Introduction to MPI (II)
How to use MPI to parallelize user applications?

Shangli Ou
LSU HPC
ou@cct.lsu.edu

Outlines

- Principles of building a good parallel code
- Three important issues when parallelizing codes: domain decomposition, ghost zone, I/O
- Example of parallelizing a serial code: 1D diffusion, 3D summation
- Optimization tips of MPI-based parallel applications

Principles of building a good parallel cde

- SPMD(single program multiple data): one single program to allow control to data on multiple processes
- Increase the fraction of your program that can be parallelized.
- Balance the workload of parallel processes.
- Minimize the time spent for communication.
  - Reduce the amount of data being transmitted
  - Reduce the number of times the data is transmitted

Outlines

- Principles of building a good parallel code
- Three important issues when parallelizing codes: domain decomposition, ghost zone, I/O
- Example of parallelizing a serial code: 1D diffusion, 3D summation
- Optimization tips of MPI-based parallel applications

Domain decomposition

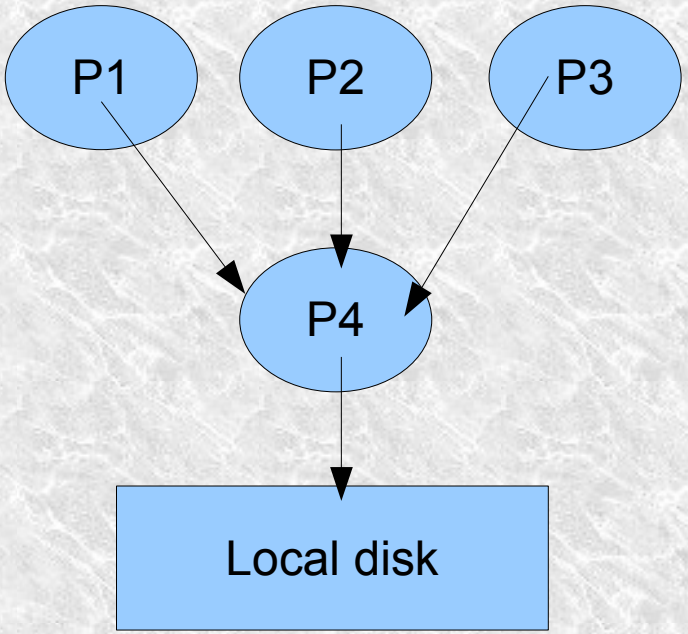
- Domain decomposition is the way how programmers divide computational domain, workload, and data across processes
- Performance depends on geometry, numerical method, load balance, etc.
- Keep in mind that: one process only has access to its own local data, not global data!!!

