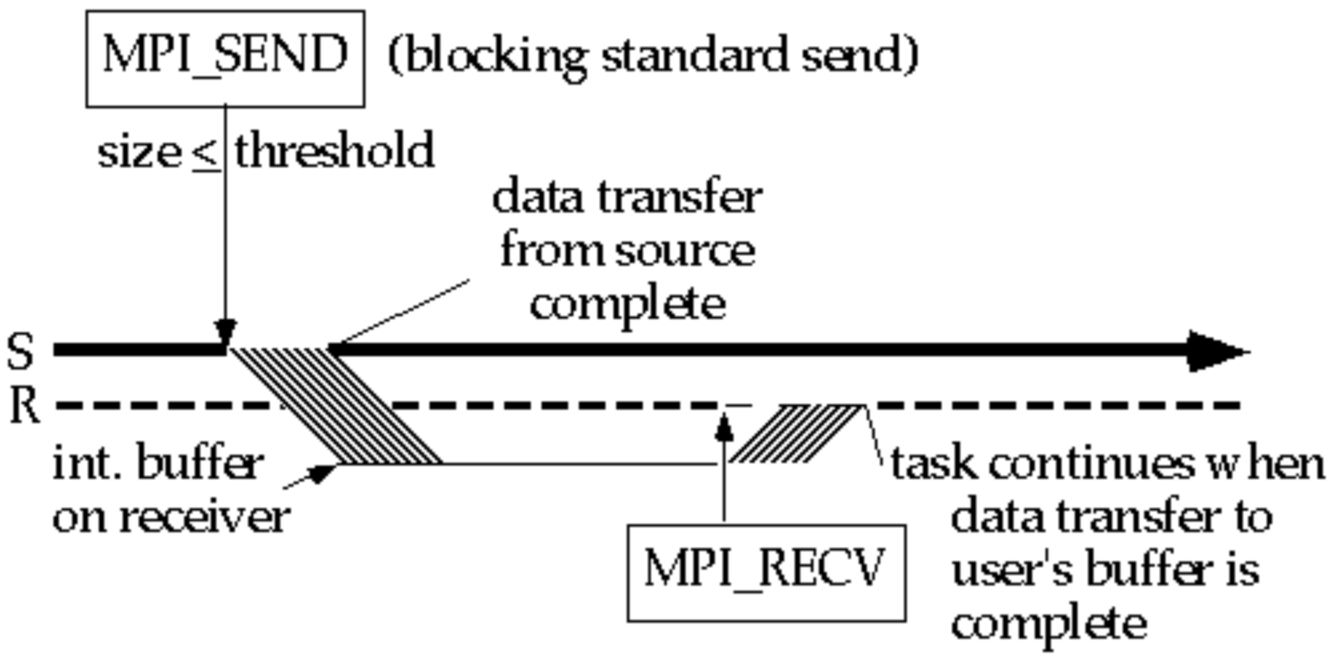
```
#include <stdio.h>
#include <mpi.h>
#include <malloc.h>

int main(int argc, char **argv )
 {
  int i,myid, numprocs;
  int *all,*ids;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  ids = (int *) malloc(numprocs * 3 * sizeof(int));
  all = (int *) malloc(numprocs * 3 * sizeof(int));
  for (i=0;i<numprocs*3;i++) ids[i] = myid;
  MPI_Alltoall(ids,3,MPI_INT,all,3,MPI_INT,MPI_COMM_WORLD);
  for (i=0;i<numprocs*3;i++)
   printf("%d\n",all[i]);
  MPI_Finalize();
  return 0;
 }
```

