High Performance Scientific Computing
MPI II.

SciNet HPC Consortium
University of Toronto

Winter 2014

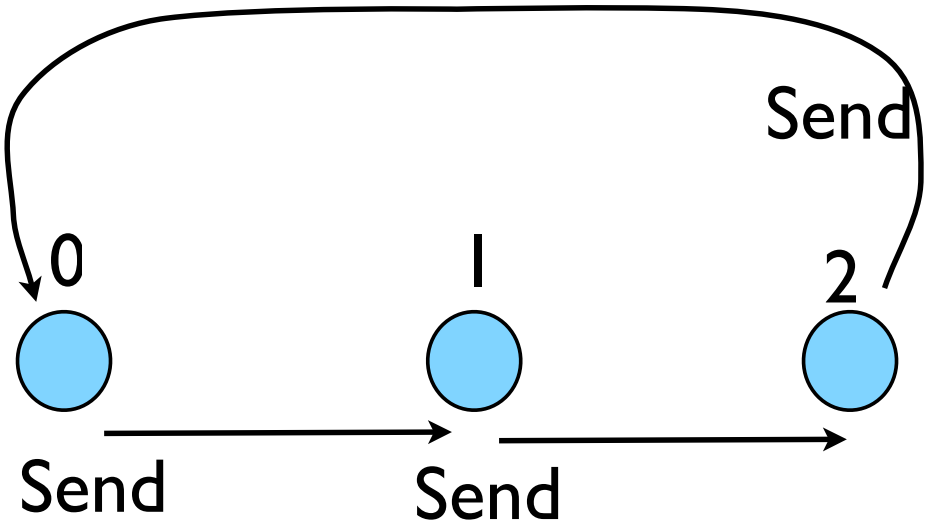
Message Passing Interface (MPI)

Review: MPI I

- MPI: Library for sending messages
- MPI Basics (MPI_INIT, size, rank, MPI_COMM_WORLD)
- MPI utils (mpirun, mpic++)
- Pairwise Communication
  - Send & Recv
  - Deadlocks

