

CSE 160

Lecture 13

Non-blocking Communication
Under the hood of MPI
Performance
Stencil Methods in MPI

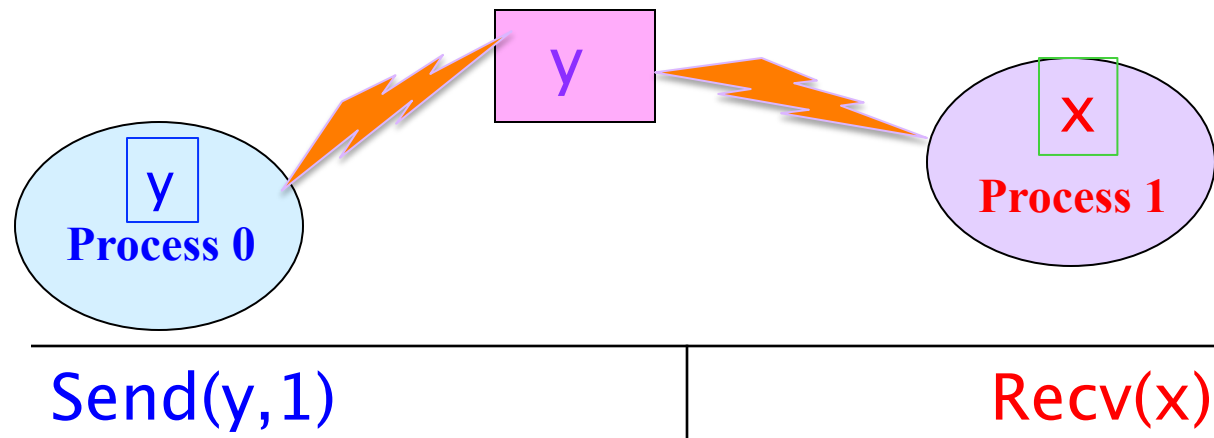
Announcements

Today's lecture

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

Recapping Send and Recv

- Primitives that implement Point to Point communication
- When **Send()** returns, the message is “in transit”
 - ♦ A return doesn't tell us if the message has been received
 - ♦ The data is somewhere in the system
 - ♦ Safe to overwrite the buffer
- **Receive()** blocks until the message has been received
 - ♦ Safe to use the data in the buffer



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: “Immediate return”
 - ♦ 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

Immediate mode send and receive

- Must synchronize with a `Wait()` before reusing buffer (`Send`) or consuming data (`Receive`)
- A `request` argument enables us to refer to a message we are waiting on

`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)`

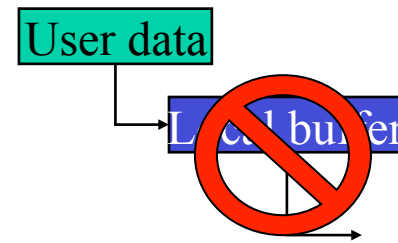
Restrictions on non-blocking communication

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

`ISend(data,destination)`

`Wait( )` on `ISend( )`

Use the data

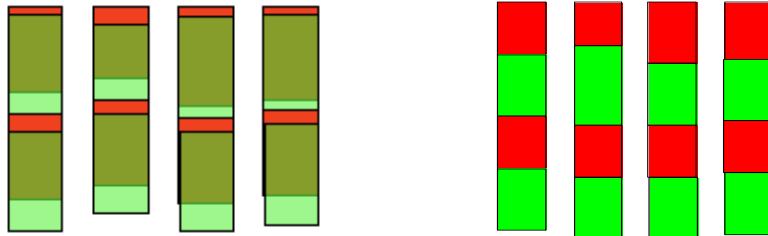


- Each pending `IRecv( )` must have a distinct buffer

Overlap

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

A message buffer may not be accessed between an IRecv() (or ISend()) and its accompanying wait()



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”

MPI has pre-allocated storage for the incoming message so there's no possibility of running out of storage

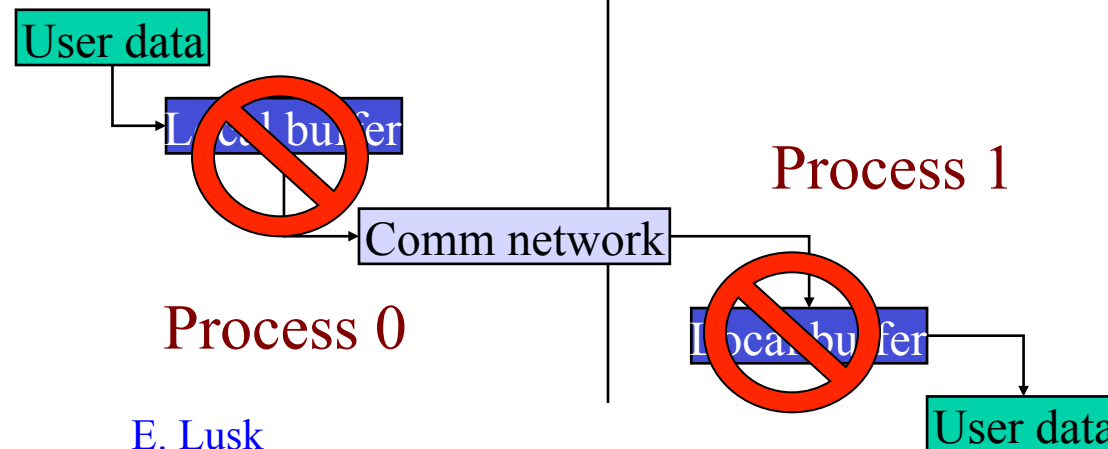
Process 0	Process 1
Send (x,1)	Recv(x,0)
Recv (y,1)	Send(y,0)

Today's lecture

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

Buffering

- Where does the message go when you send it?
- If there's not a pending receive for an incoming message, it's 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 it



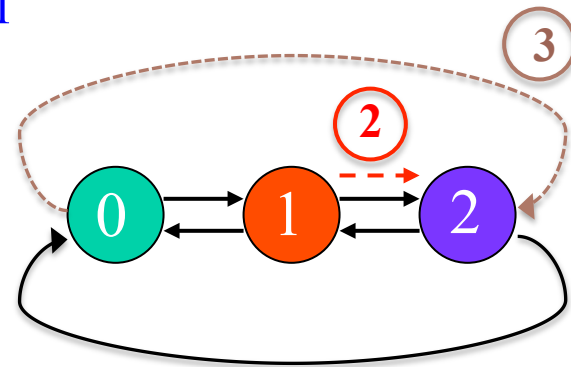
E. Lusk

Rendezvous

- When a long message is to be sent, can MPI just send the message?
- For “short” message, it can. This is *eager mode*
- *Eager limit*: 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