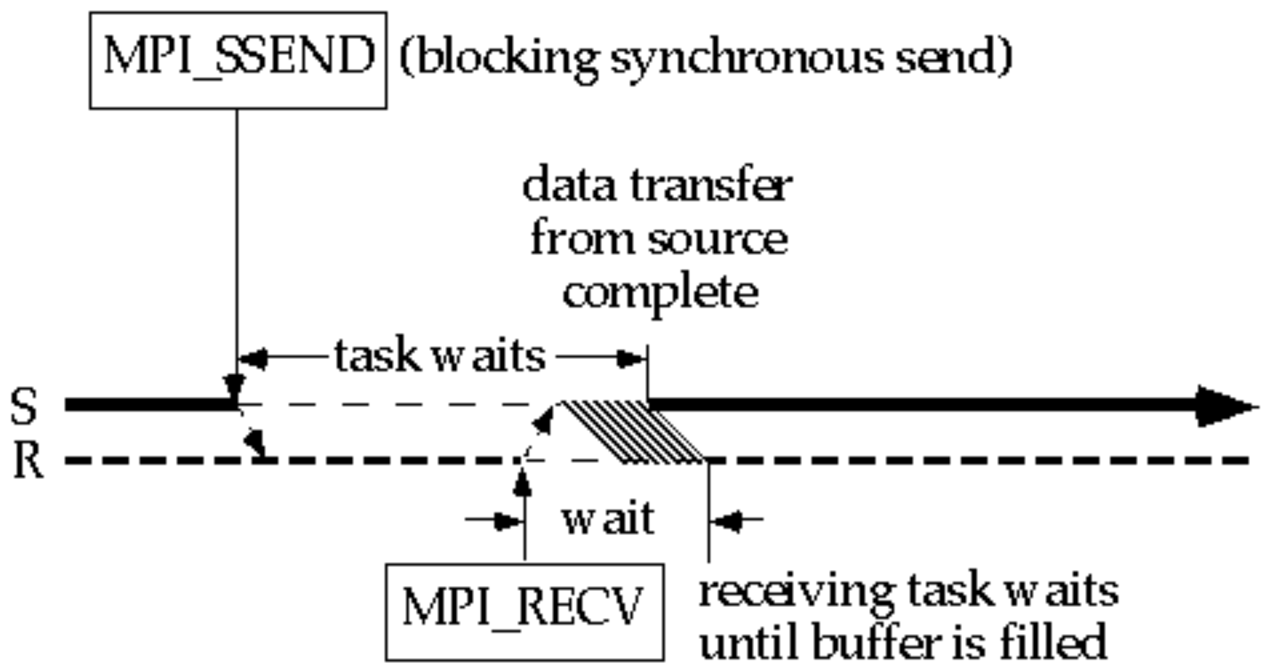
Different Modes for MPI_Send – 1 of 4
MPI_Send: Standard send

- **MPI_Send(buf, count, datatype, dest, tag, comm)**
  - Quick return based on successful “buffering” on receive side
  - Behavior is implementation dependent and can be modified at runtime



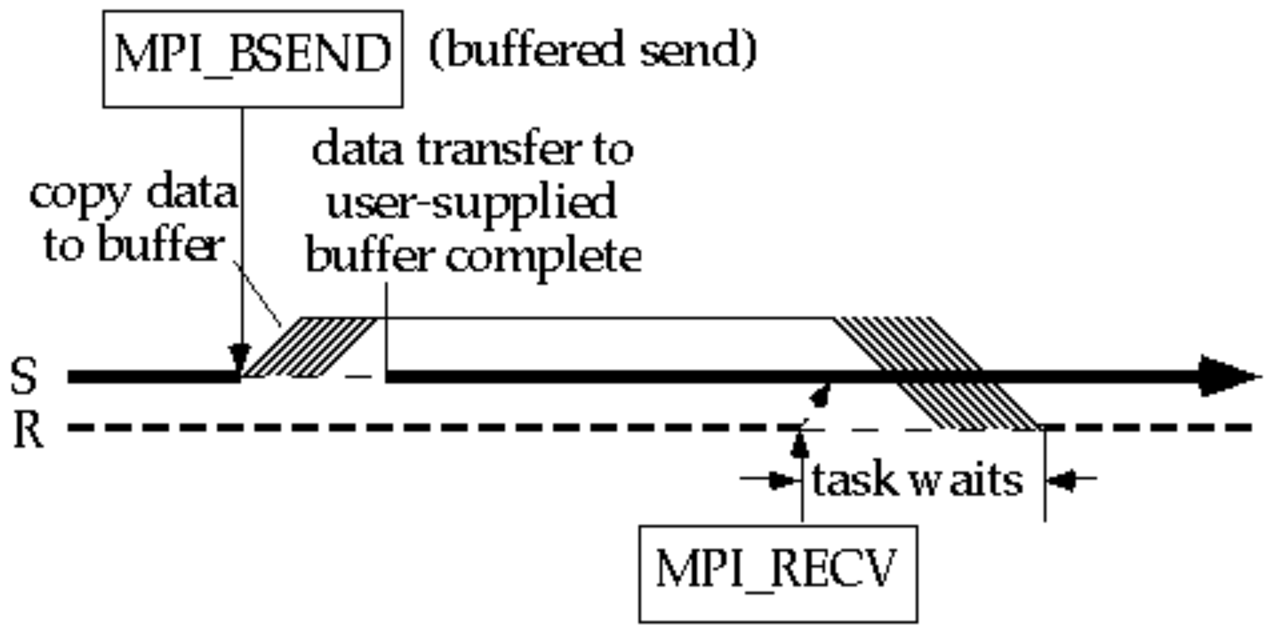
Different Modes for MPI_Send – 2 of 4
MPI_Ssend: Synchronous send