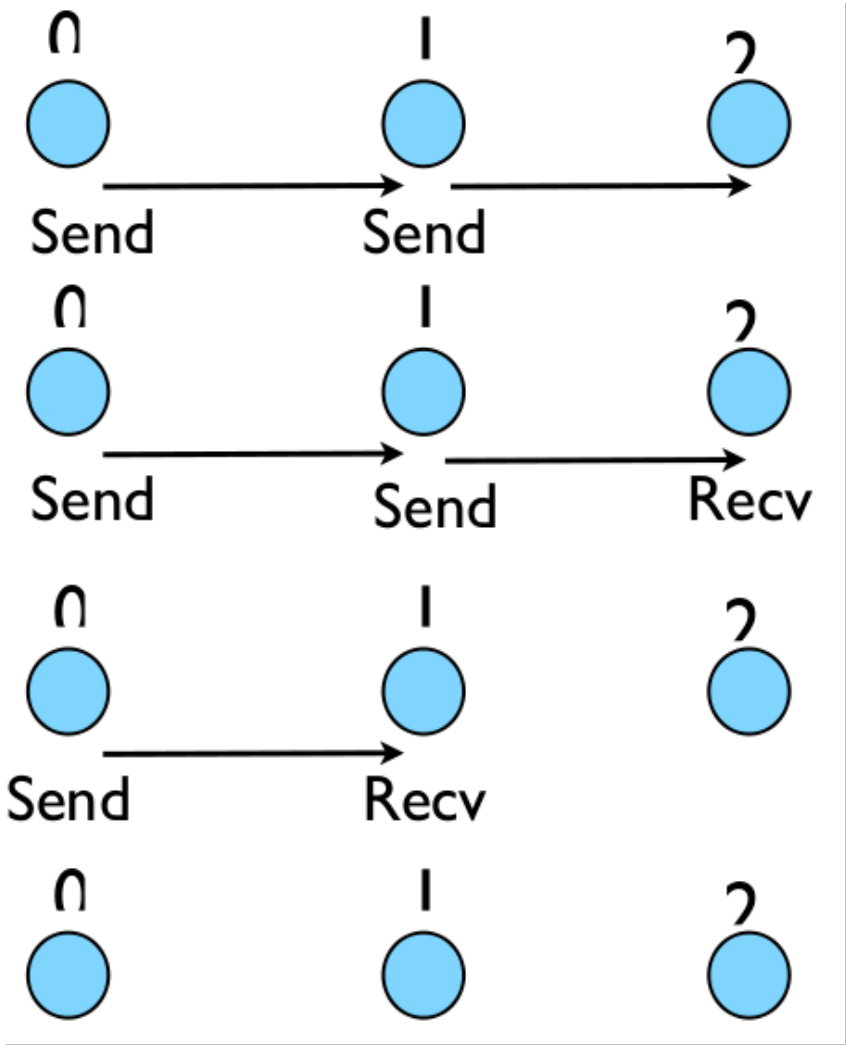
Deadlock

- A classic parallel bug
- Occurs when a cycle of tasks are for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) deadlock.



MPI: Send Left, Receive Right with Periodic BC's

Message Progression



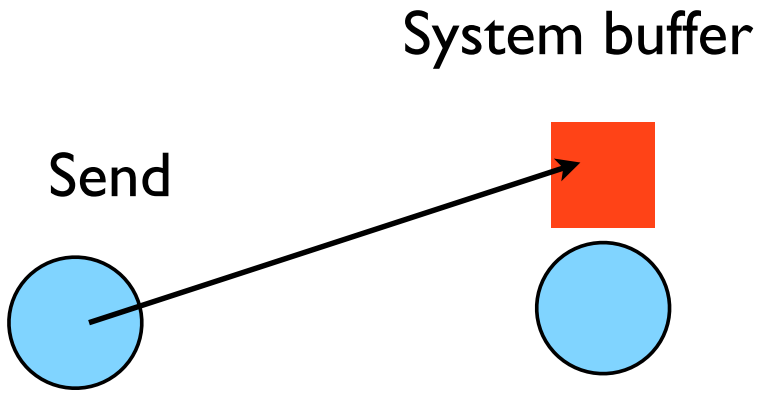
Big MPI
Lesson #1

All sends and receives must be paired, **at time of sending**

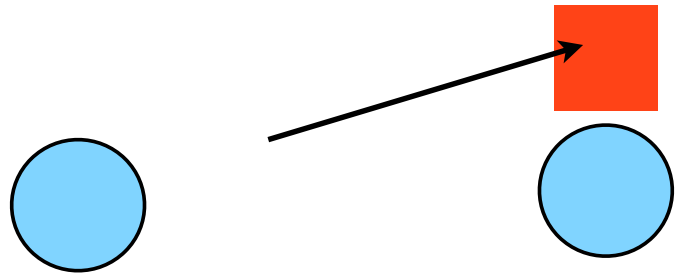
Different versions of SEND

- SSEND: safe send; doesn't return until receive has started. Blocking, no buffering.
- SEND: Undefined. Blocking, probably buffering
- ISEND : Unblocking, no buffering
- IBSEND: Unblocking, buffering

