

Introduction to Message Passing Interface

Jongsoo Kim, jskim@kasi.re.kr
Korea Astronomy & Space Science Institute

Number 1 Supercomputer



- 122.3 petaflops, 4356 nodes
- Per node: Two 22-core Power 9 CPUs + six NVIDIA Tesla V100 GPUs

Contents

- Parallel Computational Model
- OpenMP, CUDA, MPI
- Speedup, Amdahl's Law
- Examples of Simple MPI Program
 - Hello world
 - π
 - Matrix-Vector Multiplication
 - Matrix-Matrix Multiplication Laplace equation
- Arithmetic Intensity

Contents

- Examples of Intermediate MPI
 - Laplace equation
- What's New in MPI-2?
 - Parallel I/O
 - Remote Memory Operations
 - Dynamic Process Management
- Parallel isothermal (M)HD Code
- Performance Benchmarks

Parallel Computational Models

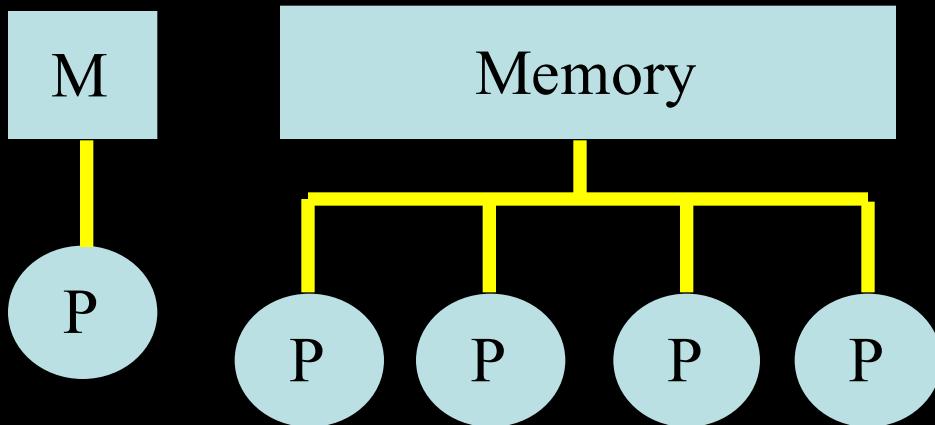
PC

SMP

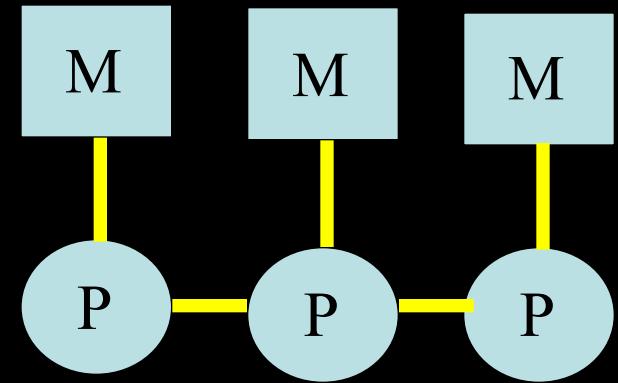
MPP

Symmetric Multiprocessors

Massively Parallel Processors



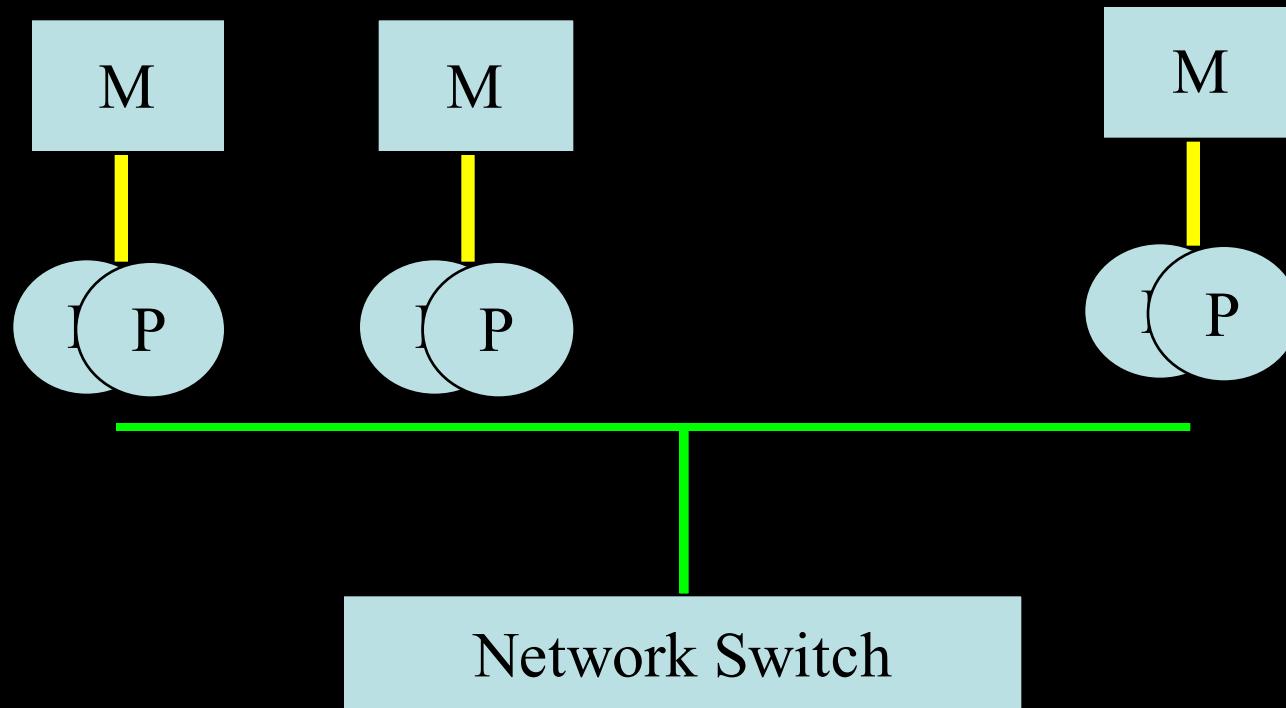
Most multi-P servers



Cray T3E

Parallel Computational Models

(Beowulf) clusters, supercomputers, etc.



