Lecture 7

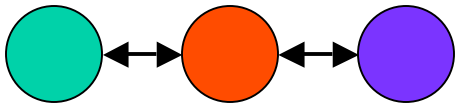
Writing parallel programs with MPI
Measuring performance

Announcements

- Project proposals
- Extra lecture on 10/26 (Monday)
  - Michael Wolfe presentation
  - Room 1202, 1pm
  - Makes up lecture on 11/17

Correctness and fairness

- When there are multiple outstanding iRecvs, MPI doesn't say how incoming messages are matched…
- Or even if the process is fair



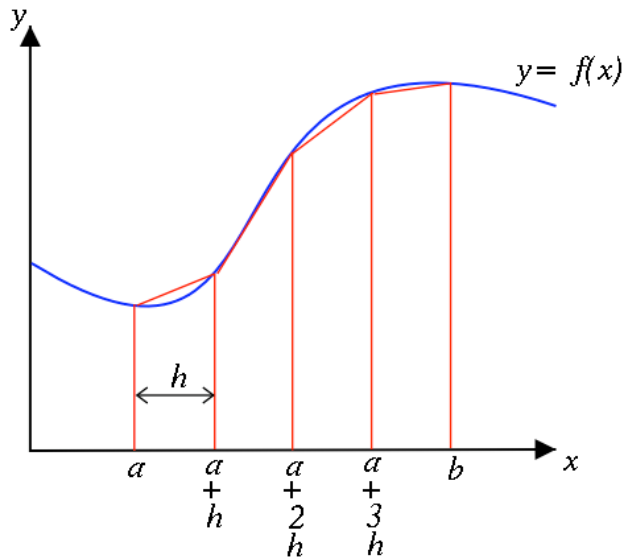
```
MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff,  len, CHAR,ANY_NODE, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
MPI_Send(buff,  len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff,  len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
```

A first MPI Application

The trapezoidal rule

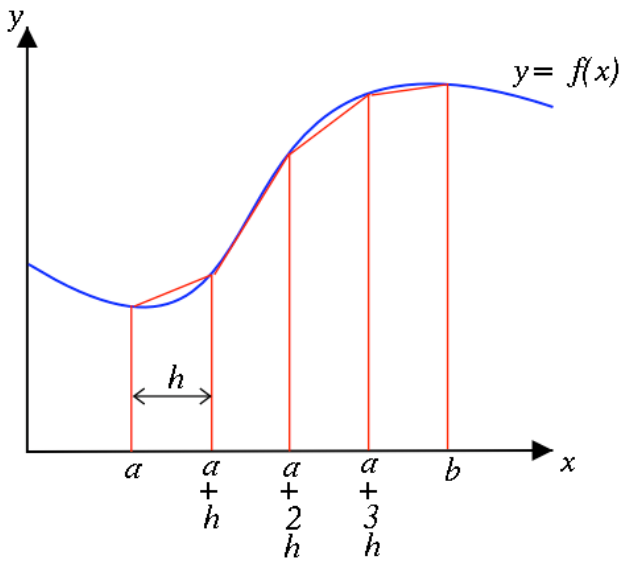
- Use the trapezoidal rule to numerically approximate the definite integral

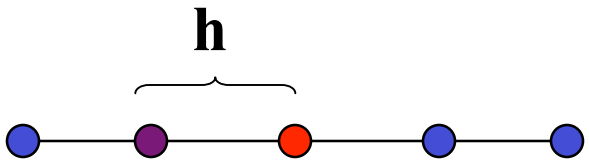
$$\int_a^b f(x)dx$$



How the trapezoidal rule works

- Divide the interval [a,b] into n segments of size h=1/n
- Approximate the area under an interval using a trapezoid
- Area under the i^{th} trapezoid
  ½ (f(a+i×h)+f(a+(i+1)×h)) ×h
- Area under the entire curve
  ≈ sum of all the trapezoids





a+i*h **a+(i+1)*h**

Reference material

- For a discussion of the trapezoidal rule

http://en.wikipedia.org/wiki/Trapezoidal_rule

- A applet to carry out integration

http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Numerical/Integration

- Code on Triton (from Pacheco hard copy text)

Serial Code

$PUB/Pacheco/ppmpi_c/chap04/serial.c

Parallel Code

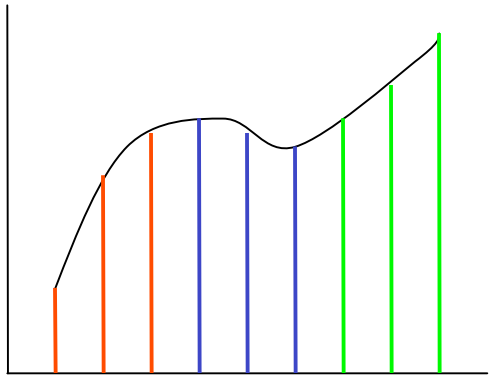
$PUB/Pacheco/ppmpi_c/chap04/trap.c

Serial code (Following Pacheco)

```
main() {
   float f(float x)   { return  x*x;  }          // Function we're integrating

   float h = (b-a)/n;                            //   h = trapezoid base width
                                                 //   a and b: endpoints
                                                 //   n = # of trapezoids
   float integral = (f(a) + f(b))/2.0;

   float  x;   int    i;

   for (i = 1, x=a; i <= n-1; i++) {
      x += h;
      integral = integral + f(x);
   }
   integral = integral*h;
 }
```

The parallel algorithm

- Decompose the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- Processors combine their local integrals into a global one