Buffering



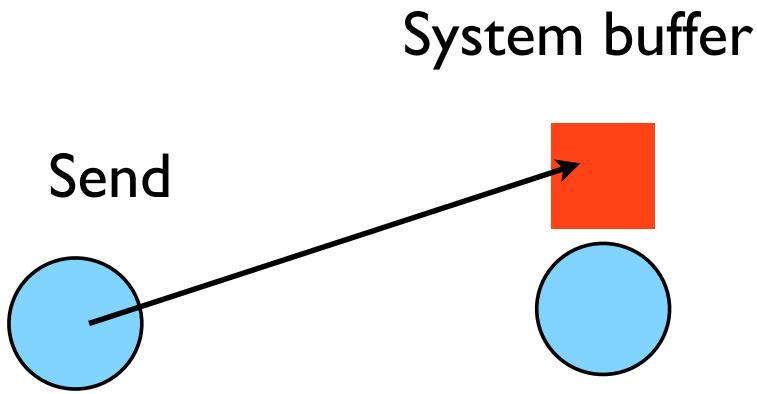
(Non) Blocking



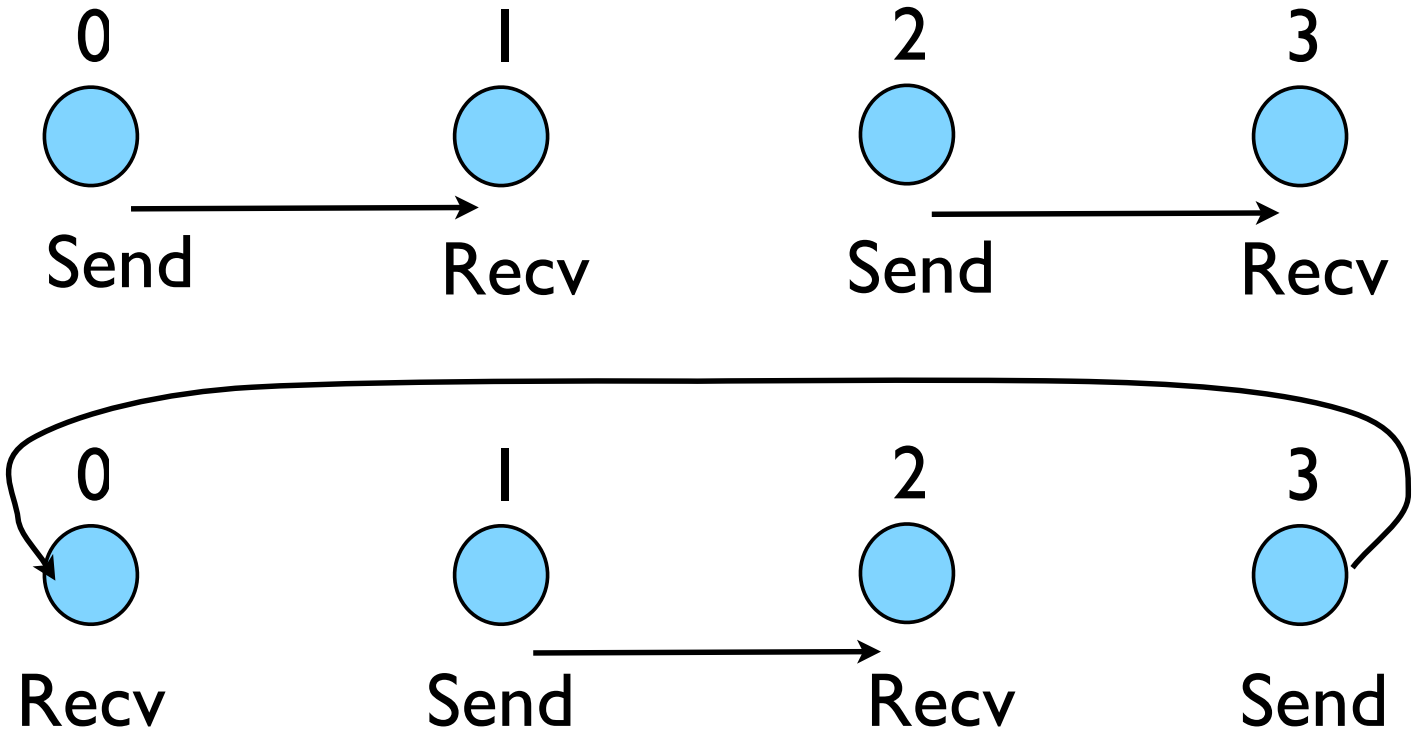
Buffering is dangerous!

- Worst kind of danger: will usually work.
- Think voice mail; message sent, reader reads when ready
- But voice mail boxes do fill
- Message fails.
- Program fails/hangs mysteriously.
- (Can allocate your own buffers)

Buffering



Without using new MPI routines, how can we fix this?



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd # of processes?
- How about 2? 1?

MPI: Send Left, Receive Right with Periodic BC's - fixed

```
{
  ...
  //Even/odd message passing to avoid deadlock
  if ((rank % 2) == 0) {
    ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag,
    MPI_COMM_WORLD);
    ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag,
    MPI_COMM_WORLD, &rstatus);
  } else {
    ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag,
    MPI_COMM_WORLD, &rstatus);
    ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag,
    MPI_COMM_WORLD);
  }
}
```

MPI: Send Left, Receive Right with Periodic BC's - fixed

```
$mpirun -np 5 ./fourthmessage
1:  Sent 1 and got 0
2:  Sent 4 and got 1
3:  Sent 9 and got 4
4:  Sent 16 and got 9
0:  Sent 0 and got 16
```

MPI: Sendrecv

```
ierr = MPI_Sendrecv(sendptr, count, MPI_TYPE,
destination, tag, recvptr, count, MPI_TYPE, source, tag,
Communicator, MPI_Status)
```

- A blocking send and receive built together
- Lets them happen simultaneously
- Can automatically pair send/recvs
- Why 2 sets of tags/types/counts?