- **MPI_Ssend(buf, count, datatype, dest, tag, comm)**
  - Returns after matching receive has begun and all data have been sent
  - This is also the behavior of MPI_Send for message size > threshold



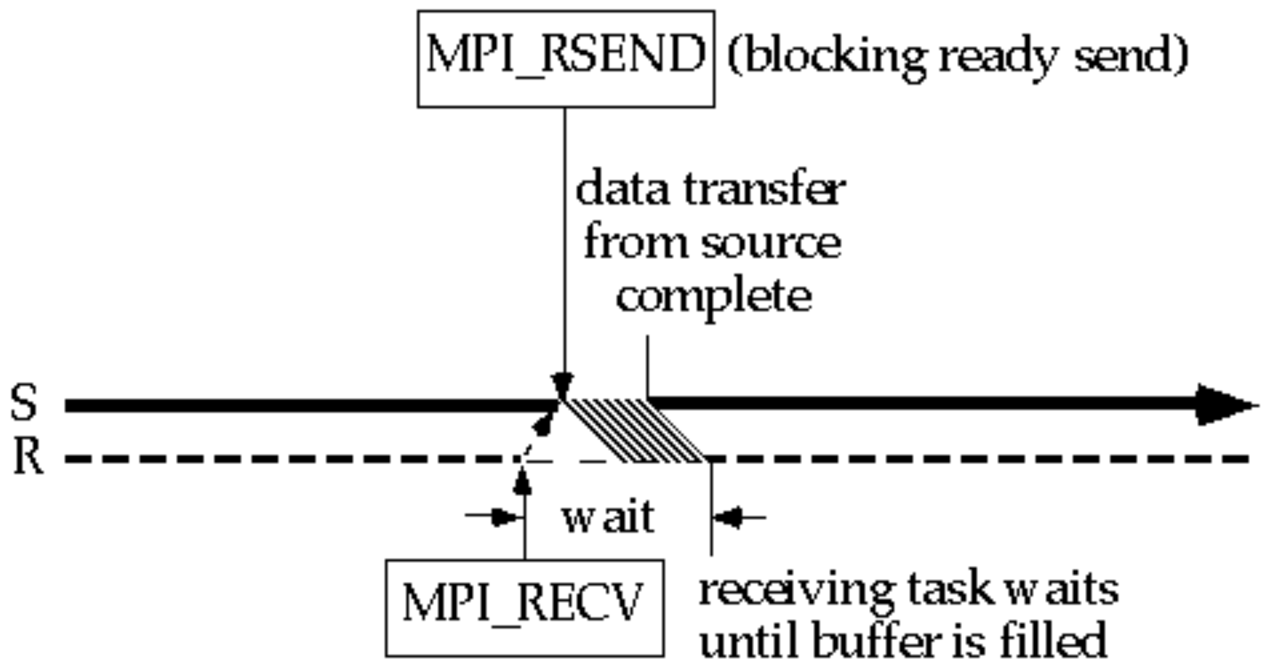
Different Modes for MPI_Send – 3 of 4
MPI_Bsend: Buffered send

- **MPI_Bsend(buf, count, datatype, dest, tag, comm)**
  - Basic send with user specified buffering via MPI_Buffer_Attach
  - MPI must buffer outgoing send and return
  - Allows memory holding the original data to be changed