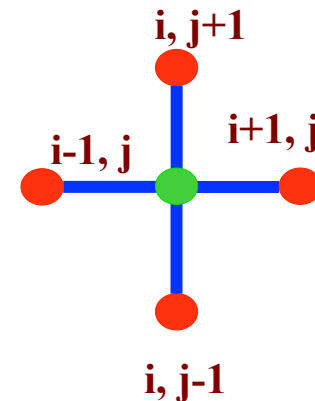
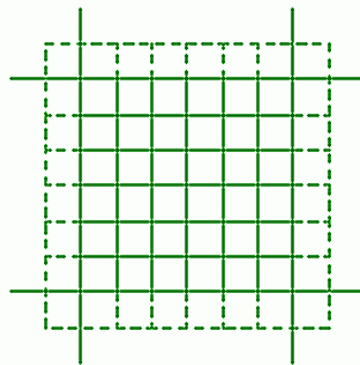
```
For i = 1 to n
MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD, &req1);
MPI_Irecv(buff2, len, CHAR, ANY_NODE, TYPE, WORLD, &req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);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

Today's lecture

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

Debugging tips

- Bugs?! Not in my code!
- The seg fault went away when I added a print statement
- Garbled output
- 2D partitioning is much more involved than 1D
- MPI is a library, not a language
- Gdb not as useful when you have many processes



Parallel print function

- Problem: how to sort out all the output on the screen
- Many messages say the same thing
 - Process 0 is alive!
 - Process 1 is alive!
 - ...
 - Process 15 is alive!
- Compare with
 - Processes[0–15] are alive!
- Parallel print facility
 - <http://www.llnl.gov/CASC/ppf>

Summary of capabilities

- Compact format list sets of nodes with common output
`PPF_Print( MPI_COMM_WORLD, "Hello world" );`
0-3: Hello world
- `%N` specifier generates process ID information
`PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );`
Message from 0-3
- Lists of nodes
`PPF_Print(MPI_COMM_WORLD,`
`(myrank % 2)`
`? "[%N] Hello from the odd numbered nodes!\n"`
`: "[%N] Hello from the even numbered nodes!\n")`
[0,2] Hello from the even numbered nodes!
[1,3] Hello from the odd numbered nodes!

Practical matters

- Installed in `$(PUB)/lib/PPF`
- Specify `ppf=1` and `mpi=1` on the “make” line or in the Makefile
 - ◆ Defined in `arch.gnu-4.7_c++11.generic`
 - ◆ Each module that uses the facility must

