

CSE 160

Lecture 20

Under the hood of MPI
Asynchronous Communication
Performance
Stencil methods with message passing

Today's lecture

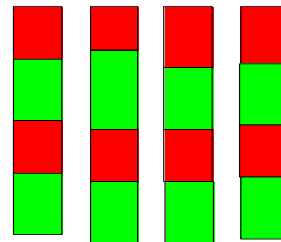
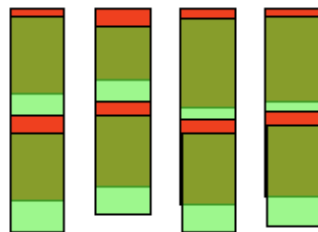
- Asynchronous non-blocking, point to point communication
- Under the hood of MPI
- Communication Performance
- Stencil methods in MPI (A4)

Asynchronous, non-blocking communication

- With Send or Receive, a return indicates the buffer may be reused, or that the data is ready
- There is also an *non-blocking asynchronous* form, that does not wait for completion
 - Required to express certain algorithms
 - Optimize performance: message flow problems
- ***Split-phased***
 - Phase 1: initiate communication with the immediate 'I' variant of the point- to-point call **IRecv(), ISend()**
 - Phase 2: synchronize **Wait()**
 - Perform unrelated computations between the two phases

Overlapping communication with computation

Overlap	No Overlap
IRecv(x, req) Send(..) Compute(y) Wait(req) Compute(x)	IRecv(x) Send(...) Wait(x) Compute(x) Compute(y)



Immediate mode send and receive

- Return does not indicate completion
- Must synchronize with a Wait() before reusing buffer (Send) or consuming data (Receive)
- An extra **request** argument is used to refer to a message we are synchronizing

MPI_Request request;

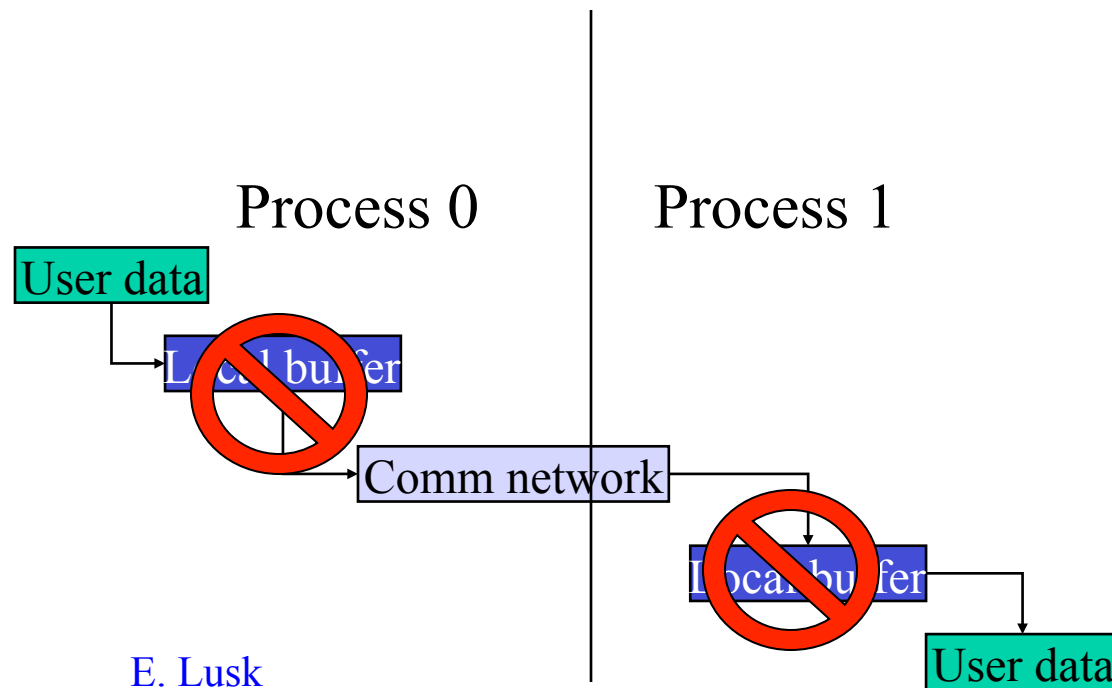
MPI_Irecv(buf, count, type, src, tag, comm, **&request**)

MPI_Wait(**&request**, **&status**)

- Irecv + Wait = Recv
MPI_Recv(buf, count, type, src, tag, comm, **&status**)
- **Immediate Send**
MPI_Isend(buf, count, type, dest, tag, comm, **&request**)