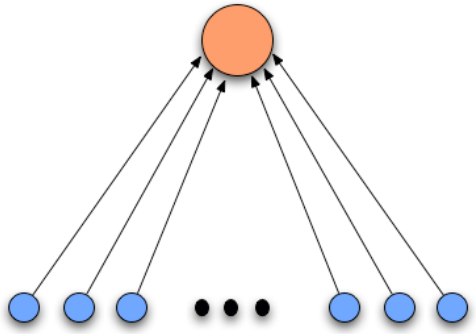
First version of the parallel code

```
local_n = n/p;                  // Number of trapezoids; assume p divides n evenly
float local_a = a + my_rank*local_n*h,
      local_b = local_a + local_n*h,
      integral = Trap(local_a, local_b, local_n, h);

if (my_rank == ROOT) {    // Sum the integrals calculated by all the processes
   total = integral;
   for (source = 1; source < p; source++) {
      MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, WORLD, &status);
      total += integral;
   }
} else
   MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, WORLD);
```

Improvements

- The result does not depend on the order in which the sums are taken, except to within roundoff
- We use a linear time algorithm to accumulate contributions, but there are other orderings

```
for (source = 1; source < p; source++) {
   MPI_Recv(&integral, 1, MPI_FLOAT,
                  MPI_ANY_SOURCE, tag,
                  WORLD, &status);

      total += integral;
}
```



Improved parallel code

- We can often improve performance by taking advantage of global knowledge about communication
- Instead of using point to point communication operations to accumulate the sum, use ***collective*** communication **nt**

```
local_n = n/p;
float local_a = a + my_rank*local_n*h,
      local_b = local_a + local_n*h,
      integral = Trap(local_a, local_b, local_n, h);
MPI_Reduce( &integral, &total, 1,
                  MPI_FLOAT, MPI_SUM,
                  ROOT,WORLD)
```

Collective communication in MPI

- Collective operations are called by **all** processes within a communicator

- Broadcast: distribute data from a designated "root" process to all the others

  MPI_Bcast(in, count, type, root, comm)

- Reduce: combine data from all processes and return to a deisgated root process

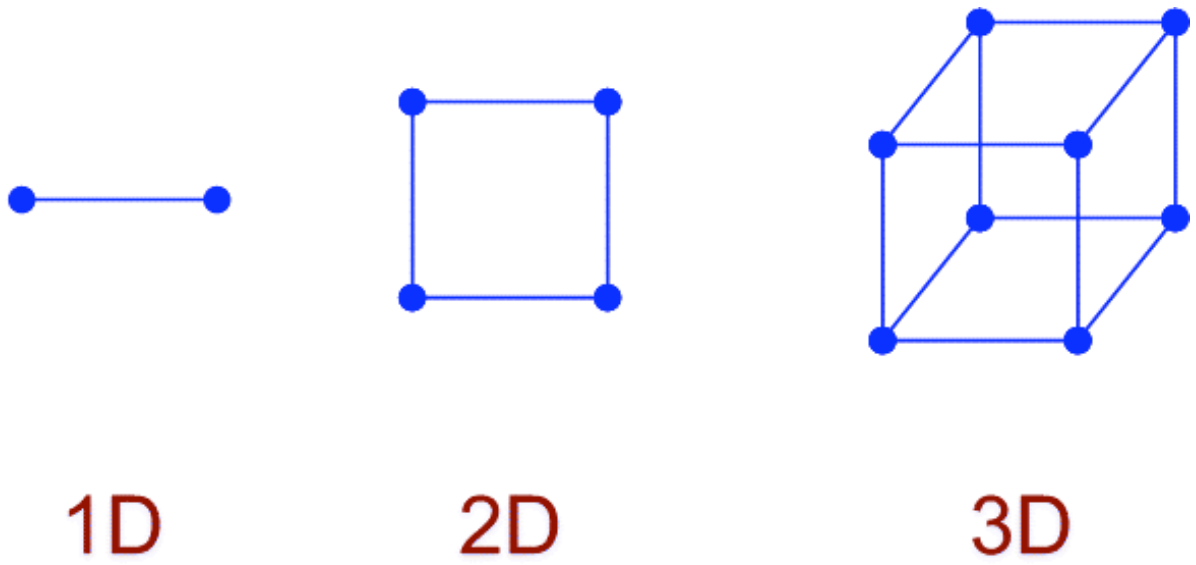
  MPI_Reduce(in, out, count, type, op, root, comm)

Broadcast

- The root process transmits of *m* pieces of data to all the *p-1* other processors
- With the linear ring algorithm this processor performs *p-1* sends of length *m*
  - Cost is $(p-1)(\alpha + \beta m)$
- Another approach is to use the *hypercube algorithm,* which has a logarithmic running time

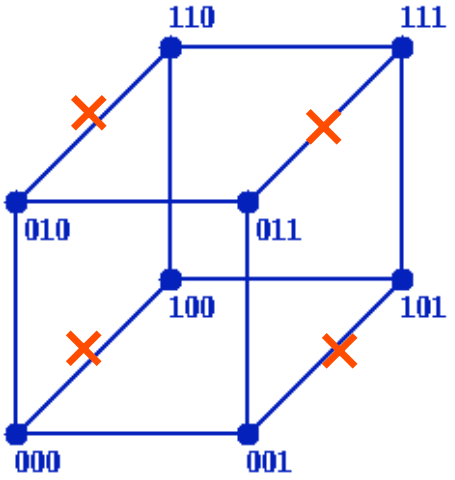
Sidebar: what is a hypercube?

