CS420 – Lectures 13, 14 and 15

Raghavendra Kanakagiri
Slides: Marc Snir

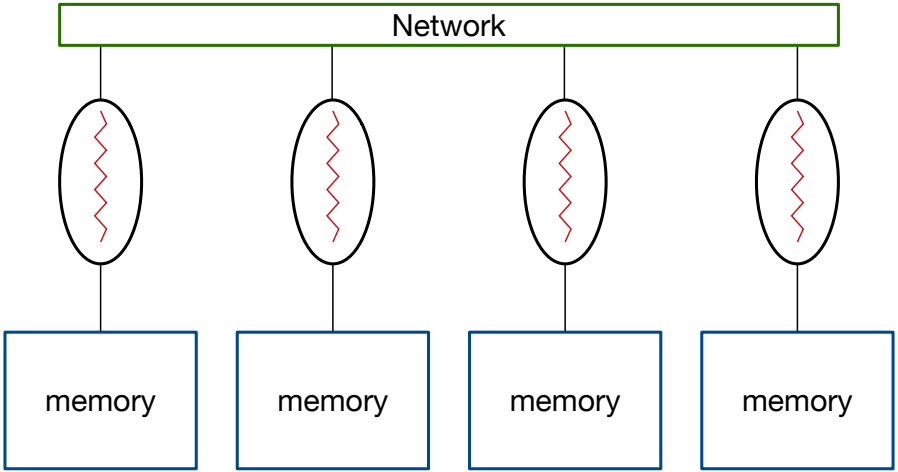
Spring 2023

Message-Passing

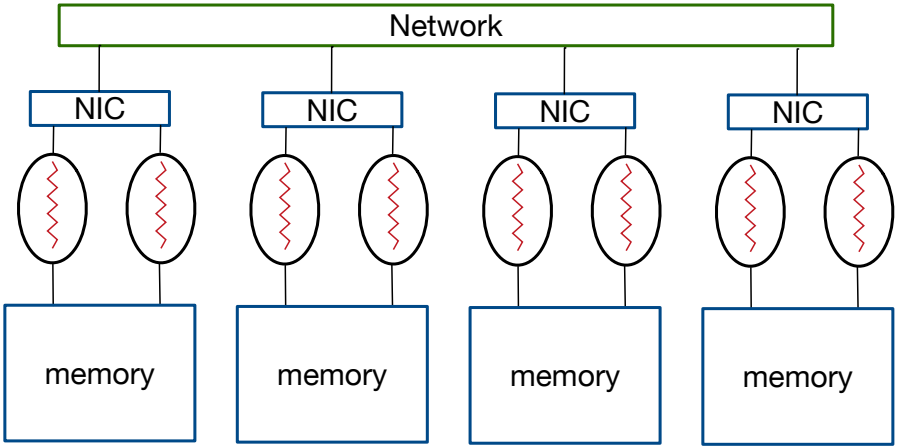
Beyond shared memory

Shared memory becomes expensive and hard to scale beyond a few sockets.
Instead we use *distributed memory*

Distributed memory



Hybrid shared memory/distributed memory



Basic communication mechanism

Hardware:
- Moves data from the memory of one node to the memory of another node.
- Provides indication that transfer is complete

Software:
- Calls to move data from one memory to another
- Calls to synchronize

Message Passing

Communication is achieved by a matching pair of a *send* and a *receive*:
`send(to, data)` $\longrightarrow$ `recv(from,data)`
The communication

- Moves data (from sender to receiver)
- Synchronizes (receive will complete after send started)

MPI

- MPI (Message Passing Interface) is a Standard Message passing library designed by an open forum that is broadly used in HPC.
- Standardization effort started in 1992, MPI-1 was published late 93.
- Has C and Fortran binding

Message Passing

Basic communication mechanism: sending & receiving messages

`send(to, data)` $\longrightarrow$ `recv(from,data)`

Message Passing

Basic communication mechanism: sending & receiving messages

`send(to, data)` $\longrightarrow$ `recv(from,data)`

- Who is communicating?
  - In MPI the communication is between *processes*. Typically, processes will be on different nodes, but they could be on the same node.

Message Passing

Basic communication mechanism: sending & receiving messages

`send(to, data)` $\longrightarrow$ `recv(from,data)`

- Who is communicating?
  - In MPI the communication is between *processes*. Typically, processes will be on different nodes, but they could be on the same node.
- Need process ids
  - An MPI computation involves a group of processes. The initial group is `MPI_COMM_WORLD`. Each process has a *rank* within `MPI_COMM_WORLD`; ranks are from 0 to $N-1$, if there are N processes. (Actually, `MPI_COMM_WORLD` is a *communicator*; more about this later.)

Hello world

```
#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;
}
```

- `main` is executed by each process. Number of processes is fixed
- `MPI_comm_rank()` returns rank of calling process
- `MPI_comm_size()` returns number of processes
- Initialization and finalization is required

Simple communication

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

int main(int argc, char ** argv)
{
int rank, val[100];

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0)
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
else if (rank == 1)
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

MPI_Finalize();
return 0;
}
```

```
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)
```

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
```

```
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)
```

- I am sending data that is stored in a buffer starting at `val` (*send buffer*)

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
```

```
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)
```

- I am sending data that is stored in a buffer starting at `val` (*send buffer*)
- I am sending 100 items

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
```

```
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)
```

- I am sending data that is stored in a buffer starting at `val` (*send buffer*)
- I am sending 100 items
- These items are integers

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
```

```
MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)
```

- I am sending data that is stored in a buffer starting at `val` (*send buffer*)
- I am sending 100 items
- These items are integers
- The *destination* is the process with rank 1 in `MPI_COMM_WORLD`

```
MPI_Send(sendbuf, count, type, dest, tag, comm)
```

`MPI_Send(val, 100, MPI_INT, 1, 0, MPI_COMM_WORLD)`

- I am sending data that is stored in a buffer starting at `val` (*send buffer*)
- I am sending 100 items
- These items are integers
- The *destination* is the process with rank 1 in `MPI_COMM_WORLD`
- The message is *tagged* with the value 0.

`MPI_Send(sendbuf, count, type, dest, tag, comm)`

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)
- I am receiving (up to) 100 items

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)
- I am receiving (up to) 100 items
- These items are integers

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)
- I am receiving (up to) 100 items
- These items are integers
- The *source* should be the process with rank 0 in `MPI_COMM_WORLD`

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)
- I am receiving (up to) 100 items
- These items are integers
- The *source* should be the process with rank 0 in `MPI_COMM_WORLD`
- The message should be *tagged* with the value 0.

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

```
MPI_Recv(val, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- I am receiving data into the buffer starting at location `val` (*receive buffer*)
- I am receiving (up to) 100 items
- These items are integers
- The *source* should be the process with rank 0 in `MPI_COMM_WORLD`
- The message should be *tagged* with the value 0.
- I don't need for MPI to return in `status` information on how the communication completed

```
MPI_Recv(recvbuf, count, type, source, tag, comm, status)
```

Rules

- The receive will match a message sent to the right destination (*communicator* and *rank*, with the correct information on the "envelope": *sender* and *tag*.
- The programmer must ensure that
  - The datatypes match
  - The sent message does not overflow the receive buffer (OK to send fewer items, the status parameter will tell how many were actually received).
- The send will complete as soon as the message was copied out of the sender memory
  - Possibly before the receive started, if there is buffering
  - Possibly only after the receive is posted, if there is no buffering
- The receive will complete as soon as all the data has been copied into the receive buffer
- Mismatched sends and receives can cause deadlocks!

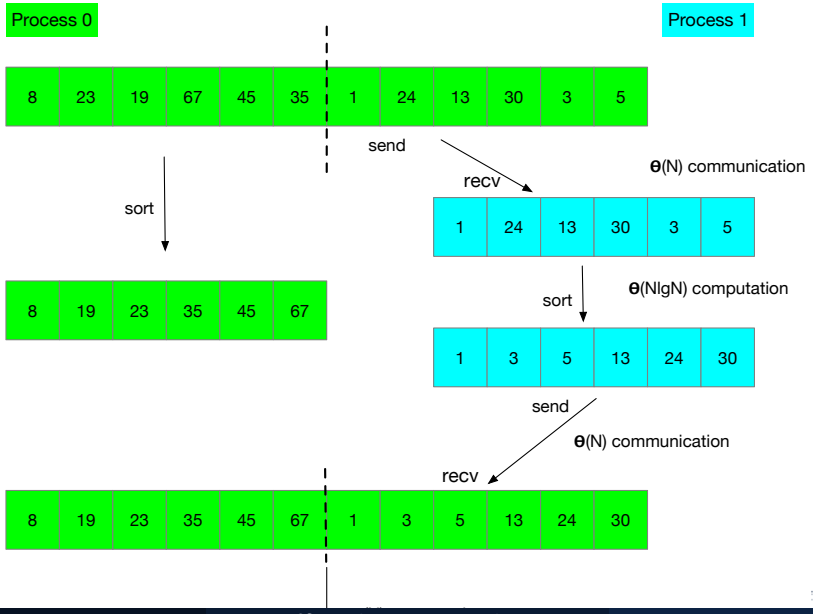
Communicators

- A *communicator* is
  - An ordered set of processes
  - A *context*, a "color"

  A process can be in multiple communicators, with a different rank in each
- Communications with different communicators do not interfere with each other
  - Important in the design of parallel libraries
- Simple programs only use `MPI_COMM_WORLD`

Example

(Naive) parallel sort with 2 processes

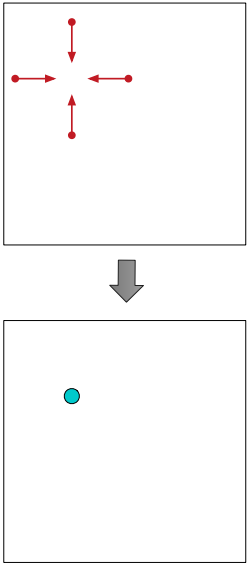


```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
int rank;
int a[1000];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
sort(a, 500);
MPI_Recv(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
merge(a);
}
else if (rank == 1) {
MPI_Recv(a, 500, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
sort(a, 500);
MPI_Send(a, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize(); return 0;
}
```

Is the parallel algorithm faster than sequential?