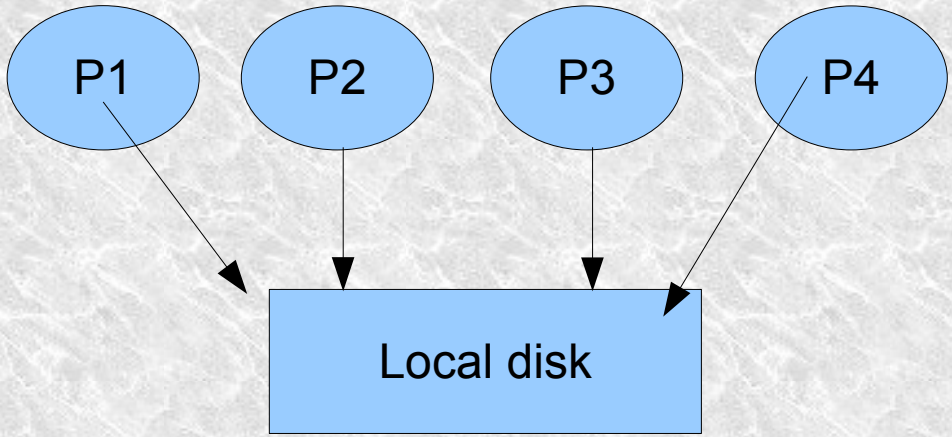
Ghost zones

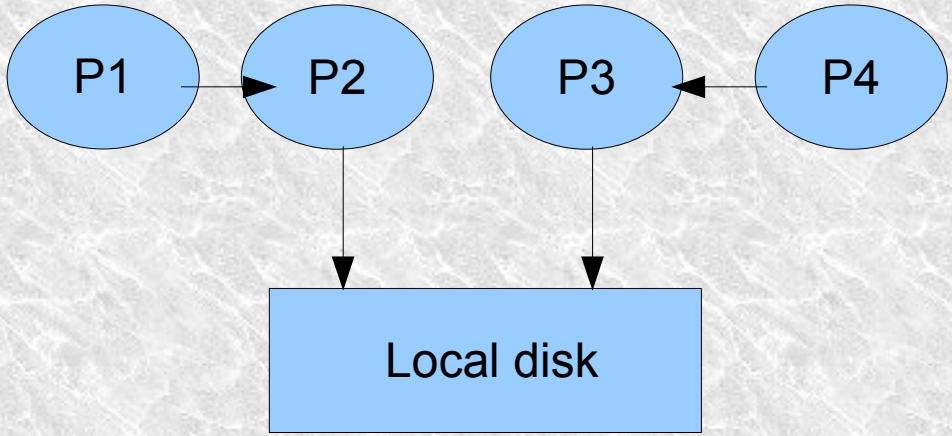
- Two kinds of grids:
  - Effective grids on which computation are performed and data are updated.
  - boundary grids used for computation but data on boundary grids are not updated.
- Ghost zones are boundary grids of one process and reside on effective grids of another process, which means their data are updated on one process but needed for the computation of another process; hence, data on ghost zones need to be communicated very frequently across processes during the computation.

I/O in parallel applications

- Different models of I/O:







Outlines

- Principles of building a good parallel code
- Three important issues when parallelizing codes: domain decomposition, ghost zone, I/O
- Example of parallelizing a serial code: 1D diffusion, 3D summation
- Optimization tips of MPI-based parallel applications

Case study 1: 1D diffusion equation

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial^2 x}$$

dx

Computational grids

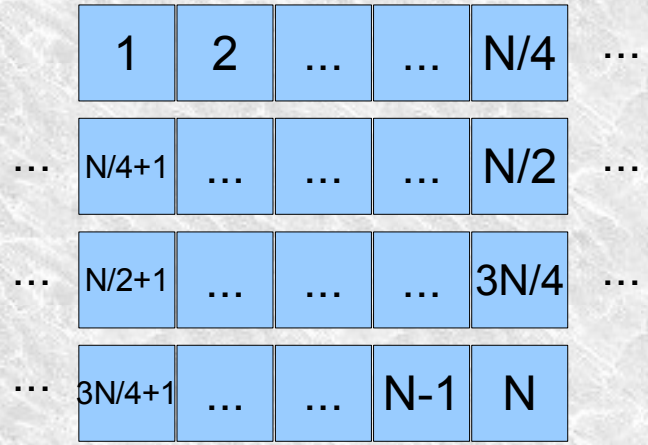
$$\frac{U_i^{new} - U_i^{old}}{h} = \frac{(U_{i+1}^{old} + U_{i-1}^{old} - 2U_i^{old})}{dx^2}$$

Forward finite differencing for grid i

```
Parameter, integer :: N=256
Double :: u(N), u2(N)
open(10, file='data.in')
read(10,*) u       ! setting up data arrays
...   ! time evolution loop on h
Do i = 2, N-1
   u2(i)= u(i) + h/ (dx*dx)* (u(i+1)+u(i-1)-2u(i))
enddo
```

Serial code, both u and u2 are in the entire memory of one single processor, hence, loop can go from 2 to N-1, but not for 1 and N because of using three point stencils.

First trial of parallelization: each process has copies of all data but divide workload to four processes



Workload for process 0

Workload for process 1

Workload for process 2

Workload for process 3

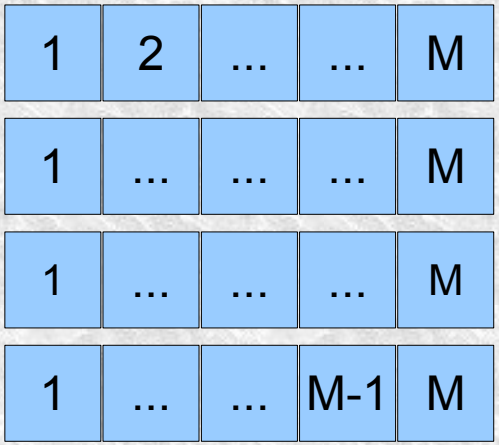
```
Parameter, integer :: N=256
Double u(N), u2(N)
call mpi_init(ierr)
call mpi_size(MPI_COMM_WORLD,nproc,ierr)
call mpi_rank(MPI_COMM_WORLD, myidi,ierr)
...   ! setting up data arrays
workload = N / nproc
istart = myid*workload + 1
iend = myid*workload
... ! time evolution loop on h
Do i = istart, iend
  u2(i)= u(i) + h/ (dx*dx)* (u(i+1)+u(i-1)-2u(i))
enddo
call MPI_finalize(ierr)
```

parallel code: both u and u2 are in the entire memory of one single processor, loops are divided into four chunks, each executed by one process.