`#include “ptools_ppf.h”`

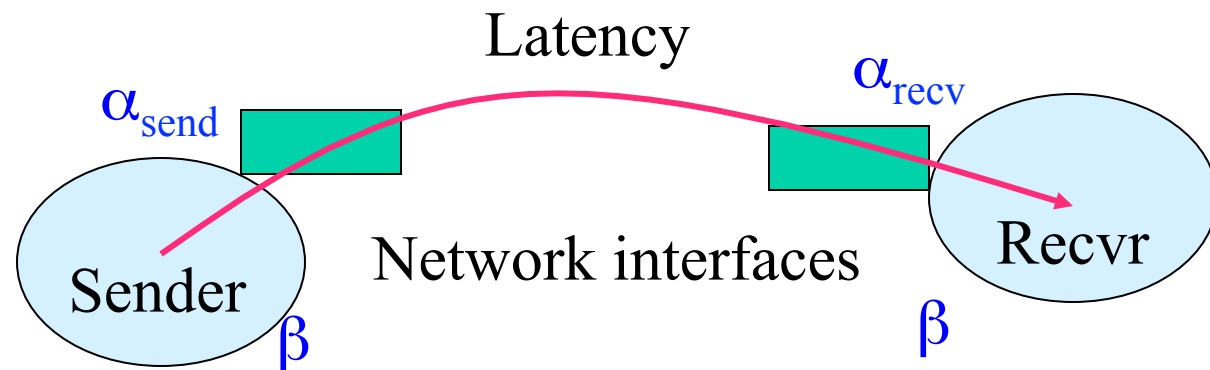
- Look in `$(PUB)/Examples/MPI/PPF` for example programs `ppfexample_cpp.C` and `test_print.c`
- Uses `MPI_Gather()`

Today's lecture

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

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

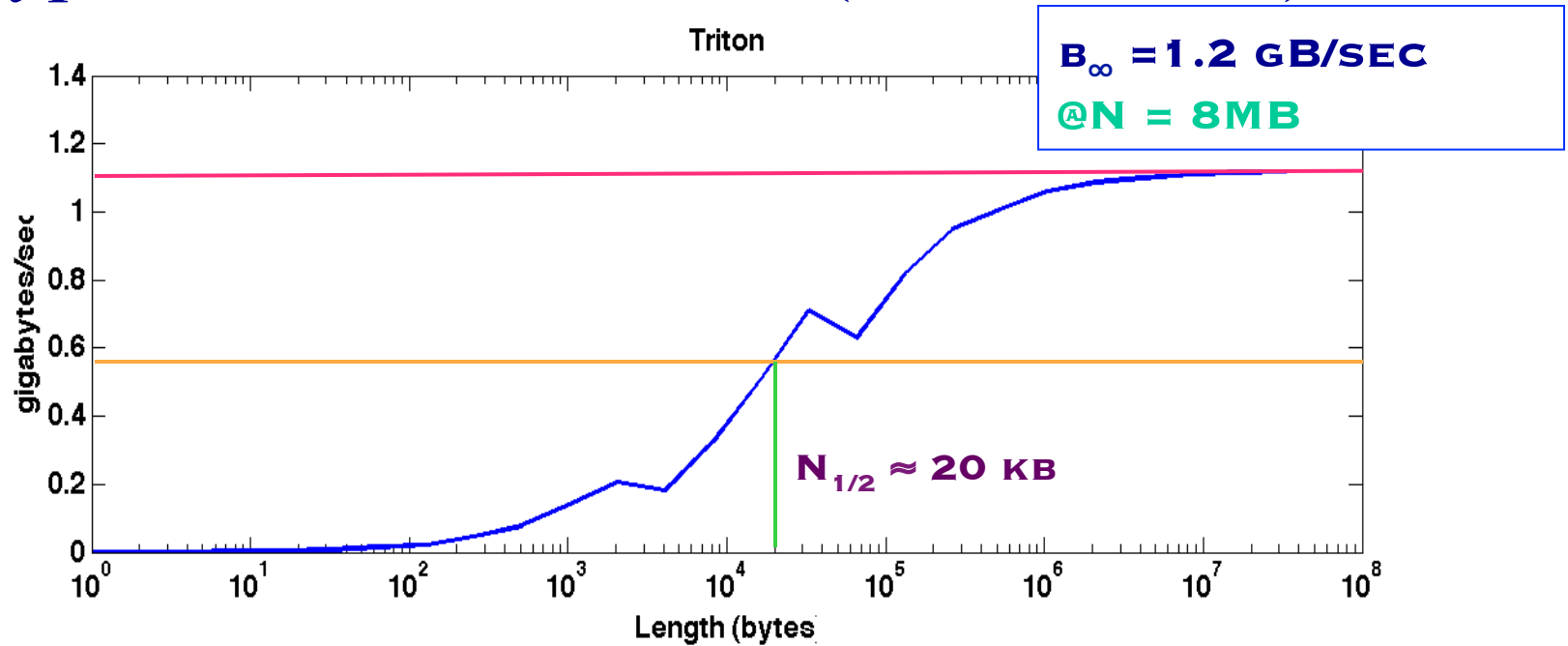
- “Short” messages: startup term dominates

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

- “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

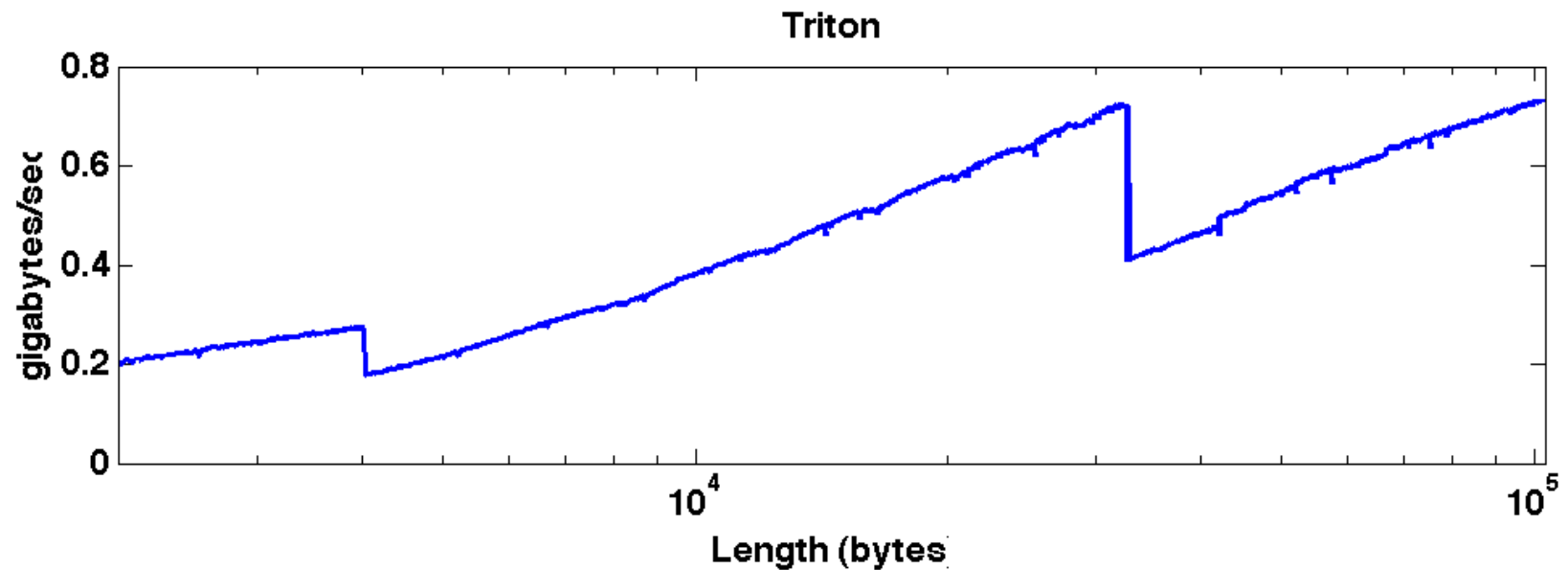
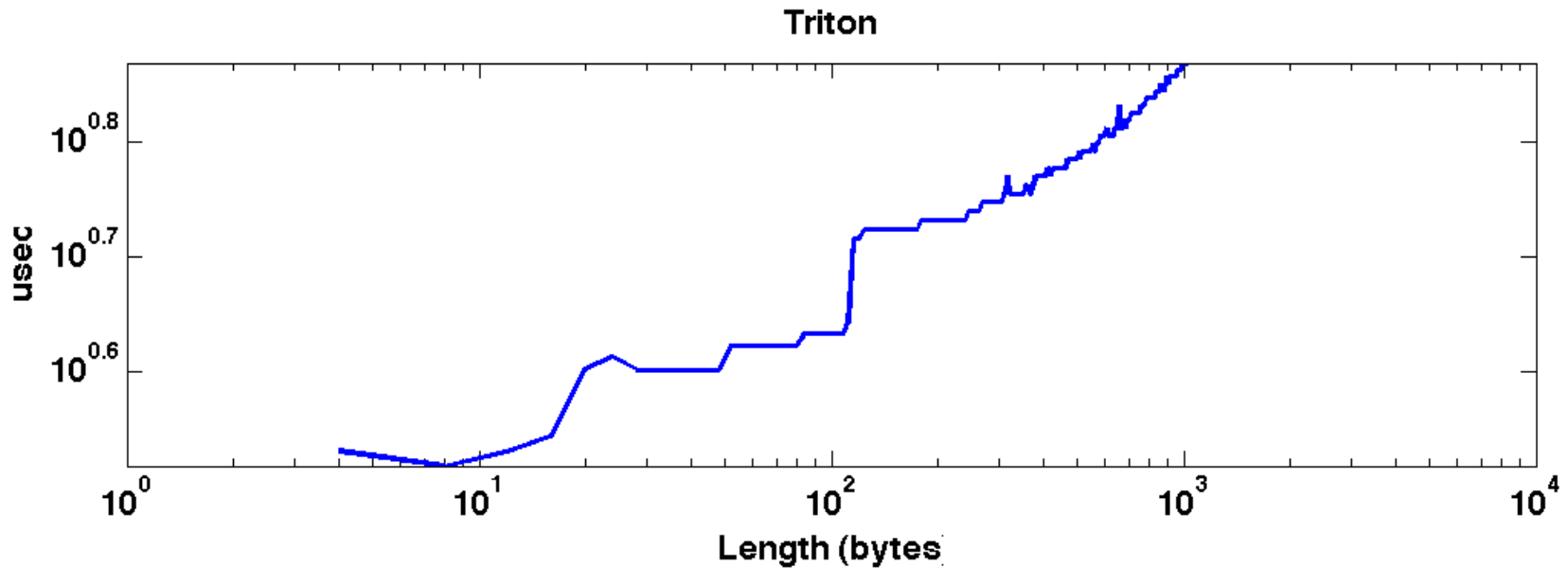
$$\beta^{-1}(n) = n / T(n)$$
- We define the *half power point* $n_{1/2}$ as the message size need to achieve $\frac{1}{2} \beta_{\infty}$

$$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$
- 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
 - ♦ $\alpha \approx 3.2 \mu s$, $\beta_{\infty} \approx 1.2 \text{ Gbytes/sec} \Rightarrow n_{1/2} \approx 3.6 \text{ KB}$
 - ♦ The actual value of $n_{1/2} \approx 20 \text{ KB}$

Length (Bytes)	Bandwidth (GB/sec)	Time (us)
1	0.31	3.247
2	0.62	3.219
4	1.24	3.216
8	2.47	3.244
16	4.91	3.258
32	8.3	3.855
64	15.81	4.047
128	25.28	5.062
256	48.25	5.305
512	86.25	5.936
1024	142.8	7.168
2048	209.3	9.786
4096	188.8	21.7
8192	334.7	24.48
16384	519.2	31.56
32768	718.6	45.6
65536	702.7	93.26
131072	897.1	146.1
262144	1039	252.4
524288	1124	466.4
1048576	1177	890.8
2097152	1201	1747
4194304	1216	3449
8388608	1223	6858

Short and intermediate message lengths



Today's lecture

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

Stencil methods under message passing

- Recall the image smoother