- Sequential time: $a + bn + cn \lg n$: fixed overhead (e.g., start program), linear overhead (e.g., read/write array) and $n \lg n$, for sorting.
- Parallel time: $a\prime + bn + c(n/2)\lg(n/2) + dn$: fixed, larger overhead to start computation on two nodes; same I/O overhead; sorting work roughly reduced by half; linear communication time added.
- Parallel algorithm is faster if $a + bn + cn \lg n > a\prime + bn + c(n/2)\lg(n/2) + dn$ or $a + \frac{cn}{2}(\lg n + 1) > a\prime + dn$
- Parallel algorithm is faster for large n, and sequential algorithm is better for small n (since $a\prime \gg a$); crossing point will depend on exact values of the various coefficients.

Jacobi



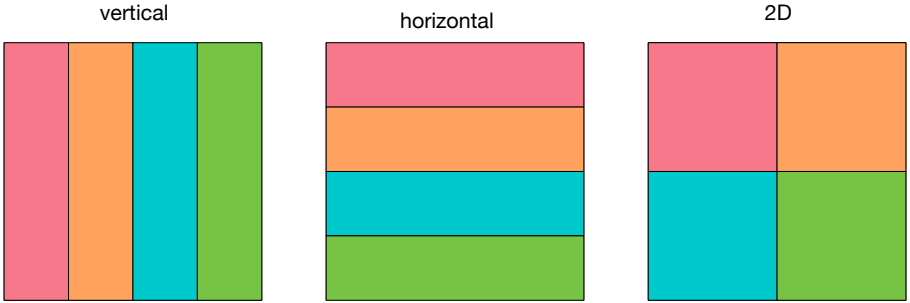
```
do {
  err=0; l=1-l;
  for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++) {
    a[1-l][i][j]=0.25*(a[l][i-1][j]
    +a[l][i+1][j]+a[l][i][j-1]
    +a[l][i][j+1]);
    err = fmax(err,fabs(a[1][i][j]
    -a[0][i][j]));
  }
} while(err>maxerr);
```

Parallel Jacobi

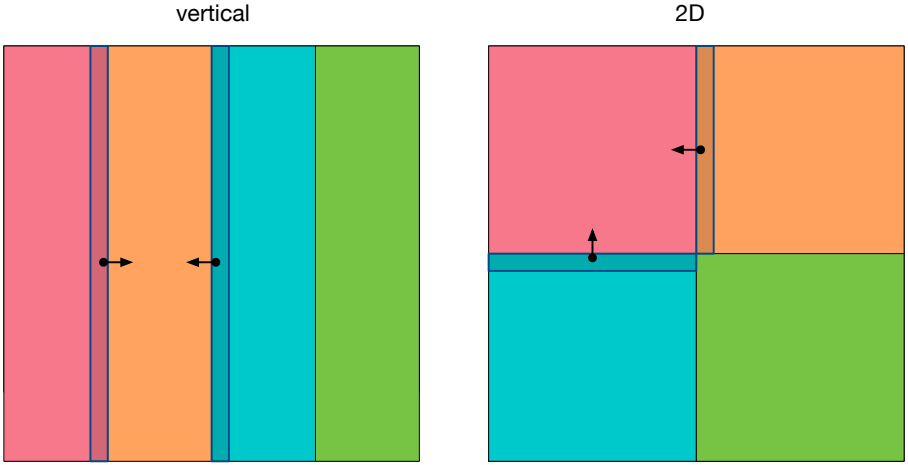
- Need to distribute the two arrays
- Need to distribute the computation
- *Data parallelism:* distribution of computation follows the distribution of the data
- Simplemost approach is to follow the *owner compute* rule: The process that "owns" an entry (has it in its local memory) is responsible for updating it.

Possible distributions

Need to split the two matrices across the processes



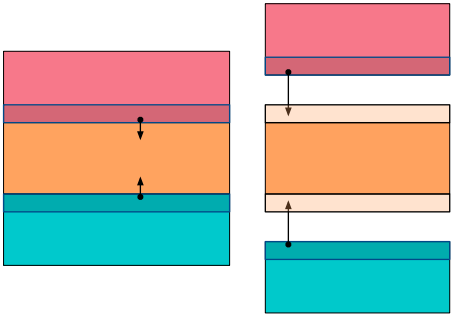
Communication among processes



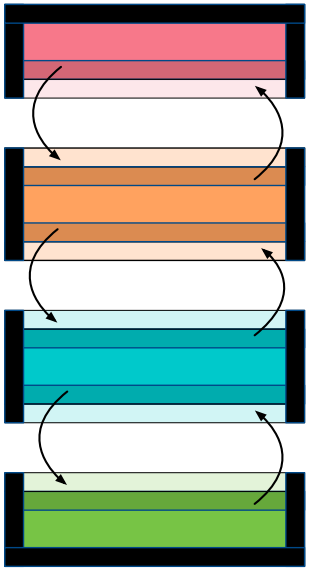
Communication will occur at the boundaries between partitions: Need to send boundary row/column to neighbor

Ghost cells (aka hallo cells)

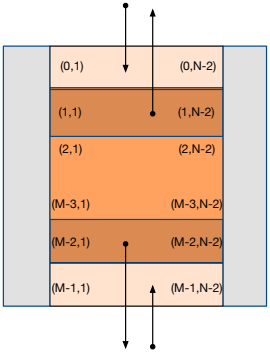
- Need place to receive copy of neighor rows.
- Will add two *ghost rows* above and beyond the rows owned by the process. These become the boundary for the local Jacobi iteration



Algorithm outline

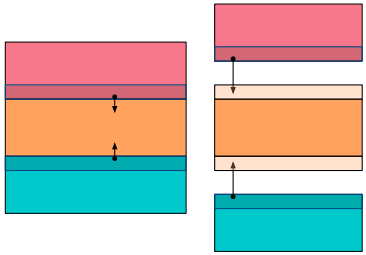


```
repeat {
update ghost rows;
compute new iteration;
}
until(converged)
```



Jacobi

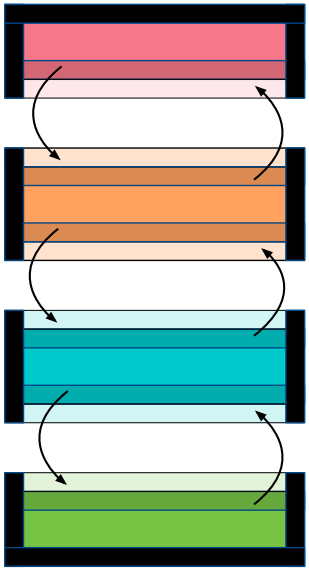
Data distribution



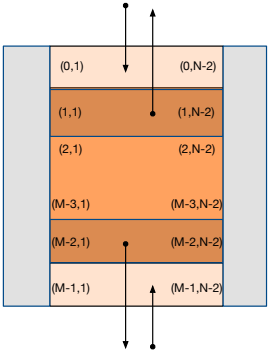
Sequential algorithm

```
do {
 err=0; l=1-l;
 for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++) {
   a[1-l][i][j]=0.25*(a[l][i-1][j]
    +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
   err = fmax(err,fabs(a[1][i][j]-a[0][i][j]));
  }
 }
while(err>maxerr);
```

Algorithm outline



```
repeat {
update ghost rows;
compute new iteration;
}
until(converged)
```

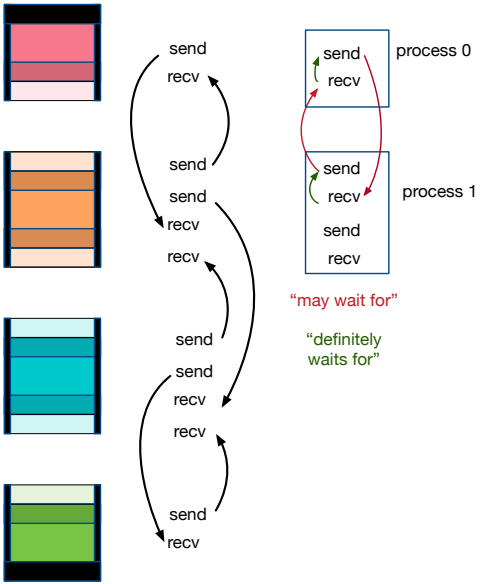


Code – first attempt (ignore convergence test)

```
double a[2][M][N];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
for(iter=0;iter<MAX;iter++) {
 if(rank>0) {
  /* send up */
  MPI_Send(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD);
   /* receive from up */
  MPI_Recv(&a[1][0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
  }
 if(rank < size-1) {
   /* send down */
  MPI_Send(&a[1][M-2][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD);
   /* receive from down */
  MPI_Recv(&a[1][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
  }
```

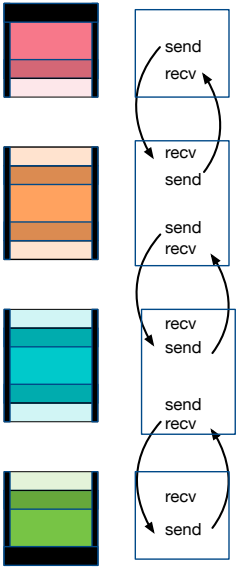
```
for (i=1;i<N-1; i++)
 for(j=1;j<N-1;j++)
  a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
l=1-l;
}
```

Deadlock is possible if send is not buffered



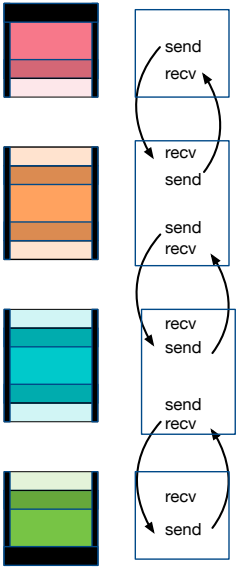
- *Deadlock*: situation where execution makes no progress.
- Typically due to a cycle of dependences: A waits for B to complete, B waits for C to complete, C waits for A to complete