Different Modes for MPI_Send – 4 of 4
MPI_Rsend: Ready send

- **MPI_Rsend(buf, count, datatype, dest, tag, comm)**
  - Send only succeeds if the matching receive is already posted
  - If the matching receive has not been posted, an error is generated

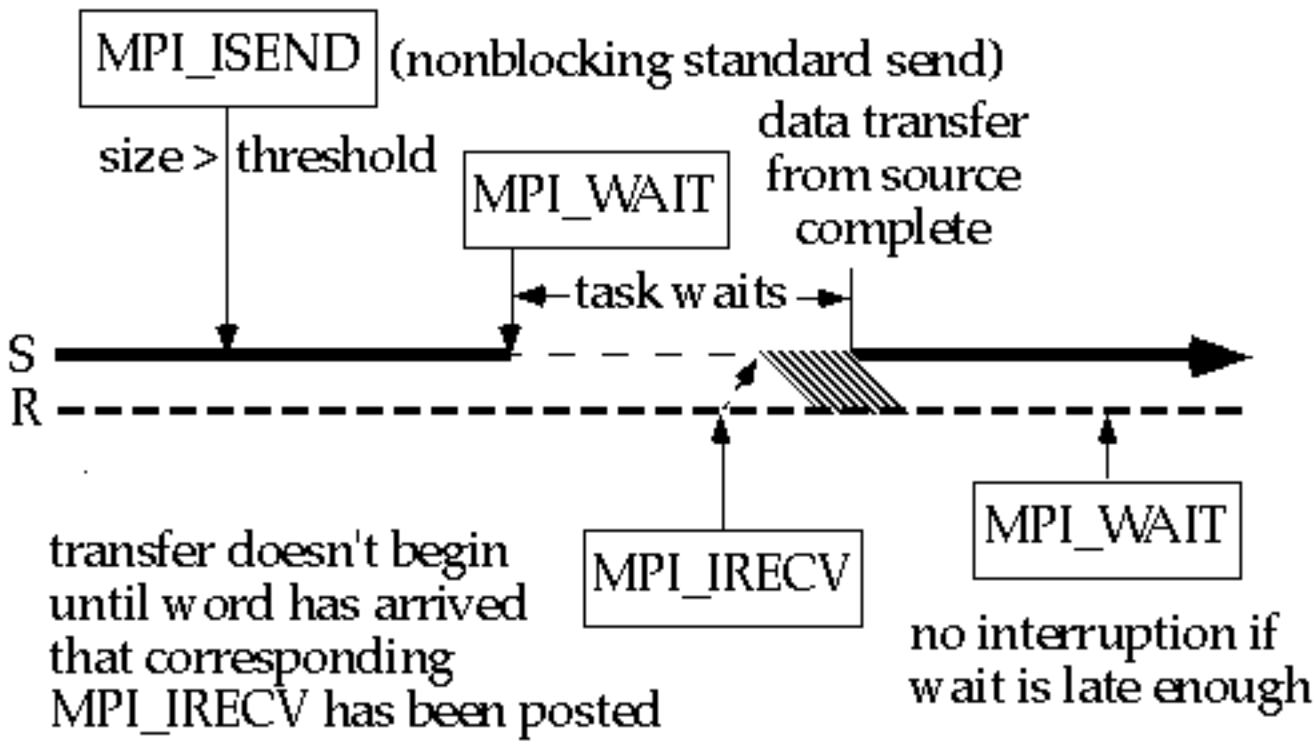


Non-Blocking Varieties of MPI_Send…

- **Do not access send buffer until send is complete!**
- **To check send status, call MPI_Wait or similar checking function**
  - Every nonblocking send *must* be paired with a checking call
  - Returned "request handle" gets passed to the checking function
  - Request handle does not clear until a check succeeds
- **MPI_Isend(buf, count, datatype, dest, tag, comm, request)**
  - Immediate non-blocking send, message goes into pending state
- **MPI_Issend(buf, count, datatype, dest, tag, comm, request)**
  - Synchronous mode non-blocking send
  - Control returns when matching receive has begun
- **MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)**
  - Non-blocking buffered send
- **MPI_Irsend (buf, count, datatype, dest, tag, comm, request)**
  - Non-blocking ready send