MPI: Send Left, Receive Right with Periodic BC's - Sendrecv

```
{
  ...
  //Replace separate Send/Recv's with Sendrecv
  ierr = MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE,
  right, tag, &msgrcvd, 1, MPI_DOUBLE, left, tag,
  MPI_COMM_WORLD, &rstatus);

  cout<<rank<<":  Sent "<<msgsent<<" and got "<<msgr-
  cvd<<endl;
  ierr = MPI_Finalize();
  return 0;
}
```

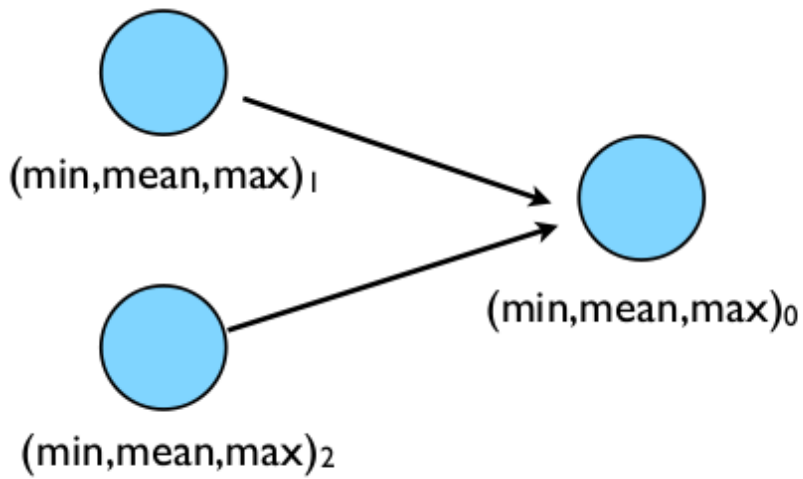
MPI: Send Left, Receive Right with Periodic BC's - Sendrec

```
$mpirun -np 5 ./fifthmessage
1:  Sent 1 and got 0
2:  Sent 4 and got 1
3:  Sent 9 and got 4
4:  Sent 16 and got 9
0:  Sent 0 and got 16
```

Collective Operations

Min, Mean, Max

- Caclulate the min/mean/max of random numbers -1..1
- Should trend to -1/0/+1 for a large N
- How to MPI it?
- Partial results on each node, collect all to node 0.



Min Mean Max

```
#include <iostream>
#include <mpi.h>
#include <stdlib.h>
using namespace std;
int main(int argc, char **argv) {
  const int nx=1500;
  float datamin, datamax, datamean
  float mmm[3], globmmm[3];
  int ierr, rank, size, tag(1);
  MPI_Status status;

  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_size(MPI_COMM_WORLD,&size);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  ...
}
```

Min Mean Max: continued

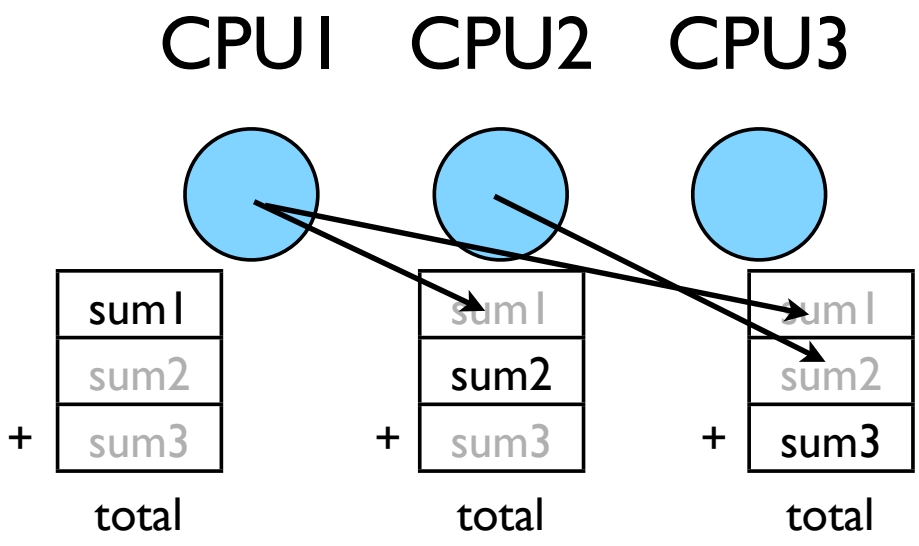
```
// generate random data
float *dat = new float [nx];
srand(0);
for (int i=0;i<nx;i++) {
  dat[i] = 2*((float)rand()/RAND_MAX)-1.;
}

//find min/mean/max
datamin = 1e+19; datamax =-1e+19; datamean = 0;
for (int i=0;i<nx;i++) {
  if (dat[i] < datamin) datamin=dat[i];
  if (dat[i] > datamax) datamax=dat[i];
  datamean += dat[i];
}
datamean /= nx;
```

