Communication Costs

- Time to send a single message from one process to another can be several orders of magnitude larger than the time to compute a single multiply.
- This communication is overhead, not present in the serial case.

Message passing takes time, just like computations do.

- Simple model of time to communicate N bytes,

$$T_{send/recv} = \alpha + \beta N$$

- *Latency*, α, is the start-up time. Fixed cost for each and every message.
- The parameter, β, is the inverse of the *bandwidth* of the communication network.
- Simple model of time to do N floating point operations,

$$T_{compute} = \gamma N$$

Message passing can take lots of time, more so than computations.

- Approximate values

  α = 3.15E-6 sec (Gahvari et al SC2012)

  β = 2.15E-9 sec/byte

  γ = 7.35E-11 sec/flop (HPC wire 2011)

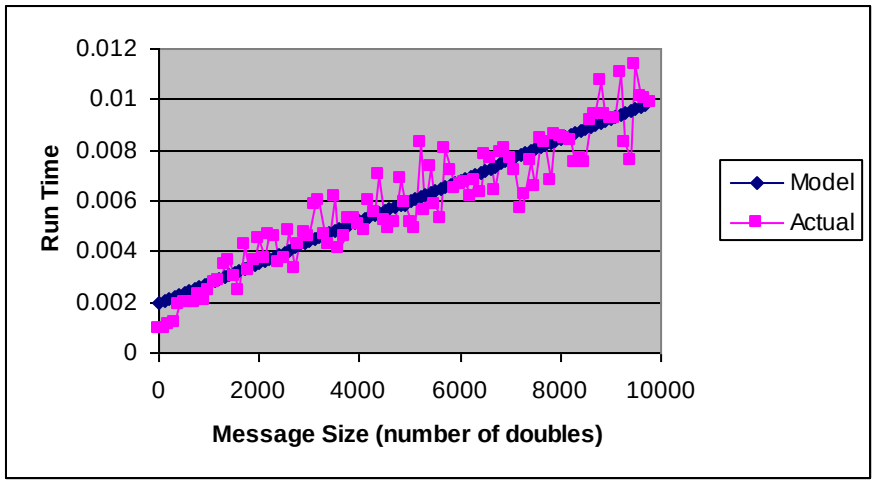
- Rescaling γ to 1

  α = 43,000 computations

  β = 29 computations

Get communication parameters by ping-pong test.

- Send messages of various sizes between 2 processors.
  - P0 sends vector X to P1, P1 receives and then sends the X vector back to P0. Elapsed time on P0 should be $2(a + bN)$.
- Find best fit of run time data to model



$$T_{comm} = a + bn$$

Get communication parameters by ping-pong test.

- Send messages of various sizes between 2 processors.
  - P0 sends vector X to P1, P1 receives and then sends the X vector back to P0. Elapsed time on P0 modeled as $2(a + bN)$.
- Use MPI timers as in assignment 3 to measure time for 100 ping pong passes.
- Pass float messages of size 1,501,1001,1501, …,9501,10001
- Find best fit values for

$$T_{comm} = a + bn$$

Your data from ping pong tests on blueshark.

- Within a node
  - α = 1E-6 sec
  - β = 1E-9 sec/float
- Across nodes
  - α = 1E-4 sec
  - β = 1E-7 sec/float
- Computations
  - γ = 2E-8 sec/rectangle

Conway's game of life on nxn grid

Survivals.

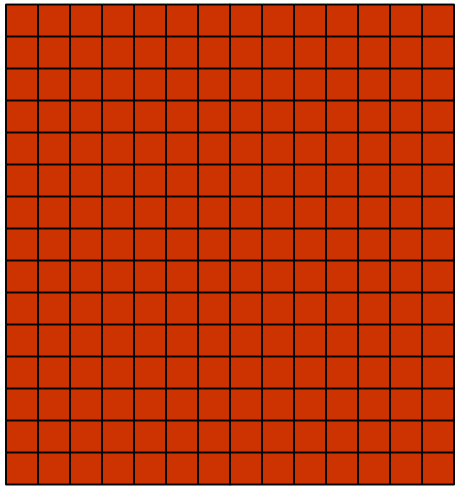
- Every counter with two or three neighboring counters survives for the next generation.

Deaths.

- Each counter with four or more neighbors dies (is removed) from overpopulation.
- Every counter with one neighbor or none dies from isolation.

Births.

- Each empty cell adjacent to exactly three neighbors--no more, no fewer--is a birth cell.



$$T_1 = \gamma n^2$$

http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

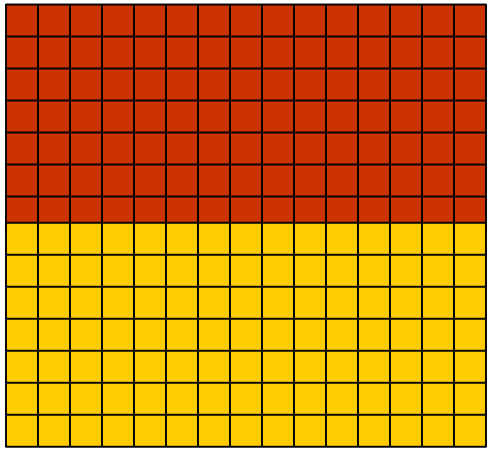
http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/

Speedup: Conway's game of life on nxn grid

Survivals.
- Every counter with two or three neighboring counters survives for the next generation.

Deaths.
- Each counter with four or more neighbors dies (is removed) from overpopulation.
- Every counter with one neighbor or none dies from isolation.

Births.
- Each empty cell adjacent to exactly three neighbors--no more, no fewer--is a birth cell.