```
for iter = 1 : nSmooth
```

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

```
     $\text{Img}^{\text{new}}[i,j] = (\text{Img}[i-1,j] + \text{Img}[i+1,j] + \text{Img}[i,j-1] + \text{Img}[i,j+1]) / 4$ 
```

```
  end
```

```
   $\text{Img} = \text{Img}^{\text{new}}$ 
```

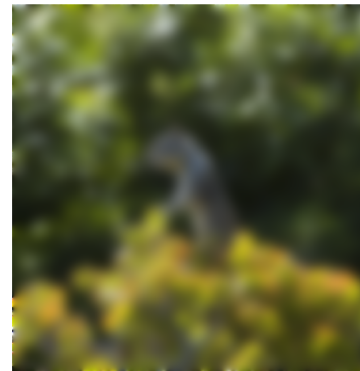
```
end
```



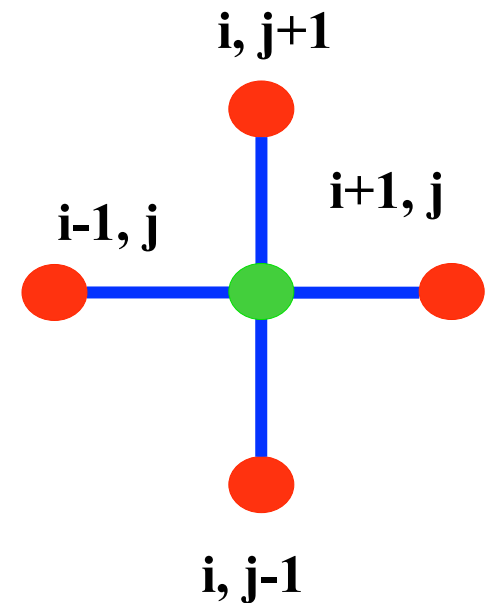
Original



100 iter

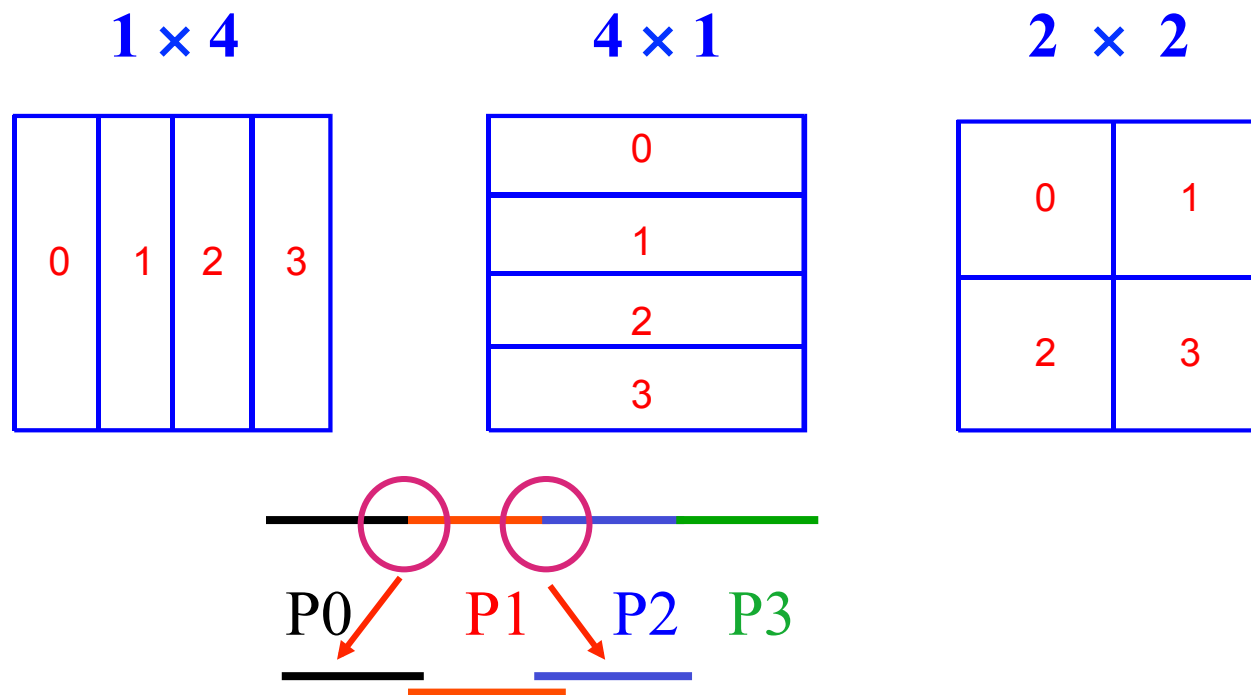


1000 iter



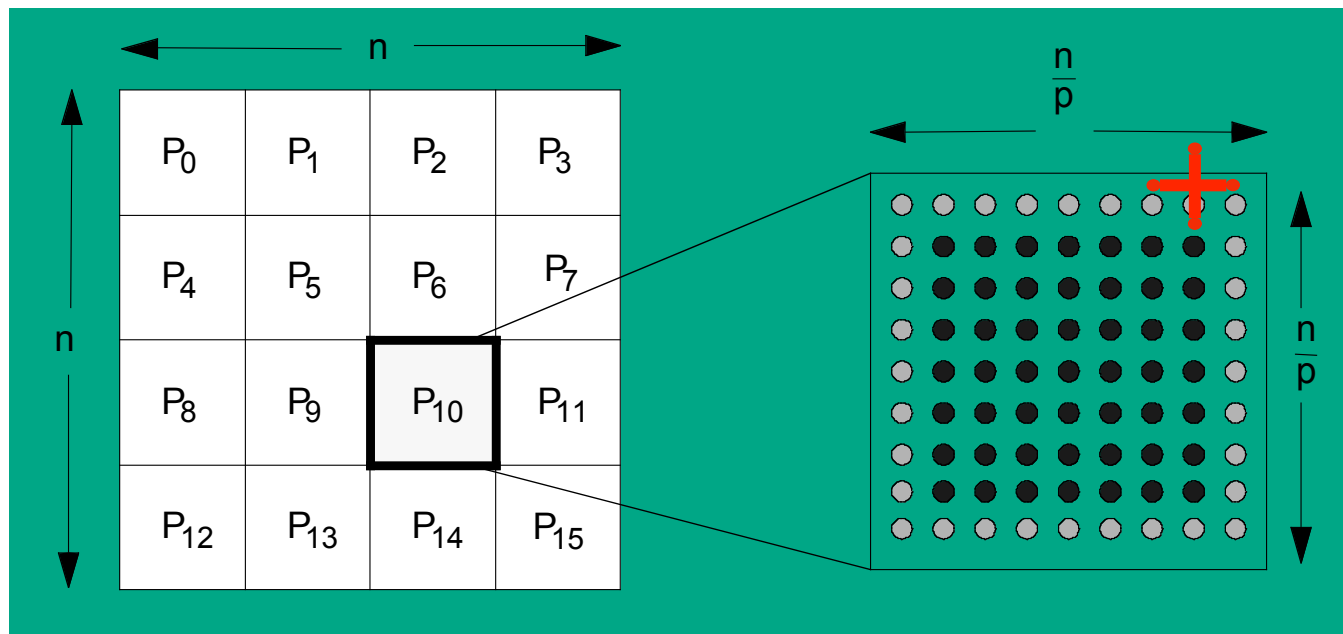
Data partitioning

- Partition computation and data, assigning each partition to a unique process: “Owner computes rule”
- The *processor geometry* defines the partitionings
- Dependences on values found on neighboring processes
- Communicate off-processor data



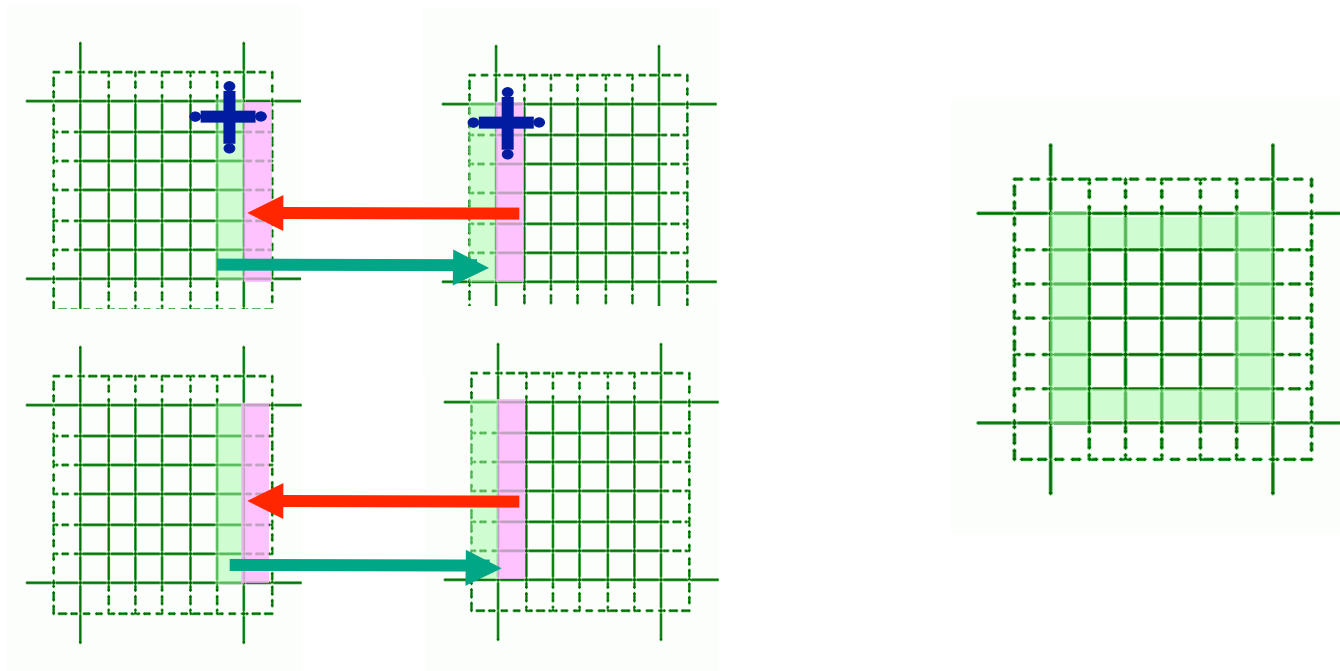
Ghost cells

- “Overlap” or “ghost” cells hold a copies off-process values
- Surround each local subproblem