What is the problem with this code?
What is the limitation?

```
if (myid.eq.0) istart = 2
if (myid.eq.3) iend=N-1
```

Second step of parallelization: divide data onto four processes



data for process 0

data for process 1

data for process 2

data for process 3

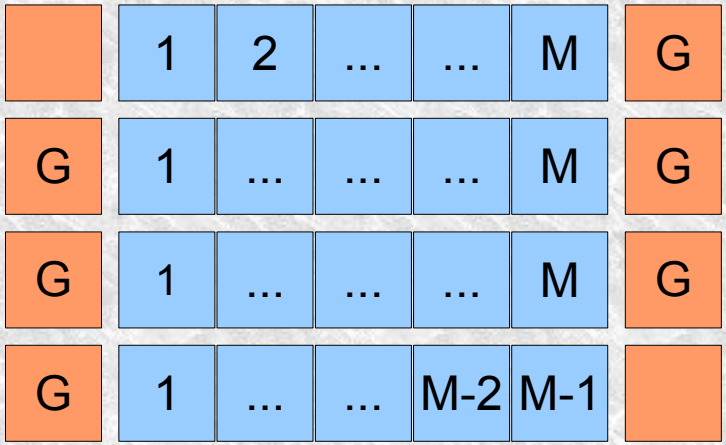
```
Parameter, integer :: M=64, N=256
Double u(M), u2(M)
...   ! setting up data arrays
istart = 1
if (myid.eq.0) istart = 2
iend = M
if (myid.eq.3) iend=M-1
... ! time evolution loop on h
Do i = istart, iend
   u2(i)= u(i) + h/ (dx*dx)* (u(i+1)+u(i-1)-2u(i))
enddo
...
```

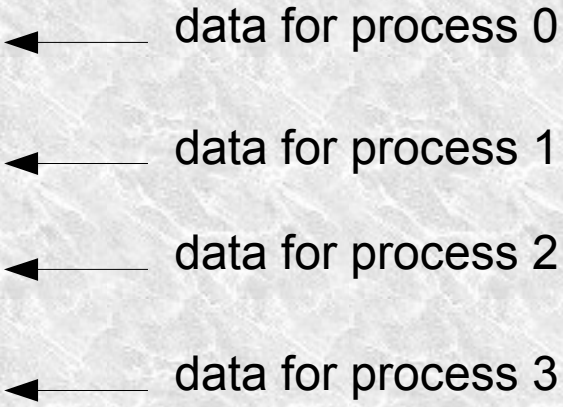
parallel code: u and u2 are divided into four chunks and distributed to each process, note the change of array index.

What else is missing?

Needs ghost zones!

Third step of parallelization: taking care of ghost zones

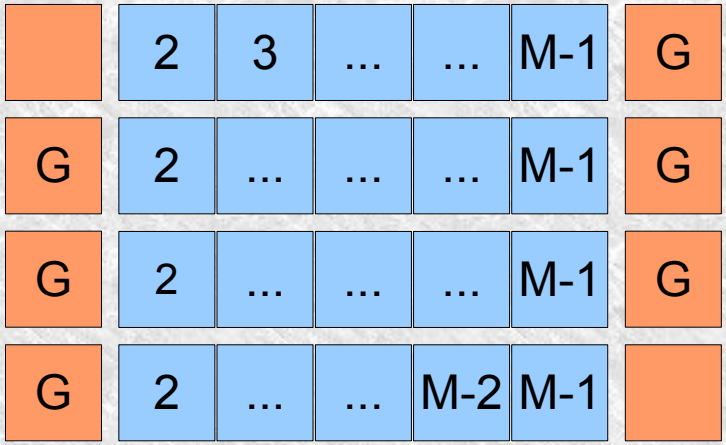


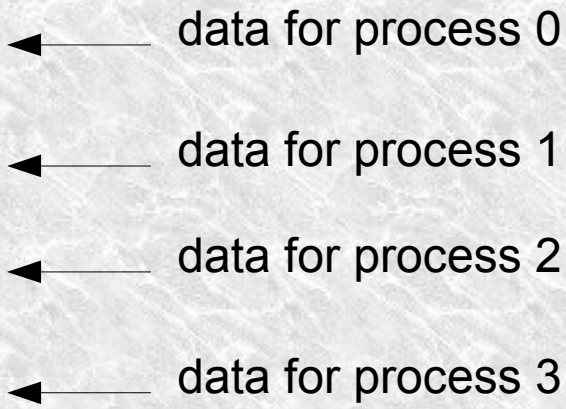



```
Parameter, integer :: K=66, N=256
Double u(K), u2(K)
...   ! setting up data arrays
istart = 2
if (myid.eq.0) istart = 3
iend = K -1
if (myid.eq.3) iend=K-2
... ! time evolution loop on h
call send_ghostzones
Do i = istart, iend
   u2(i)= u(i) + h/ (dx*dx)* (u(i+1)+u(i-1)-2u(i))
enddo
...
```

```
Subroutine send_ghostzones
if (myid.eq.0) then
  sendbuf=u(K-1)
  call mpi_send(sendbuf,1,mpi_double, 1...)
  call mpi_receive(recvbuf,1,mpi_double,1,...)
  u(M)=recvbuf
else if (myid.eq.1) then
   call mpi_receive(recvbuf,1,mpi_double, 0, ...)
   u(1) = recvbuf
   sendbuf=u(2)
   call mpi_send(sendbuf,1,mpi_double, 0...)
...
```