Count time on one of the procs
Time for send + Time for recv

$$T_2 = \frac{1}{2}\gamma n^2 + 2(\alpha + \beta n)$$

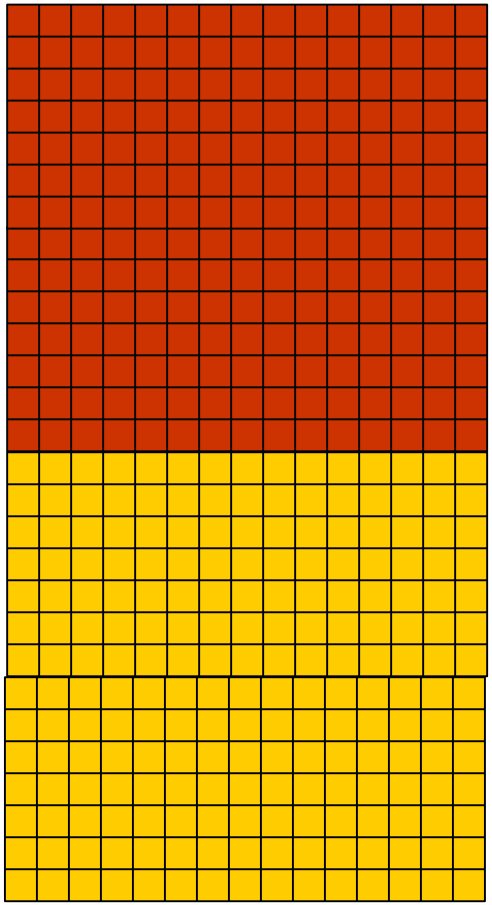
$$T_p = \frac{1}{p}\gamma n^2 + 4(\alpha + \beta n)$$

$$S = \frac{\gamma n^2}{\frac{1}{p}\gamma n^2 + 4(\alpha + \beta n)} < \frac{\gamma n^2}{4(\alpha + \beta n)}$$

Scaled Efficiency: Conway's game of life on nxpn grid

$$T_p = \gamma n^2 + 4(\alpha + \beta n)$$

$$E = \frac{\gamma n^2}{\gamma n^2 + 4(\alpha + \beta n)} \approx 1$$



For large n which is the "subdomain" size because:

- ***Computation Time $O(n^2)$***
- ***Communication Time $O(n)$***

Communication limits

- Clever programming can allow "hiding" of some of the communication costs by overlapping communication and computation.
- Some machines may have dedicated communication processors so that computation and communication can take place concurrently.
- The send/recv we used thus far are *blocking*. The code doesn't advance until they complete*.

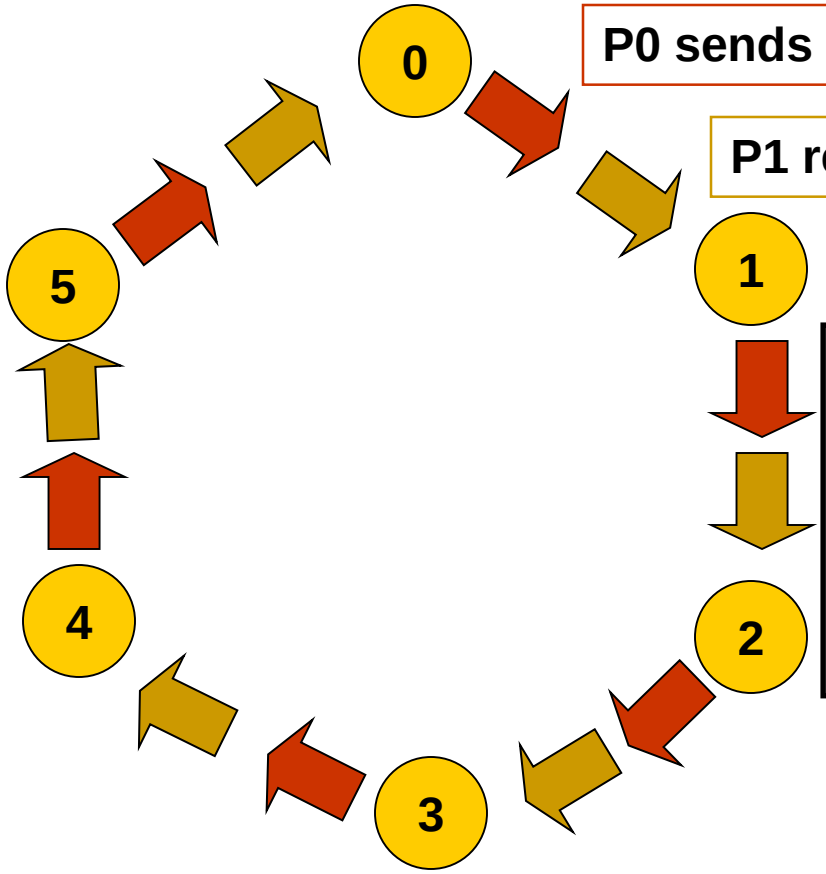
*MPI_Send and MPI_Recv are blocking

- MPI_Send does not complete until its safe for sender to modify (or delete) message contents.
  - When MPI_Send completes, you should not assume that the data has been received on the other processor. Its safe to modify your copy of the data, but the message may be buffered somewhere or in transit.
- MPI_Recv does not complete until its safe for receiver to use message contents.

Blocking Send: Ring pass

```
up_proc = myid + 1;
if (up_proc == num_procs) up_proc = 0;
down_proc = myid - 1;
if (down_proc == -1) down_proc = num_procs - 1;

MPI_Send( X, count, MPI_DOUBLE, up_proc,
                    tag_up, MPI_COMM_WORLD);
MPI_Recv( Y, count, MPI_DOUBLE, down_proc,
                    tag_up, MPI_COMM_WORLD, &status);
```