- Non-contiguous data on some boundaries



Managing ghost cells

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



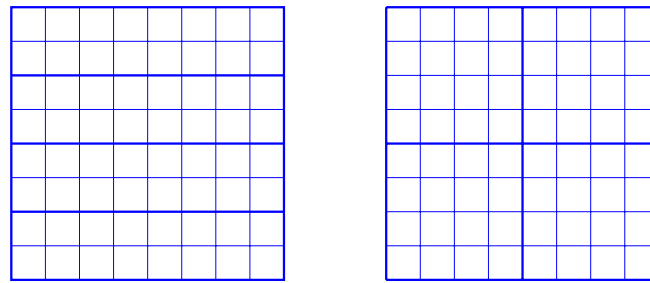
Performance is sensitive to processor geometry

- 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

Modeling the parallel running time

- The model has two parts
 - ♦ Local computation
 - ♦ Communication
- We may ignore the convergence test (check infrequently)
- Communication overheads are due to ghost cell updates
- 2 kinds of geometries: strips vs boxes



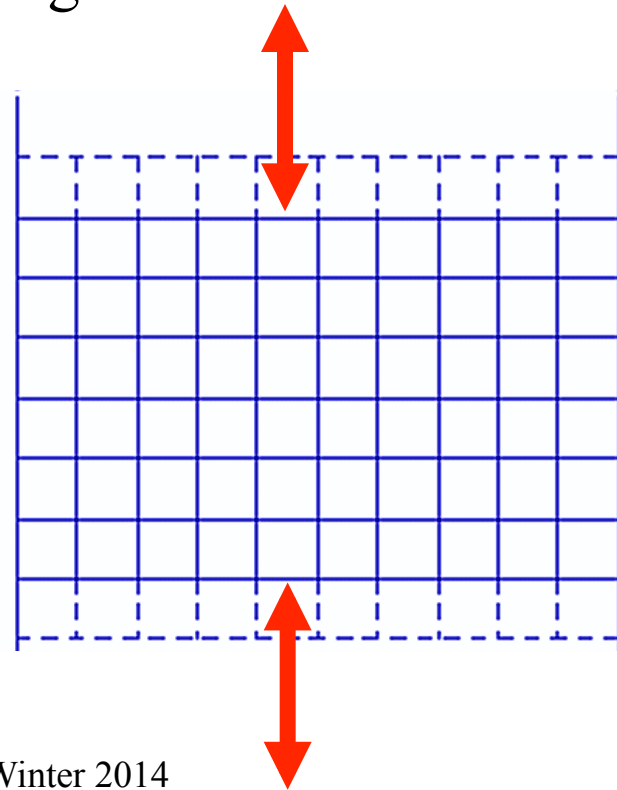
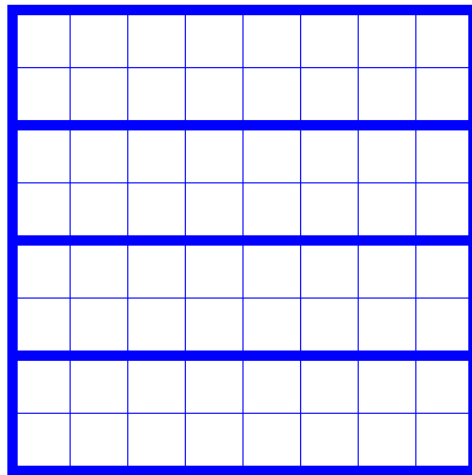
Model assumptions and definitions

- $T(1,(m,n))$ = running time of the **best serial algorithm** on a problem of size $m \times n$
- $T(P,(m,n))$ = running time on P processors
- $T_\gamma(P,(m,n))$ = **grind time** on P processors
 - ♦ $T_\gamma(P,(m,n)) = T(P,(m,n))/(m \cdot n \cdot \text{Niter})$
 - ♦ Ideally T_γ is independent of m , n , and P
