CS420 – Lecture 9

Marc Snir

Fall 2018

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!

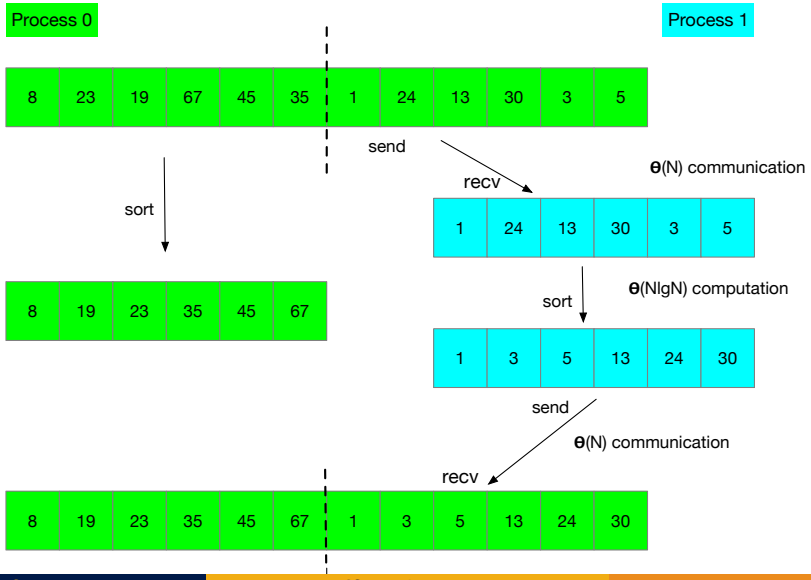
Communicators

- 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

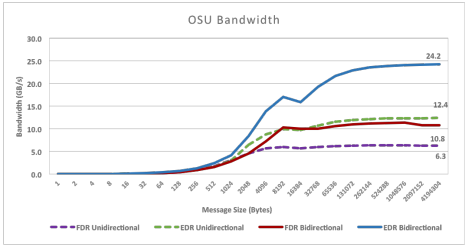


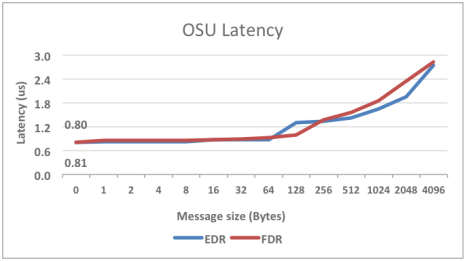
```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
int rank;
int a[1000];

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.

- We assumed communication time is linear in message size.
- Actual measurements on Blue Waters



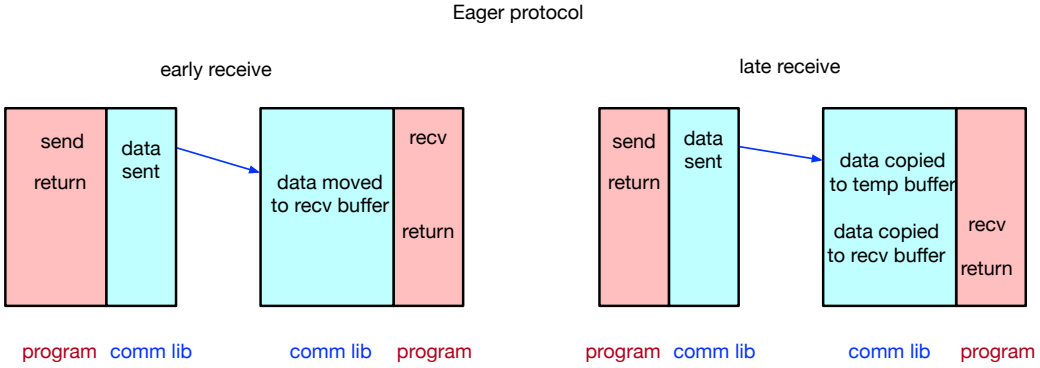


A better approximation to communication time is $\ell + n/b$ or $\max(\ell, n/b)$: ℓ is *latency* (fixed overhead of send and receive calls and transfer time in network) and b is *bandwidth*.