Blocking Send: Ring Pass

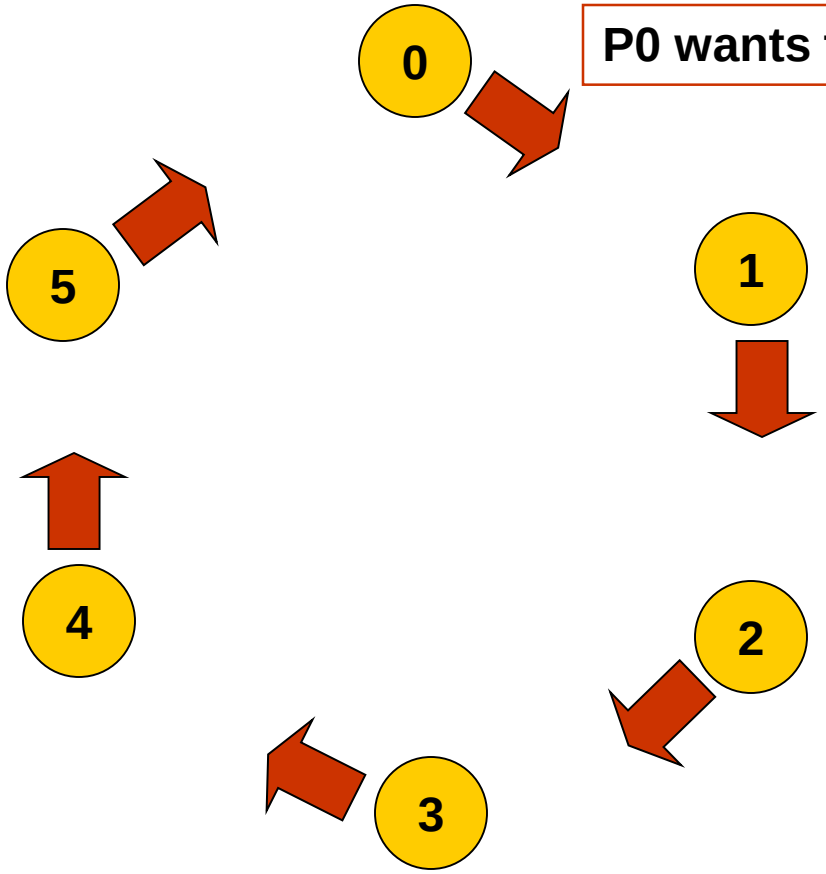


P0 sends contents of his X

P1 receives contents into his Y

Think of it as passing a note to your left neighbor
- Lean to left and get your neighbor's attention
- Pass the note