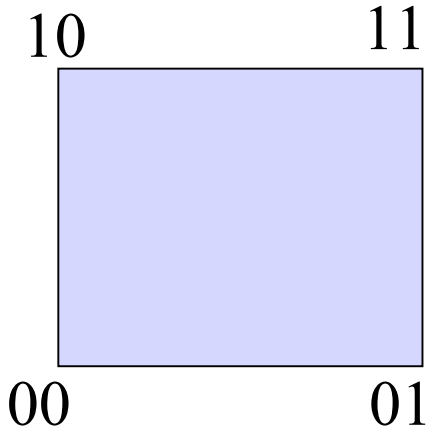
- A hypercube is a d-dimensional graph with 2^d nodes
- A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
- Each node has d neighbors



Properties of hypercubes

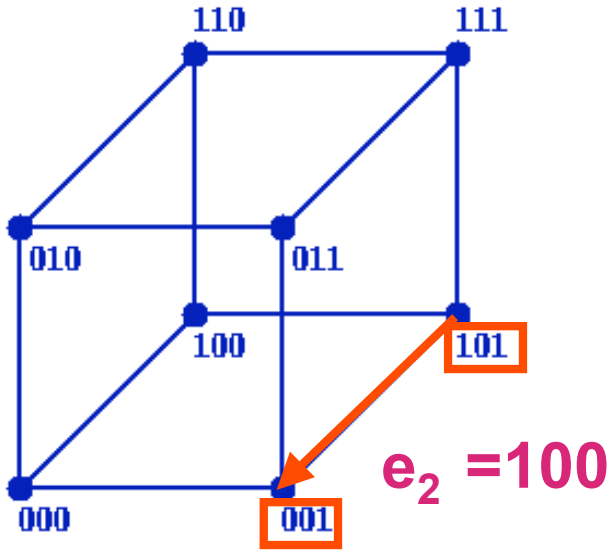
- A hypercube with p nodes has lg(p) dimensions
- *Inductive construction*: we may construct a d-cube from two (d-1) dimensional cubes
- **Diameter:** What is the maximum distance between any 2 nodes?
- **Bisection bandwidth:** How many cut edges (mincut)





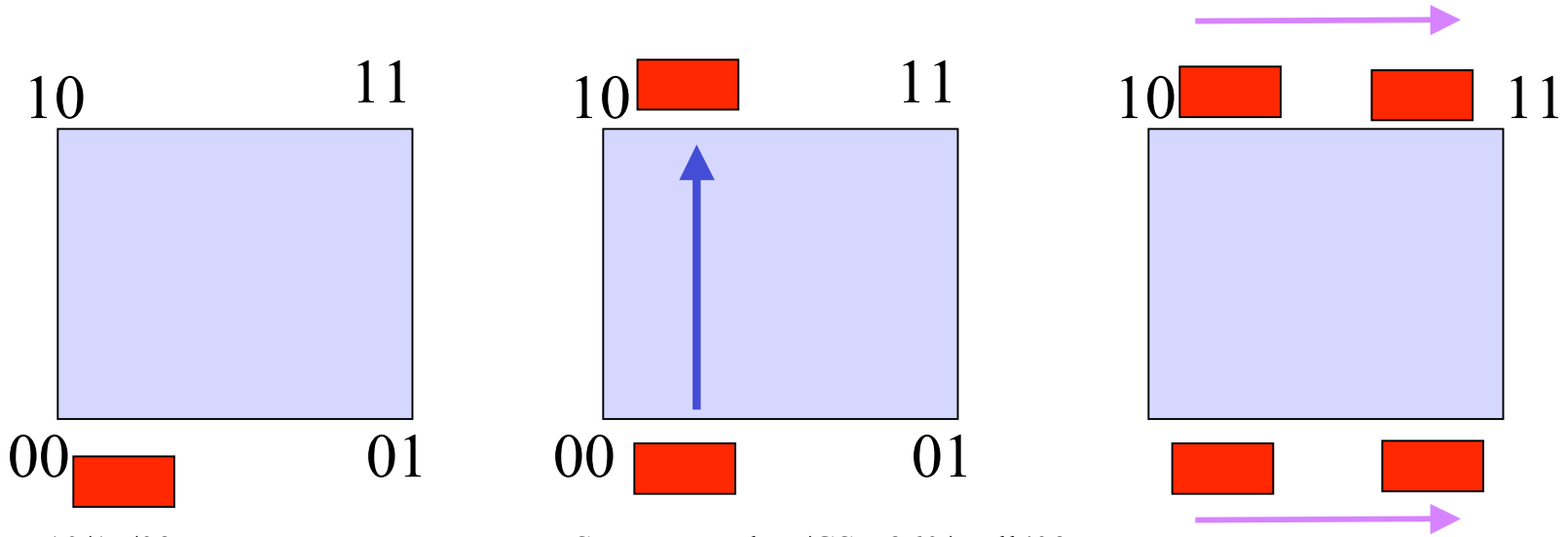
Bookkeeping

- Label nodes with a binary reflected grey code
  http://www.nist.gov/dads/HTML/graycode.html
- Neighboring labels differ in exactly one bit position **$001 = 101 \otimes e_2$, $e_2 = 100$**



Hypercube broadcast algorithm with p=4

- Processor 0 is the root, sends its data to its hypercube "buddy" on processor 2 (10)
- Proc 0 & 2 send data to respective buddies



Reduction

- We may use the hypercube algorithm to perform reductions as well as broadcasts
- Another variant of reduction provides all processes with a copy of the reduced result

  Allreduce( )
- Equivalent to a Reduce + Bcast
- A clever algorithm performs an Allreduce in one phase rather than having perform separate reduce and broadcast phases

Final version

```
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
      local_b = local_a + local_n*h,
      integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce( &integral, &total, 1,
                    MPI_FLOAT, MPI_SUM, WORLD)
```