Min Mean Max: MPI

```
mmm[0] = datamin; mmm[1] = datamean; mmm[2] = datamax;
if (rank != 0) {
  ierr = MPI_Ssend(mmm,3,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
} else {
  globmmm[0] = datamin; globmmm[1] = datamean;
  globmmm[2] = datamax;
  for (int i=1;i<size;i++) {
    ierr = MPI_Recv(mmm,3,MPI_FLOAT,MPI_ANY_SOURCE,tag,
    MPI_COMM_WORLD,&status);
    globmmm[1] += mmm[1];
    if (mmm[0] < globmmm[0]) globmmm[0] = mmm[0];
    if (mmm[2] > globmmm[2]) globmmm[2] = mmm[2];
  }
  globmmm[1] /= size;
  cout<<"Global Min/mean/max "<<globmmm[0]<<" "<<
  globmmm[1]<<" "<<globmmm[2]<<endl;
}
ierr = MPI_Finalize(); return 0;
```

Inefficient!

Requires (P-1) messages, 2(P-1) if everyone then needs to get the answer.



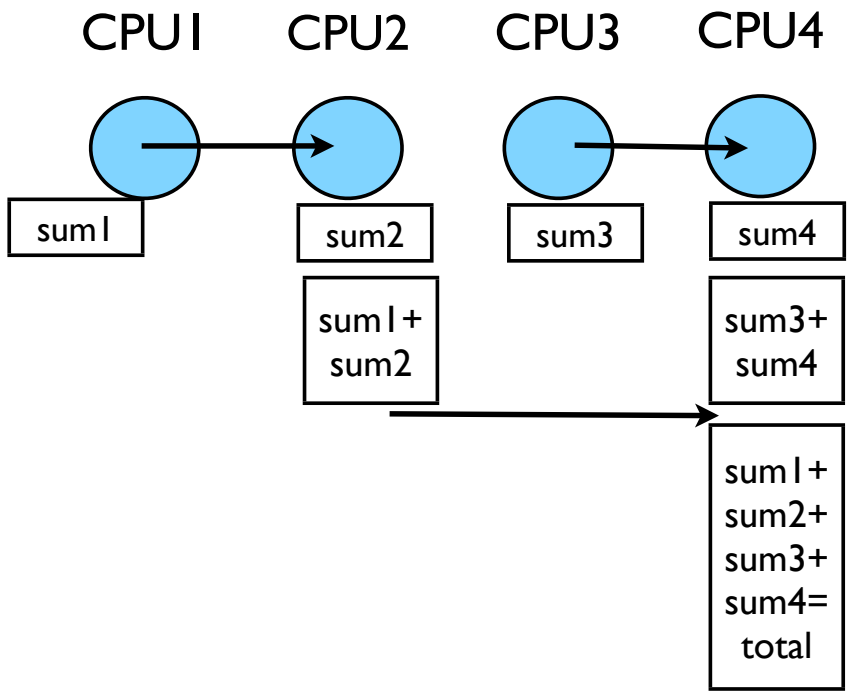
Better Summing

Pairs of processors; send partial sums

Max messages received $\log_2(P)$

Can repeat to send total back

$$T_{\text{comm}} = 2\log_2(P)C_{\text{comm}}$$



Reduction; works for a variety of operators (+,*,min,max...)

SciNet compute • calcul CANADA

MPI Collectives

```
ierr = MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE,
MPI_OP, Communicator);
```