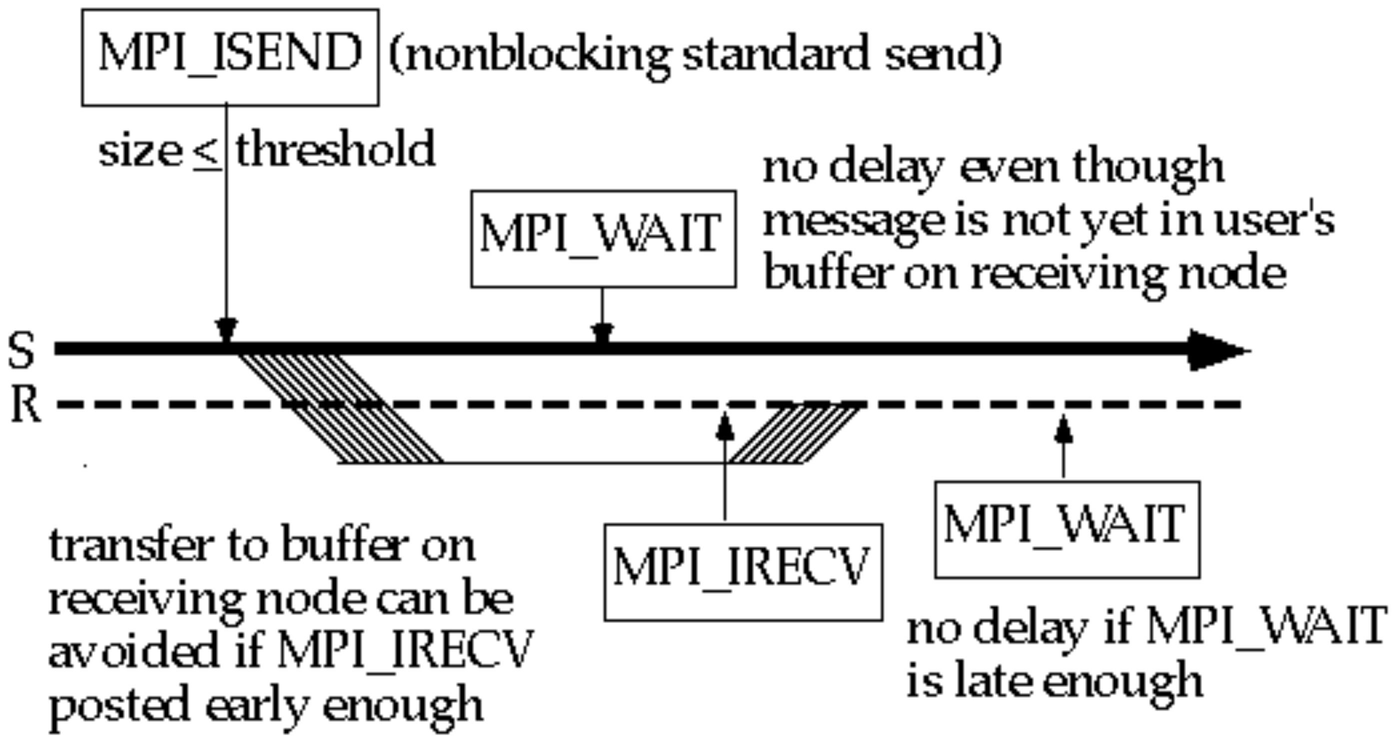
MPI_Isend for Size > Threshold: Rendezvous Protocol

- **MPI_Wait blocks until receive has been posted**
- **For Intel MPI, I_MPI_EAGER_THRESHOLD=262144 (256K by default)**



MPI_Isend for Size <= Threshold: Eager Protocol

- No waiting on either side is MPI_Irecv is posted after the send…
- What if MPI_Irecv or its MPI_Wait is posted before the send?