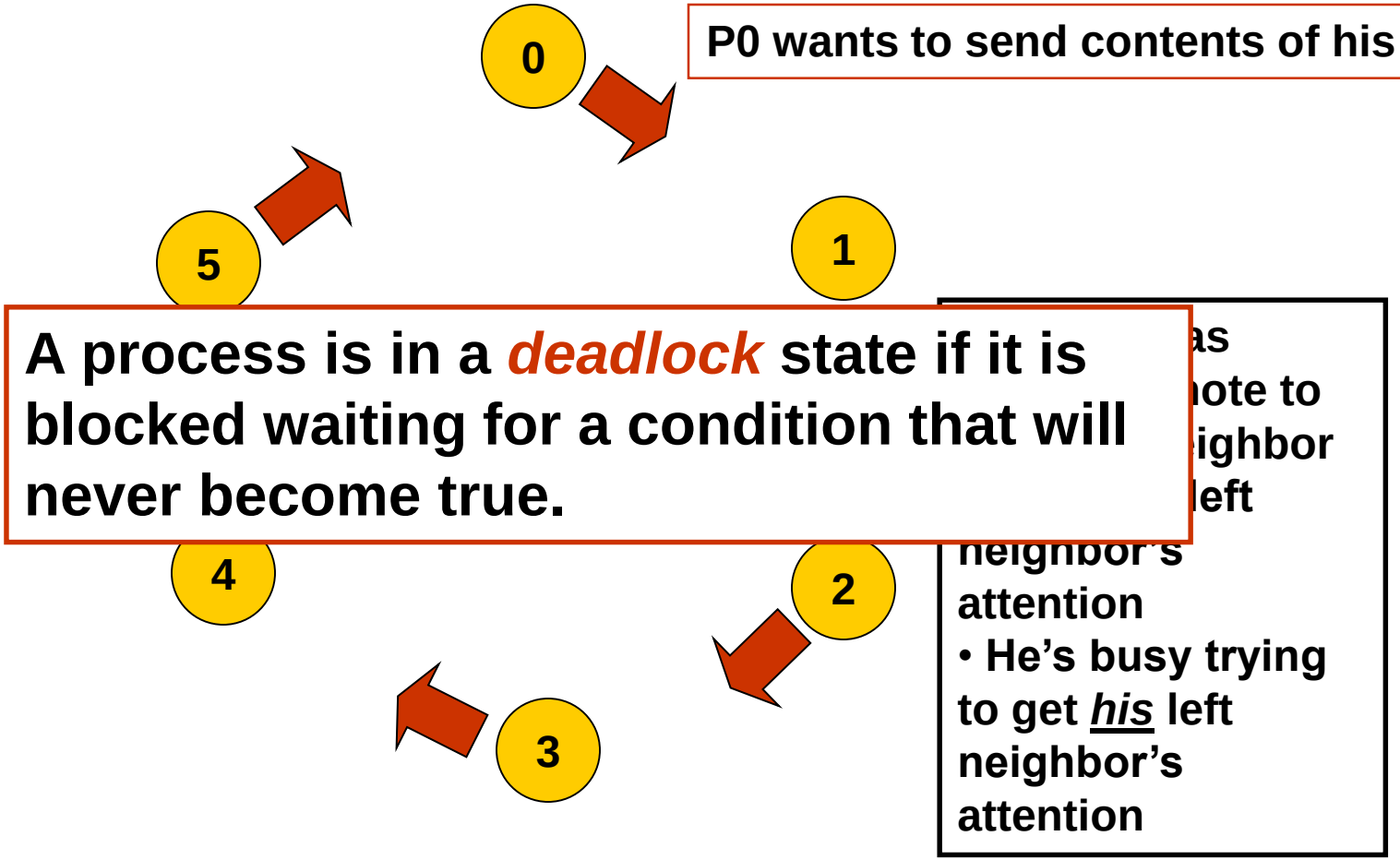
Deadlock in Ring Pass



P0 wants to send contents of his X

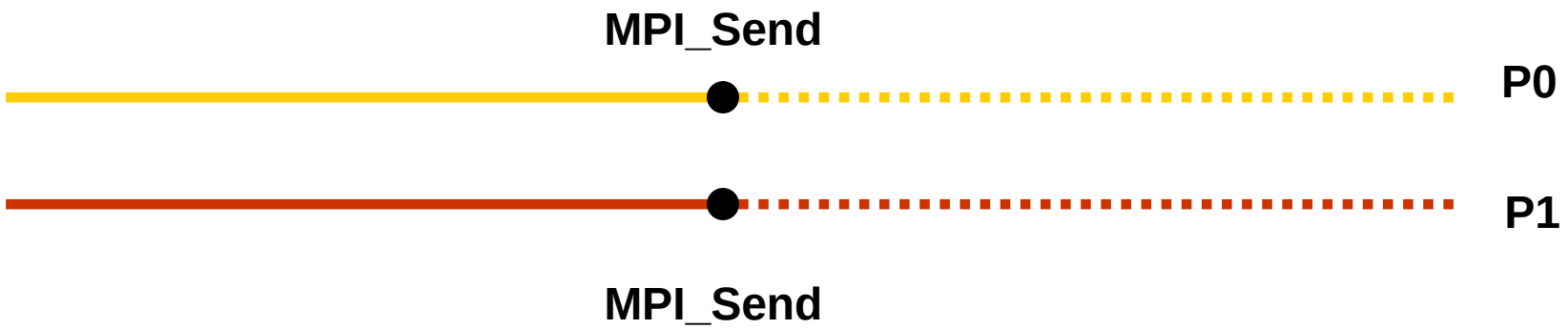
Think of it as passing a note to your left neighbor
- **Can't get left neighbor's attention**
- **He's busy trying to get *his* left neighbor's attention**

Deadlock in Ring Pass



Blocking Send: potential deadlock problem with num_procs = 2

```
up_proc = myid + 1;
if (up_proc == num_procs) up_proc = 0;
down_proc = myid - 1;
If (down_proc == -1) down_proc = num_procs - 1;

MPI_Send( X, count, MPI_DOUBLE, up_proc,
                    tag_up, MPI_COMM_WORLD);
MPI_Recv( Y, count, MPI_DOUBLE, down_proc,
                    tag_up, MPI_COMM_WORLD, &status);
```



Blocking Send: potential deadlock problem

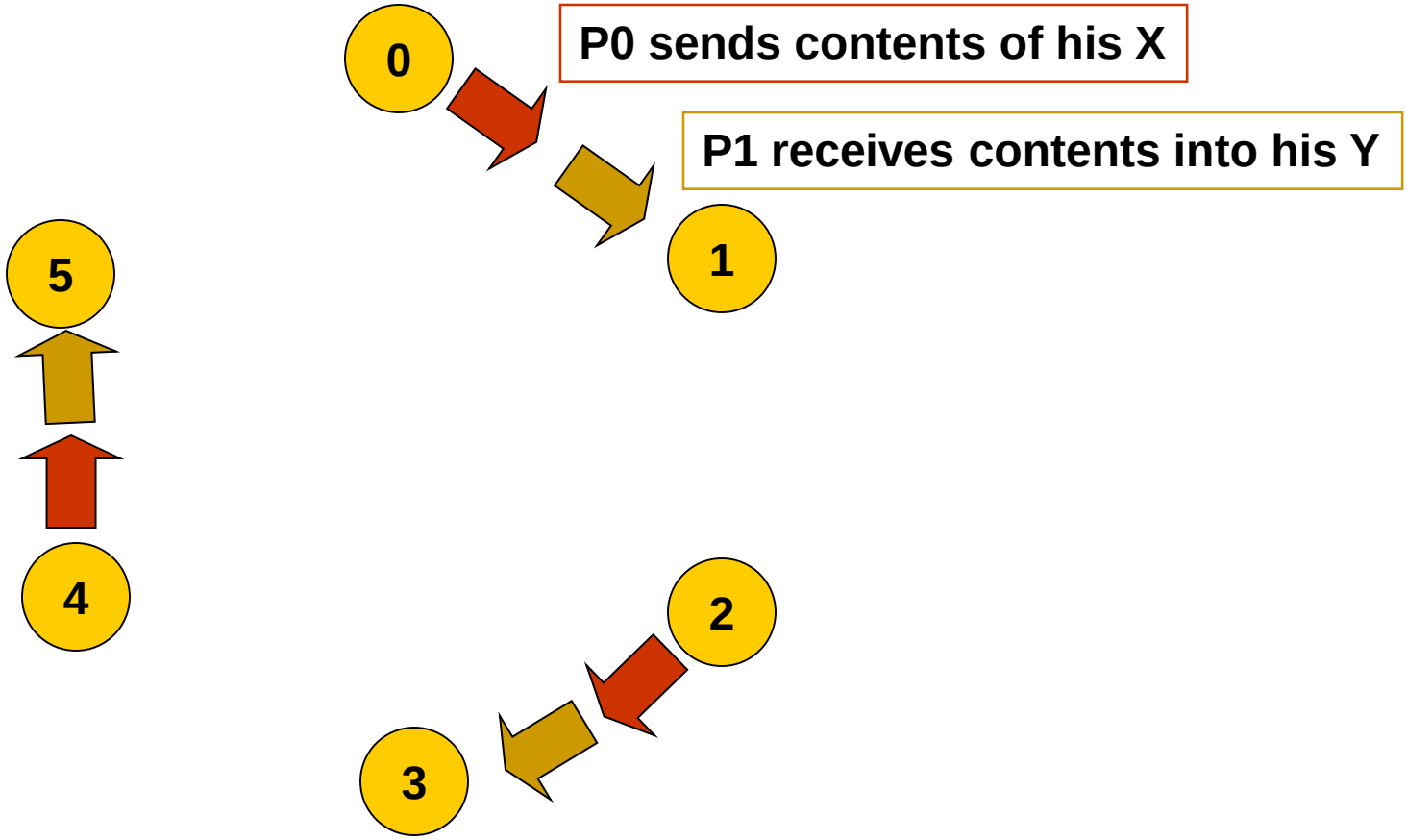
```
up_proc = myid + 1;
if (up_proc == num_procs) up_proc = 0;
down_proc = myid - 1;
if (down_proc == -1) down_proc = num_procs - 1;

MPI_Send( X, count, MPI_DOUBLE, up_proc,
                    tag, MPI_COMM_WORLD);
MPI_Recv( Y, count, MPI_DOUBLE, down_proc,
                    tag, MPI_COMM_WORLD, &status);
```

- Think how to remove potential deadlock by reordering send and recv.