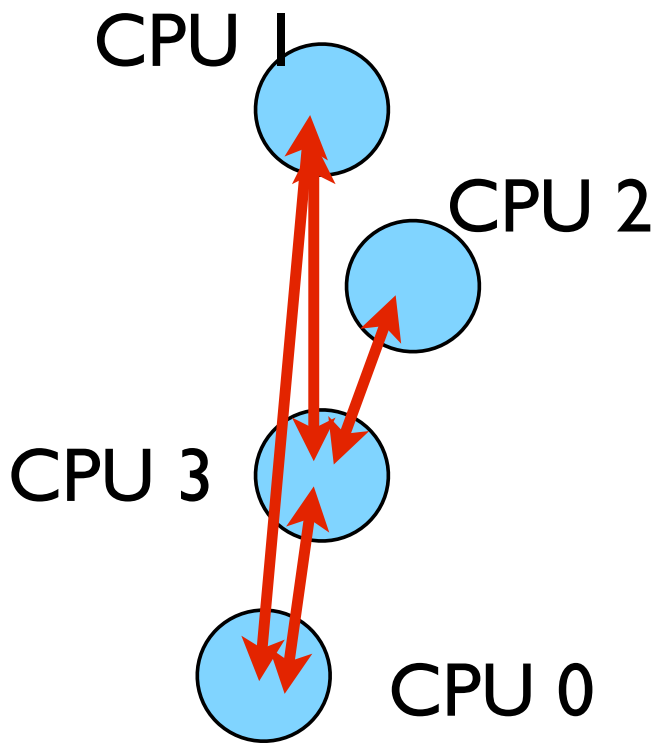
- `sendptr/rcvptr`: pointer to buffers
- `count`: number of elements in ptr
- `MPI_TYPE`: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `MPI_OP`: one of MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX, etc.
- `Communicator`: MPI_COMM_WORLD or user created

Min Mean Max: Allreduce

```
ierr = MPI_Allreduce(&datamin, &globalmin, 1, MPI_FLOAT,
MPI_MIN, MPI_COMM_WORLD);
ierr = MPI_Allreduce(&datamax, &globalmax, 1, MPI_FLOAT,
MPI_MAX, MPI_COMM_WORLD);
ierr = MPI_Allreduce(&datamean, &globalmean, 1,
MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
globalmean /= size;

if (rank == 0) {
  cout<<"Global Min/mean/max "<<globalmin<<" "<< global-
  max<<" "<<globalmax<<endl;
}
ierr = MPI_Finalize();
return 0;
```

Collective Operations

As opposed to the pairwise messages we've seen
All processes in the communicator must participate
Cannot proceed until all have participated
Don't necessarily know what goes on 'under the hood'



SciNet compute • calcul CANADA

Scientific MPI Example

MPI "Real" problems

- Finite Difference Stencils
- Time-Marching Method
- Domain Decomposition
- Load Balancing
- Global Norms
- BC's

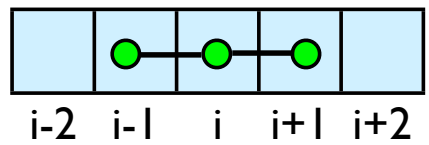
Discretizing Derivatives

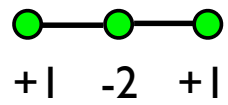
$$\left.\frac{d^2Q}{dx^2}\right|_i \approx \frac{Q_{i+1} - 2Q_i + Q_{i-1}}{\Delta x^2}$$

Done by finite differencing the discretized values

Implicitly or explicitly involves interpolating data and taking derivative of the interpolant

More accuracy - larger 'stencils'





SciNet compute • calcul CANADA

Diffusion Equation

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

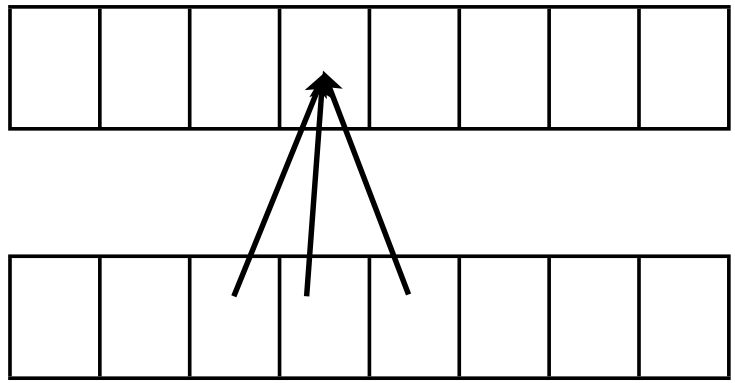
$$\frac{\partial T_i^{(n)}}{\partial t} \approx \frac{T_i^{(n)} + T_i^{(n-1)}}{\Delta t}$$

$$\frac{\partial T_i^{(n)}}{\partial x} \approx \frac{T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}}{\Delta x^2}$$

$$T_i^{(n+1)} \approx T_i^{(n)} + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}\right)$$