MPI_Recv and MPI_Irecv

- **MPI_Recv(buf, count, datatype, source, tag, comm, status)**
  - Blocking receive
- **MPI_Irecv(buf, count, datatype, source, tag, comm, request)**
  - Non-blocking receive
  - Make sure receive is complete before accessing buffer

- **Nonblocking call must always be paired with a checking function**
  - Returned “request handle” gets passed to the checking function
  - Request handle does not clear until a check succeeds
- **Again, use MPI_Wait or similar call to ensure message receipt**
  - MPI_Wait(MPI_Request request, MPI_Status status)

MPI_Irecv Example
Task Parallelism fragment (tp1.c)

```
while(complete < iter)
 {
  for (w=1; w<numprocs; w++)
   {
    if ((worker[w] == idle) && (complete < iter))
     {
      printf ("Master sending UoW[%d] to Worker %d\n",complete,w);
      Unit_of_Work[0] = a[complete];
      Unit_of_Work[1] = b[complete];
      // Send the Unit of Work
      MPI_Send(Unit_of_Work,2,MPI_INT,w,0,MPI_COMM_WORLD);
      // Post a non-blocking Recv for that Unit of Work
      MPI_Irecv(&result[w],1,MPI_INT,w,0,MPI_COMM_WORLD,&recv_req[w]);
      worker[w] = complete;
      dispatched++;
      complete++; // next unit of work to send out
     }
   } // foreach idle worker
  // Collect returned results
  returned = 0;
  for(w=1; w<=dispatched; w++)
   {
    MPI_Waitany(dispatched, &recv_req[1], &index, &status);
    printf("Master receiving a result back from Worker %d c[%d]=%d
\n",status.MPI_SOURCE,worker[status.MPI_SOURCE],result[status.MPI_SOURCE]);
    c[worker[status.MPI_SOURCE]] = result[status.MPI_SOURCE];
    worker[status.MPI_SOURCE] = idle;
    returned++;
   }
```

MPI_Probe and MPI_Iprobe

- **MPI_Probe**
  - MPI_Probe(source, tag, comm, status)
  - Blocking test for a message
- **MPI_Iprobe**
  - int MPI_Iprobe(source, tag, comm, flag, status)
  - Non-blocking test for a message
- **Source can be specified or MPI_ANY_SOURCE**
- **Tag can be specified or MPI_ANY_TAG**

MPI_Get_count

MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

- **The status variable returned by MPI_Recv also returns information on the length of the message received**
  - This information is not directly available as a field of the MPI_Status struct
  - A call to MPI_Get_count is required to "decode" this information
- **MPI_Get_count takes as input the status set by MPI_Recv and computes the number of entries received**
  - The number of entries is returned in count
  - The datatype argument should match the argument provided to the receive call that set status
  - Note: in Fortran, status is simply an array of INTEGERs of length MPI_STATUS_SIZE

MPI_Sendrecv

- **Combines a blocking send and a blocking receive in one call**
- **Guards against deadlock**
- **MPI_Sendrecv**
  - Requires two buffers, one for send, one for receive
- **MPI_Sendrecv_replace**
  - Requires one buffer, received message overwrites the sent one
- **For these combined calls:**
  - Destination (for send) and source (of receive) can be the same process
  - Destination and source can be different processes
  - MPI_Sendrecv can send to a regular receive
  - MPI_Sendrecv can receive from a regular send
  - MPI_Sendrecv can be probed by a probe operation

BagBoy Example
1 of 3

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>
#include <malloc.h>
#define Products 10

int main(int argc, char **argv )
 {
  int myid,numprocs;
  int true = 1;
  int false = 0;
  int messages = true;
  int i,g,items,flag;
  int *customer_items;
  int checked_out = 0;
  /* Note, Products below are defined in order of increasing weight */
  char Groceries[Products][20] = {"Chips","Lettuce","Bread","Eggs","Pork
    Chops","Carrots","Rice","Canned Beans","Spaghetti Sauce","Potatoes"};
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

BagBoy Example
2 of 3

```
if (numprocs >= 2)
   {
    if (myid == 0) // Master
     {
      customer_items = (int *) malloc(numprocs * sizeof(int));
      /* initialize customer items to zero - no items received yet */
      for (i=1;i<numprocs;i++) customer_items[i]=0;
      while (messages)
       {
        MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&flag,&status);
        if (flag)
         {
          MPI_Recv(&items,
    1,MPI_INT,status.MPI_SOURCE,status.MPI_TAG,MPI_COMM_WORLD,&status);
      /* increment the count of customer items from this source */
          customer_items[status.MPI_SOURCE]++;
          if (customer_items[status.MPI_SOURCE] == items) checked_out++;
          printf("%d: Received %20s from %d, item %d of %d
    \n",myid,Groceries[status.MPI_TAG],status.MPI_SOURCE,customer_items[status.MP
    I_SOURCE],items);
         }
        if (checked_out == (numprocs-1)) messages = false;
       }
     } // Master
```