Ethernet, Omni-Path, (Myrinet), Infiniband

Serial Programming

- Languages
 - Fortran, C, C++, python, ...
- Optimization of Serial Codes
 - Compilers
 - Optimization flags
 - Libraries, e.g., math kernel,

Parallel Programming

- OpenMP
 - SMP, incremental parallelization
- CUDA (openCL)
 - Coprocessor, incremental parallelization
- MPI
 - Cluster, needs sometimes many changes of serial codes

Hybrid Programming

- OpenMP (MPI) + CUDA
 - SMP, multiple GPUs
- MPI + OpenMP
 - Cluster of SMP nodes
- MPI + CUDA
 - Cluster, multiple GPUs
- MPI + OpenMP + CUDA

Speedup & Amdahl's Law

- Speedup

$$S_p = \frac{T_1}{T_p}$$

- Linear (ideal) Speedup $S_p = p$
- Amdahl's Law
 - f: fraction of serial operations
 - p: number of processors

$$S_{p,\max} = \frac{1}{f + \frac{1-f}{p}}$$

MPI Forums

- MPI-1
 - Point-to-point communication, collective communication, datatypes, virtual process topologies, error codes and classes, bindings for both C and Fortran 77 (MPICH implementation)
 - MPI-1.0 May 1994
 - MPI-1.1 (minor modification) Jun 1995;
 - MPI-1.2 July 1997
 - MPI-1.3: (final end of MPI-1 series) May 2008
- MPI-2
 - Parallel file I/O, one-sided (put/get) communication, dynamic process management, multithreaded MPI communications, bindings for Fortran 90 and C++
 - MPI-2.0 1997
 - MPI-2.1 Sep., 2008
 - MPI-2.2 Sep., 2009
- MPI-3
 - Nonblocking Collective operations, and others...
 - MPI-3.0 September 2012
 - MPI-3.1 June 2015

MPI References

Gropp, W., Lusk, E., & Skjellum, A. 2014, Using MPI, 3rd (Cambridge: MIT)

Gropp, W., Hoefler, T., Thakur, R, & Lusk, E. 2014, Using Advanced MPI (Cambridge, MIT)



MPI; Message Passing Interface

- What is MPI?
 - Communication library for Fortran, C, and C++
- (Six) Nine basic routines
 - MPI_INIT
 - MPI_FINALIZE
 - MPI_COMM_SIZE
 - MPI_COMM_RANK
 - MPI_SEND
 - MPI_RECV
 - MPI_BCAST
 - MPI_REDUCE
 - MPI_BARRIER
- One timing routine
 - MPI_WTIME

Termial emulator for Windows

- PuTTY

<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

Chalawan

- 1 master + 16 nodes (28 cores) + 7 nodes (16 cores)
 - CPU: Intel Xeon E5-2697 v3 (2.6GHz)
 - Memory: 4GB / core
 - INTERCONNECT: 56 Gbps FDR Infiniband
 - OS : CentOS 6.5
 - Scheduler : SGE
 - openMPI, mpich-2
- 90TB lustre file system with FDR Infiniband

Access to Chalawan

- Wireless Network
- From outside of the NARIT
 - ssh -p 22240 -Y guest@stargate.narit.space
- From inside of the NARIT
 - ssh guest -Y 192.168.5.100
 - ssh compute-0-[0-18]
 - mkdir (your usual userid)
- cp -r /home/guest/jskim/mpi/* .

Hello World (Fortran or C, serial)

```
program main          #include <stdio.h>
  print*, 'hello world' int main(int argc, char **argv)
end                      {
                           printf("Hello, World!\n");
                           return 0;
                         }

gfortran hello_serial.f gcc hello_serial.c
./a.out                  ./a.out
```

Hello World (Fortran+mpi)

```
program main
use mpi
integer ierr
call MPI_INIT(ierr)
print*, 'hello world'
call MPI_FINALIZE(ierr)
end
```

```
-----  
mpif90 hello.f90  
mpiexec -n 2 ./a.out
```

Hello World (C+MPI)

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    printf("Hello World!\n");
    MPI_Finalize();
    return 0;
}

mpicc hello.c
mpiexec -n 2 ./a.out
```

Rank and Size (fortran)

```
program main
use mpi

integer ierr, rank, size

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
print*, "I am rank", rank, " out of ", size, "processors"
call MPI_FINALIZE(ierr)
end

mpif90 ranksize.f90
mpiexec -n 2 ./a.out
```

Rank and Size (c version)

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("I am rank %d out of %d processors\n", rank, size
);
    MPI_Finalize();
    return 0;
}
mpicc ranksize.c
mpiexec -n 4 ./a.out
```

Processor name

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Get the name of the processor
    char name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(name, &name_len);
    printf("My name is %s, rank %d out of %d processors\n"
    ,name,rank,size);
    MPI_Finalize();
    return 0;
}
mpicc processor_name.c
mpiexec -host compute-0-0,compute-0-1 -n 4 ./a.out
```

Processor name

```
program main
```

```
    use mpi
```

```
    integer ierr, rank, size  
    character*(MPI_MAX_PROCESSOR_NAME) name  
    integer name_len
```

```
    call MPI_INIT(ierr)  
    call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)  
    call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
```

```
    call MPI_Get_processor_name(name,name_len,ierr)  
    print*, "My name is", name,"rank",rank,"out of ",size,"processors"
```