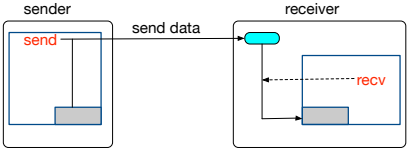
Communication Protocols

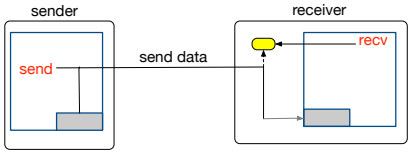
- Basic problem of send-receive communication (aka 2-sided communication): The processes have no common clock, so the send can occur before the receive or after the receive.
- If send data as soon as send occurs, then it may arrive before the receive is posted and needs to be buffered (eager protocol)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

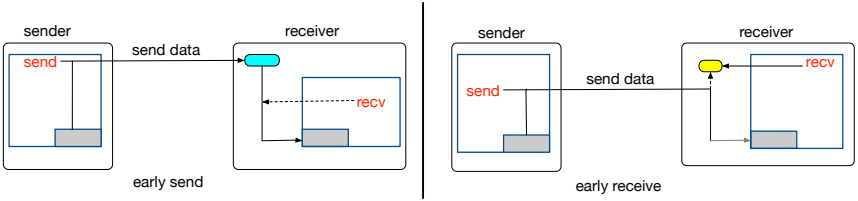


Eager protocol



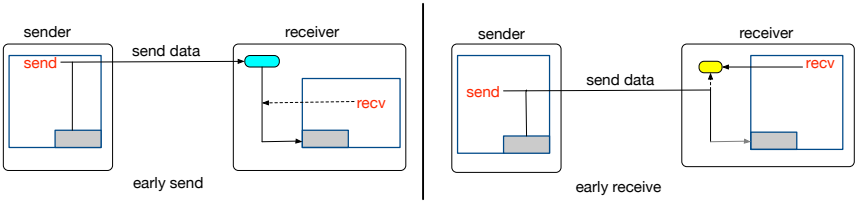


Eager protocol



Good: Simple protocol; send completes rapidly

Eager protocol

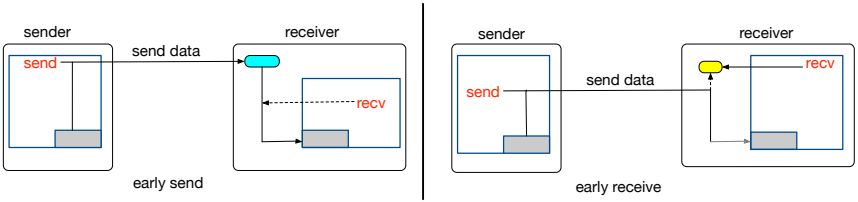


Good: Simple protocol; send completes rapidly

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



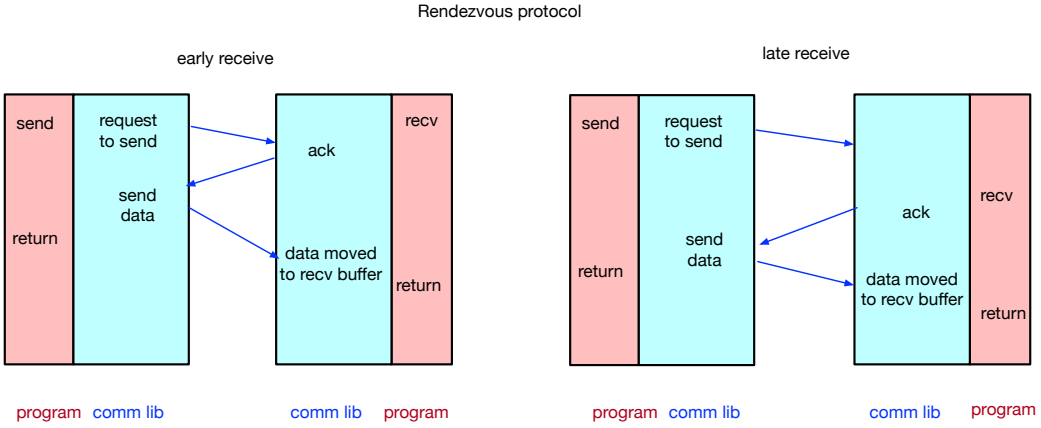
Good: Simple protocol; send completes rapidly

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