- Processor geometry is $p \times q$
 - ♦ Strips or box –like partitions
- $T(P,(N,N)) = T(1,(m,n)) + T_{\text{comm}}$, $m = N/p$, $n = N/q$
- Following analysis applies to a Poisson Solver

$$u'[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 \cdot b[i-1,j-1])/4$$

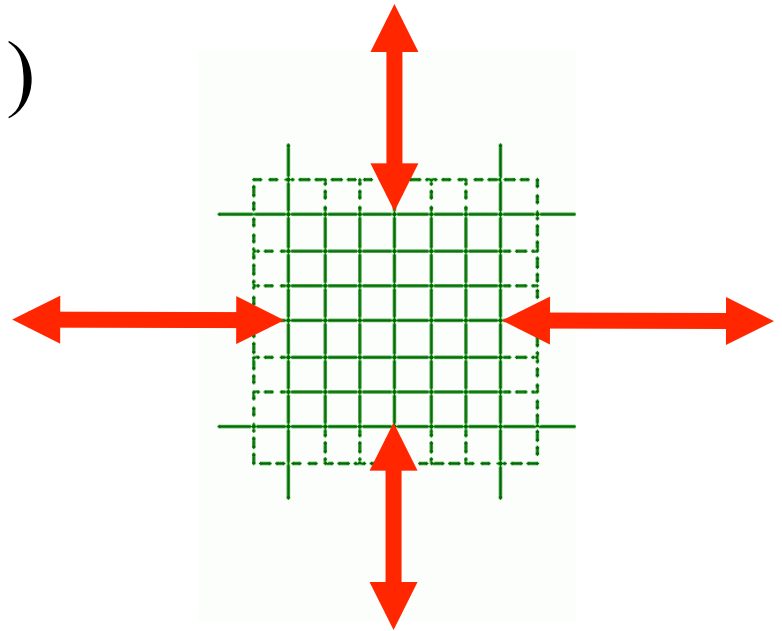
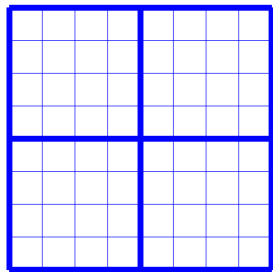
Communication costs for 1D geometries

- Assumptions
 - ♦ P divides N evenly
 - ♦ $N/P > 2$
 - ♦ 1 word = double precision floating point = 8 bytes
- For horizontal strips, data are contiguous
 $T_{\text{comm}} = 2(\alpha + 8\beta N)$



2D Processor geometry

- Assumptions
 - ♦ $\sqrt{P}$ divides N evenly
 - ♦ $N/\sqrt{P} > 2$
 - ♦ 1 word = double precision floating pt. = 8 bytes
- 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 Bang
 - $\alpha = 1.2 \text{ us}, \beta = 1/(1.4 \text{ GB/sec}),$
 - ♦ $N < 210(\sqrt{P}/(\sqrt{P} - 2))$