Fourth step: Taking care of I/O






```
Parameter, integer :: M=66, N=256
Double u(M), u2(M)
if (myid.eq.0) then
  open(2,file='indata.0',..)
  read(2) u
else if (myid.eq.1) then
  open(2,file='indata.1',..)
  read(2) u
...
```

```
Double u(M), u2(M), chunk(M)
if (myid.eq.0) then
  open(2,file='indata')
  read(2) u  !read own data
  read(2) chunk ! chunk for PE 1
  call mpi_send(chunk,..., 1...)
  read(2) chunk ! Chunk for PE 2
  call mpi_send(chunk,..., 2...)
  ...
else if (myid.eq.1) then
   call mpi_receive(u,..., 0, ...)
...
```

One process is in charge of reading all data and send them to other processes

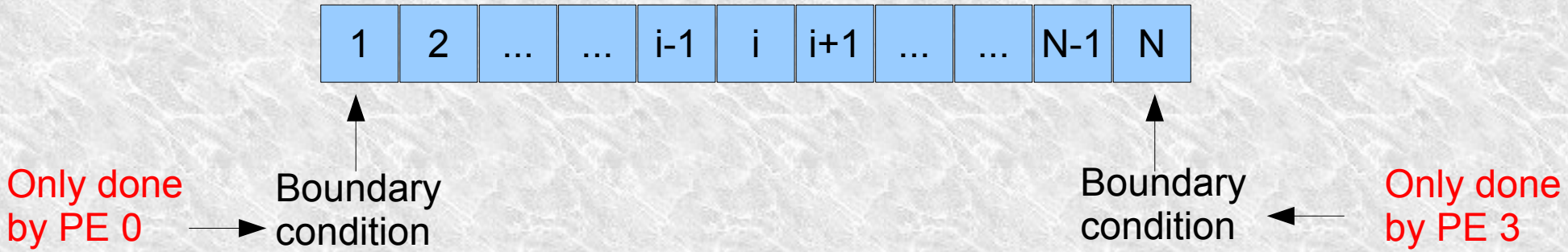
Every process read in data from its own local data file

Further considerations

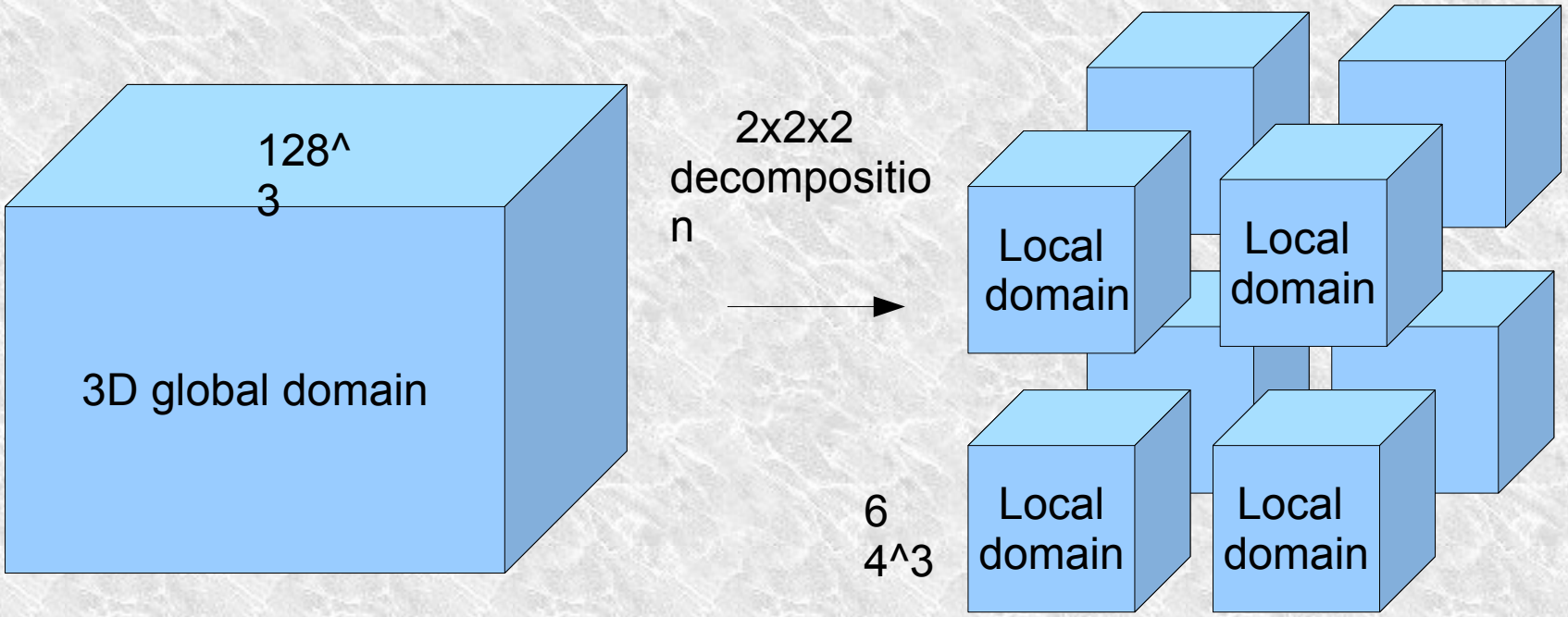
- Pop Quiz:
- What is special when computing the limiting minimum time step of the Courant condition?
  - Which mpi function needs to be used?