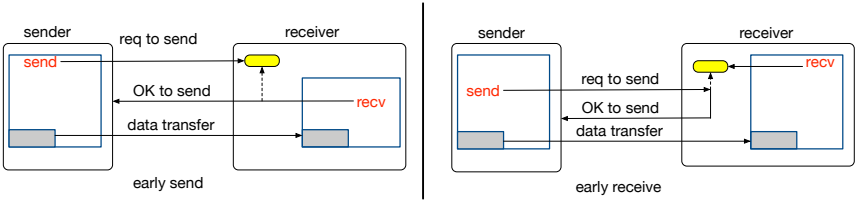
Send returns before or after receive starts

Best if receive is posted ahead of send

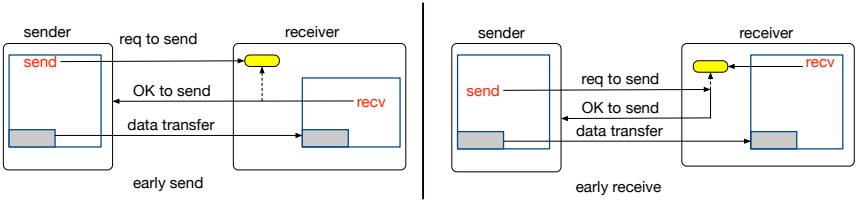
Rendezvous protocol



Rendezvous protocol

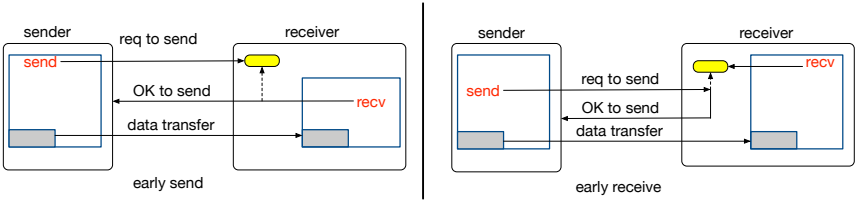


Rendezvous protocol



Good: Data moved once; need much less buffering

Rendezvous protocol

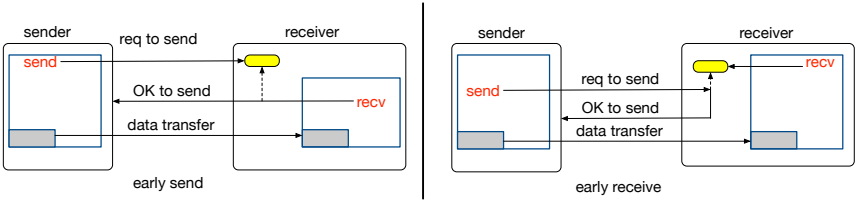


Good: Data moved once; need much less buffering

Bad: Extra protocol messages

Send returns only after receives start

Rendezvous protocol



Good: Data moved once; need much less buffering

Bad: Extra protocol messages

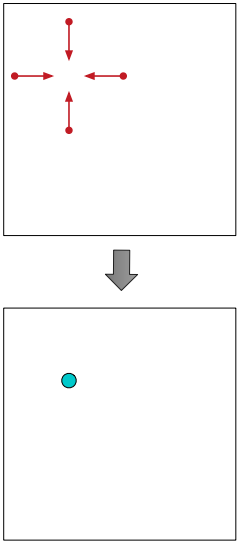
Send returns only after receives start

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

Jacobi



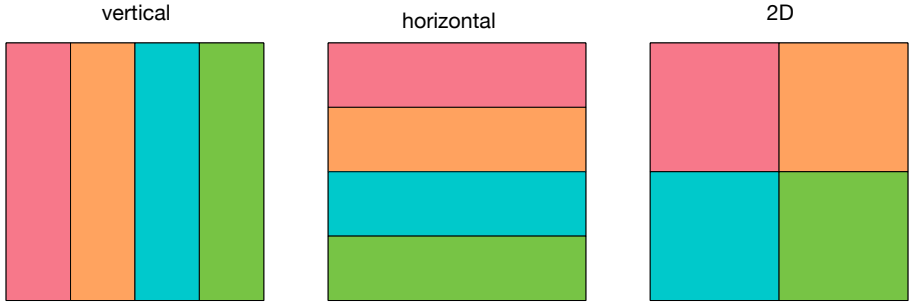
```
do {
  err=0; l=1-l;
  for (i=1;i<N-1; i++)
  for(j=1;j<N-1;j++) {
    a[1-l][i][j]=0.25*(a[l][i-1][j]