Avoid deadlock, first solution



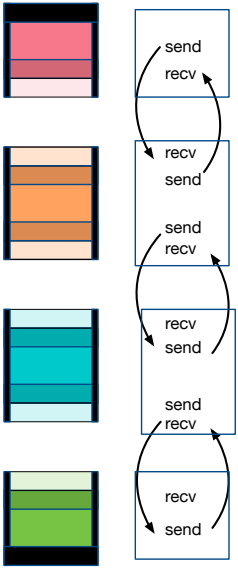
- Alternate the order of sends and receives

Avoid deadlock, first solution



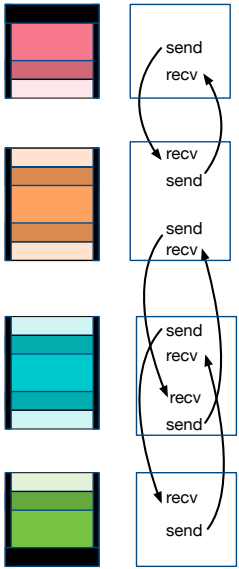
- Alternate the order of sends and receives
- Code is now deadlock-free

Avoid deadlock, first solution



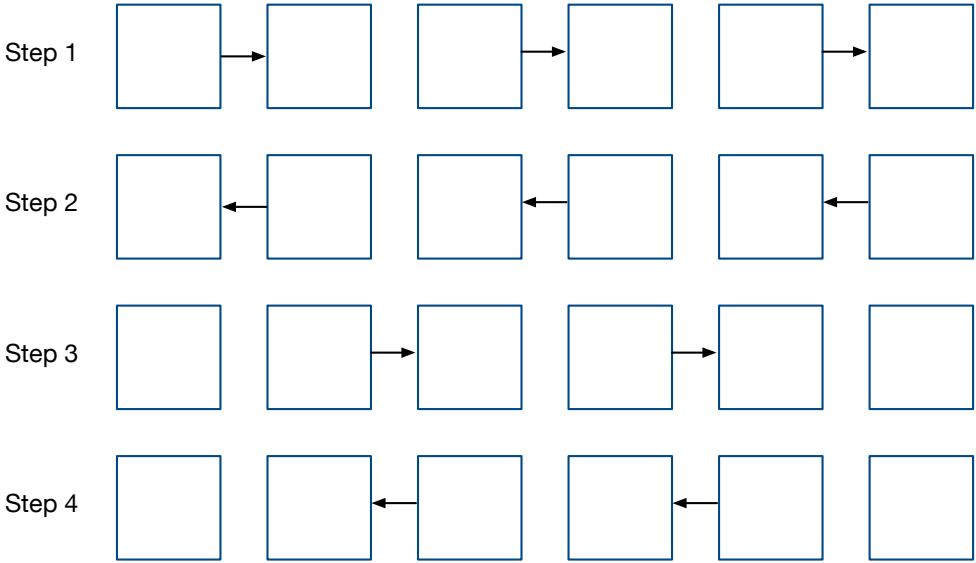
- Alternate the order of sends and receives
- Code is now deadlock-free
- But communications are serialized!

Avoid deadlock, second solution



Communicate in four rounds:

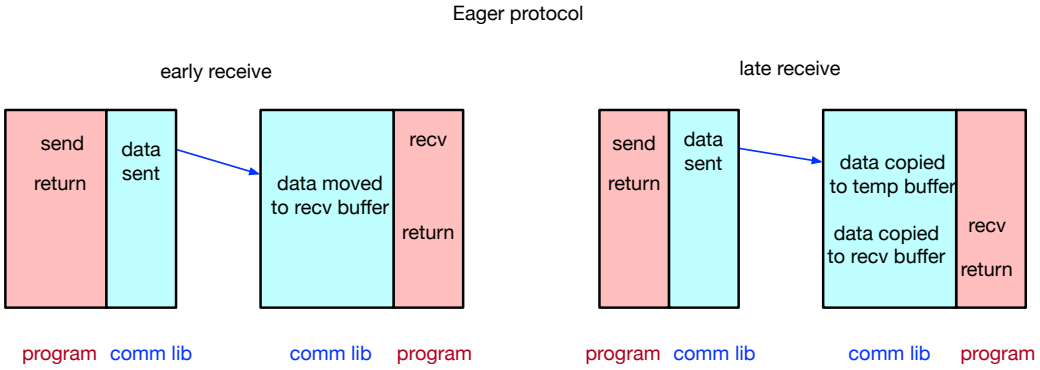
- $2i$ to $2i + 1$
- $2i + 1$ to $2i$
- $2i - 1$ to $2i$
- $2i$ to $2i - 1$


Step 1
Step 2
Step 3
Step 4

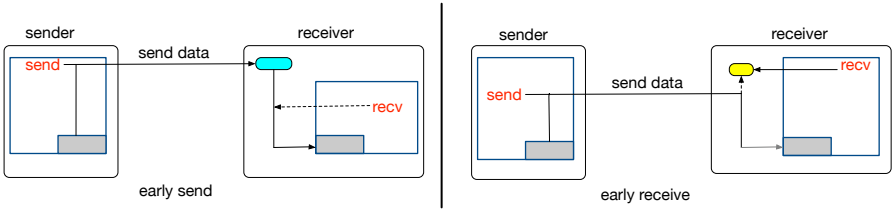
Communication Protocols

- Basic problem of send-receive communication (aka 2-sided communication): The processes have no common clock, so the send can occur before the receive or after the receive.
- If send data as soon as send occurs, then it may arrive before the receive is posted and needs to be buffered (eager protocol)
- If send data only after receive is posted, then additional communication is needed to inform the sender that the receive is posted (rendezvous protocol)

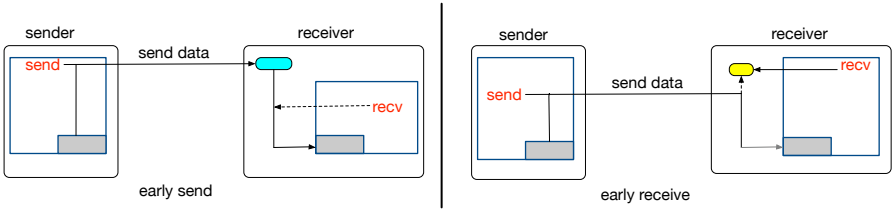
Eager protocol



Eager protocol

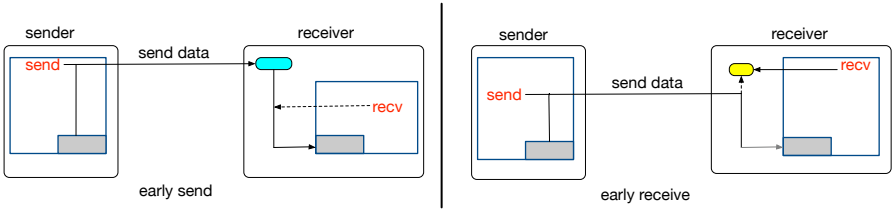


Eager protocol



Good: Simple protocol; send completes rapidly

Eager protocol

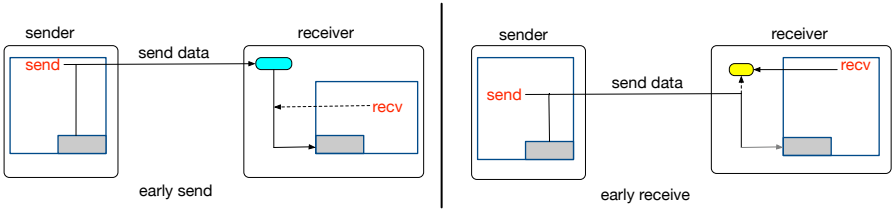


Good: Simple protocol; send completes rapidly

Bad: Extra copying; need extra buffer space and need to run protocol to prevent buffer overflow

Send returns before or after receive starts

Eager protocol



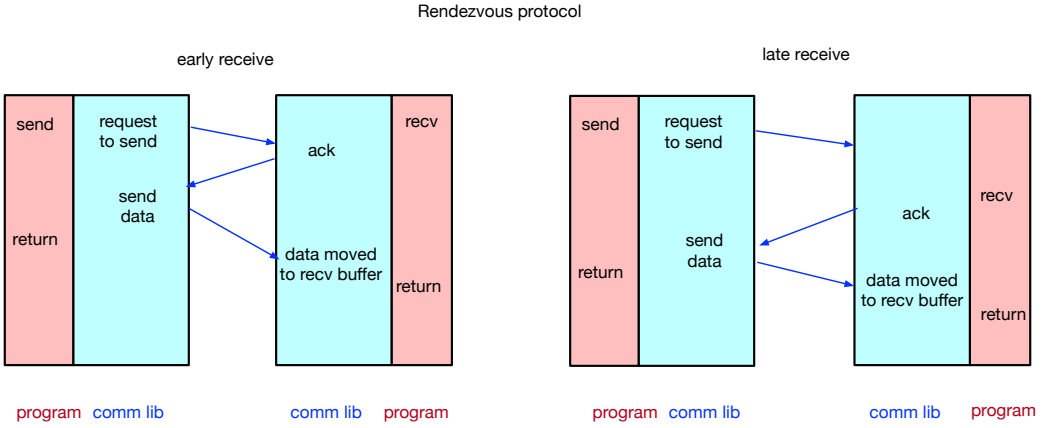
Good: Simple protocol; send completes rapidly

Bad: Extra copying; need extra buffer space and need to run protocol to prevent buffer overflow