 - ♦ For $P = 16$, strips are preferable when $N < 280$
- 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 $N < 2340$

Parallel speedup and efficiency

- 1-D decomposition

$$S_p = T_1/T_p = 16N^2\beta / (16N^2\beta/P + 2(\alpha+8\beta N))$$

$$\begin{aligned} E_p = S_p / P &= 16N^2\beta / (16N^2\beta + 2P(\alpha+8\beta N)) \\ &= 1 / (1 + (\alpha+8\beta N)P / (8N^2\beta)) \end{aligned}$$

- 2-D decomposition

$$S_p = T_1/T_p = 16N^2\beta / (16N^2\beta/P + 4(\alpha+8\beta N/\sqrt{P}))$$

$$\begin{aligned} E_p = S_p / P &= 16N^2\beta / ((16N^2\beta) + 4(\alpha P + 8\beta N\sqrt{P})) \\ &= 1 / (1 + (\alpha P + 8\beta N\sqrt{P}) / (4N^2\beta)) \end{aligned}$$

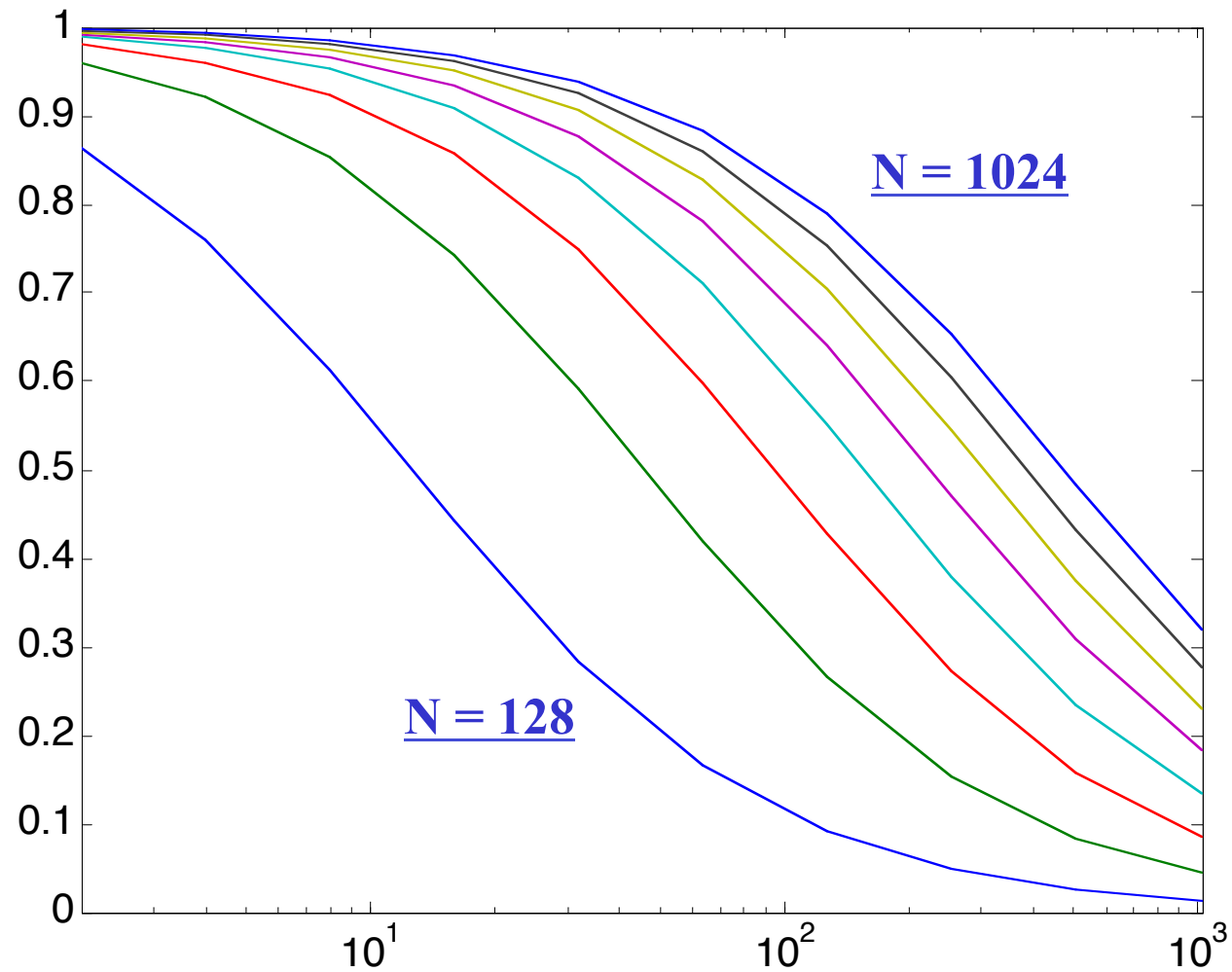
Putting these formulas to work

- 1-D decomposition
- Plot E_p as a function of N , varying P as a parameter

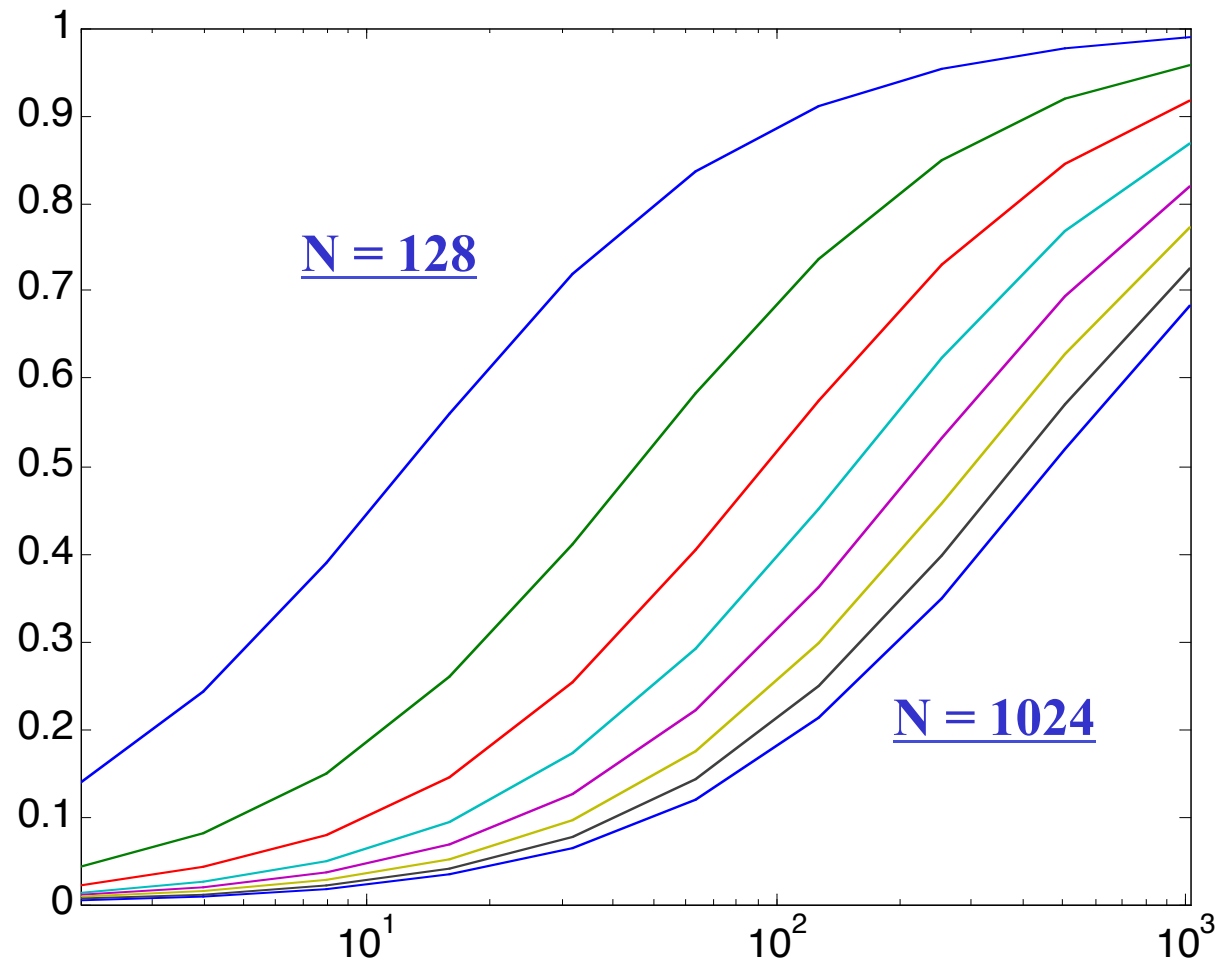
$$E_p = 1 / (1 + (\alpha + 8\beta N)P / (8N^2\beta))$$