    +a[l][i+1][j]+a[l][i][j-1]
    +a[l][i][j+1]);
    err = fmax(err,fabs(a[1][i][j]
    -a[0][i][j]));
  }
} while(err>maxerr);
```

Parallel Jacobi

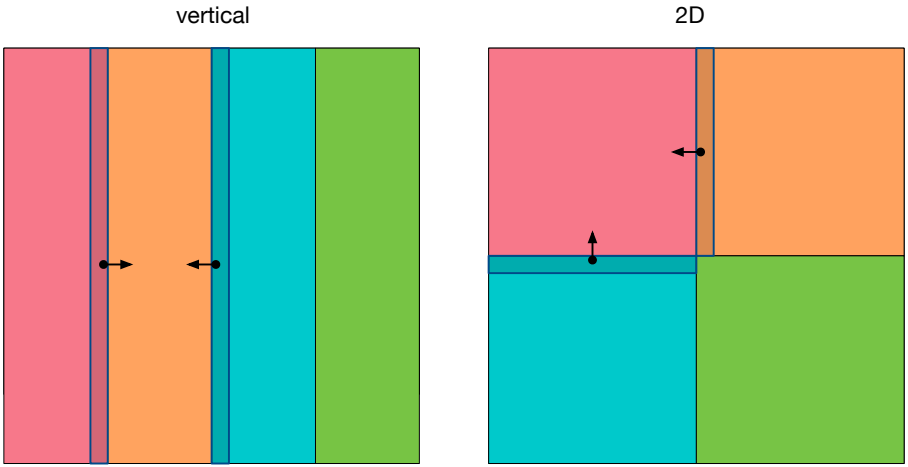
- Need to distribute the two arrays
- Need to distribute the computation
- *Data parallelism:* distribution of computation follows the distribution of the data
- Simplemost approach is to follow the *owner compute* rule: The process that "owns" an entry (has it in its local memory) is responsible for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

Possible distributions

Need to split the two matrices across the processes



Communication among processes



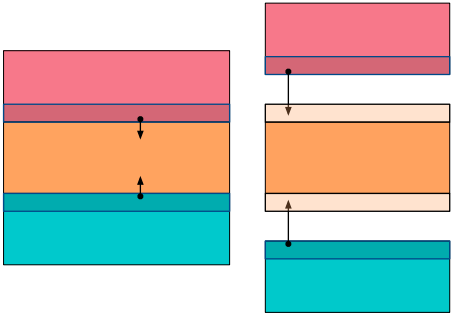
Communication will occur at the boundaries between partitions: Need to send boundary row/column to neighbor

Distribution

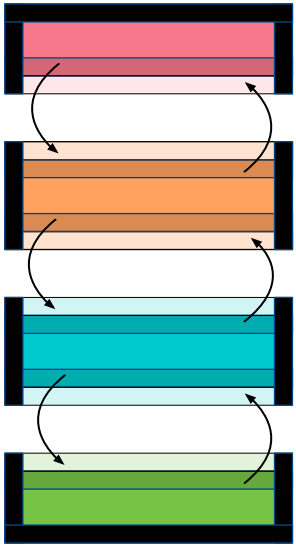
- How should we split the matrices across processes?