```
    call MPI_FINALIZE(ierr)
```

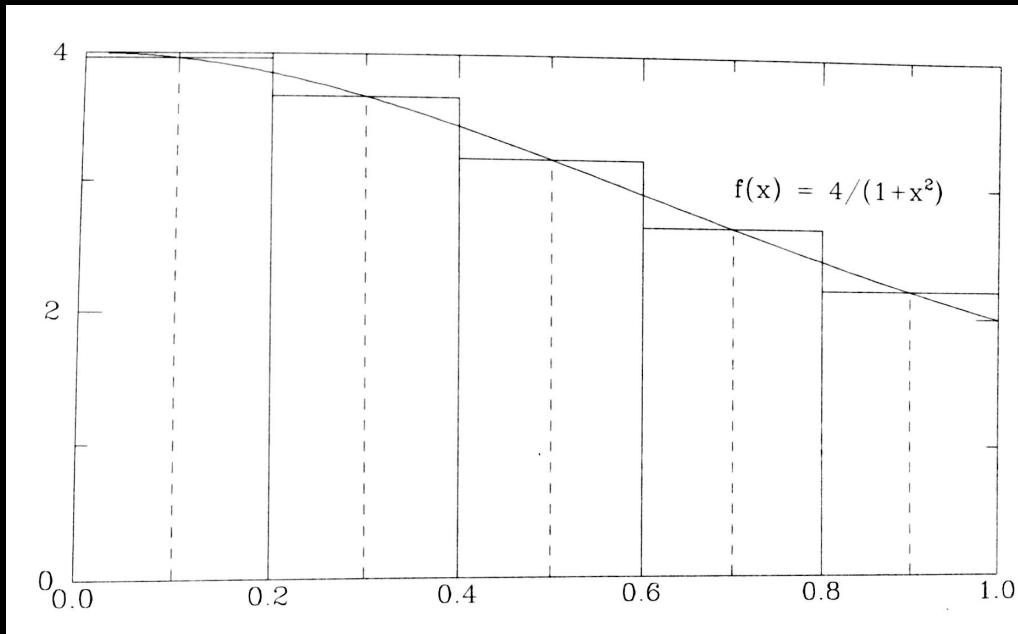
```
end
```

```
mpif90 processor_name.f90
```

```
mpiexec -host gpu01,gpu02 -n 4 ./a.out
```

π

$$\int_0^1 \frac{4}{1+x} dx = 4 \arctan(x) \Big|_0^1 = \pi$$



- program
input: nx
output: sum

π (f90, serial)

```
program main
double precision PI25DT
parameter          (PI25DT = 3.141592653589793238462643d0)
double precision pi, h, sum, x, f, a
integer n, i
!
!                                     function to integrate
f(a) = 4.d0 / (1.d0 + a*a)
do
    print *, 'Enter the number of intervals: (0 quits) '
    read(*,*) n
!
!                                     check for quit signal
    if (n .le. 0) exit
!
!                                     calculate the interval size
    h = 1.0d0/n
    sum = 0.0d0
    do i = 1, n
        x = h * (dbl(i) - 0.5d0)
        sum = sum + f(x)
    enddo
    pi = h * sum
    print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT)
enddo
end
```

```
program main
use mpi
double precision PI25DT
parameter      (PI25DT = 3.141592653589793238462643d0)
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
!          function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

do
.....
enddo

call MPI_FINALIZE(ierr)
end
```

```

do
  if (myid .eq. 0) then
    print *, 'Enter the number of intervals: (0 quits)'
    read(*,*) n
  endif
!
!           broadcast n
  call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
!
!           check for quit signal
  if (n .le. 0) exit
!
!           calculate the interval size
  h = 1.0d0/n
  sum = 0.0d0
  do i = myid+1, numprocs
    x = h * (dbe(i) - 0.5d0)
    sum = sum + f(x)
  enddo
  mypi = h * sum
!
!           collect all the partial sums
  call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION, &
                  MPI_SUM, 0, MPI_COMM_WORLD, ierr)
!
!           node 0 prints the answer.
  if (myid .eq. 0) then
    print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT)
  endif
enddo

```

π (c, serial)

```
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int n, i;
    double PI25DT = 3.141592653589793238462643;
    double pi, h, sum, x;
    while (1) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
        if (n == 0)
            break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = 1; i <= n; i++) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            pi = h * sum;
            printf("pi is approximately %.16f, Error is %.16f\n",
                   pi, fabs(pi - PI25DT));
        }
    }
    return 0;
}
```

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    while (1) {
    }
    MPI_Finalize();
    return 0;
}
```

```
while (1) {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0)
        break;
    else {
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                   MPI_COMM_WORLD);
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n",
                   pi, fabs(pi - PI25DT));
    }
}
```

```
program main

include 'mpif.h'
double precision f, a, dx, x, sum, pip, pi
integer nx, nxp, ix
integer ierr, np, irank

f(a) = 4.d0 / (1.d0 + a*a)

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,irank,ierr)

if (irank .eq. 0) then
    print*, 'number of intervals:'
    read(*,*) nx
endif

call MPI_BCAST(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

dx = 1.0d0 / dfloat(nx)
nxp = nx / np
```

```
sum = 0.0d0
do 10 ix = 1, nxp
    x = dx*(dfloat(irank*nxp+ix)-0.5d0)
    sum = sum + f(x)
10 continue
pip = dx*sum

call
MPI_REDUCE(pip,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0
,
$      MPI_COMM_WORLD,ierr)

if (irank .eq. 0) then
    print*, pi
endif

call MPI_FINALIZE(ierr)
stop
end
```

```
program main
```

```
    integer MAX_ROWS, MAX_COLS, rows, cols
```

```
    parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
```

```
    double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_ROWS)
```

```
    integer i, j
```

```
    rows = 100
```

```
    cols = 100
```

```
    do j = 1,cols
```

```
        b(j) = 1.0
```

```
        do i = 1,rows
```

```
            a(i,j) = i
```

```
        enddo
```

```
    enddo
```

```
    do i=1,rows
```

```
        c(i) = 0.0
```

```
        do j=1,cols
```

```
            c(i) = c(i) + a(i,j)*b(j)
```

```
        enddo
```

```
        print*, i, c(i)
```

```
    enddo
```

```
end
```

Matrix-Vector Multiplication

- $c = A b$
- This example shows the use of the MPI send and receive
- Manager-workers (self-scheduling) algorithm
 - worker processes do not have to communicate with each other
 - The amount of work that each slave must perform is difficult to predict

Send and Receive

$$A \begin{bmatrix} \text{Process 0} \\ \hline \text{Process 1} \\ \hline \text{Process 2} \\ \hline \text{Process 3} \end{bmatrix} * b = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$$