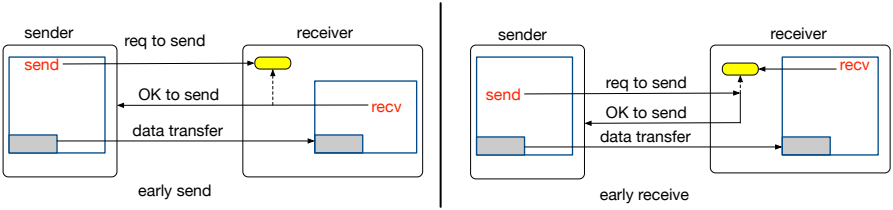
Send returns before or after receive starts

Best if receive is posted ahead of send

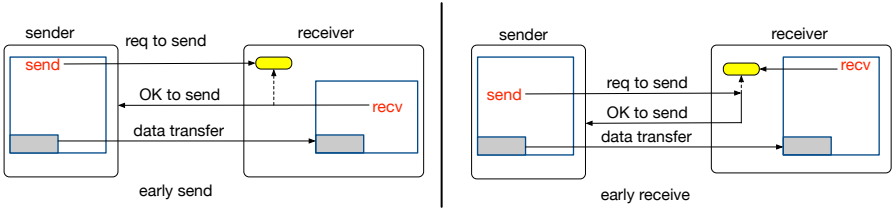
Rendezvous protocol



Rendezvous protocol

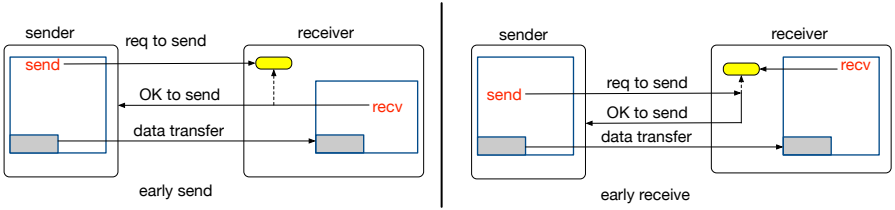


Rendezvous protocol



Good: Data moved once; need much less buffering

Rendezvous protocol

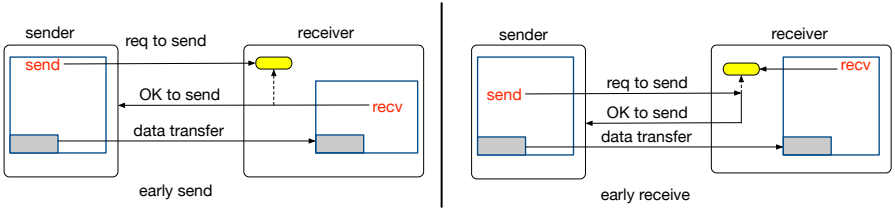


Good: Data moved once; need much less buffering

Bad: Extra protocol messages

Send returns only after receives start

Rendezvous protocol



Good: Data moved once; need much less buffering

Bad: Extra protocol messages

Send returns only after receives start

Best if receive is posted ahead of send

Typical implementation

- Uses eager protocol for short messages
- Uses rendezvous protocol for long message, when overhead of extra copy larger than overhead of extra messages .
- Always good to post receives ahead of matching sends

Back to Jacobi: Better – use *nonblocking communication*

Separate *start* of communication from *completion* of communication

```
...
int a[1000], b[1000];
MPI_Request req[2];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Isend(a,1000,MPI_INT,rank+1,0,
MPI_COMM_WORLD,&req[0])
MPI_IRecv(b,1000,MPI_INT,rank-1,0,
MPI_COMM_WORLD,&req[1]);
MPI_Wait(&req[0],MPI_STATUS_IGNORE);
MPI_Wait(&req[1],MPI_STATUS_IGNORE);
```

- A nonblocking send returns (does not block), even if matching receive has not occurred
- The *request* object identifies the started communication
- `MPI_WAIT` blocks until the communication identified by the request is complete.
- Once both send and receive are started, the communication will complete – no further MPI call is needed.

Better

```
...
int a[1000], b[1000];
MPI_Request req[2];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Isend(a,1000,MPI_INT,rank+1,0,
MPI_COMM_WORLD,&req[0])
MPI_IRecv(b,1000,MPI_INT,rank-1,0,
MPI_COMM_WORLD,&req[1]);
MPI_Waitall(2,req,MPI_STATUSES_IGNORE);
```

- Can wait for the completion of a set of communications (sends or receives)
- Also have `MPI_Wait_any`, `MPI_Wait_some`

Jacobi

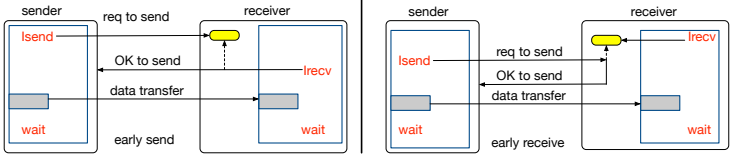
```
...
double a[2][M][N];
MPI_Request req[4];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
for(iter=0;iter<MAX;iter++) {
/* up */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[1][0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
```

```
/*down */
 if(rank<size-1) {
  MPI_Isend(&a[l][M-2][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
 if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
 else if(rank==(size-1)) MPI_Waitall(2,&req[0],MPI_STATUSES_IGNORE);
 else MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

 for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++)
   a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
 l=1-l;
}
```

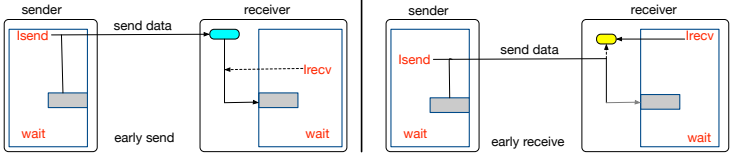
Nonblocking communication

- If wait is too early, process blocks until communication is complete.
- Start communication as soon as possible and wait as late as possible

Rendezvous



Eager



Back to Jacobi

```
...
double a[2][M][N];
MPI_Request req[4];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
for(iter=0;iter<MAX;iter++) {
/* up */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[1][0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
```

```
/*down */
 if(rank<size-1) {
  MPI_Isend(&a[l][M-2][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
 if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
 else if(rank==(size-1)) MPI_Waitall(2,&req[0],MPI_STATUSES_IGNORE);
 else MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

 for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++)
   a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
 l=1-l;
}
```

Reduction

- Convergence check: Need to compute the max of the local errors.
- Can use *collective communication*

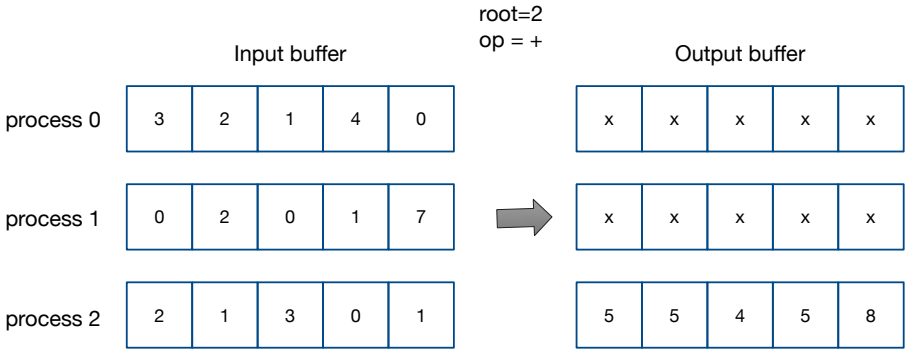
```
...
int rank,maxrank,sumrank,size;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
MPI_Reduce(&rank,&sumrank,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if(rank==0) printf('' %d %d \n'', maxrank,sumrank);
...
```

```
MPI_Reduce(&rank,&maxrank,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
```

```
int MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

- sendbuf: the location where my input comes from
- recvbuf: the location where the result goes to
- count: the length of the vector being reduced element-wise
- datatype: the type of each vector element
- op: the reduction operation
- root: the rank of the process that is gathering the result
- comm: the group of processes involved in the reduction

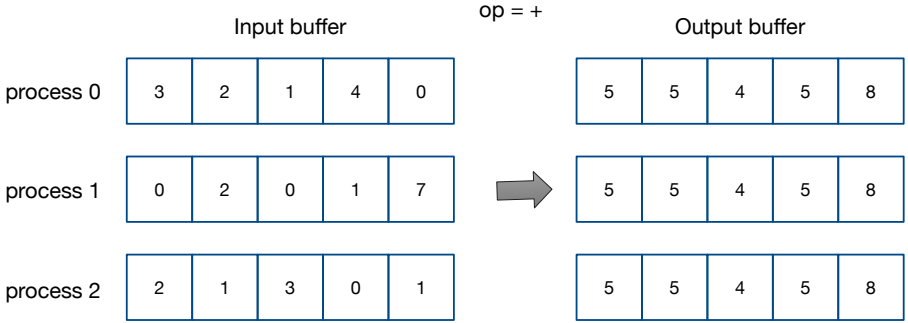
MPI_Reduce


root=2
op = +
Input buffer
Output buffer
process 0
3 2 1 4 0
x x x x x
process 1
0 2 0 1 7
x x x x x
process 2
2 1 3 0 1
5 5 4 5 8

MPI_Allreduce

- With reduce one process gets the result of the reduction and can decide whether the iteration converged. But we need *all* processes to break out of the loop.
- Use `allreduce`, instead: The result of the reduction is broadcast to all participating processes
- `MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm)`
- Same as `MPI_Reduce`, except that there is no `root` argument

MPI_Allreduce

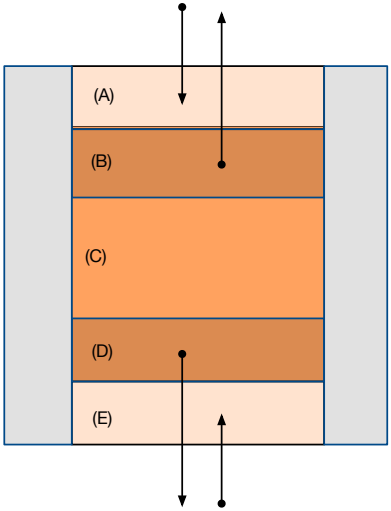


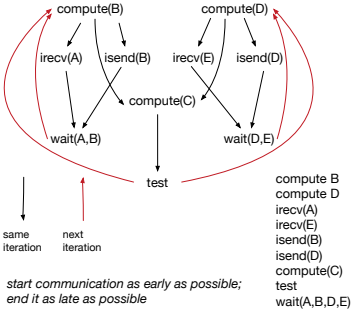
Jacobi, again