BagBoy Example
3 of 3

```
else  // Workers
     {
      srand((unsigned)time(NULL)+myid);
      items = (rand() % 5) + 1;
      for(i=1;i<=items;i++)
       {
        g = rand() % 10;
        printf("%d: Sending %s, item %d of %d\n",myid,Groceries[g],i,items);
        MPI_Send(&items,1,MPI_INT,0,g,MPI_COMM_WORLD);
       }
     } // Workers
   }
  else
   printf("ERROR: Must have at least 2 processes to run\n");

  MPI_Finalize();
  return 0;
 }
```

Using Message Passing Interface, MPI
Bubble Sort

Bubble Sort

```
#include <stdio.h>
#define N 10


int main (int argc, char *argv[])
 {
   int a[N];
   int i,j,tmp;

   printf("Unsorted\n");
   for (i=0; i<N; i++) { a[i] = rand(); printf("%d\n",a[i]); }
   for (i=0; i<(N-1); i++)
    for(j=(N-1); j>i; j--)
     if (a[j-1] > a[j])
      {
       tmp = a[j];
       a[j] = a[j-1];
       a[j-1] = tmp;
      }
   printf("\nSorted\n");
   for (i=0; i<N; i++) printf("%d\n",a[i]);
 }
```

Serial Bubble Sort in Action

N = 5	3 8 4 5 2
i=0, j=4	3 8 4 2 5
i=0, j=3	3 8 2 4 5
i=0, j=2	3 2 8 4 5
i=0, j=1	2 3 8 4 5
i=1, j=4	2 3 8 4 5
i=1, j=3	2 3 4 8 5
i=1, j=2	2 3 4 8 5
i=2, j=4	2 3 4 5 8
i=2, j=3	2 3 4 5 8

Step 1: Partitioning
Divide Computation & Data into Pieces

- **The primitive task would be each element of the unsorted array**

Goals:

- ✓ **Order of magnitude more primitive tasks than processors**
- ✓ **Minimization of redundant computations and data**
- ✓ **Primitive tasks are approximately the same size**
- ✓ **Number of primitive tasks increases as problem size increases**

Step 2: Communication
Determine Communication Patterns between Primitive Tasks

- **Each task communicates with its neighbor on each side**

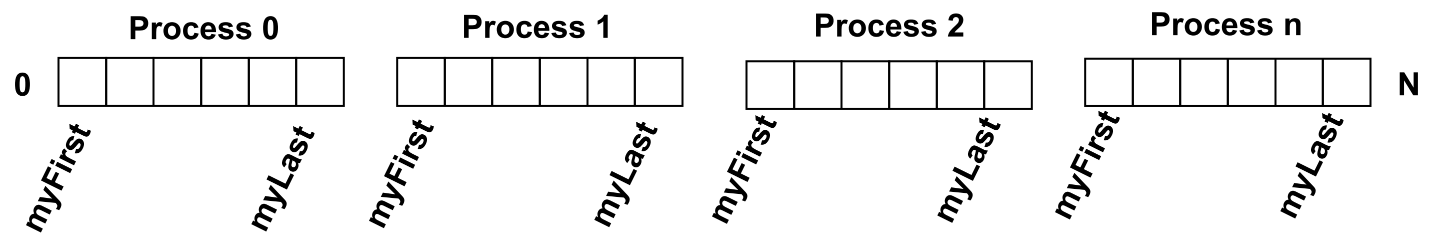
Goals:

- ✓ **Communication is balanced among all tasks**
- ✓ **Each task communicates with a minimal number of neighbors**
- ✓ ***Tasks can perform communications concurrently**
- ✓ ***Tasks can perform computations concurrently**