- It is good to have one partition per process
- It is good to have both matrices distributed the same way
- Communication is minimized for 2D partition
  - Assume $P = p^2$ processes/partitions
  - vertical/horizontal partition: Communication volume is $2(P-1) \cdot N = 2(p^2-1) \cdot N$
  - 2D partition: Communication volume is $4(p-1) \cdot N$
  - $2(p^2-1) > 4(p-1)$ if $p \geq 3$
- Horizontal tiles better than vertical tiles: The have longer rows and communicate items consecutive in memory. 2D reduces communication for large number of processes, but communication is more expensive
- Horizontal is probably better for small N/P and 2D for large N/P.

Ghost cells (aka hallo cells)

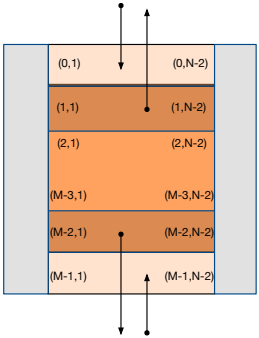
- Need place to receive copy of neighor rows.
- Will add two *ghost rows* above and beyond the rows owned by the process. These become the boundary for the local Jacobi iteration



Algorithm outline

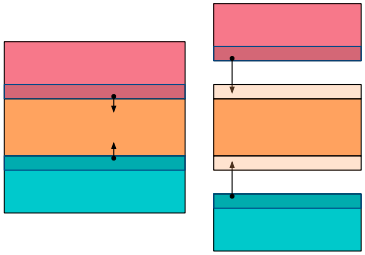


```
repeat {
update ghost rows;
compute new iteration;
}
until(converged)
```



Jacobi

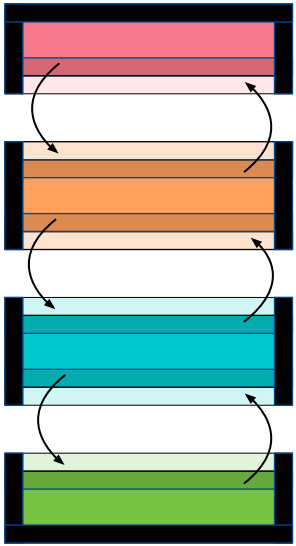
Data distribution



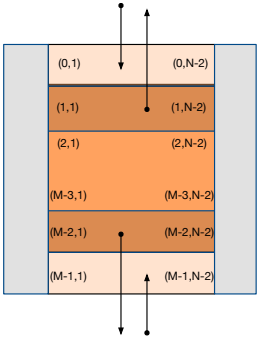
Sequential algorithm

```
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)
```

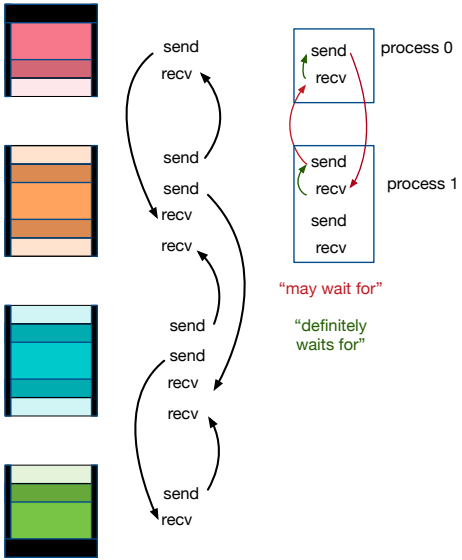


Code – first attempt (ignore convergence test)

```
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);
  }
```

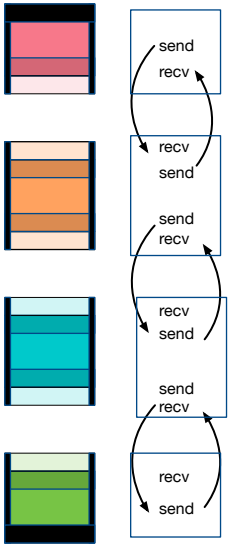
```
for (i=1;i<N-1; i++)
 for(j=1;j<N-1;j++)
  a[1-l][i][j]=0.25*(a[l][i-1][j]+a[l][i+1][j]+a[l][i][j-1]+a[l][i][j+1]);
l=1-l;
}
```

Deadlock is possible if send is not buffered



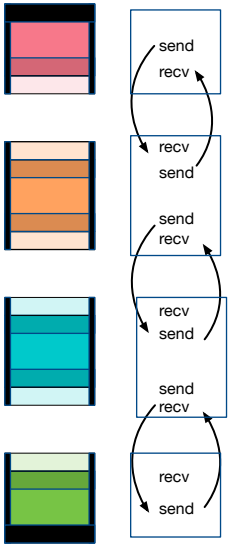
- *Deadlock*: situation where execution makes no progress.