```
double a[2][M][N];
double local_err,global_err;
MPI_Request req[4];
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
...
/* assume (M-2)*size=N */
do {
/* up */
 if(rank>0) {
  MPI_Isend(&a[l][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
/*down */
 if(rank < size-1) {
  MPI_Isend(&a[l][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
```

```
if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
else if(rank==(size-1)) MPI_Waitall(2,&req[0],MPI_STATUSES_IGNORE);
else MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

local_err=0;
for (i=1;i<N-1; i++)
 for(j=1;j<N-1;j++) {
  a[1-l][i][j]=0.25*(a[l][i-1][j]
      +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
 local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
 }
l=1-l;
MPI_Allreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);
} while(global_err>maxerr);
```

Optimize: overlap computation and communication





Optimized jacobi

```
...
MPI_Request req[5];
/* loop */
do {
 local_err=0; l=1-l;
/* compute top, bottom rows */
 for(j=1;j<N-1;j++) {
  a[1-l][1][j]=0.25*(a[l][0][j]
     +a[l][2][j]+a[l][1][j-1]+a[l][1][j+1]);
  local_err = fmax(local_err,fabs(a[1][1][j]-a[0][1][j]));
  }
 for(j=1;j<N-1;j++) {
  a[1-l][M-2][j]=0.25*(a[l][M-3][j]
   +a[l][M-1][j]+a[l][M-2][j-1]+a[l][M-2][j+1]);
  local_err = fmax(local_err,fabs(a[1][M-2][j]-a[0][M-2][j]));
  }
```

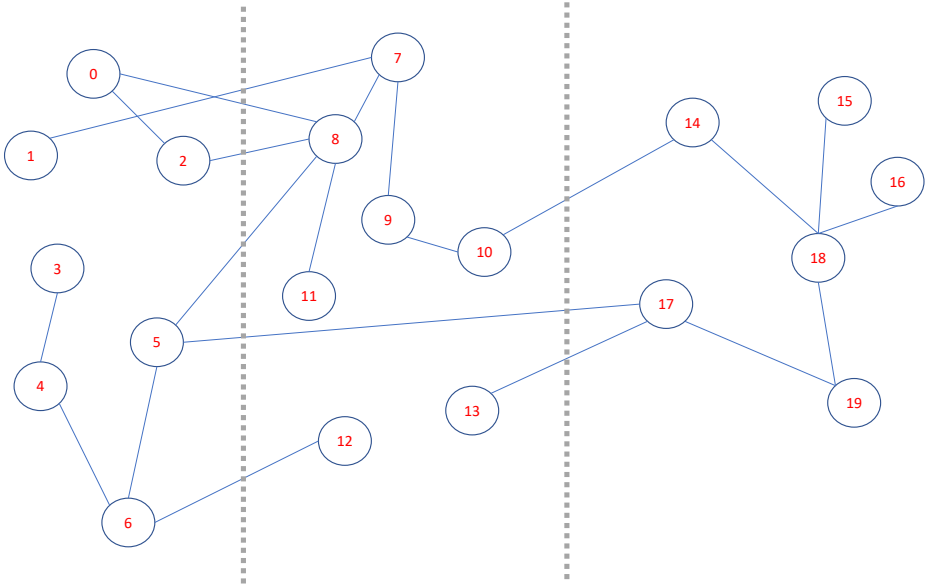
```
/* start communications */
 if(rank>0) {
  MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[0]);
  MPI_Irecv(&a[0][1],N-2,MPI_DOUBLE,rank-1,0,MPI_COMM_WORLD,&req[1]);
  }
 if(rank < size-1) {
  MPI_Isend(&a[1][M-1][1],N-2, MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[2]);
  MPI_Irecv(&a[1][M-1][1],N-2, MPI_DOUBLE,rank+1,0,MPI_COMM_WORLD,&req[3]);
  }
/* compute interior */
 for (i=2;i<M-2; i++)
  for(j=1;j<N-1;j++) {
   a[1-l][i][j]=0.25*(a[l][i-1][j]
        +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
   local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
   }
```

```
/* start reduction */
 MPI_Iallreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,
      MPI_COMM_WORLD,&req[4]);

/* end pt-2-pt communications */
 if(rank==0) MPI_Waitall(2,&req[2],MPI_STATUSES_IGNORE);
 else if(rank==(size-1)) MPI_Waitall(2,&req[0], MPI_STATUSES_IGNORE);
 else MPI_Waitall(4, req, MPI_STATUSES_IGNORE);
/* end reduction */
 MPI_Wait(req[4], MPI_STASTUS_IGNORE)
 }  while(global_err>maxerr);
```

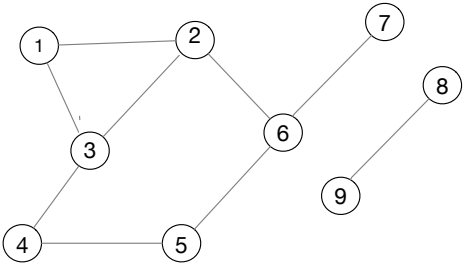
Shared memory vs. distributed memory Jacobi

- Computation partitioned in similar ways (tiling)
- Dynamic partitioning and overpartitioning seldom used for distributed memory
- Distributed memory (with MPI) adds the burden of
  - Distributing data, and using local indices/pointers
  - Communicating data (ghost cells)
  - Replicating sequential code

Distributed memory graph traversal



Distributed memory graph traversal



1	2 3
2	1 3 6
3	1 2 4
4	3 5
5	4 6
6	2 5 7
7	6
8	9
9	8

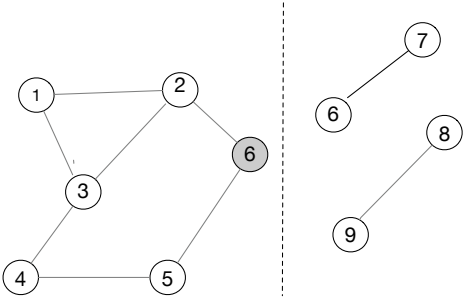
Distributed graph representation

- Nodes are distributed across processes
- If node u owned by a process connects to a node v owned by another process then a "ghost copy" of v is stored at the first process

Algorithm:

- Process 0 starts by traversing its nodes, starting from node 0
- If traversal encounters a ghost node, then a request is send to node owner to start a traversal from that node

Distributed graph representation (2 processes)



1	2	3	
2	1	3	6
3	1	2	4
4	3	5	
5	4	6	

6	2	5	7
7	6		
8	9		
9	8		

Partial, inefficient and incomplete traversal code

```
...
typedef struct {
 bool visited;
 int degree; // number of neighbors
 int neighbor[];// indices of neighbors; use reversed sign to mark ghosts
} Node;

Node *node; // local array of nodes
int nnodes; // number of nodes
MPI_Requests reqs[max];
int reqcount // index for requests;

void graph_init() {}
```

```
void visit(int i) {
 int j,k,m;
 if (!node[i]->visited) {
  node[i]->visited=true;
  for(j=0;j<node[i]->degree;j++) {
   k=node[i]->neighbor[j];
   if(isGhostCopy(k))
    MPI_Isend(k, 1, MPI_INT,
      k*size/nnodes, 0, MPI_COMM_WORLD,reqs[reqcount++]);
   else
    visit(k)}
 }
}
```

```
int main(int argc, char **argv) {
 int i,j;
 MPI_Request recvreq;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 graph_init();

 MPI_Irecv(&i, 1, MPI_INT, MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &recvreq);
 if(rank==0)
  visit(0);

 while(notdone) {
  MPI_Wait(&recvreq,MPI_STATUS_IGNORE);
  j=i;
  MPI_Irecv(&i, 1, MPI_INT, MPI_ANY_SOURCE,0, MPI_COMM_WORLD, &recvreq);
  visit(j);
 // complete all sends of visit
  MPI_Waitall(reqs,reqcount,MPI_STATUSES_IGNORE);
  reqcount=0;
  }

 MPI_Finalize();
}
```

Problems

- Correctness: How do we know computation has completed?
  - Need to periodically test for completion; i.e., that all processes are done
- Performance: Sending too many small messages
  - Need to aggregate messages and send them in bulk

Bulk-synchronous code

- Computation proceeds in phases, with communication at end of each phase
- At each phase do as much work as possible without communicating

```
typedef struct {
  bool visited;
  int degree;
  int neighbor[]; // mark ghost cell with negative index
} Node;
Node *node[nnodes];
void graph_init() {}

int rank, size;
int **out;     // send buffers (one per destination)
int *out_ptr; // points to first empty slot in output buffer
int *in;       // receive buffer
int in_ptr;    // points to first empty slot in input buffer
MPI_Request *reqs;
MPI_Status status;
MPI_Request *reqs;
MPI_Status status;
 ...
```

```
void visit(int i) {
  int j,k,m;
  if (!node[i]->visited) {
    node[i]->visited=true;
    for(j=0;j<node[i]->degree;j++) {
      k= node[i]->neighbor[j];
      if(isGhostCopy(k)) { // ghost node; append to  appropiate output list
        m = k*size/nnodes; // destination
        out[m][out_ptr[m]++]=k;
      }
      else
      visit(k);
    }
  }
}
```

```
int main(int argc, char **argv) {
  int i,j,k;
  bool done, global_done;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  graph_init();
  for(i=0;i<size;i++) out[size] = (int*)malloc(max*sizeof(int));
  in = (int *)malloc(max*size*sizeof(int));
  reqs = (MPI_Request *)malloc(size*sizeof(MPI_Request));
  out_ptr = (int *)malloc(size*sizeof(int));
  if(rank==0)
     in[in_ptr++] =0;
```