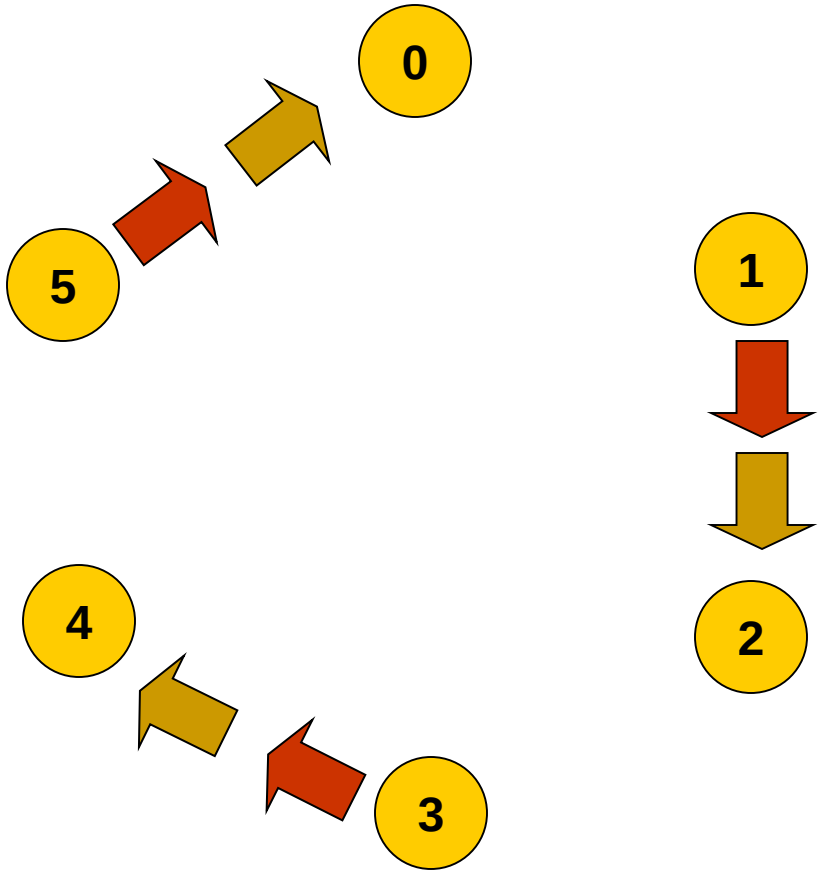
Ring Pass with reordered send/recv to avoid deadlock

```
if (myid % 2 == 0)
{
      MPI_Send( X, count, MPI_DOUBLE, up_proc,
                  tag, MPI_COMM_WORLD);
      MPI_Recv( Y, count, MPI_DOUBLE, down_proc,
                  tag, MPI_COMM_WORLD);
}
else
{
      MPI_Recv( Y, count, MPI_DOUBLE, down_proc,
                  tag, MPI_COMM_WORLD);
      MPI_Send( X, count, MPI_DOUBLE, up_proc,
                  tag, MPI_COMM_WORLD);
}
```