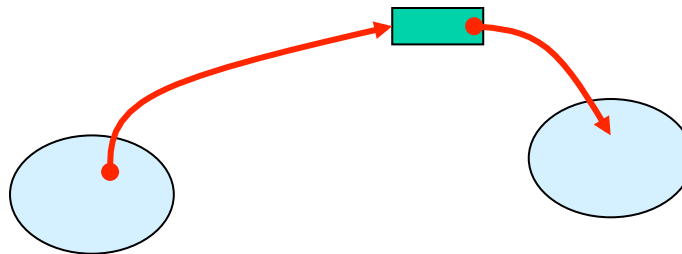
Buffering

- Where does the data go when you send it?
- It may be stored in a system buffer
- Preferable to avoid the extra copy when possible



Anonymous Buffering

- If there is not a pending receive, then an incoming message is placed in an anonymous system buffer
- When the receive gets posted, the message is moved into the user specified buffer
- Double copying reduces communication performance
- Non-blocking communication can help avoid this problem
- *MPI: The Complete Reference*, by Marc Snir et al.
www.netlib.org/utk/papers/mpi-book/mpi-book.html
“Buffering and Safety”
- *Send modes* are also useful
www-unix.mcs.anl.gov/mpi/sendmode.html



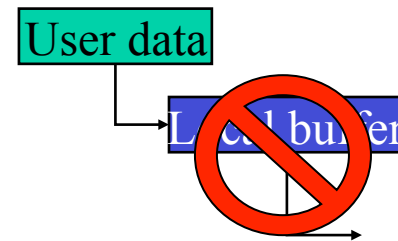
Restrictions on non-blocking communication

- The message buffer may not be accessed between an `IRecv()` (or `ISend()`) & its accompanying `Wait()`

`ISend(data,destination)`

`Wait()` on `ISend()`

Use the data



- Each pending `IRecv()` must have a distinct buffer

Message completion

- A `Send( )` *may or may not* complete...
- ... before a `Recv ( )` has been posted
- “May or may not” depends on the implementation
- Some programs may deadlock on certain message passing implementations

This program may deadlock

Process 0	Process 1
Send (x,1)	Send(y,0)
Recv (y,1)	Recv(x,0)

This program is “safe”

Process 0	Process 1
Send (x,1)	Recv(x,0)
Recv (y,1)	Send(y,0)

Avoiding an unsafe program

- The program on the left may deadlock if there isn't enough storage to receive the message
- In the program on the right, MPI has pre-allocated storage for the incoming message so there's no possibility of running out of storage

Process 0	Process 1
Send(x,1) Recv(y)	Send(x,0) Recv(y)

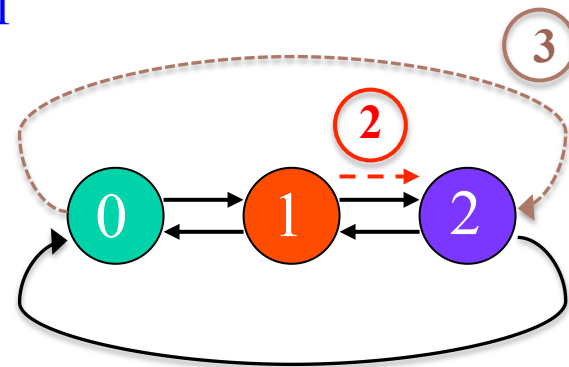
Process 0	Process 1
IRecv(x) Send(y,1)	IRecv(x) Send(y,0)

Rendezvous

- When a long message is to be sent, can MPI just send the message?
- For “short” message, it can. This is *eager mode* (How do we implement this?)
- The *eager limit* is the longest message that can be sent in eager mode
- See M. Banikazemi et al., IEEE TPDS, 2001, “MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems”
- For long messages, MPI first sends a scout to get permission to send the message
- This is called *rendezvous* mode

Correctness and fairness

- ① Iteration 1: $1 \rightarrow 2 \& 0$ $0 \rightarrow 1$ ($0 \rightarrow 2$) $2 \rightarrow 0 \& 1$
- ② 1 begins iteration 2: $1 \rightarrow 2$
- ③ $0 \rightarrow 2$ (but for iteration 1)
- ④ Problem: irecv in P2 receiving data from P1 in iteration 2 while it expects data from P0 in iteration 1