Simple 1d PDE

Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]

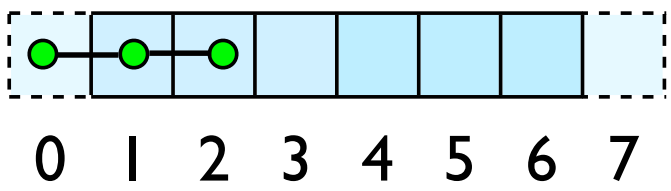


SciNet compute • calcul CANADA

Guardcells

How to deal with boundaries?
Because stencil juts out, need information on cells beyond those you are updating
Pad domain with 'guard cells' so that stencil works even for the first point in domain
Fill guard cells with values such that the required boundary conditions are met

Global Domain



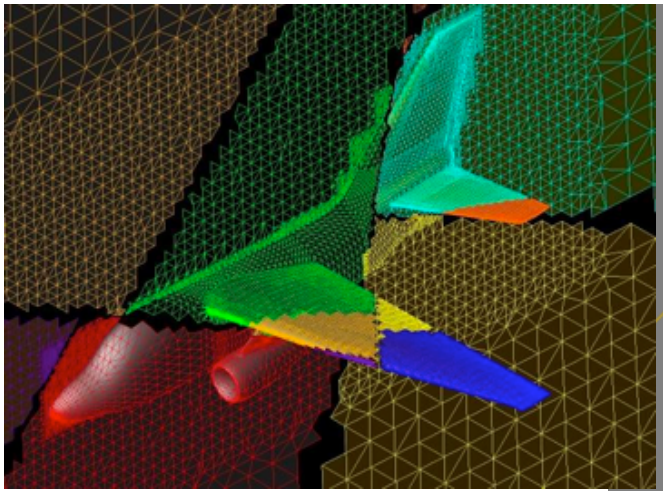
ng = 1
loop from ng, N - 2 ng

SciNet compute • calcul CANADA

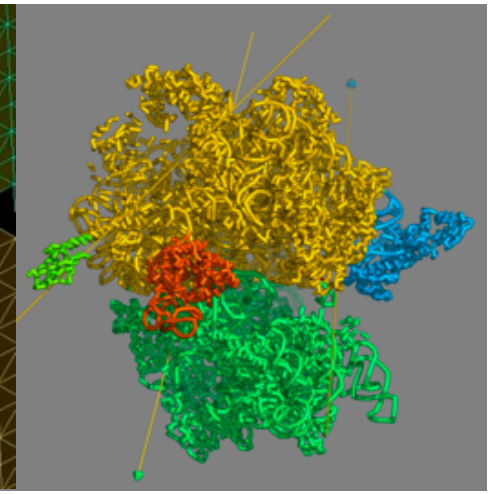
Domain Decomposition

A very common approach to parallelizing on distributed memory computers

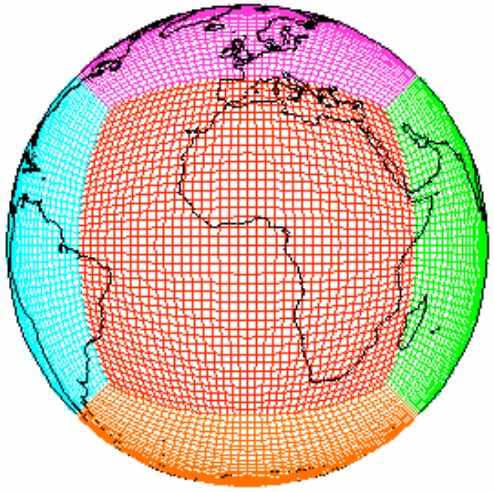
Maintain Locality; need local data mostly, this means only surface data needs to be sent between processes.



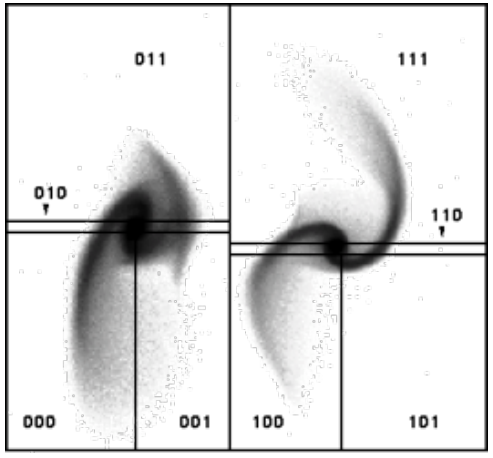
http://adg.stanford.edu/aa241/design/compaero.html