```
while (1) {
 for(i=0;i<size;i++)
  out_ptr[i]=0;

// visit all newly marked nodes
 for(i=0; i<in_ptr ;i++)
  visit(in[i]);
 in_ptr=0;

// send out marked ghost nodes
 k=0;
 for(i=0;i<size;i++)
  if (i!=rank)
 MPI_Isend(out[i],out_ptr[i], MPI_INT, i, 0, MPI_COMM_WORLD, &reqs[k++]);
```

```
 // receive marked ghost nodes
  done=true;
  for(i=0;i<size;i++)
   if(i !=rank) {
    MPI_Recv(&in[in_ptr], max, MPI_INT, MPI_ANY_SOURCE, 0,
    MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT,&j);
    if(j>0) done=false;
    in_ptr+=j;
    }

 // complete sends
  MPI_Waitall(reqs,size-1,MPI_STATUSES_IGNORE)

 // test for completion
  MPI_Allreduce(&done,&global_done, 1, MPI_C_BOOL,MPI_LAND, MPI_COMM_WORLD);
  if(global_done) break;
  }
}
```

New Constructs

- `MPI_ANY_SOURCE` – wildcard source
- Also have `MPI_ANY_TAG`
- `status` object, of type `MPI_Status`
- `MPI_Get_count(status, datatype, count)`
- Also can directly access `status.MPI_SOURCE` and `status.MPI_TAG`

Graph traversal Possible Improvements

- All processes first send to process 0 then to process 1, etc.;
- Communication not overlapped optimally
- Better: Process i sends to $i+1, i+2, \ldots$ and receives from $i-1, i-2, \ldots$

```
k=0;
m = rank;
for(i=0;i<size-1;i++) {
 m=(m+1)%size;
MPI_Isend(out[m],out_ptr[m], MPI_INT, m, 0, MPI_COMM_WORLD, &reqs[k++]);
```

- Use multiple receive buffers, so that all receives can be started at the same time

- Use multiple receive buffers, so that all receives can be started at the same time
- But, then may need to allocate more space
- Check how much space each send needs, then allocate receive buffer

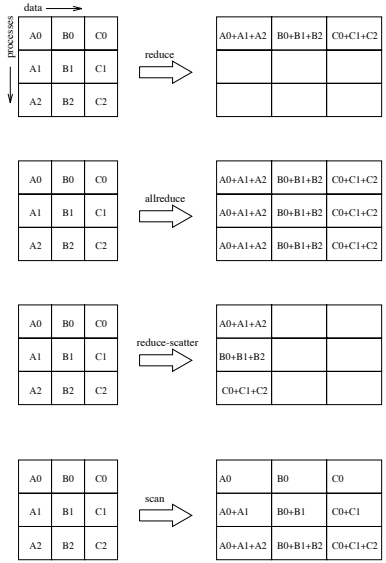
```
k=0;
m = rank;
for(i=0;i<size-1;i++) {
 m=(m-1)%size;
 MPI_Probe(m,0,MPI_COMM_WORLD,status);
 MPI_Get_count(&status, MPI_INT,&j);
 MPI_Irecv(in[in_ptr], j, MPI_INT, m, 0, MPI_COMM_WORLD, &reqs[k++]);
 int_ptr += j;
}
```

- `MPI_Probe(source, tag, comm, status)`: Returns information about incoming message without actually receiving it

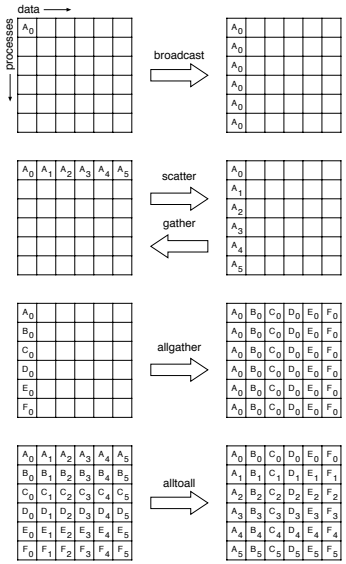
Collective communications

- Computation collectives: `Reduce`, `Allreduce`, `Reducescatter`
- Synchronization collective: `Barrier`
- Data movement collectives: `broadcast`, `scatter`, `gather`, `allgather`, `alltoall`

Computation collectives



Communication collectives



alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcount` number of elements received from any process
- `recvtype` data type of receive buffer elements
- `comm` communicator
- `request` communication request

Alltoall

```
...
// run on 4 processes, each process has 2 elements, and the recv is filled
// with 2 elements from each process (including itself)
int send[2];
int recv[8];
MPI_Alltoall(&send, 2, MPI_INT, recv, 2, MPI_INT, MPI_COMM_WORLD);
```

`MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)`

Each process sends to every process different amounts of data

- `sendbuf` send buffer
- `sendcounts` array with number of elements sent to each process
- `sdispls` array with displacement from `sendbuf` start of each sent block
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcounts` array with number of elements received from each process
- `rdispls` array with displacement from `recvbuf` start of each received block
- `recvtype` data type of receive buffer elements
- `comm` communicator
- `request` communication request

Alltoallv

```
...
/* run on 3 processes; p0 has 3 elements (a,b,c), p1 has 3 elements (d,e,f),
p2 has 5 elements (g,h,i,j,k);
after the call, the recv buffer at p0 contains (a,g), p1 contains (b,c,h),
and p3 contains (e,f) */

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int send_counts[3], recv_counts[3], send_displs[3], recv_displs[3];
int *send, *recv;
if (rank == 0) {
  send = (int *) malloc(3 * sizeof(int));
  recv = (int *) malloc(2 * sizeof(int));
  send[0] = 'a'; send[1] = 'b'; send[2] = 'c';
  send_counts[0] = 1; send_counts[1] = 2; send_counts[2] = 0;
  send_displs[0] = 0; send_displs[1] = 1; send_displs[2] = 3;
  recv_counts[0] = 1; recv_counts[1] = 0; recv_counts[2] = 1;
  recv_displs[0] = 0; recv_displs[1] = 1; recv_displs[2] = 1;
}
```

Alltoallv

```
else if (rank == 1) {
  send = (int *) malloc(3 * sizeof(int));
  recv = (int *) malloc(3 * sizeof(int));
  send[0] = 'd'; send[1] = 'e'; send[2] = 'f';
  send_counts[0] = 0; send_counts[1] = 0; send_counts[2] = 2;
  send_displs[0] = 0; send_displs[1] = 0; send_displs[2] = 1;
  recv_counts[0] = 2; recv_counts[1] = 0; recv_counts[2] = 1;
  recv_displs[0] = 0; recv_displs[1] = 2; recv_displs[2] = 2;
}
```

Alltoallv

```
else if (rank == 2) {
  send = (int *) malloc(5 * sizeof(int));
  recv = (int *) malloc(2 * sizeof(int));
  send[0] = 'g'; send[1] = 'h'; send[2] = 'i'; send[3] = 'j'; send[4] = 'k';
  send_counts[0] = 1; send_counts[1] = 1; send_counts[2] = 0;
  send_displs[0] = 0; send_displs[1] = 1; send_displs[2] = 0;
  recv_counts[0] = 0; recv_counts[1] = 2; recv_counts[2] = 0;
  recv_displs[0] = 0; recv_displs[1] = 0; recv_displs[2] = 0;
}
MPI_Alltoallv(send, send_counts, send_displs, MPI_INT,
  recv, recv_counts, recv_displs, MPI_INT, MPI_COMM_WORLD);
```

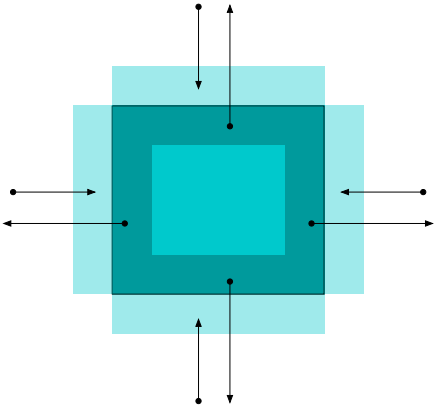
Example: graph traversal with Alltoallv

First communicate size of messages, then communicate data
Reminder:

- `out[i][]` contains indices to be sent to process `i`. Each row has `max` elements
- `out_ptr[i]` contains number of indices to be sent to process `i`
- `in[]` is the receive buffer for all indices.

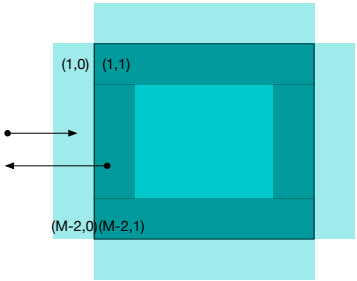
```
for(i=0;i<size;i++)
 sdispl[i] = rank*max;
out_ptr(rank) = 0;
    ...
MPI_Alltoall(out_ptr,1,MPI_INT,recvcounts,1,MPI_INT,MPI_COMM_WORLD)
rdipls[0]=0;
for (i=1; i<size; i++)
 rdispls[i]= rdispls[i-1]+recvcounts[i];
MPI_Ialltoallv(out,out_ptr,sdipls,MPI_INT,in,recvcounts,rdispls,MPI_INT,MPI_CO
...
```

Jacobi – 2D decomposition



- Problem: Columns are not stored in consecutive locations
- Solution 1: Pack to consecutive locations to send; unpack after receiving
- Solution 2: Define a suitable datatype for the send and receive locations
- Solution 2 is better *if the MPI implementation does a good job*

Option 1




```
...
MPI_Irecv(fromleft,M-2,MPI_DOUBLE,myrank-1,
    0,MPI_COMM_WORLD,&req[0]);
for(i=1;i<M-1;i++)
 toleft[i]=a[1][i][1];
MPI_Isend(toleft,M-2,MPI_DOUBLE,myrank-1,
    0,MPI_COMM_WORLD,&req[1]);
...
MPI_Waitall(...)
for(i=1;i<M-1;i++)
 a[1][i][0]=fromleft[i];
 ...
```

Derived datatypes