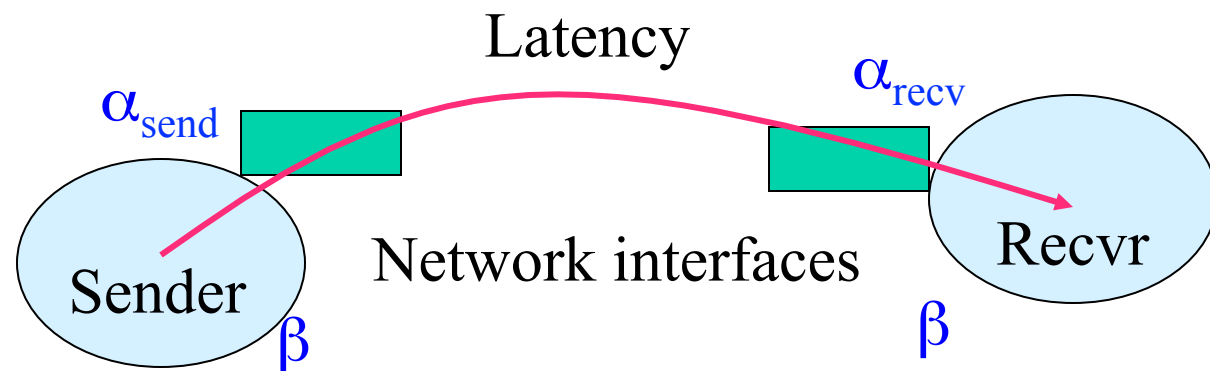
```
For i = 1 to n
  MPI_Request req1, req2;
  MPI_Status status;
  MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD, &req1);
  MPI_Irecv(buff2, len, CHAR, ANY_NODE, TYPE, WORLD, &req2);
  MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
  MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
  MPI_Wait(&req1, &status);
  MPI_Wait(&req2, &status);
End for
```

Today's lecture

- Asynchronous non-blocking, point to point communication
- Under the hood of MPI
- **Communication Performance**
- Stencil methods in MPI (A4)

Where does the time go?

- Communication performance can be a major factor in determining application performance
- Under ideal conditions...
 - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the user's message buffer
 - There is no other communication traffic
- Assume a contiguous message
- LogP model (Culler et al, 1993)



Communication performance

- The so-called α β model is often good enough
- Message passing time = $\alpha + \beta^{-1} n$

α = message startup time

β_{∞} = peak bandwidth (bytes per second)

n = message length

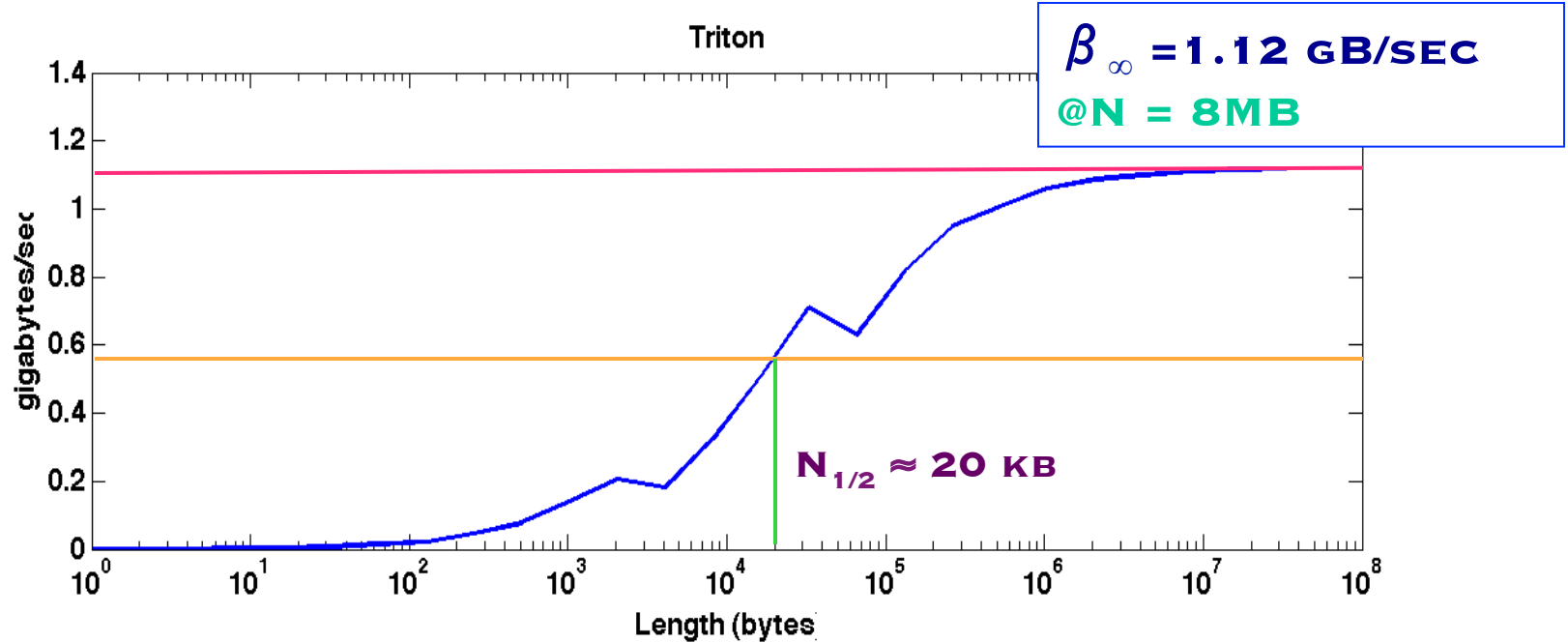
- “Short” messages: startup term dominates

$$\alpha \gg \beta^{-1} n$$

- “Long” messages: bandwidth term dominates

$$\beta^{-1} n \gg \alpha$$

Typical bandwidth curve (SDSC Triton)



$$\alpha = 3.2 \mu\text{sec}$$

Long Messages: $\beta^{-1} \infty n \gg \alpha$



Half power point

- $T(n)$ = time to send a message of length n
- Let $\beta(n)$ = the effective bandwidth