Call mpi_reduce(hlocal,hglobal,1,MPI_REAL,mpi_min,0,MPI_COMM_WORLD,ierr)

- What needs to be taken care when implementing boundary condition?



Case Study 2: 3D arrays




```
Double, dimension(128,128,128) :: density
Mass = 0.0
Do k = 1,128
Do j = 1,128
Do i = 1,128
   mass =  mass+density(i,j,k)*dv
enddo
enddo
enddo
```

```
Double, dimension(64,64,64) :: density
...  temp = 0.0
Do k,j,i = 1,64
   temp =  temp+density(i,j,k)*dv   ! local sum
enddo
! get the global sum
call mpi_reduce(temp,mass,1,MPI_REAL,
        mpi_sum,0, MPI_COMM_WORLD,ierr)
```

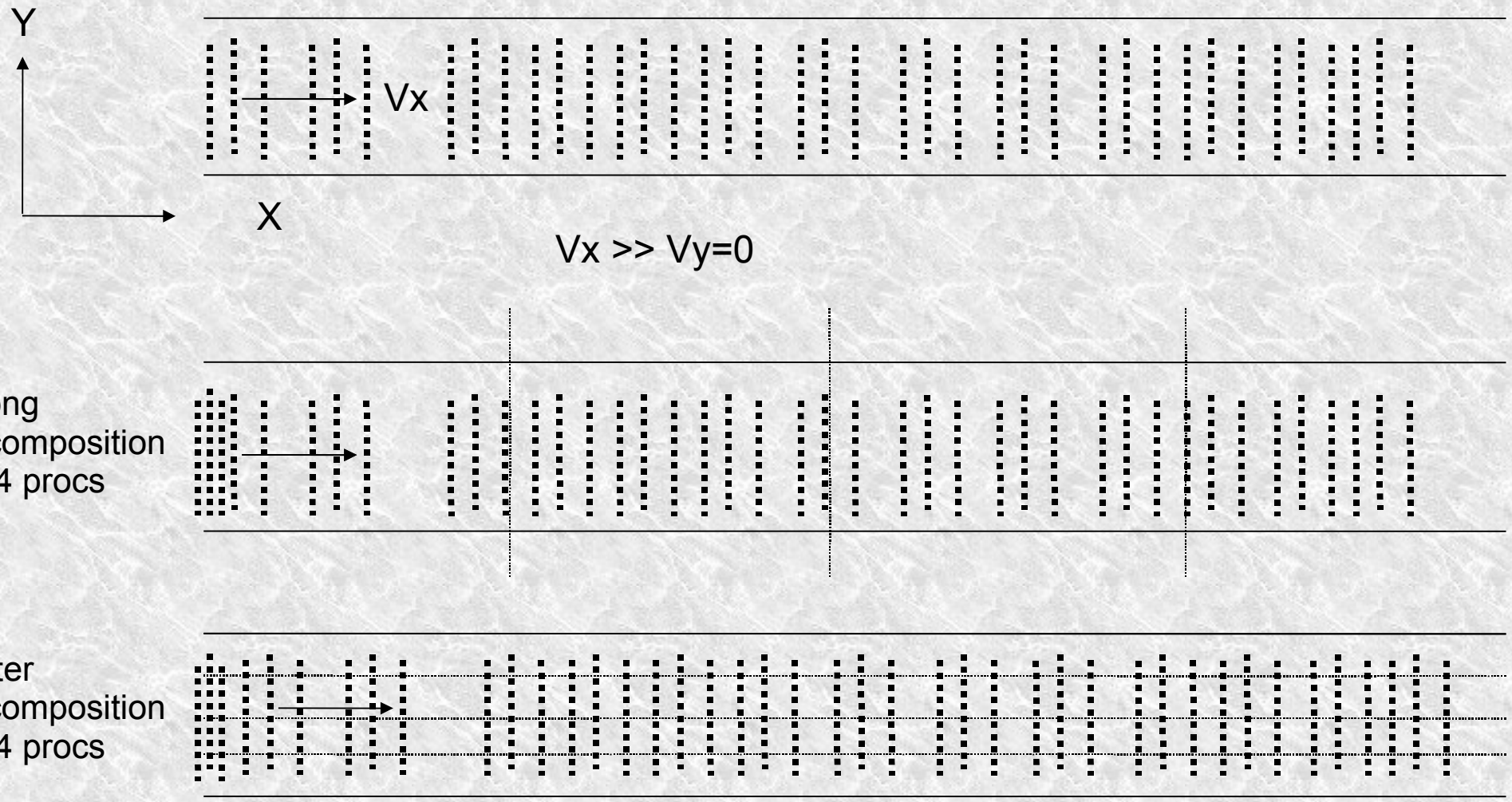
Outlines

- Principles of building a good parallel code