- Manager
 - Send: b and each row of A
 - Receive: each component of c
- Workers
 - Receive: b and each row of A
 - Send: each component of c

Serial code

program main

```
integer MAX_ROWS, MAX_COLS, rows, cols
parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_ROWS)
integer i, j
rows = 100
cols = 100
```

```
do j = 1,cols
    b(j) = 1.0
    do i = 1,rows
        a(i,j) = i
    enddo
enddo
```

```
do i=1,rows
    c(i) = 0.0
    do j=1,cols
        c(i) = c(i) + a(i,j)*b(j)
    enddo
    print*, i, c(i)
enddo
end
```

Code for all processes

```
program main
use mpi
integer MAX_ROWS,MAX_COLS,rows,cols
parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS)
double precision c(MAX_ROWS), buffer(MAX_COLS), ans

integer myid, manager, numprocs, ierr, status(MPI_STATUS_SIZE)
integer i, j, numsent, sender
integer anstype, row

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
manager = 0
rows  = 100
cols  = 100
```

Code for all processes

```
if ( myid .eq. manager ) then
    master initializes and dispatches
    ...
else
    workers receive b, and compute dot
product
    ...
endif

call MPI_FINALIZE(ierr)
end
```

code for master

```
! manager initializes and then dispatches
! initialize a and b (arbitrary)
do j = 1,cols
    b(j)= 1
    do i = 1,rows
        a(i,j)= i
    enddo
enddo
numsent=0
! send b to each worker process
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, manager, &
               MPI_COMM_WORLD, ierr)
! send a row to each worker process; tag with row number
do i = 1,min(numprocs-1,rows)
    do j = 1,cols
        buffer(j)= a(i,j)
    enddo
    call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i, &
                  i, MPI_COMM_WORLD, ierr)
    numsent=numsent+1
enddo
```

code for master

```
do i = 1,rows
    call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, &
                  MPI_ANY_SOURCE, MPI_ANY_TAG, &
                  MPI_COMM_WORLD, status, ierr)
    sender = status(MPI_SOURCE)
    anstype = status(MPI_TAG) ! row is tag value
    c(anstype)=ans
    if (numsent .lt. rows) then ! send another row
        do j = 1,cols
            buffer(j) = a(numsent+1,j)
        enddo
        call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, &
                      sender, numsent+1, MPI_COMM_WORLD, ierr)
        numsent = numsent+1
    else ! Tell sender that there is no more work
        call MPI_SEND(MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION, &
                      sender, 0, MPI_COMM_WORLD, ierr)
    endif
enddo
! print out the answer
do i = 1,cols
    print *, "c(", i, ")= ", c(i)
enddo
```

code for slave

```
else
!    workers receive b, then compute dot products until
!    done message received
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, manager, &
               MPI_COMM_WORLD, ierr)
if (myid .le. rows) then
    ! skip if more processes than work
    do
        call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, &
                      manager, MPI_ANY_TAG, MPI_COMM_WORLD, &
                      status, ierr)
        if (status(MPI_TAG) .eq. 0) exit
        row = status(MPI_TAG)
        ans = 0.0
        do i = 1,cols
            ans = ans+buffer(i)*b(i)
        enddo
        call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, manager, &
                      row, MPI_COMM_WORLD, ierr)
    enddo
endif
endif
```

Matrix-Matrix Multiplication

- $C = A \cdot B$
- The algorithm for MM is the same as Mv.
- Manager-workers (self-scheduling) algorithm
 - worker processes do not have to communicate with each other
 - The amount of work that each slave must perform is difficult to predict

Code for all processes

program main

```
include "mpif.h"
```

```
integer MAX_AROWS, MAX_ACOLS, MAX_BCOLS  
parameter (MAX_AROWS=20, MAX_ACOLS=1000, MAX_BCOLS=20)  
double precision a(MAX_AROWS,MAX_ACOLS), b(MAX_ACOLS,MAX_BCOLS)  
double precision c(MAX_AROWS,MAX_BCOLS)  
double precision buffer(MAX_ACOLS), ans(MAX_ACOLS)
```

```
integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)  
integer i, j, numsent, sender  
integer anstype, row, arows, acols, brows, bcols, crows, ccols
```

```
call MPI_INIT( ierr )  
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )  
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )  
print *, "Process ", myid, " of ", numprocs, " is alive"
```

```
master=0  
arows = 10  
acols = 100  
brows = 100  
bcols = 10  
crows = arows  
ccols = bcols
```

Code for all processes

```
if ( myid .eq. manager ) then
    master initializes and dispatches
    ...
else
    workers receive b, and compute dot
product
    ...
endif

call MPI_FINALIZE(ierr)
end
```

code for master

```
if( myid .eq. master ) then
c      master initializes and then dispatches
c      initialize a and b
do i=1,acols
  do j = 1,arows
    a(j,i)=i
  enddo
enddo
do i=1,bcols
  do j = 1,brows
    b(j,i)=i
  enddo
enddo

numsent=0

c      send b to each other process
do i=1,bcols
  call MPI_BCAST(b(1,i), brows, MPI_DOUBLE_PRECISION, master,
$      MPI_COMM_WORLD, ierr)
enddo

c      send a row of a to each other process; tag with row number
do i=1,numprocs-1
  do j = 1,acols
    buffer(j)=a(i,j)
  enddo
  call MPI_SEND(buffer, acols, MPI_DOUBLE_PRECISION,i,
$      i, MPI_COMM_WORLD, ierr)
  numsent=numsent+1
enddo
```

code for master

```
if( myid .eq. master ) then
c      master initializes and then dispatches
c      initialize a and b
do i=1,acols
  do j = 1,arows
    a(j,i)=i
  enddo
enddo
do i=1,bcols
  do j = 1,brows
    b(j,i)=i
  enddo
enddo

numsent=0

c      send b to each other process
do i=1,bcols
  call MPI_BCAST(b(1,i), brows, MPI_DOUBLE_PRECISION, master,
$      MPI_COMM_WORLD, ierr)
enddo

c      send a row of a to each other process; tag with row number
do i=1,numprocs-1
  do j = 1,acols
    buffer(j)=a(i,j)
  enddo
  call MPI_SEND(buffer, acols, MPI_DOUBLE_PRECISION,i,
$      i, MPI_COMM_WORLD, ierr)
  numsent=numsent+1
enddo
```