```
MPI_Type_vector(3,2,6,MPI_DOUBLE,&vector-type);
MPI_TYPE_VECTOR(count,blocklength,stride,
oldtype,newtype
```

count: number of blocks

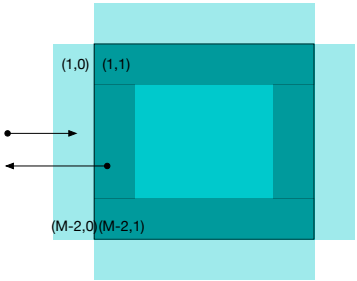
blocklength: number of elements per block (usually 1)

stride: distance from start of block to start of next block

oldtype: type of each element

newtype: name of new derived datatype

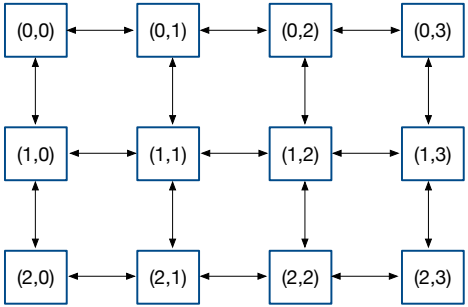
Option 2




```
...
MPI_Type_vector(M-2,1,N,MPI_DOUBLE,&vector_type
MPI_Type_commit(&vector_type);
MPI_Irecv(&a[1][1][0],1,vector_type,myrank-1,
  0,MPI_COMM_WORLD,&req[0]);
MPI_Isend(&a[1][1][1],1,vector_type,myrank-1,
   0,MPI_COMM_WORLD,&req[1]);
...
MPI_Waitall(...)
...
```

Cartesian grid of processes

- For the 2D Jacobi, it is convenient to think of the processes as organized in a 2D mesh
- Can do this with `MPI_Cart_create`




```
dims[0]=4; dims[1]=3;
periods[0]=false; periods[1]=false;
reorder= true;
ndim=2;
MPI_Cart_create(MPI_COMM_WORLD,ndim,
dims,periods,reorder,&comm2d);
```

- May want to know my Cartesian coordinates: `MPI_Cart_Coords()`
- May want to know my rank and the rank of one of my neighbors: `MPI_Cart_shift(comm2d,direction,disp,&rank_source,&rank_dest)`
- What happens if there is no neighbor in the chosen direction? The call returns `MPI_PROC_NULL`; a send to `MPI_PROC_NULL` or a receive from `MPI_PROC_NULL` is a noop.

2D Jacobi, in all its glory...

```
...
/* initialization */
MPI_Comm_size(MPI_Comm_World,&size)
/* pick grid dimensions */
MPI_Dims_create(size,2,dims)
/* dims[0]*dims[1] = size */
period[0]=period[1]=false;
MP_Cart_create(MPI_COMM_WORLD,2,dims,periods,true,&comm2d);
MPI_Cart_shift(comm2d,0,-1,&myrank,&up);
MPI_Cart_shift(comm2d,0,1,&myrank,&down);
MPI_Cart_shift(comm2d,1,-1,&myrank,&left);
MPI_Cart_shift(comm2d,1,1,&myrank,&right);

MPI_Type_vector(M-2,1,N,MPI_DOUBLE,&vector_type);
MPI_Type_commit(&vector_type);
```

```
/* loop */
do {
 local_err=0;
/* compute boundaries */
/* top */
for(j=1;j<N-1;j++) {
 a[1-1][1][j]=0.25*(a[1][0][j]
   +a[1][2][j]+a[1][1][j-1]+a[1][1][j+1]);
 local_err = fmax(local_err,fabs(a[1][1][j]-a[0][1][j]));
}
/* bottom */
for(j=1;j<N-1;j++) {
 a[1-1][M-2][j]=0.25*(a[1][M-3][j]
  +a[1][M-1][j]+a[1][M-2][j-1]+a[1][M-2][j+1]);
 local_err = fmax(local_err,fabs(a[1][M-2][j]-a[0][M-2][j]));
}
```

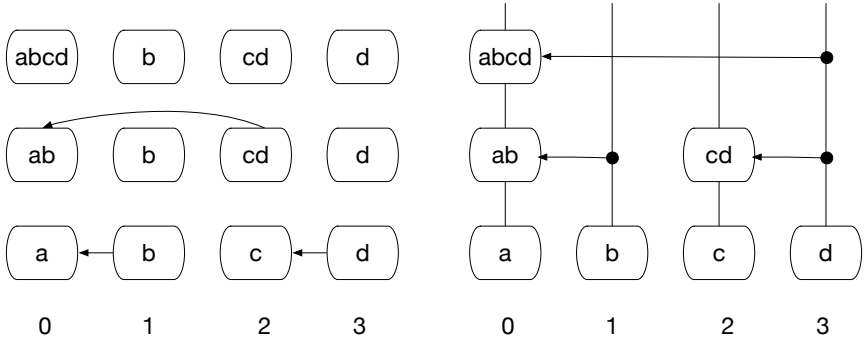
```
/* left */
for(i=1;i<M-1;i++) {
 a[1-l][i][1]=0.25*(a[l][i-1][1]
  +a[l][i+1][1]+a[l][i][0]+a[l][i][2]);
 local_err = fmax(local_err,fabs(a[1][i][1]-a[0][i][1]));
}
/* right */
for(i=1;i<M-1;i++) {
 a[1-l][i][N-2]=0.25*(a[l][i-1][N-3]
  +a[l][i+1][N-1]+a[l][i][N-3]+a[l][i][N-1]);
 local_err = fmax(local_err,fabs(a[1][i][N-2]-a[0][i][N-2]));
}
/* start communications */
MPI_Irecv(&a[l][0][1],N-2,MPI_DOUBLE,up,0,MPI_COMM_WORLD,&req[1]);
MPI_Irecv(&a[l][M-1][1],N-2,MPI_DOUBLE,down,0,MPI_COMM_WORLD,&req[2]);
MPI_Irecv(&a[l][1][0],1,vector_type,left,MPI_COMM_WORLD,&req[3]);
MPI_Irecv(&a[l][1][N-1],1,vector_type,right,MPI_COMM_WORLD,&req[4])
MPI_Isend(&a[l][1][1],N-2,MPI_DOUBLE,up,MPI_COMM_WORLD,&req[5]);
MPI_Isend(&a[l][M-2][1],N-2,MPI_DOUBLE,down,MPI_COMM_WORLD,&req[6]);
MPI_Isend(&a[l][1][1],1,vector_type,left,MPI_COMM_WORLD,&req[7]);
MPI_Isend(&a[l][1][N-2],1,vector_type,right,MPI_COMM_WORLD,&req[8]);
```

```
/* compute interior */
for (i=2;i<M-2; i++)
 for(j=2;j<N-2;j++) {
  a[1-l][i][j]=0.25*(a[l][i-1][j]
   +a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
  local_err = fmax(local_err,fabs(a[1][i][j]-a[0][i][j]));
}
l=1-l;

/* start convergence test */
MPI_Iallreduce(&local_err,&global_err,1,MPI_DOUBLE,MPI_MAX,
  MPI_COMM_WORLD,&req[9]);

/* end communications */
MPI_Waitall(9,req,MPI_STATUSES_IGNORE);

}  while(global_err>maxerr);
```

How is reduce implemented?

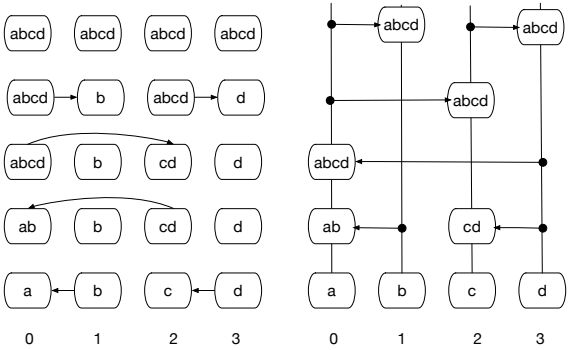
- Assume we reduce vector of length n
- Simple implementation: All processes send message to root; root sums them all.
- Assume receive of n words takes time $\ell + n/b$; only one receive at a time
- Communication time is $(p-1)(\ell + n/b) = \Theta(pn)$ – not good

Binary reduction



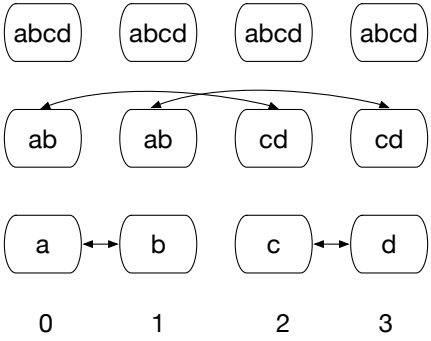
- Communication time is $\lg(p)(\ell + n/b) = \Theta(log(p) \cdot n)$ – much better
- Different processes exit collective operation at different times

Allreduce – reduce followed by broadcast



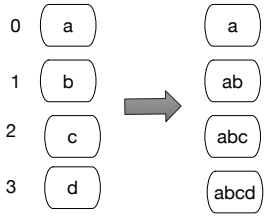
- Communication time is $2\lg(p)(\ell + n/b)$

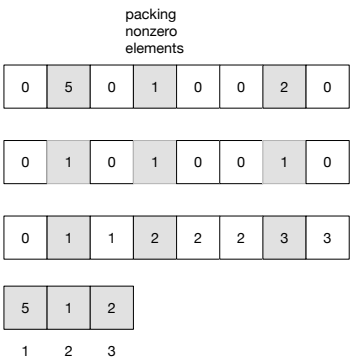
Allreduce – alternative implementation



- Assume send and receive can be simultaneous: Communication time is $\lg(p)(\ell + n/b)$ – half as much as previous algorithm
- Assume no overlap at all between send and receive; then the two algorithms take as much time.

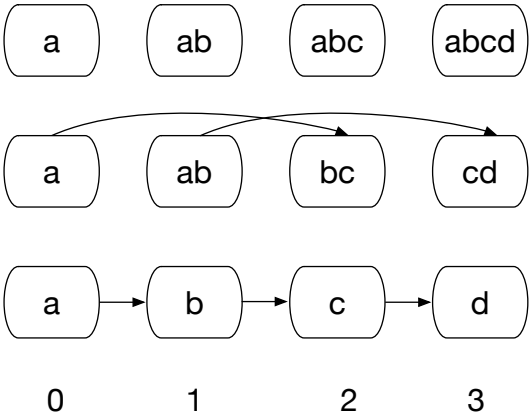
Examples


0
a
1
b
2
c
3
d
a
ab
abc
abcd


packing
nonzero
elements
0 5 0 1 0 0 2 0
0 1 0 1 0 0 1 0
0 1 1 2 2 2 3 3
5 1 2
1 2 3


finding
leading
nonzero
0 0 0 1 0 1 1 0
0 0 0 1 1 2 3 3
0 0 0 1 1 2 3 3

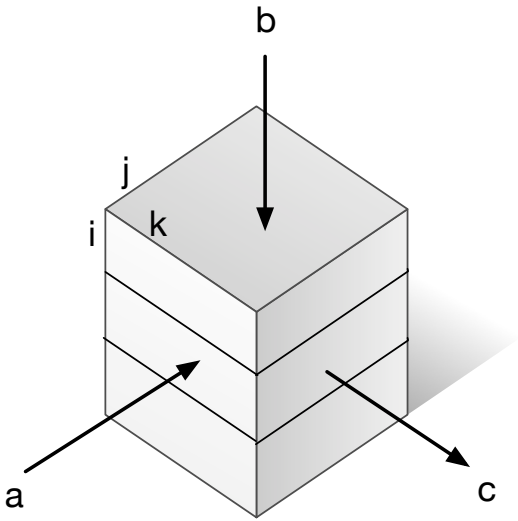
Scan evaluation



- Assuming simultaneous send and receive
- Time is $\lg(p)(\ell + n/b)$

Good MPI implementation uses *polyalgorithm*: It chooses the right algorithm, based on machine properties (latency, bandwidth), number of involved processes, length of messages, etc.

Matrix product 1D tiling



Will parallelize with MPI, by assigning a tile to each process

Code