*Note: there are some exceptions in the actual implementation

Step 3: Agglomeration
Group Tasks to Improve Efficiency or Simplify Programming

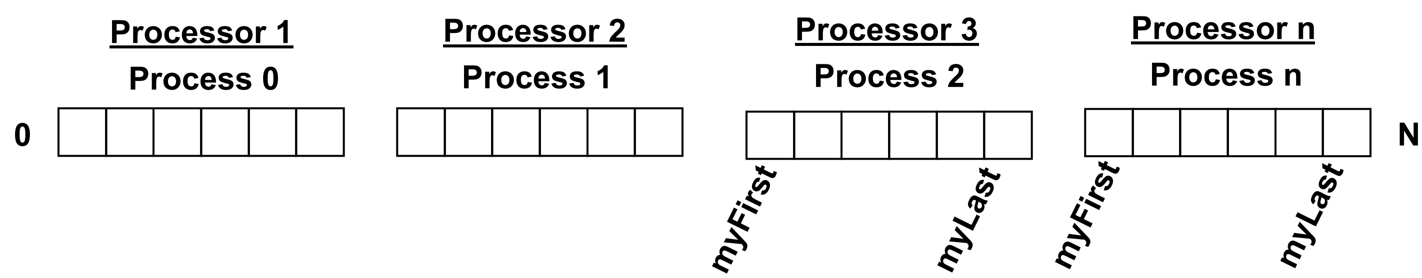
- **Divide unsorted array evenly amongst processes**
- **Perform sort steps in parallel**
- **Exchange elements with other processes when necessary**



- ✓ **Increases the locality of the parallel algorithm**
- ✓ **Replicated computations take less time than the communications they replace**
- ✓ **Replicated data is small enough to allow the algorithm to scale**
- ✓ **Agglomerated tasks have similar computational and communications costs**
- ✓ **Number of tasks can increase as the problem size does**
- ✓ **Number of tasks as small as possible but at least as large as the number of available processors**
- ✓ **Trade-off between agglomeration and cost of modifications to sequential codes is reasonable**

Step 4: Mapping
Assigning Tasks to Processors

- Map each process to a processor
- This is not a CPU intensive operation so using multiple tasks per processor should be considered
- If the array to be sorted is very large, memory limitations may compel the use of more machines



✓ Mapping based on one task per processor and multiple tasks per processor have been considered

✓ Both static and dynamic allocation of tasks to processors have been evaluated

(NA) If a dynamic allocation of tasks to processors is chosen, the task allocator (master) is not a bottleneck

✓ If static allocation of tasks to processors is chosen, the ratio of tasks to processors is at least 10 to 1

Hint – Sketch out Algorithm Behavior *BEFORE* Implementing 1 of 2

7	6	5	4		3	2	1	0
j=3					j=7			
7	6	4	5		3	2	0	1
j=2					j=6			
7	4	6	5		3	0	2	1
j=1					j=5			
4	7	6	5		0	3	2	1
j=0					j=4			
				<->				
4	7	6	0		5	3	2	1
j=3					j=7			
4	7	0	6		5	3	1	2
j=2					j=6			
4	0	7	6		5	1	3	2
j=1					j=5			
0	4	7	6		1	5	3	2
j=0					j=4			
				<->				
0	4	7	1		6	5	3	2
j=3					j=7			
0	4	1	7		6	5	2	3
j=2					j=6			
0	1	4	7		6	2	5	3
j=1					j=5			
0	1	4	7		2	6	5	3
j=0					j=4			
				<->				

Hint 2 of 2

0	1	4	2		7	6	5	3
j=3					j=7			
0	1	2	4		7	6	3	5
j=2					j=6			
0	1	2	4		7	3	6	5
j=1					j=5			
0	1	2	4		3	7	6	5
j=0					j=4			
				<->				
0	1	2	3		4	7	6	5
j=3					j=7			
0	1	2	3		4	7	5	6
j=2					j=6			
0	1	2	3		4	5	7	6
j=1					j=5			
0	1	2	3		4	5	7	6
j=0					j=4			
				<->				
0	1	2	3		4	5	7	6
j=3					j=7			
0	1	2	3		4	5	6	7
j=2					j=6			
0	1	2	3		4	5	6	7
j=1					j=5			
0	1	2	3		4	5	6	7
j=0					j=4			

…?... <-> how many times?...