code for slave

```
c    slaves receive b, then compute rows of c until done message
do i = 1,bcols
    call MPI_BCAST(b(1,i), brows, MPI_DOUBLE_PRECISION, master,
$        MPI_COMM_WORLD, ierr)
enddo
do
    call MPI_RECV(buffer, acols, MPI_DOUBLE_PRECISION, master,
$        MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
    if (status(MPI_TAG) .eq. 0) exit
    row = status(MPI_TAG)
    do i = 1,bcols
        ans(i) = 0.0
        do j = 1,acols
            ans(i) = ans(i) + buffer(j)*b(j,i)
        enddo
    enddo
    call MPI_SEND(ans, bcols, MPI_DOUBLE_PRECISION, master, row,
$        MPI_COMM_WORLD, ierr)
enddo
```

Timing MPI programs

double precision starttime, endtime

starttime = MPI_WTIME()

-

-

-

endtime = MPI_WTIME()

print*, endtime-starttime, ‘ seconds’

AI (Arithmetic Intensity)

- Definition: number of operations per byte
- Matrix (N, N) – Vector(N) Multiplication
$$N(N+N-1)/(4N^2+4N) \sim \frac{1}{4} \text{ ops/byte}$$
- Matrix (N, N) – Matrix(N, N) Multiplication
$$N^2(N+N-1)/(2 \times 4N^2) \sim N/8 \text{ ops/byte}$$

Laplace Equation

Solution: Gauss Iteration

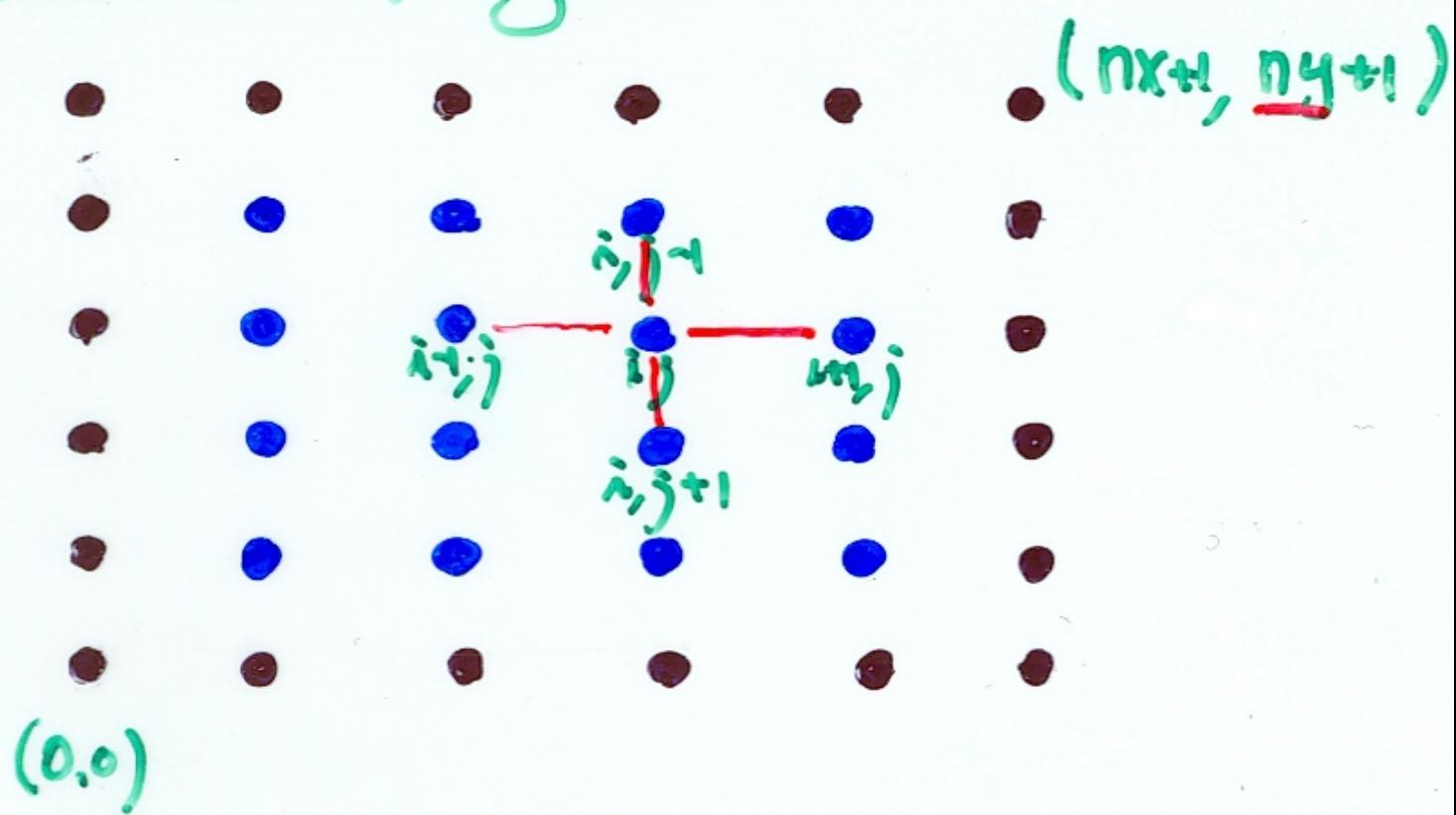
$$\nabla^2 u = 0$$

$$u_{i,j}^{n+1} = \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4}$$

$$\epsilon = \sum_{i,j} |u_{i,j}^{n+1} - u_{i,j}^n|$$

Laplace Equation (cont.)