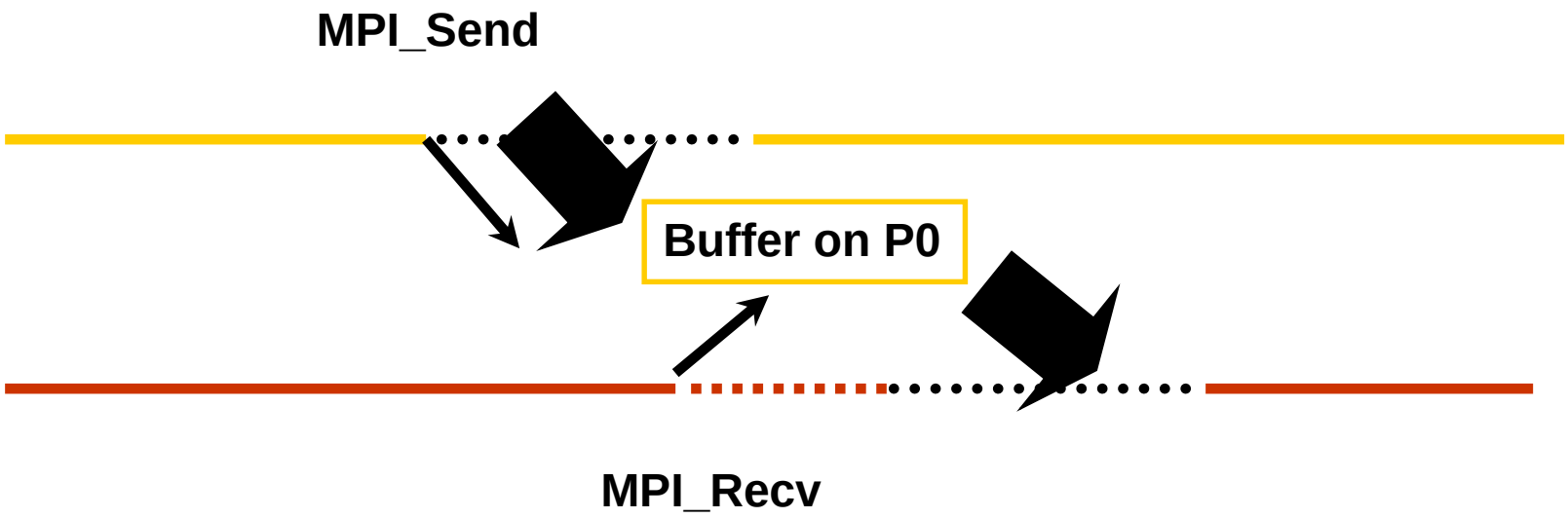
Ring Pass without deadlock, first step



Ring Pass without deadlock, second step

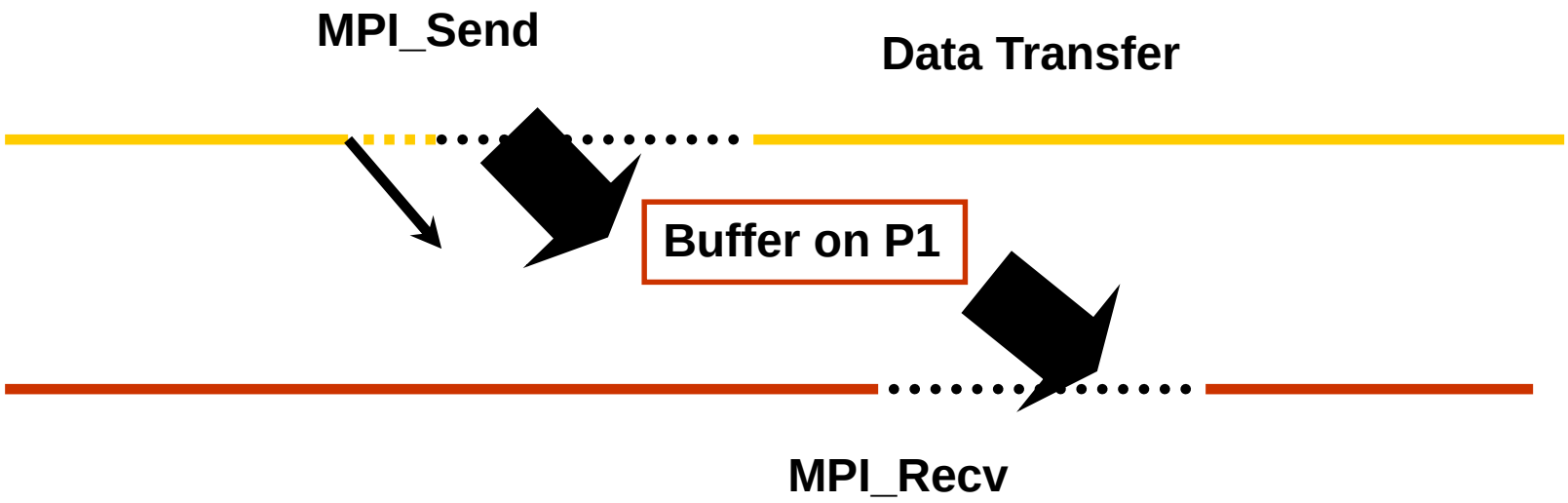


MPI implementations of MPI_Send and MPI_Recv typically provide buffering



- When P0 makes send call, message contents are immediately copied (buffered) into temporary storage (buffer) in P0's local memory
- When P1 makes receive call, data is sent from buffer to P1's local memory

MPI implementations of MPI_Send and MPI_Recv typically provide buffering

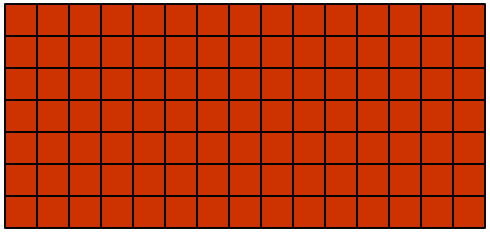


- Buffer may be in receivers memory instead (or in addition)
- When P0 makes send call, message contents are immediately sent and copied (buffered) into temporary storage (buffer) in P1's local memory
- When P1 makes receive call, data is copied from buffer to P1's local memory

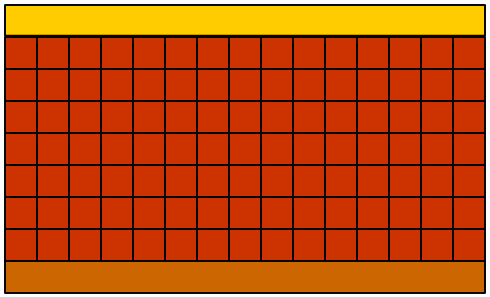
However implemented: MPI_Send and MPI_Recv are blocking

- MPI_Send does not complete until its safe for sender to modify (or delete) message contents.
- MPI_Recv does not complete until its safe for receiver to use message contents.

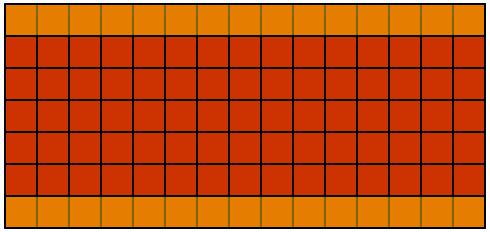
Game of life: Data I own



Game of life: Data I'll need from others



Game of life: Data others need from me



Game of Life codes

- Download the game of life codes from CANVAS.
- If you want to experiment, turn on the debug flag and see results for odd and even number of generations. You should see the “beacon” pattern.
- Run a speedup study with NumCells=1280 and NumGenerations=10 with procs=1,2,4,8,16,32,64,128

Nonblocking communication:Send

- Break the message passing event into two parts (think of send for now):
  - Initiating the send. This can happen as soon as I have the data required to be sent.
  - Finalizing the send. This needs to happen before I modify (or delete) the data to be sent.

Nonblocking communication

- Break the message passing event into two parts (think of send for now):
  - Initiating the send. This can happen as soon as I have the data required to be sent.
  - Finalizing the send. This needs to happen before I modify (or delete) the data to be sent.
- Avoids deadlock
- Allows computations in what might otherwise be dead times.