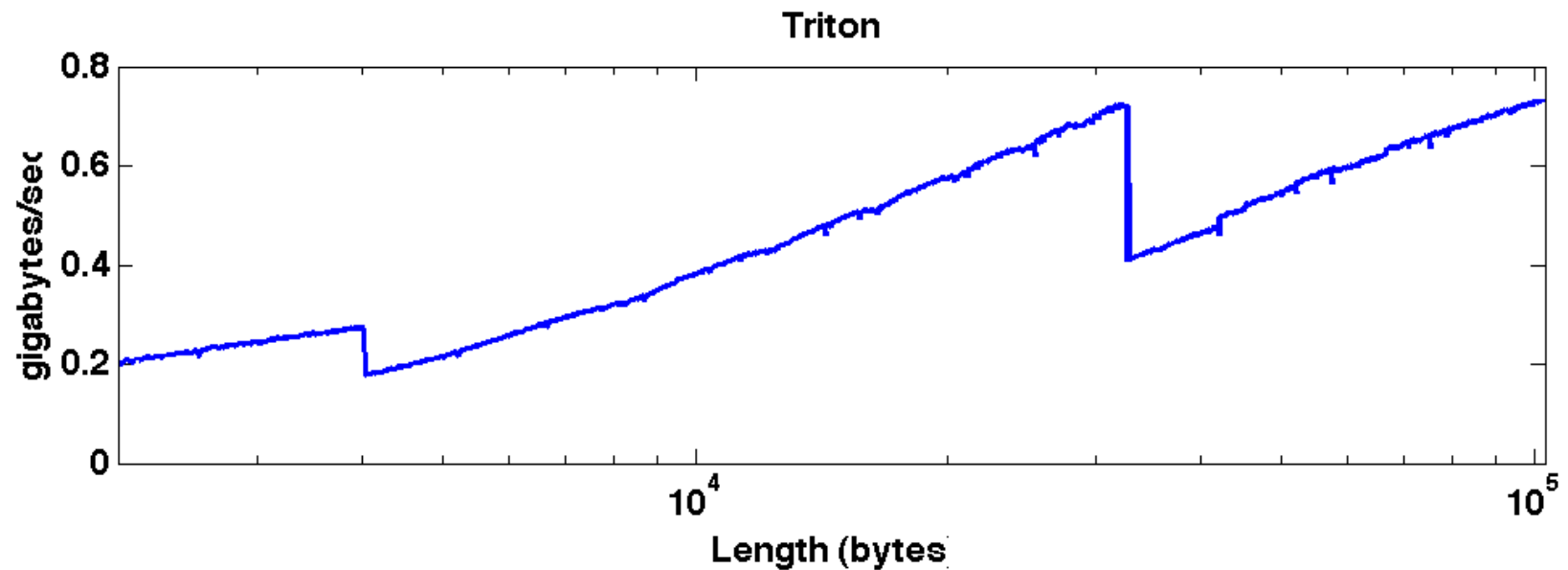
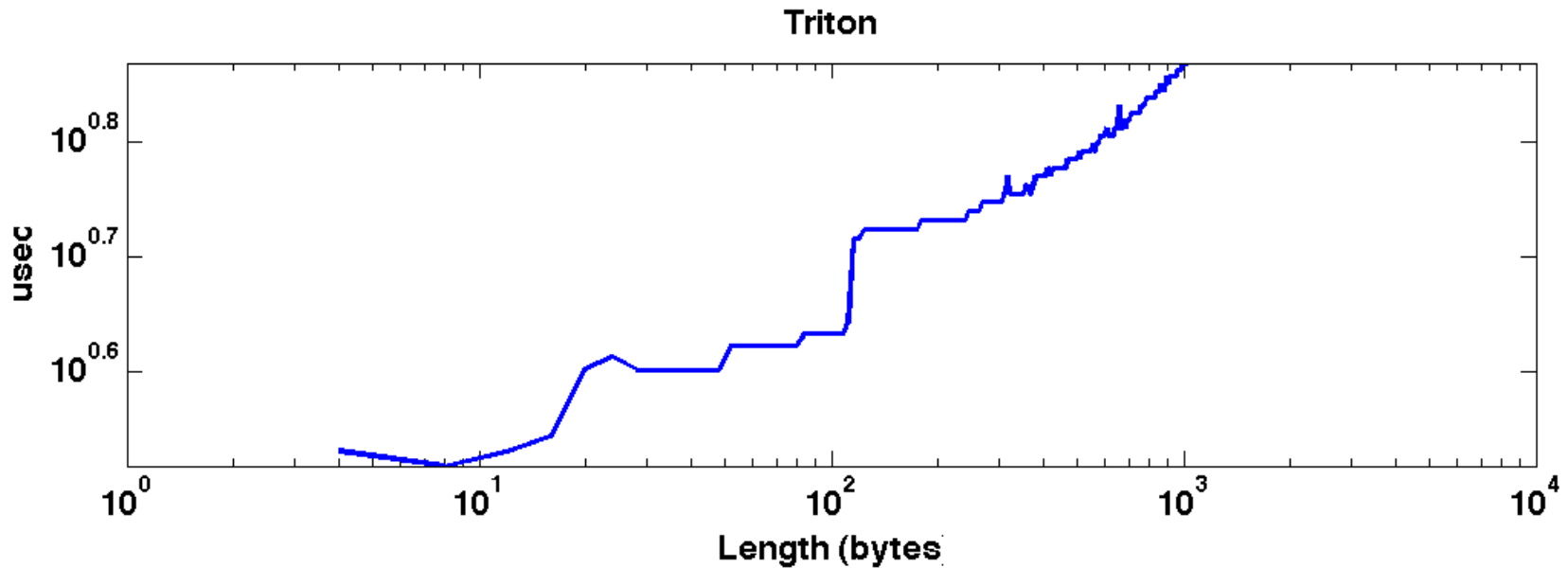
$$\beta^{-1}(n) = n / T(n)$$
- We define the *half power point* $n_{1/2}$ as the message size need to achieve $\frac{1}{2} \beta_{\infty}$