Sequential Program



```
program main
c
integer nx,ny
parameter(nx=16,ny=16)
double precision u(0:nx+1,0:ny+1),unew(0:nx+1,0:ny+1)
double precision eps,anorm
c-----
c tolerance
c-----
c-----eps = 1.d-5
c-----
c initialization of u
c-----
do 10 j=0,ny+1
do 10 i=0,nx+1
  u(i,j) = 0.d0
10 continue
c-----
c boundary condition
c-----
call bound(nx,ny,u)
c-----
c Gauss iteration
c-----
100 continue
c
do 30 j=1,ny
do 30 i=1,nx
  unew(i,j) = 0.25d0*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))
30 continue
```

```
c-----  
c Compute a norm of difference between u and unew  
c-----  
      anorm = 0.d0  
      do 40 j=1,ny  
      do 40 i=1,nx  
         anorm = anorm + abs(unew(i,j)-u(i,j))  
40  continue  
c-----  
c Update u  
c-----  
      do 50 j=1,ny  
      do 50 i=1,nx  
         u(i,j)=unew(i,j)  
50  continue  
c  
      if (anorm .gt. eps) go to 100  
c-----  
c write output  
c-----  
      open(unit=10,file='u.dat')  
c  
      do 60 j=1,ny  
         write(10,200) (u(i,j),i=1,nx)  
60  continue  
c  
200  format(16f5.2)  
c-----  
c end  
c-----  
      stop  
end
```

```
subroutine bound(nx,ny,u)
c-----
c Dirichlet boundary conditions
c-----
integer i, j
integer nx, ny
double precision u(0:nx+1,0:ny+1)
```

```
c-----
c left and right boundary condition
```

```
c-----
do 10 j = 1,ny
  u( 0,j) = 1.0d0
  u(nx+1,j)=0.0d0
```

```
10 continue
```

```
c-----
c lower and upper boundary condition
```

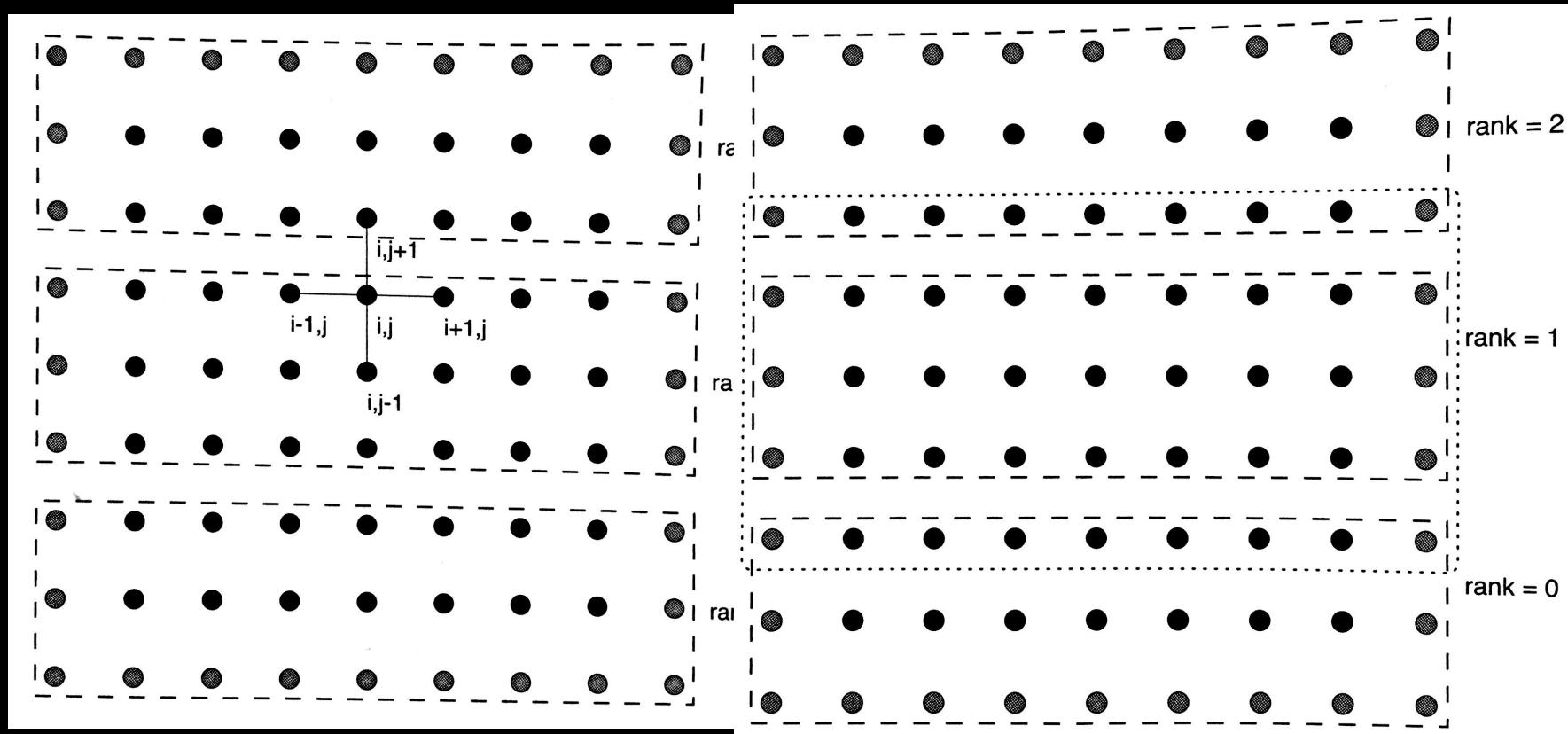
```
c-----
do 20 i = 1,nx
  u(i,0)  = 1.0d0
  u(i,ny+1)=0.0d0
```

```
20 continue
```