- Typically due to a cycle of dependences: A waits for B to complete, B waits for C to complete, C waits for A to complete

Avoid deadlock, first solution



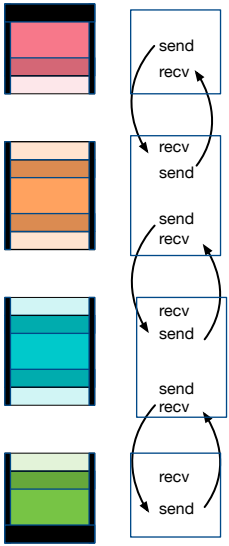
- Alternate the order of sends and receives

Avoid deadlock, first solution



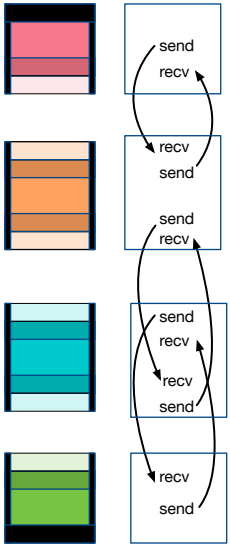
- Alternate the order of sends and receives
- Code is now deadlock-free

Avoid deadlock, first solution



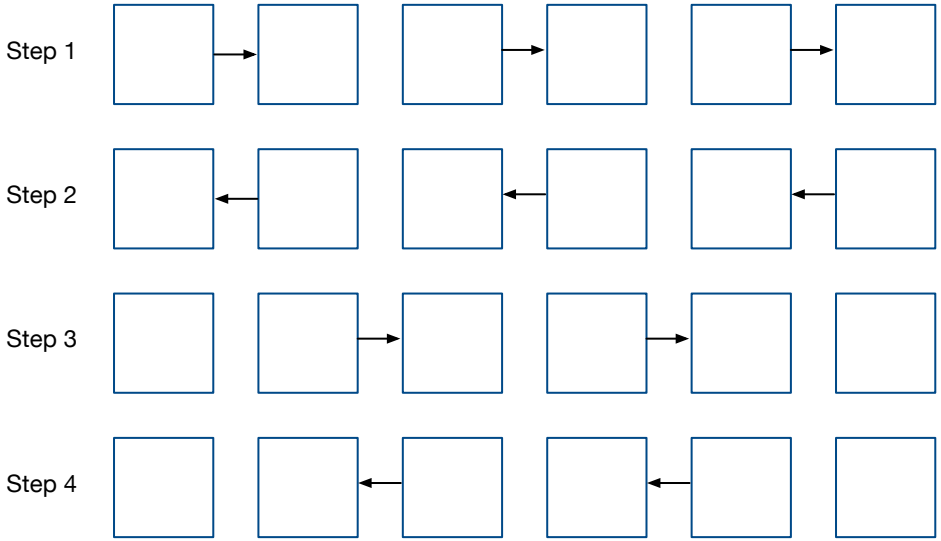
- Alternate the order of sends and receives
- Code is now deadlock-free
- But communications are serialized!

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

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`

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;
}
```