$$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$
- In theory, this occurs when $\alpha = \beta^{-1}_{\infty} n_{1/2}$

$$\Rightarrow n_{1/2} = \alpha \beta_{\infty}$$
- Generally not a good predictor of $n_{1/2}$
- For SDSC's Triton Cluster
 - $\alpha \approx 3.2 \mu s$, $\beta_{\infty} \approx 1.12 \text{ Gbytes/sec} \Rightarrow n_{1/2} \approx 3.6 \text{ KB}$
 - The actual value of $n_{1/2} \approx 20 \text{ KB}$

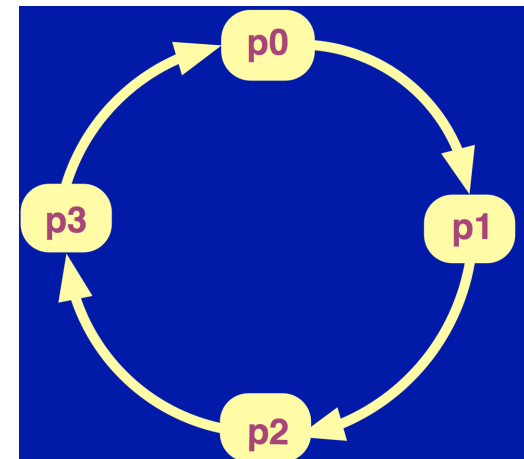
Message Length	BW (GB/sec)
1	0.29
2	0.61
4	1.16
8	2.46
16	4.99
32	8.31
64	15.98
128	24.86
256	47.83
512	85.39
1024	141.28
2048	211.05
4096	183.35
8192	330.86
16384	519.23
32768	718.64
65536	702.74
131072	897.12
262144	1038.6
524288	1124.1
1048576	1177.18
2097152	1200.74
4194304	1216.02
8388608	1223.27

Short and intermediate message lengths



Measuring communication performance with the Ring program

- Configure the processors logically in a ring and pass messages around the ring multiple times
- Assume there are p processors
- Neighbors of processor k are
 - $(k + 1) \bmod p$
 - $(k + p - 1) \bmod p$
- See `$PUB/Examples/MPI/Ring`



Measurement technique with Ring

```
for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
if (myid == 0) {
// (WARM UP CODE)
    const double start = MPI_Wtime( );
    for (int i = 0; i < trips; i++) {
        PROCESSOR 0 CODE
    }
    const double delta = MPI_Wtime( ) - start;
    Bandwidth = (long)((trips*len*nodes)/ delta /1000.0);
} else { // myid != 0
    // (WARM UP CODE)
    for (int i = 0; i < trips; i++) {
        ALL OTHER PROCESSORS
    }
}
```