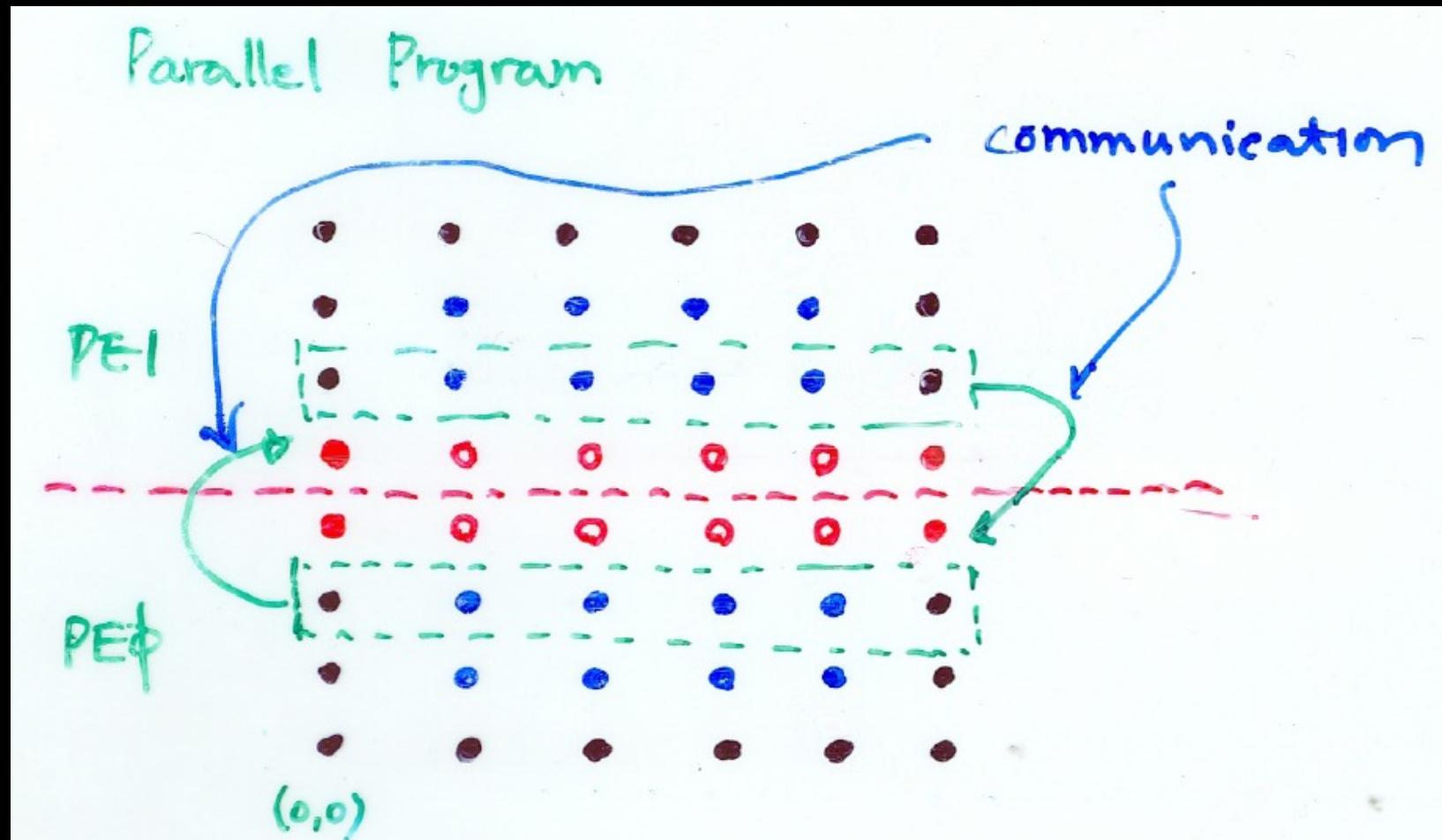
```
c-----
c end
```

```
c-----
      return
end
```

Domain Decomposition



Communications



```
program main
c
c implicit none
c include 'mpif.h'
c
c integer nx, ny, np, nyp
c parameter (nx=16,ny=16,np=2,nyp=8)
c integer i, j
c double precision u(0:nx+1,0:nyp+1), unew(0:nx+1,0:nyp+1)
c double precision eps, anormp, anorm
c
c integer ierr, irank, status(MPI_STATUS_SIZE), ndest
c
c character*8 fname
c-----
c tolerance
c-----
c     eps = 1.d-5
c-----
c MPI initialization
c-----
c     call MPI_INIT(ierr)
c     call MPI_COMM_RANK(MPI_COMM_WORLD,irank,ierr)
c-----
c initialization of u
c-----
c     do 10 j=0,nyp+1
c     do 10 i=0,nx+1
c         u(i,j) = 0.d0
c 10    continue
c-----
c boundary condition
c-----
c     call bound (nx,nyp,u,irank,np)
```

```
c-----  
c Gauss iteration  
c-----  
100 continue  
c  
do 20 j=1,nyp  
do 20 i=1,nx  
  unew(i,j) = 0.25d0*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))  
20 continue  
c-----  
c Compute a norm of difference between u and unew  
c-----  
anormp = 0.d0  
do 30 j=1,nyp  
do 30 i=1,nx  
  anormp = anormp + abs(unew(i,j)-u(i,j))  
30 continue  
call MPI_REDUCE(anormp,anorm,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,  
&           MPI_COMM_WORLD,ierr)  
call MPI_BCAST(anorm,1,MPI_DOUBLE_PRECISION,0,  
&           MPI_COMM_WORLD,ierr)  
c-----  
c Update u  
c-----  
do 40 j=1,nyp  
do 40 i=1,nx  
  u(i,j) = unew(i,j)  
40 continue
```

```
c-----  
c boundary condition between processors  
c-----  
c-----  
c from i-th ranked process to (i+1)-th ranked process  
c-----  
if (irank .eq. (np-1)) then  
    ndest = 0  
else  
    ndest = irank+1  
endif  
c  
call MPI_SEND (u(1,nyp),nx,MPI_DOUBLE_PRECISION,  
&           ndest,irank,MPI_COMM_WORLD,ierr)  
c  
call MPI_BARRIER (MPI_COMM_WORLD,ierr)  
c  
call MPI_RECV (u(1,0),nx,MPI_DOUBLE_PRECISION,  
&           MPI_ANY_SOURCE,MPI_ANY_TAG,  
&           MPI_COMM_WORLD,status,ierr)  
c-----  
c from i-th ranked process to (i-1)-th ranked process  
c-----  
if (irank .eq. 0) then  
    ndest = np-1  
else  
    ndest = irank-1  
endif  
c  
call MPI_SEND (u(1,1),nx,MPI_DOUBLE_PRECISION,  
&           ndest,irank,MPI_COMM_WORLD,ierr)  
c  
call MPI_BARRIER (MPI_COMM_WORLD,ierr)  
c  
call MPI_RECV (u(1,nyp+1),nx,MPI_DOUBLE_PRECISION,  
&           MPI_ANY_SOURCE,MPI_ANY_TAG,  
&           MPI_COMM_WORLD,status,ierr)
```

```
c-----  
c boundary condition  
c-----  
    call bound (nx,nyp,u,irank,np)  
c-----  
c Gauss iteration  
c-----  
    if (anorm .gt. eps) go to 100  
c-----  
c write output  
c-----  
    write(fname,900) 'u',irank,'.dat'  
c  
900  format (a1,i3.3,a4)  
c  
    open(unit=10,file=fname)  
c  
    do 50 j=1,nyp  
        write(10,200) (u(i,j),i=1,nx)  
50  continue  
c  
200  format(16f5.2)  
c-----  
c MPI finalization  
c-----  
    call MPI_FINALIZE(ierr)  
c-----  
c end  
c-----  
    stop  
end
```

```
subroutine bound (nx,nyp,u,irank,np)
```

```
c-----
```

```
c Dirichlet boundary conditions
```

```
c-----
```

```
implicit none
```

```
integer i, j
```

```
integer nx, nyp, irank, np
```

```
double precision u(0:nx+1,0:nyp+1)
```

```
c-----
```

```
c left and right boundary condition
```

```
c-----
```

```
do 10 j = 1,nyp
```

```
    u( 0,j) = 1.0d0
```

```
    u(nx+1,j) = 0.0d0
```

```
10 continue
```

```
c-----
```

```
c lower boundary condition
```

```
c-----
```

```
if (irank .eq. 0) then
```

```
    do 20 i = 1,nx
```

```
        u(i,0) = 1.0d0
```

```
20 continue
```

```
endif
```

```
c-----
```

```
c upper boundary condition
```

```
c-----
```

```
if (irank .eq. np-1) then
```

```
    do 30 i = 1,nx
```

```
        u(i,nyp+1)= 0.0d0
```

```
30 continue
```

```
endif
```

```
c-----
```

```
c end
```

```
c-----
```

```
return
```

```
end
```