- Three important issues when parallelizing codes: domain decomposition, ghost zone, I/O
- Example of parallelizing a serial code: 1D diffusion, 3D summation
- Optimization tips of MPI-based parallel applications

Optimization tips of MPI-based parallel applications

Principle: Analyse your algorithm and domain decomposition, implement the most optimal way to reduce communication as possible as you can, because communication is more expensive than computation.

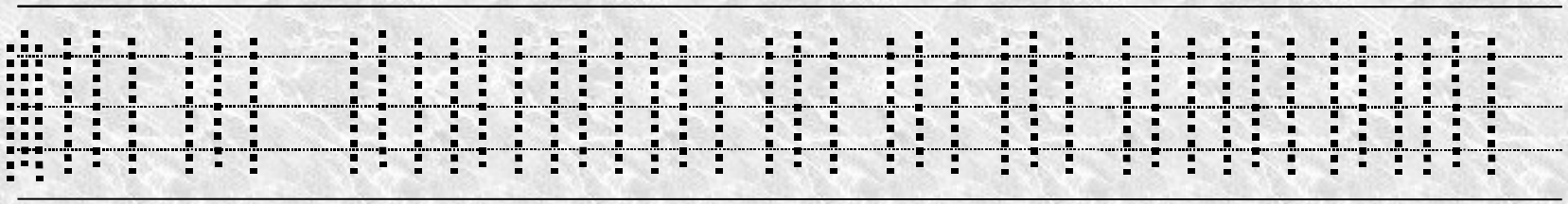
Example of optimizing domain decomposition



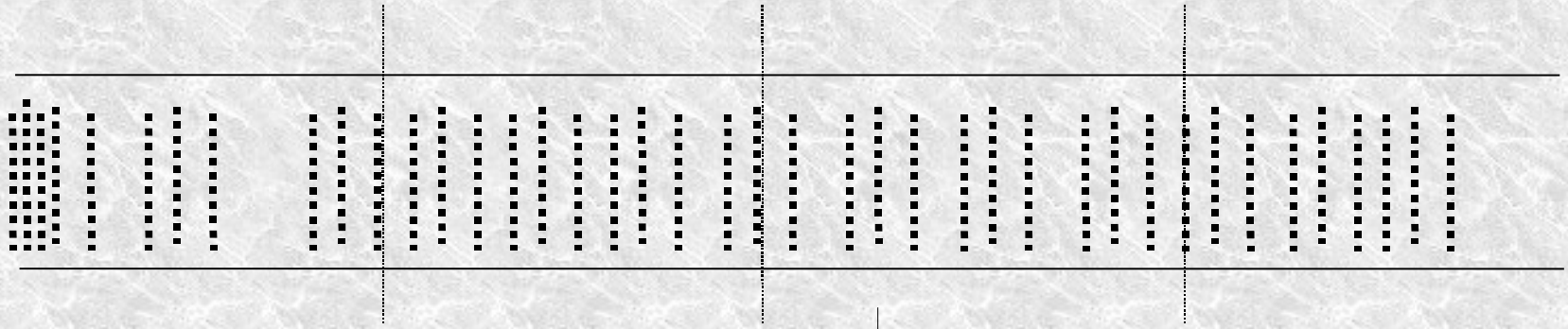
Example of optimizing domain decomposition