Performance of iterative methods for solving systems of equations

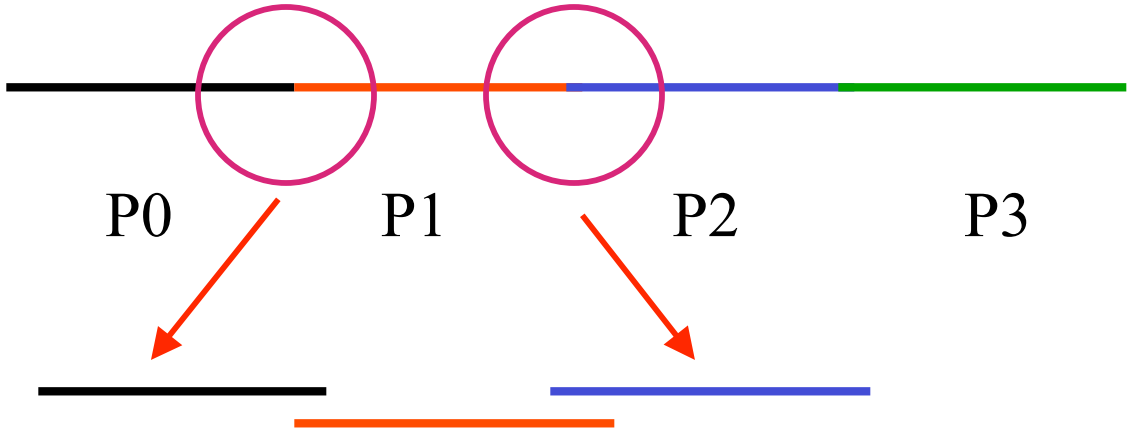
Recall an iterative method in 1D

- Jacobi's Method

```
Repeat until the result is satisfactory
  for i = 1 : N-1
     unew[i] = (u[i-1]+u[i+1] +h*h*f[i])/2
  end for
  Swap u and unew
end Repeat
```

Parallel Implementation

- We partition the data into intervals, assigning each to a unique processor
- Each interval depends on two endpoint values found on neighboring processes
- We add “overlap” or “ghost” cells to hold a copies off-processor values
- Collective communication for convergence check

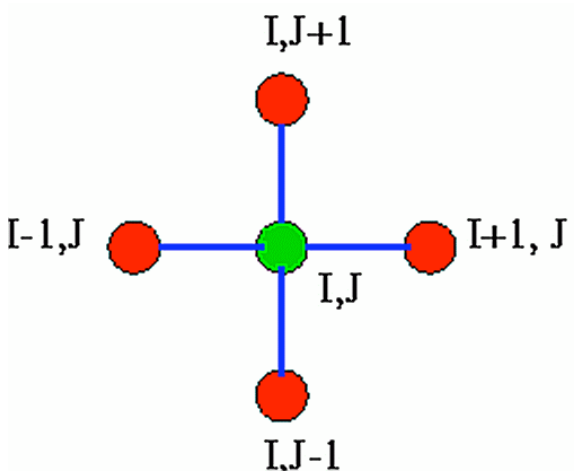


Jacobi's Method in 2D

- The update formula

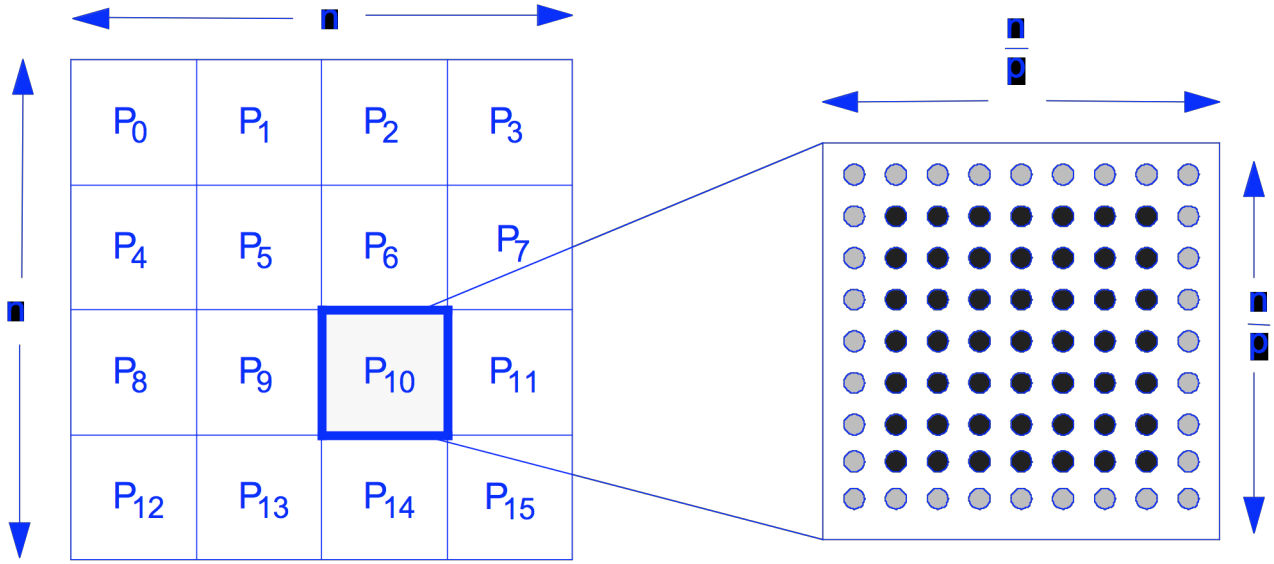
```
for (i,j) in 0:N-1 x 0:N-1
  u'[i,j] = (u[i-1,j] + u[i+1,j] +
             u[i,j-1] + u[i, j+1]
             - h²f[i,j])/4
u =u'
```