```
...
int matmult(int nra, int nca, int ncb, double a[nra][nca],
double b[nca][ncb], double c[nra][ncb], MPI_Comm comm) {
    /*******************
    Computes the product c = a*b using the processes in the group of
    communicator comm
    The three matrices are stored on the process with rank 0 in comm
    ************************/
    int numprocs,                /* number of processes in group */
    procid,                  /* a process identifier */
    numworkers,              /* number of worker processes */
    source,                  /* process id of message source */
    dest,                    /* process id of message destination */
    rows,                    /* rows of matrix A sent to each worker */
    averow, extra, offset, /* used to determine rows sent to each worker */
    i, j, k, rc;             /* misc */

    MPI_Comm_rank(comm,&procid);
    MPI_Comm_size(comm,&numprocs);
```

```
if (numprocs < 2 ) {
    /* run sequential code */
    for (i=0;i<nra;i++)
    for (j=0;j<ncb;j++)
    for(k=0;k<nca;k++)
    c[i][j] += a[i][k]*b[k][j];
    exit(1);
}
```

```
if(procid==0) {
/**************  master process ***************/
numworkers = numprocs-1;
/* Send matrix data to the worker tasks */
averow = nra/numworkers;
extra = nra%numworkers;
offset = 0;
for (dest=1; dest<=numworkers; dest++)
{
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, 0, comm);
    MPI_Send(&rows, 1, MPI_INT, dest, 0, comm);
    MPI_Send(&a[offset][0], rows*nca, MPI_DOUBLE, dest, 0,
    comm);
    MPI_Send(b, nca*ncb, MPI_DOUBLE, dest, 0, comm);
    offset = offset + rows;
}
```

```
/* Receive results from worker processes */
for (source=1; source<=numworkers; source++)
{
MPI_Recv(&offset, 1, MPI_INT, source, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&rows, 1, MPI_INT, source, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&c[offset][0], rows*ncb, MPI_DOUBLE, source, 0,
comm, MPI_STATUS_IGNORE);
}
}
```

```
/**************************** worker task ************************************/
else {
MPI_Recv(&offset, 1, MPI_INT, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(&rows, 1, MPI_INT, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(a, rows*nca, MPI_DOUBLE, 0, 0, comm, MPI_STATUS_IGNORE);
MPI_Recv(b, nca*ncb, MPI_DOUBLE, 0, 0, comm, MPI_STATUS_IGNORE);

for (i=0; i<rows; i++)
for (j=0; j<ncb; j++) {
c[i][j] = 0;
for (k=0;k<nca;k++)
c[i][j] += a[i][j] * b[j][k];
}
MPI_Send(&offset, 1, MPI_INT, 0, 0, comm);
MPI_Send(&rows, 1, MPI_INT, 0, 0, comm);
MPI_Send(&c, rows*ncb, MPI_DOUBLE, 0, 0, comm);
}
return(1);
}
```

- Matmult is invoked on all processes – should be a collective invocation
- Matrices `a`,`b` and `c` are allocated on all processes – not a big problem?
- (Usually matmult is invoked while matrices are already distributed)

Possible improvements:

- Involve process 0 in computation as well
- Avoid communicating `offset` and `rows`
- Use collective communications (scatter `a`, broadcast `b`, gather `c`)

uneven scatter-gather

Problem: Not each worker receives same number of rows; need to use scatterv and gatherv functions.

`MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)`

- sendcounts: Number of elements sent to each process
- displs: `displs[i]` is displacement from start of buffer to start of elements sent to process `i`

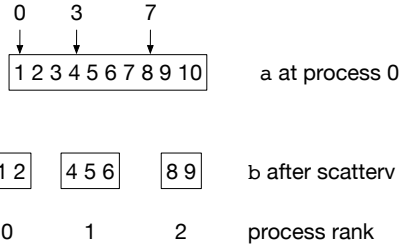
`MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)`

Arguments that are not needed can be NULL

Example

Assume 3 processes

```
/* array arguments needed at root */
int a[10] ={1,2,3,4,5,6,7,8,9,10};
int counts ={2,3,2}
int displs={0,3,7}]

int b[3];

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Scatterv(&a, counts, displs, MPI_INT, &b,
count[rank], MPI_INT,0 , MPI_COMM_WORLD);
```



Improved code

```
...
int matmult(int nra, int nca, int ncb, double a[nra][nca],
double b[nca][ncb], double c[nra][ncb], MPI_Comm comm) {

int numprocs,                /* number of processes in group */
procid,                   /* a process identifier */
source,                   /* process id of message source */
dest,                     /* process id of message destination */
averow, extra, offset,    /* used to determine rows sent to each worker */
i, j, k;                   /* misc */

MPI_Comm_rank(comm,&procid);
MPI_Comm_size(comm,&numprocs);

int  rows[numprocs],            /* # rows sent to each process */
countsa[numprocs],          /* # elements of a sent to each process */
countsc[numprocs],          /* # elements of c received form each process */
displsa[numprocs],          /* displacements in matrix a */
displsc[numprocs];          /* displacements in matrix c */
```

```
if (numprocs < 2 ) {
/* run sequential code */
for (i=0;i<nra;i++)
for (j=0;j<ncb;j++)
for(k=0;k<nca;k++)
c[i][j] += a[i][k]*b[k][j];
exit(1);
}


/* computed by all processes */
averow = nra/numprocs;
extra = nra%numprocs;
offset=0;
for (i=0; i<numprocs; i++) {
rows[i] = (i < extra) ? averow+1 : averow;
countsa[i] = rows[i]*nca;
countsc[i] = rows[i]*ncb;
displsa[i] = offset*nca;
displsc[i] = offset*ncb;
offset += rows[i];
}
```

```
/* Scatter matrix data to all processes */

double aa[rows[procid]][nca];  /* local tile of a */
double cc[rows[procid]][ncb]; /* local tile of c */

MPI_Scatterv(&a[0][0], countsa, displsa, MPI_DOUBLE, &aa[0][0],
countsa[procid], MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&b[0][0], nca*ncb, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* local computation */

for (i=0; i<rows[procid]; i++)
for (j=0; j<ncb; j++) {
cc[i][j] = 0;
for (k=0;k<nca;k++)
cc[i][j] += aa[i][k] * b[k][j];
}
/* gather results */
MPI_Gatherv(&cc[0][0], countsc[procid], MPI_DOUBLE,
&c[0][0], countsc, displsc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
return(1);
}
```

Consider two MPI processes, P1 and P2. Answer on the likelihood of the following program deadlocking:

P1:	P2:
MPI_Send(P2) MPI_Recv(P2)	MPI_Send(P1) MPI_Recv(P1)

Answer

- Never deadlocks
- May deadlock
- Always deadlocks

Consider two MPI processes, P1 and P2. Answer on the likelihood of the following program deadlocking:

P1:	P2:
MPI_Send(P2) MPI_Recv(P2)	MPI_Send(P1) MPI_Recv(P1)

Answer

- Never deadlocks
- May deadlock
- Always deadlocks

Consider two MPI processes, P1, P2, and P3. Answer on the likelihood of the following program deadlocking:

P1:	P2:	P3:
MPI_Send(P2) MPI_Recv(P3)	MPI_Send(P3) MPI_Recv(P1)	MPI_Recv(P2) MPI_Send(P1)

Answer

- Never deadlocks
- May deadlock
- Always deadlocks

Consider two MPI processes, P1, P2, and P3. Answer on the likelihood of the following program deadlocking:

P1:	P2:	P3:
MPI_Send(P2)	MPI_Send(P3)	MPI_Recv(P2)
MPI_Recv(P3)	MPI_Recv(P1)	MPI_Send(P1)

Answer

- Never deadlocks
- May deadlock
- Always deadlocks