Nonblocking communication

- Break the message passing event into two parts (think of send for now):
  - Initiating the send. This can happen as soon as I have the data required to be sent.
  - Finalizing the send. This needs to happen before I modify (or delete) the data to be sent.
- Allows computations in what might otherwise be dead times.

```
Initialize sending X;
Do some computations changing stuff other than X;
Finalize sending of X;
Mess with X;
```

Nonblocking communication

- Break the message passing event into two parts (think of send for now):
  - Initiating the send. This can happen as soon as I have the data required to be sent.
  - Finalizing the send. This needs to happen before I modify (or delete) the data to be sent.
- Allows computations in w
  otherwise be dead times

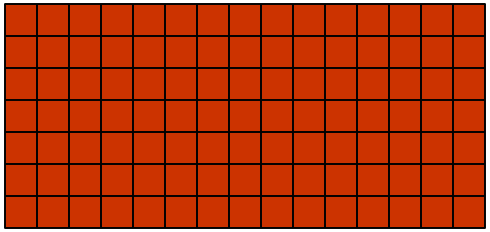
```
Initialize sending X;
Do some computations changing s
Finalize sending of X;
Mess with X;
```

Overlapping of communication and computation. This may allow the cost of communication to be "hidden".

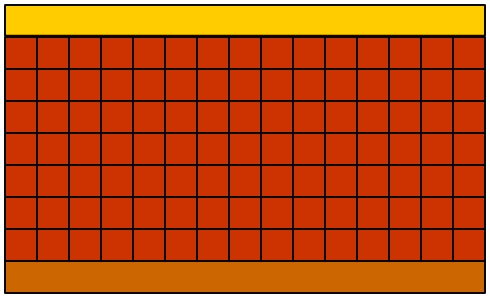
Nonblocking communication:Recv

- Break the message passing event into two parts (think of send for now):
  - Initiating the recv. This can happen as soon as I have the space ready for data to be recieved.
  - Finalizing the recv. This needs to happen before I read the data.

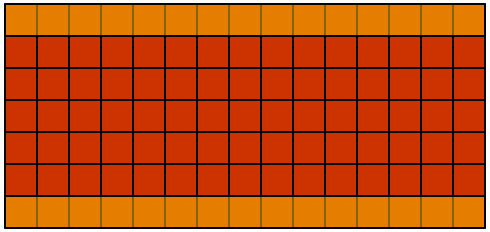
Game of life: Data I own



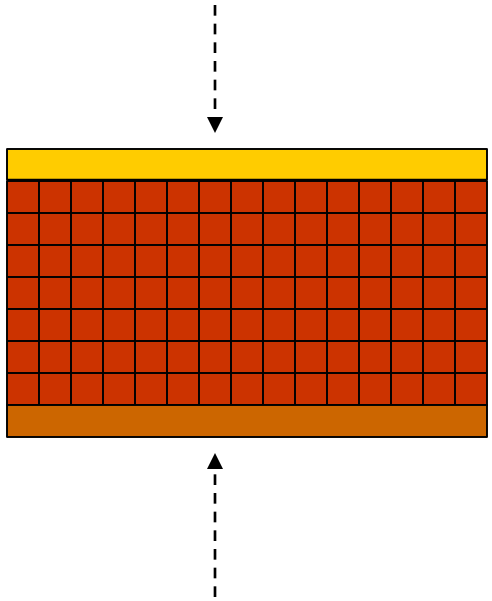
Game of life: Data I'll need from others