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

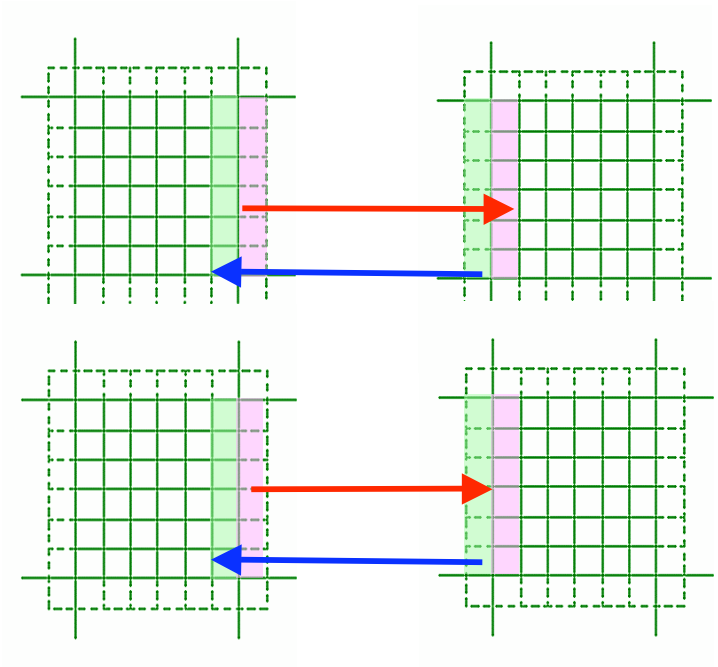


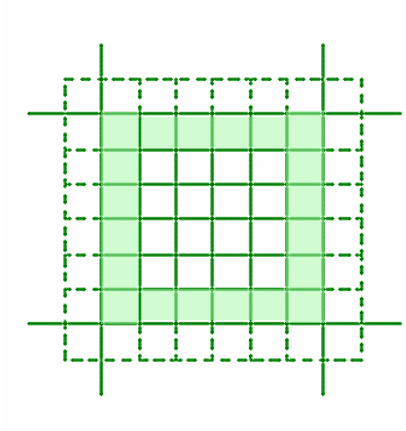
Ghost cells 2D

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



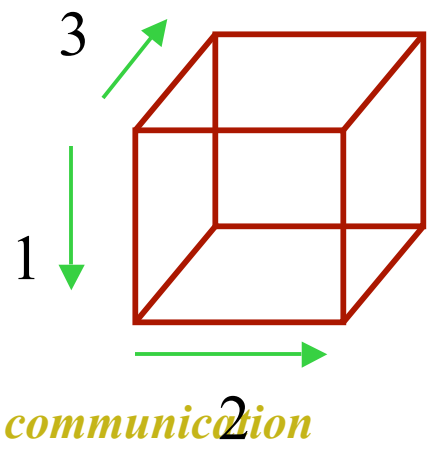
Periodically refresh the ghost cells





Some results in 3D on an IBM SP2 with 16 processors

	Total		Without communication			
Geometry	MF/s	Time	Comm	MF/s	Time	
16x 1x1	510	15.80	2.41	601	13.39	
1x16x 1	655	12.29	2.43	816	9.87	
1x 1x 16	547	14.72	8.33	**1260**	**6.39**	*57% communication*
4x 4x 1	611	13.18	**0.90**	656	12.27	*6.8% communication*
4x 1x 4	674	11.94	2.71	872	9.23	
1x 4x 4	621	12.96	2.77	790	10.19	
8x 2x 1	554	14.54	1.43	614	13.11	
8x 1x 2	619	13.00	1.21	683	11.79	
4x 2x 2	629	12.81	1.10	688	11.71	
2x 4x 2	629	12.81	1.07	686	11.74	
2x 2x 4	671	12.00	2.48	846	9.52	
2x 8x 1	669	12.03	1.39	757	10.64	
2x 1x 8	617	13.06	4.40	931	8.65	
1x 2x 8	560	14.37	4.43	810	9.95	
1x 8x 2	**693**	**11.63**	1.84	823	9.79	*15.8% communication*



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
- Let's start with the 1D ODE solver and then move to higher dimensional spaces
- We'll use valkyrie.ucsd.edu

Model assumptions and definitions: 1D case

- T(1,n) = running time of the **best serial algorithm** on a problem of size n
- T(P,n) = running time on P processors
- $T_\gamma(P,n)$ = **grind time**
  - Time to perform a single mesh update
  - Helps us normalize with respect to problem size
  - $T_\gamma(P,n0) = T(P,n)/(n \bullet Niter)$

Local Computation time

- $T(1,n) = n\ T_\gamma\ N_{Iter}$, where T_γ is the cost of performing an update

  $$u_i = (u_{i+1}+u_{i-1} +h^2 f_i)/2$$

- On Valkyrie

  $$T_\gamma \approx 20\beta$$

- Datum are 8-byte double precision numbers, message passing time = $\alpha+8\beta N$

More on the Performance model

- Make the naïve assumption that $T_\gamma(1,n)$ is independent of n
- T_{comm} = local communication for ghost cells
- $T(P,N) = T(1,N/p) + T^{local}_{comm}$
- $= 20N\beta/P + 2(\alpha+8\beta)$

  $\approx 20N\beta/P$

The curse of dimensionality

- In higher dimensional spaces
  - Many more possible processor geometries
  - Communication involves higher dimensional arrays
  - Fraction of communication increases for a fixed number of unknowns
- In 1D
  - There is only one possible processor geometry
  - Each process communicates at most 2 points
- In 2D
  - There are 1D and 2D geometries
  - Each process communicate a set of 1D arrays
- In D dimensions
  - D different sets of geometries
  - Each process communicates several (D-1) dimensional arrays

Model assumptions and definitions : 2D case

- T(1,(m,n)) = running time of the **best serial algorithm** on a problem of size m × n
- T(P,(m,n)) = running time on P processors
- T_γ(P,(m,n)) = **grind time** on P processors
  - T_γ(P,(m,n)) = T(P,(m,n))/(m • n • Niter)
  - Ideally T_γ is independent of m, n, and P
- Following analysis applies to Laplace's equation, a special case of Poisson's Equation with f=0

Performance

- 2 components
  - Local computation
  - Ghost cell communication
  - (Convergence check, including global communication)

Local Computation time

- T(1,(m,n)) = $m \times n \times T_\gamma$ where

  T_γ = grind time on one processor

- For the Blue Horizon IBM SP3 system at the San Diego Supercomputer Center (with Power3 CPUs)