The Ring program

Processor 0:

```
MPI_Request req;  
MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1)%p,  
          tag, MPI_COMM_WORLD, &req);  
MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p,  
          tag, MPI_COMM_WORLD);  
MPI_Status status;  
MPI_Wait(&req,&status);
```

All others:

```
MPI_Status status1;  
MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p,  
          tag, MPI_COMM_WORLD, &status1);  
MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p,  
          tag, MPI_COMM_WORLD);
```

Today's lecture

- Asynchronous non-blocking, point to point communication
- Under the hood of MPI
- Communication Performance
- **Stencil methods in MPI (A4)**

Stencil methods under message passing

- Recall the image smooth algorithm
- Repeat as many times as necessary

for (i,j) in 0:N-1 x 0:N-1

$I_{\text{new}}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$

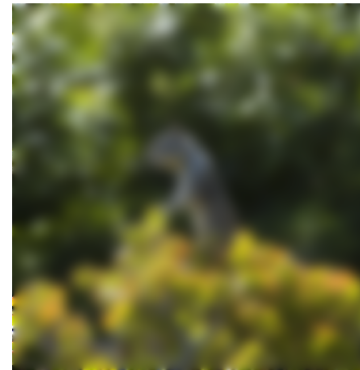
$I = I_{\text{new}}$



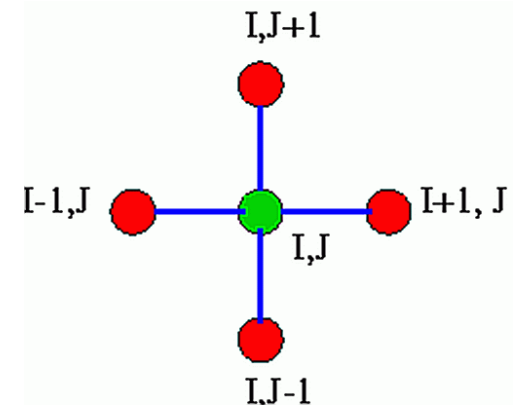
Original



100 iter



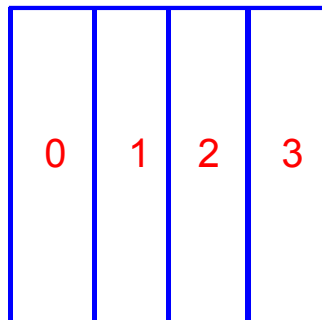
1000 iter



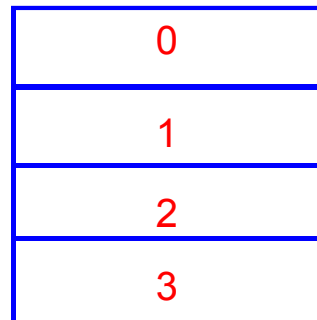
Parallel Implementation

- Partition data, assigning each partition to a unique process
- Different partitionings according to the *processor geometry*
- Dependences on values found on neighboring processes
- “Overlap” or “ghost” cells hold a copies off-process values

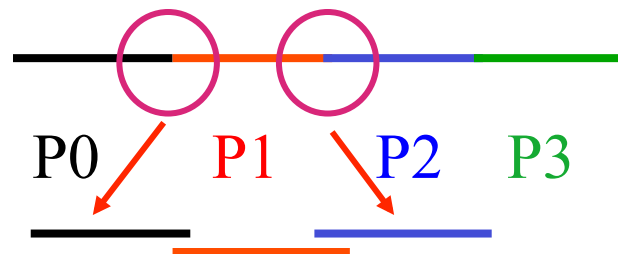
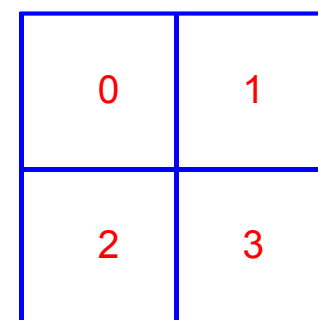
1×4