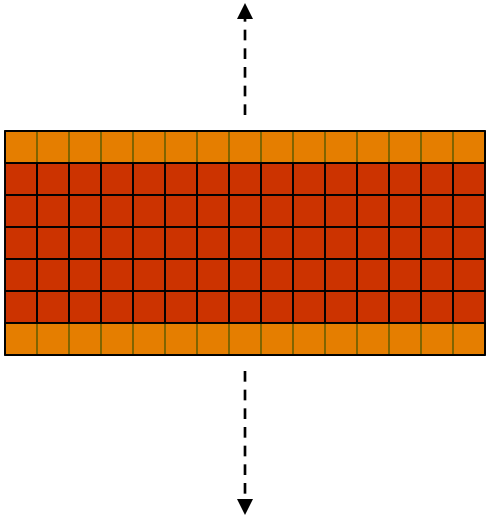
Game of life: Data others need from me



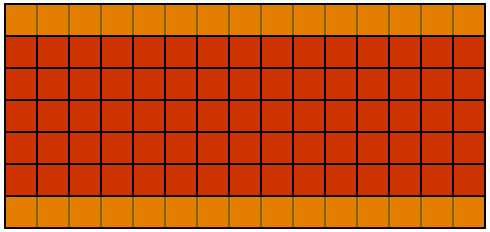
Game of life:
Step 1: Initiate Recv



Game of life:
Step 1: Initiate Send

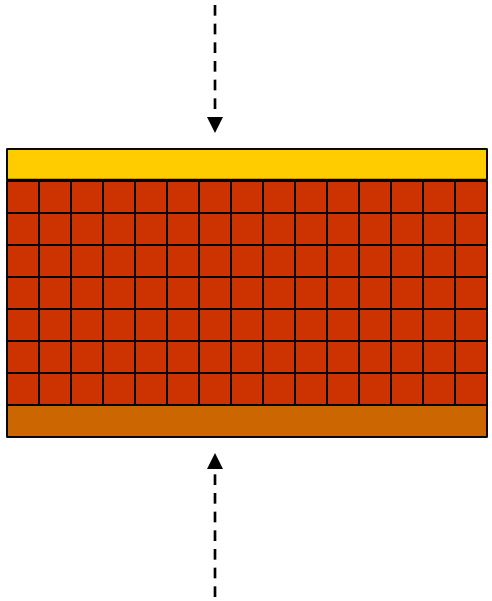


Game of life:
Step 2: compute update in interior

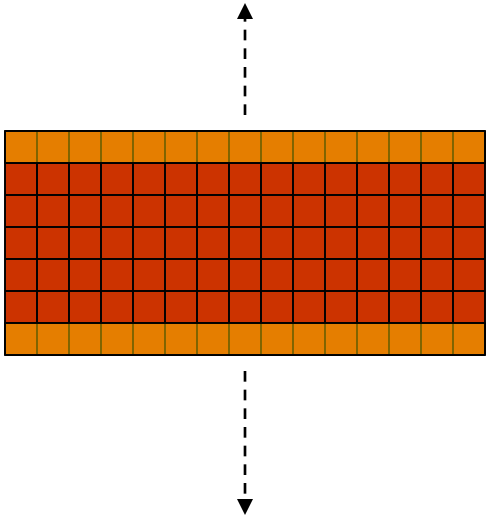


Solid red cells don't need neighbor processor data

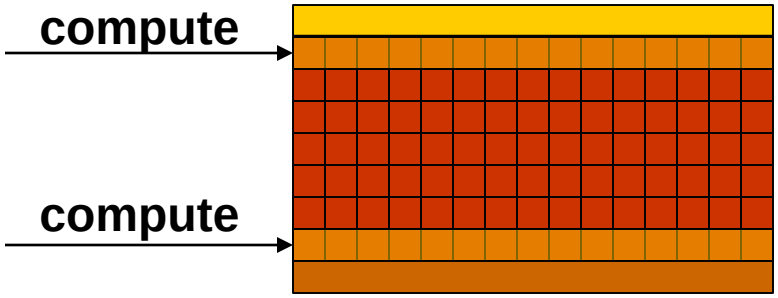
Game of life:
Step 3: Finalize Recv



Game of life:
Step 3: Finalize Send



Game of life:
Step 4: compute update on boundary



Implement nonblocking communication

- Call MPI_Isend & MPI_Irecv initialize communication
- Do computations unaffected by the communication
- Call MPI_Wait to finalize communication
- Do computations that depend on communication

nonblocking implementation

```
for (step=0; step<NumGenerations; step++)
{
        MPI_Isend and MPI_Irecv
        for (row=2; row<N_local; row++)
         {
                /* compute new generation*/
                u_new[row]=…
        }
        MPI_Wait or MPI_Waitall
        row=1;
        u_new[row]= …
        row=Nlocal;
        u_new[row]= …

}
```

nonblocking implementation (P1's view)

```
for (step=0; step<NumGenerations; step++)
{
      MPI_Isend and MPI_Irecv
      for (row=2; row<N_local; row
       {
             /* compute new generation*/
             u_new[row]=…
      }
      MPI_Wait or MPI_Waitall
      row=1;
      u_new[row]= …
      row=Nlocal;
      u_new[row]= …

}
```

Begin communications for top and bottom

nonblocking implementation (P1's view)

```
for (step=0; step<NumGenerations; step++)
{
        MPI_Isend and MPI_Irecv
        for (row=2; row<N_local; row++)
        {
                /* compute new generation*/
                u_new[row]=…
        }
        MPI_Wait or MPI_Waitall
        row=1;
        u_new[row]= …
        row=Nlocal;
        u_new[row]= …

}
```

Do computations in interior while communications finish

nonblocking implementation (P1's view)

```
for (step=0; step<NumGenerations; step++)
{
        MPI_Isend and MPI_Irecv
        for (row=2; row<N_local; row++)
         {
                /* compute new generation*/
                u_new[row]=…
        }
        MPI_Wait or MPI_Waitall
        row=1;
        u_new[row]= …
        row=Nlocal;
        u_new[row]= …

}
```

Finalize communications at endpoints

nonblocking implementation (P1's view)

```
for (step=0; step<NumGenerations; step++)
{
        MPI_Isend and MPI_Irecv
        for (row=2; row<N_local; row++)
         {
                /* compute new generation*/
                u_new[row]=…
        }
        MPI_Wait or MPI_Waitall
        row=1;
        u_new[row]= …
        row=Nlocal;
        u_new[row]= …

}
```

Do remaining computations at endpoints

MPI_Isend: a nonblocking send (I for immediate return)

```
MPI_Isend(
        void*          data,
        int            count,
        MPI_Datatype   type,
        int            destination,
        int            tag,
        MPI_Comm       comm,
        MPI_Request*   request)
```

- This call initializes or "posts" the send.
- request is a *handle* to an *opaque object*.

MPI_Wait: finalizes a nonblocking send identified by its handle

```
MPI_Wait(
        MPI_Request* request,
        MPI_Status*  status)
```

- This call completes when the send identified by the request handle is done.
- request is returned as MPI_REQUEST_NULL.
- status has no meaning for Isend.
- The data should not be ***written to*** between the Isend and Wait calls.

MPI_Irecv: a nonblocking receive (I for immediate return)

```
MPI_Irecv(
        void*          data,
        int            count,
        MPI_Datatype   type,
        int            source,
        int            tag,
        MPI_Comm       comm,
        MPI_Request*   request)
```

- This call initializes or "posts" the receive.
- request is a *handle* to an *opaque object*.

MPI_Wait: finalizes a nonblocking recv identified by its handle

```
MPI_Wait(
        MPI_Request* request,
        MPI_Status*  status)
```

- This call completes when the receive identified by the request handle is done.
- request is returned as MPI_REQUEST_NULL.
- status has same meaning as with MPI_Recv.
- The data should not be ***read from or written to*** in between IRecv and Wait calls.

Best Practices:

- Use nonblocking communication if there is useful work to do
  - post sends and receives as early as possible
    - Sender: as soon a data to send is available
    - Receiver: as soon as storage for data is available
  - Do waits as late as possible
    - Sender: just before data will be overwritten or deleted
    - Receiver: just before data will be read

In class experiment

- Copy the gameBlock.c code to gameNonBlock.c.
- Implement the non blocking communication as outlined on the previous slides.
- Do a speedup study with the nonblocking code and compare results to what you saw Tuesday with the blocking code.