  $T_\gamma \approx 16\ \beta$

$$u_{i,j} = (u_{i+1,j}+u_{i-1,j} + u_{i,j+1}+u_{j,i-1})/4$$

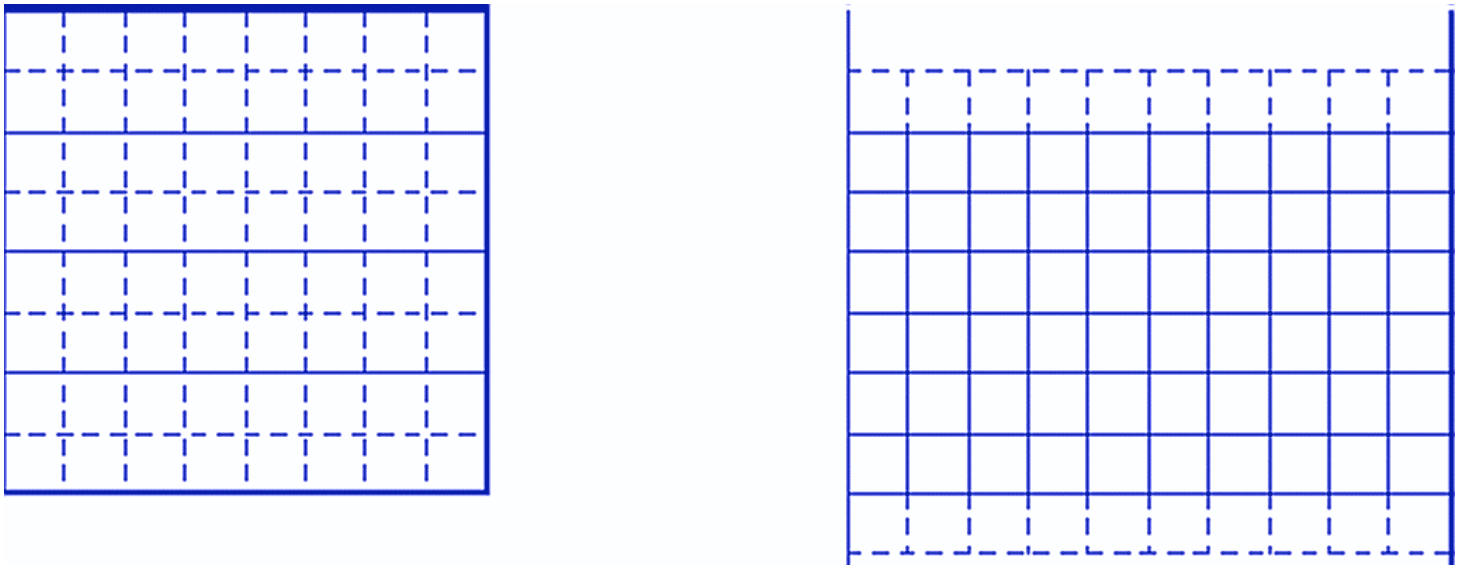
Completing the Performance model

- Message passing time $T(N) = \alpha + 8\beta N$
  (8-byte double precision numbers)
- Processor geometry is p × q
  - Strips or box –like partitions
- $T(P,(N,N)) = T(1,(m,n)) + T_{comm}^{local}$

  $m = N/p$, $n = N/q$

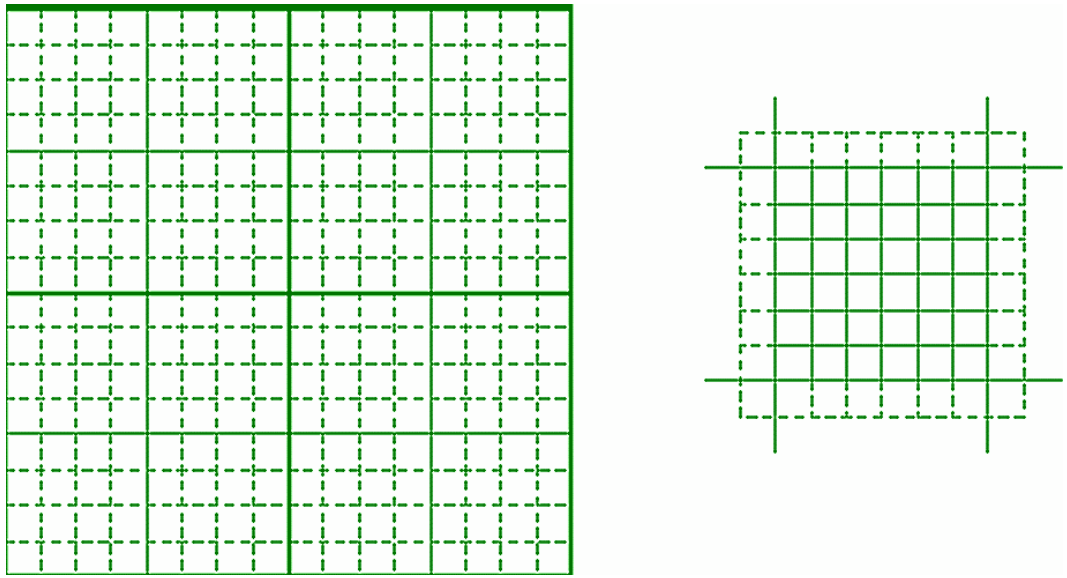
Communication performance for 1D

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



2D Processor geometry

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



Summing up the performance models

- 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≥2: $N < (\sqrt{P}/(\sqrt{P}-2))(\alpha/8\beta)$
- On SDSC's IBM SP3 system "Blue Horizon"

  $\alpha = 24$ *us*

  $\beta = 1/(390$ *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

$$S_P = T_1/T_P = 16N^2\beta / (16N^2\beta/P + 2(\alpha+8\beta N))$$
$$E_P = S_P / P = 16N^2\beta / (16N^2\beta + 2P(\alpha+8\beta N))$$
$$= 1 /(1 + (\alpha+8\beta N)P/ (8N^2\beta))$$