4×1

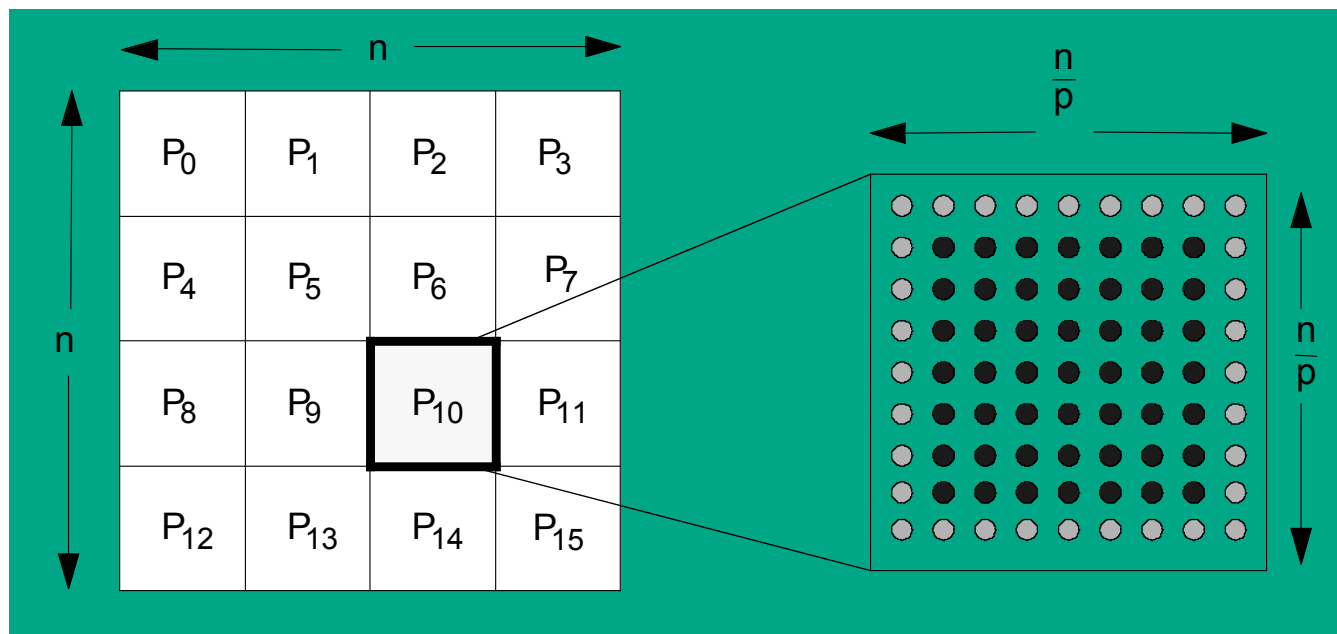


2×2



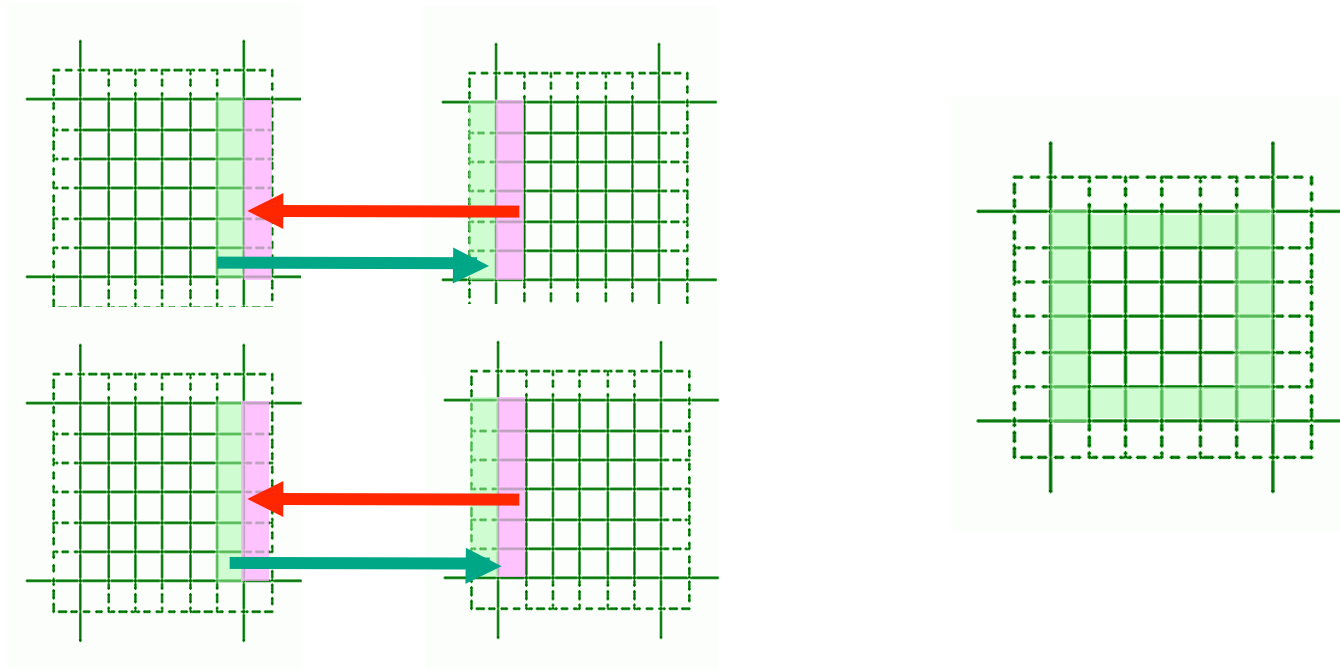
Ghost cells in higher dimensions

- Ghost cells surround each local subproblem
- Non-contiguous data
- Inefficient to communicate individual values