Random distribution of Vx Vy

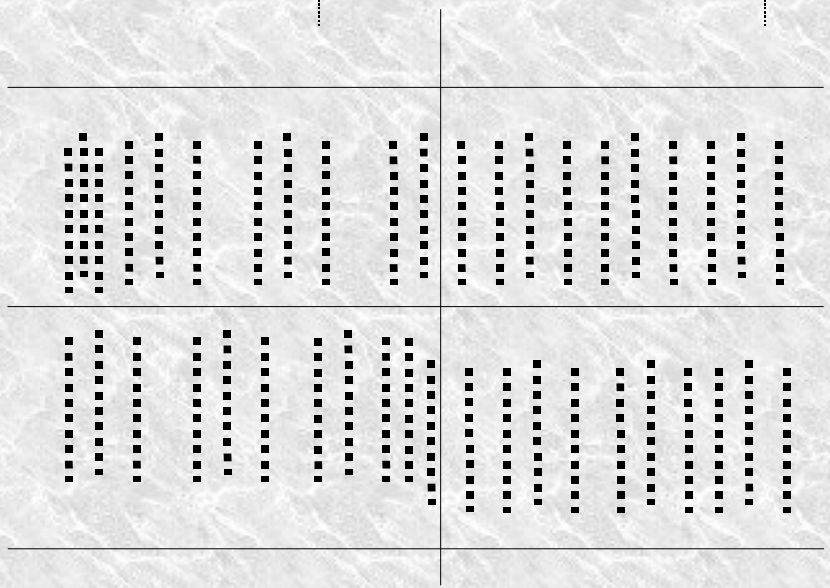
bad decomposition on 4 procs



right decomposition on 4 procs

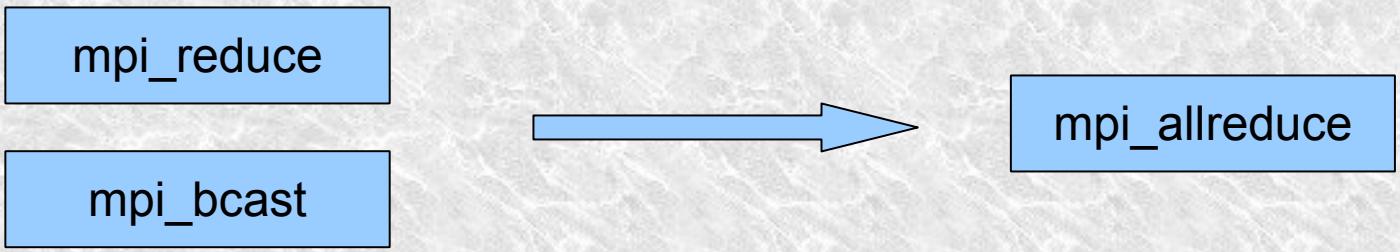


right decomposition if the domain is more square-like, the key point is to reduce the area of interface between processes to reduce possible communication



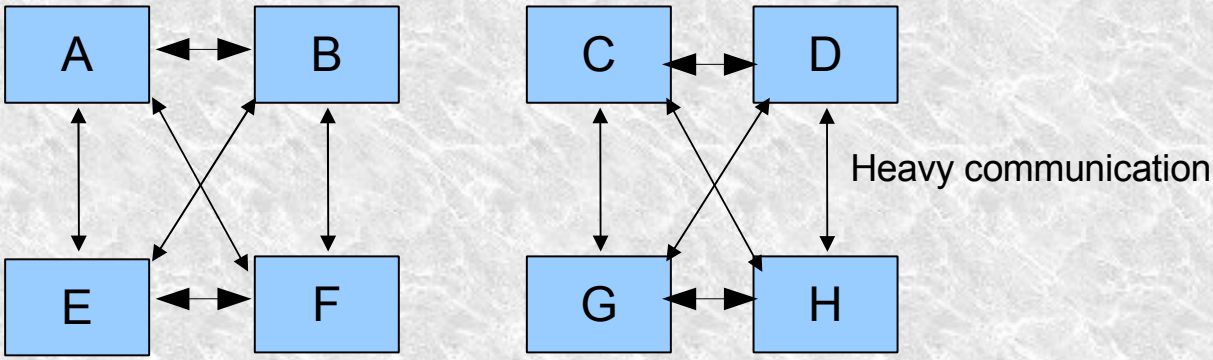
Avoid excessive MPI calls to reduce overhead.