- 2-D decomposition

$$S_P = T_1/T_P = 16N^2\beta/(16N^2\beta/P+4(\alpha+8\beta N/\sqrt{P})))$$
$$E_P = S_P / P = 16N^2 \beta/((16N^2\beta)+4(\alpha P+8\beta N\sqrt{P}))$$
$$= 1 / (1 + (\alpha P+8\beta N\sqrt{P})/(4N^2 \beta))$$

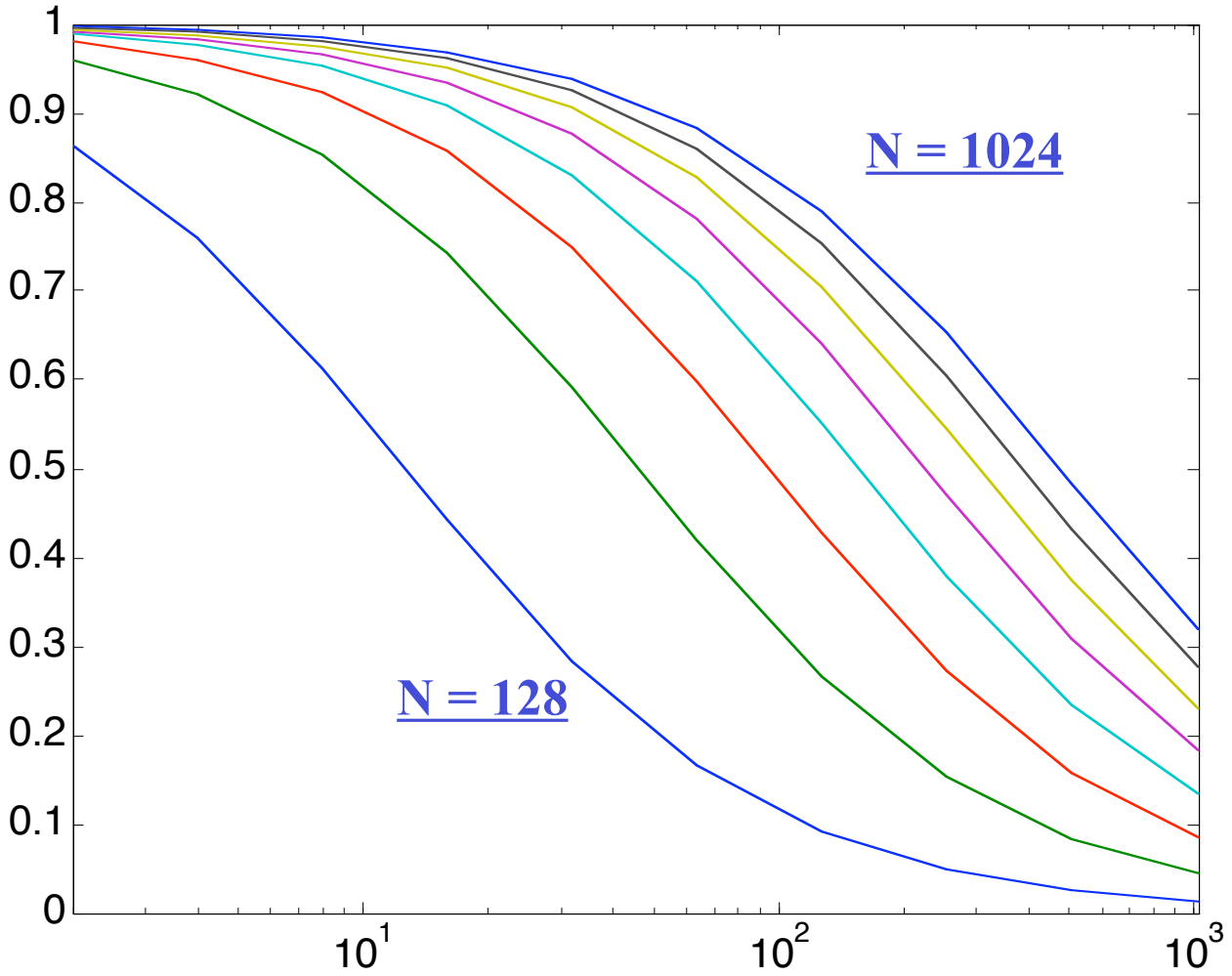
Putting these formulas to work

- 1-D decomposition
- Let's plot E_P as a function of N, varying P as a parameter