Managing ghost cells

- Post **IReceive** () for all neighbors
- **Send** data to neighbors
- **Wait** for completion



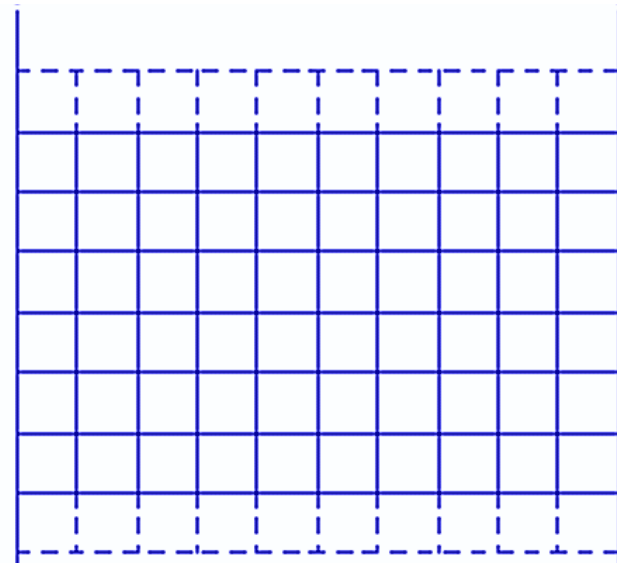
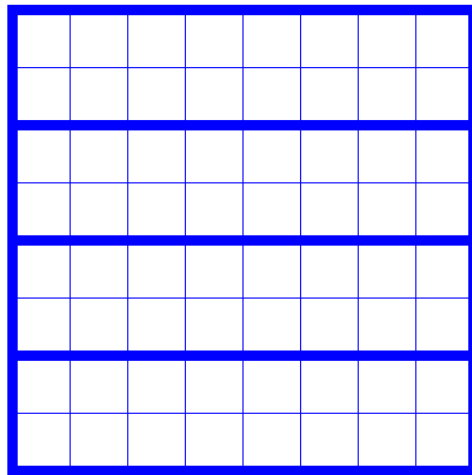
Performance is sensitive to processor geometry

- Aliev- Panfilov method running on triton.sdsc.edu (Nehalem Cluster)
- 256 cores, $n=2047$, $t=10$ (8932 iterations)

Geometry	GFlops	Gflops w/o Communication
32 x 8	573	660
8 x 32	572	662
16 x 16	554	665
2 x 128	508	658
4 x 64	503	668
128 x 2	448	658
256 x 1	401	638

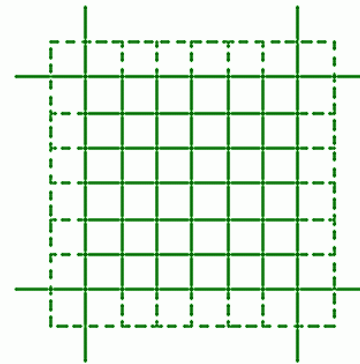
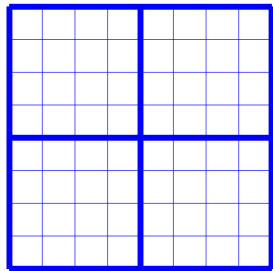
Communication costs for 1D geometries

- Assumptions
 - P divides N evenly
 - $N/P > 2$
- For horizontal strips, data are contiguous
 $T_{\text{comm}} = 2(\alpha + 8\beta N)$



2D Processor geometry

- Assumptions
 - $\sqrt{P}$ divides N evenly
 - $N/\sqrt{P} > 2$
- Ignore the cost of packing message buffers
- $T_{\text{comm}} = 4(\alpha + 8\beta N/\sqrt{P})$



Summing up communication costs

- Substituting $T_\gamma \approx 16 \beta$
- 1-D decomposition

$$(16N^2 \beta/P) + 2(\alpha + 8\beta N)$$

- 2-D decomposition

$$(16N^2 \beta/P) + 4(\alpha + 8\beta N/\sqrt{P})$$

Comparative performance

- Strip decomposition will outperform box decomposition ...
resulting in lower communication times ...
when $2(\alpha + 8\beta N) < 4(\alpha + 8\beta N/\sqrt{P})$
- Assuming $P \geq 2$: $N < (\sqrt{P}/(\sqrt{P} - 2))(\alpha/8\beta)$
- On SDSC's IBM SP3 system "Blue Horizon"
 $\alpha = 24 \text{ us}$
 $\beta = 1/(390 \text{ MB/sec})$
- $N < 1170 (\sqrt{P}/(\sqrt{P} - 2))$
- For $P = 16$, strips are preferable when $N < 2340$