- An MPI call involves certain system overhead, i.e., encapsulating data, copying data to system buffer, waiting for handles, etc.
- To reduce the overhead, merge multiple MPI calls into a single MPI call as possible as you can.
- Use collective mpi calls as much as you can (opposite to point to point communication, better for homogeneous environments).

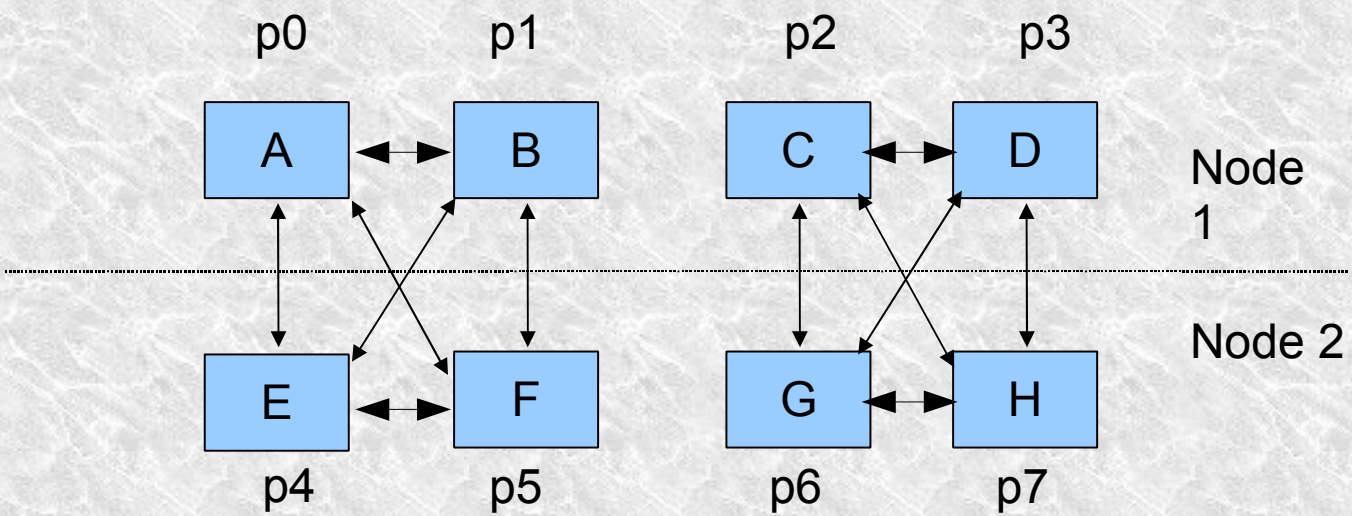


Using Intranode Communication on multi-core clusters

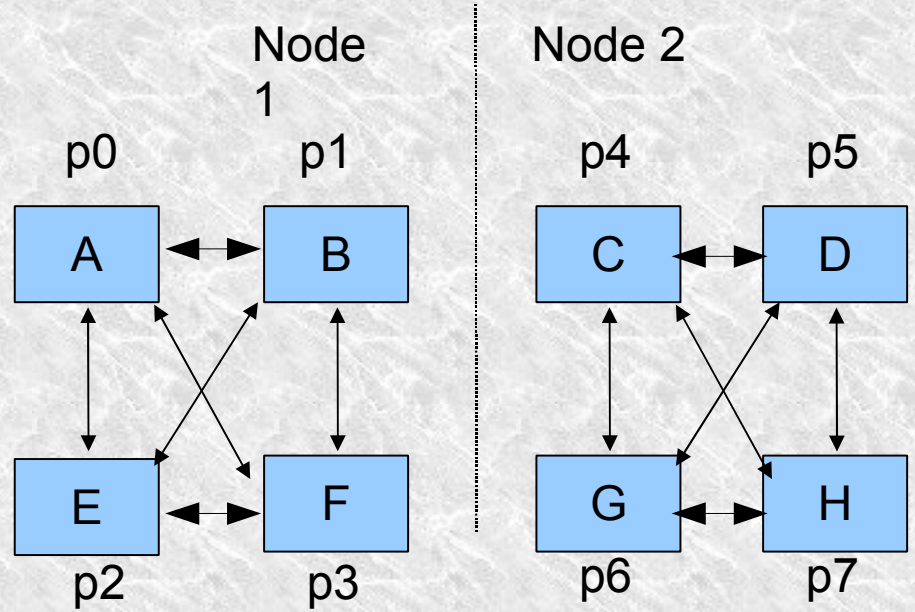
Locate communication-intensive MPI processes and put them on the same node, because communication within the same node is much faster.



Wrong way to distribute processes: lot of vertical communication has to go cross nodes.



Right way to distribute processes: intensive vertical communication is limited to intranode.



The End