- Plot the fraction of time spent communicating

Parallel efficiency



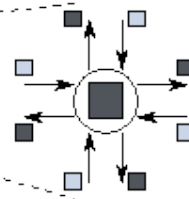
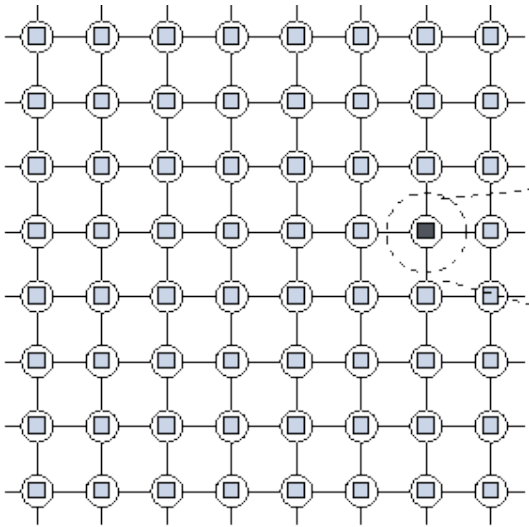
Communication fraction



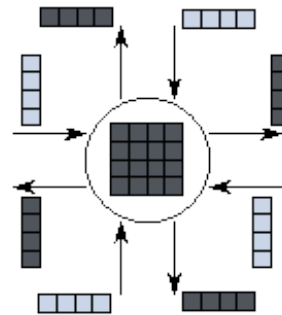
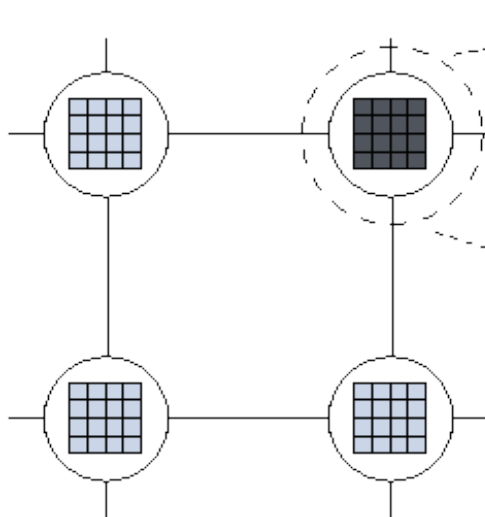
Surface to volume ratio affects performance

- The *surface to volume ratio* of a geometry is the maximum number of points on the surface (perimeter) over all partitions divided by the volume
- As we increase N while leaving P fixed, we decrease the surface to volume ratio, which gives us a measure of the relative cost of communication
- As volume increases, S/V drops

Surface to volume ratio



1 unit of work
4 units of communication



16 units of work
16 units of communication

The curse of dimensionality

- As we move to higher dimensional spaces, communication becomes relatively more costly
 - ▶ In 2D: $4N / N^2 = 4/N$
 - ▶ In 3D: $6N^2 / N^3 = 6/N$
- Large memory strides