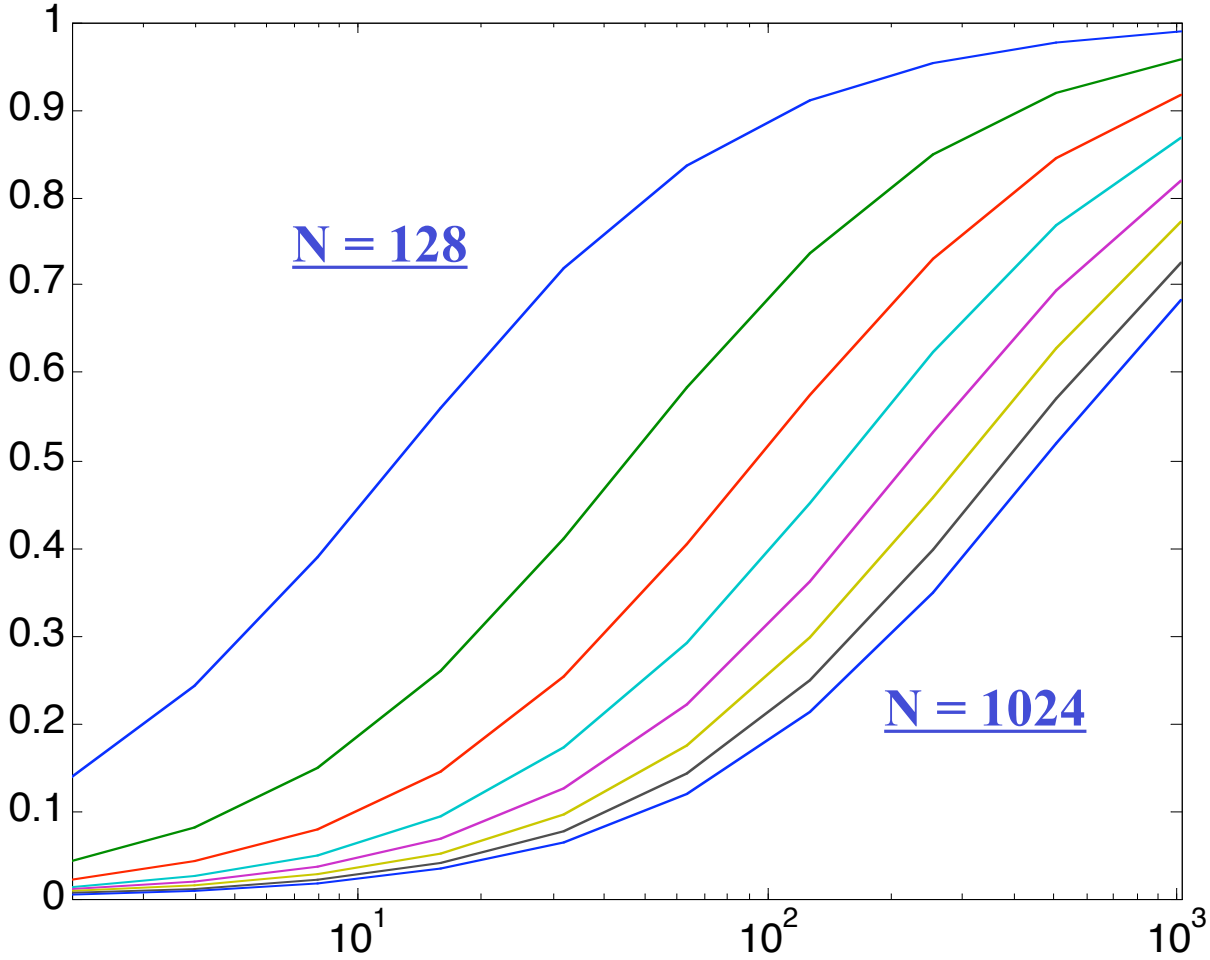
$$E_P = 1 /(1 + (\alpha+8\beta N)P/ (8N^2\beta))$$

- Let's also plot the fraction of time spent communicating

Parallel efficiency



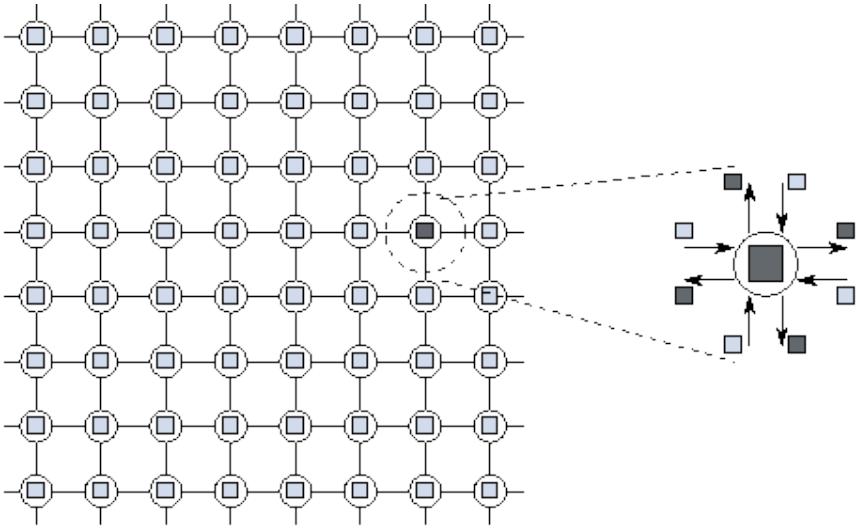
Communication fraction



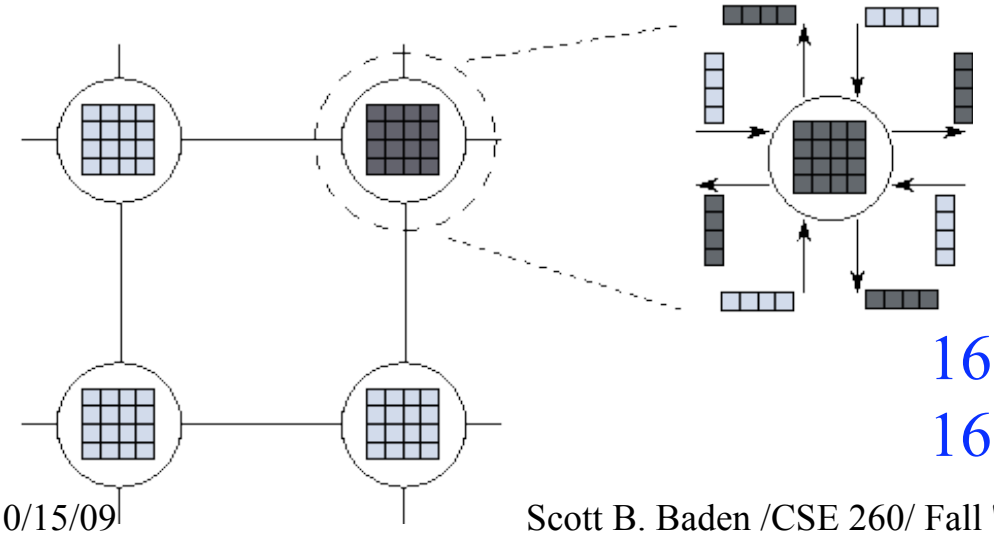
Surface to volume ratio affects performance

- 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: $4N / N^2 = \mathbf{4/N}$
  - In 3D: $6N^2 / N^3 = \mathbf{6/N}$