http://www.uea.ac.uk/cmp/research/cmpbio/Protein+Dynamics,+Structure+and+Function



http://sivo.gsfc.nasa.gov/cubedsphere_comp.html



http://www.cita.utoronto.ca/~dubinski/treecode/node8.html

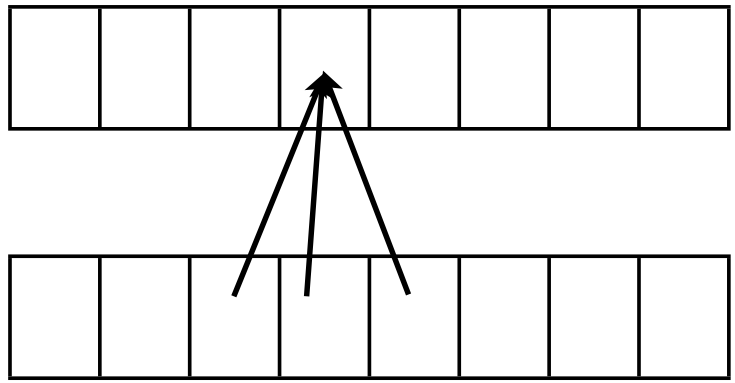
SciNet compute • calcul CANADA

Implement a diffusion equation in MPI

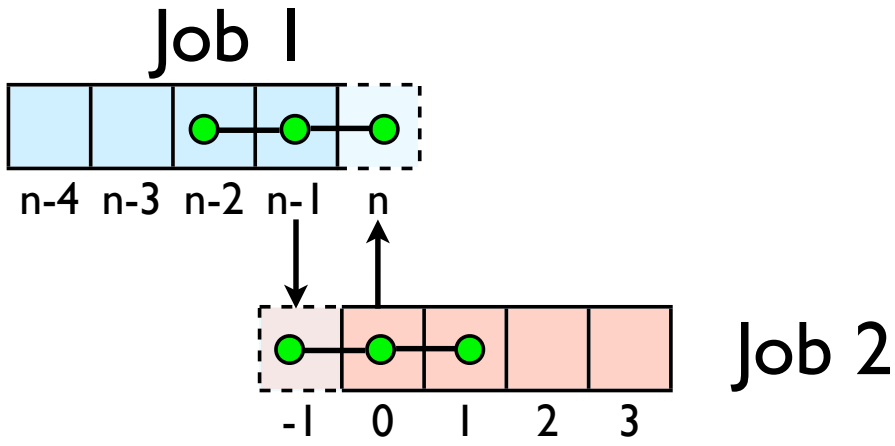
$$\frac{dT}{dt} = D\frac{d^2T}{dx^2}$$

$$T_i^{n+1} = T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)$$

Need one neighboring number per neighbor per timestep



SciNet compute • calcul CANADA



Do computation

guardcell exchange: each cell has to do 2 sendrecvs

its rightmost cell with neighbors leftmost

its leftmost cell with neighbors rightmost

Use even/odd trick from before; if even do rightmost

For simplicity, fixed-temperature BCs; temperature in first, last zones are fixed

SciNet compute • calcul CANADA

Assignment 11

Parallelize 1D Diffusion Equation with MPI

- Start with same serial base code from Assignment 10
  - git clone /scinet/course/sc3/homework3
  - cd homework3
  - source setup
- Add standard MPI calls (MPI_Init, MPI_Comm_Rank, etc)
- Load balance (totalpoints/size on each proc)
- At each step, exchange guardcells(Hint: use Sendrecv)
- Calculate the total error.
- Provide a strong scaling plot (without I/O) using 32 cores (4 GPC nodes).
  - Disscuss the scaling. How many points do you need to use to scale well?
- Submit code, Makefile, git log, scaling plot, and txt file with discussion.

C syntax

```
MPI_Status status;

ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_{size,rank}(Communicator, &{size,rank});
ierr = MPI_Send(sendptr, count, MPI_TYPE, destination,
                tag, Communicator);
ierr = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag,
                Communicator, &status);
ierr = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination,tag,
                    recvptr, count, MPI_TYPE, source, tag,
                    Communicator, &status);
ierr = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE,
                     MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

Examples

MPI Examples

Sample codes used in these two lectures are avaiable here:
git clone /scinet/course/sc3/lc20