Several SEND/RECV routines

- SEND/RECV
- BSEND/BRECV
- SENDRECV
- SSEND/SRECV
- ISEND/IRecv

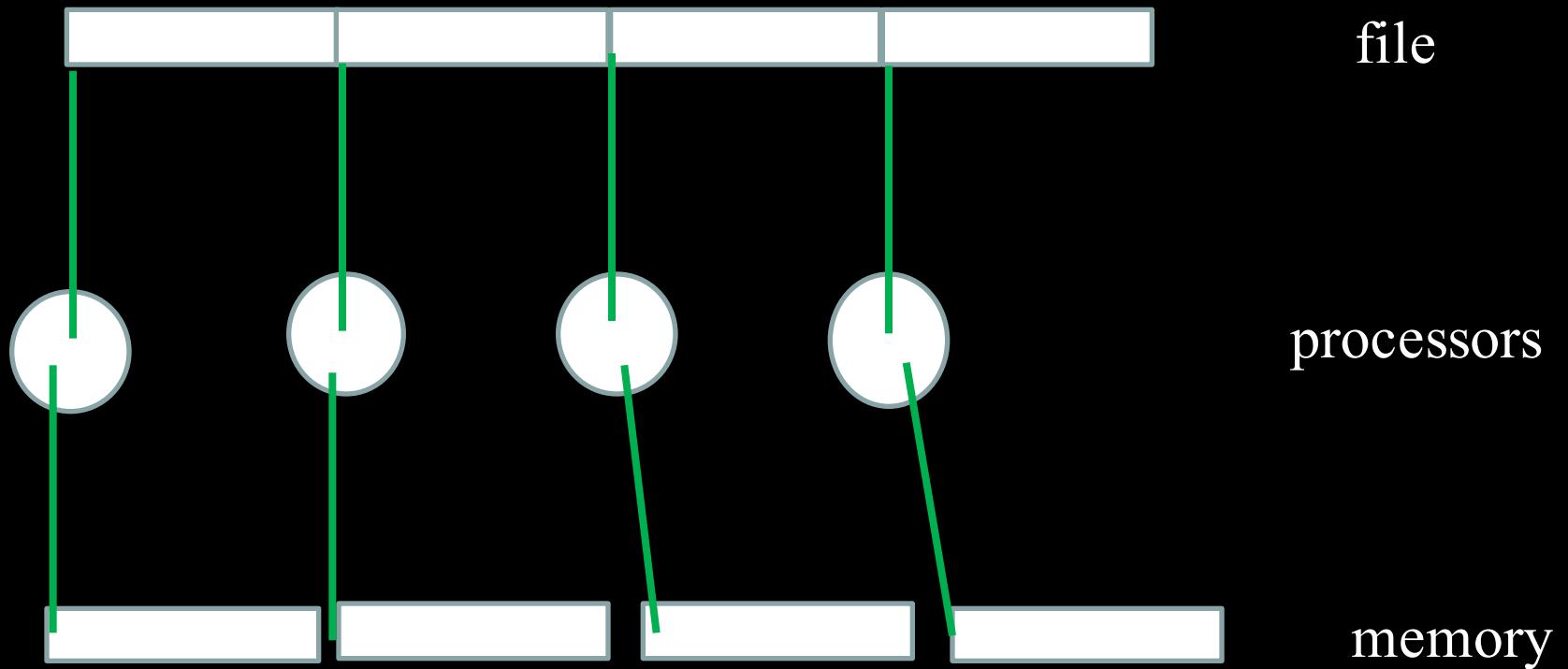
fwrite

```
#include <stdio.h>

#define SIZE (1024*1024)

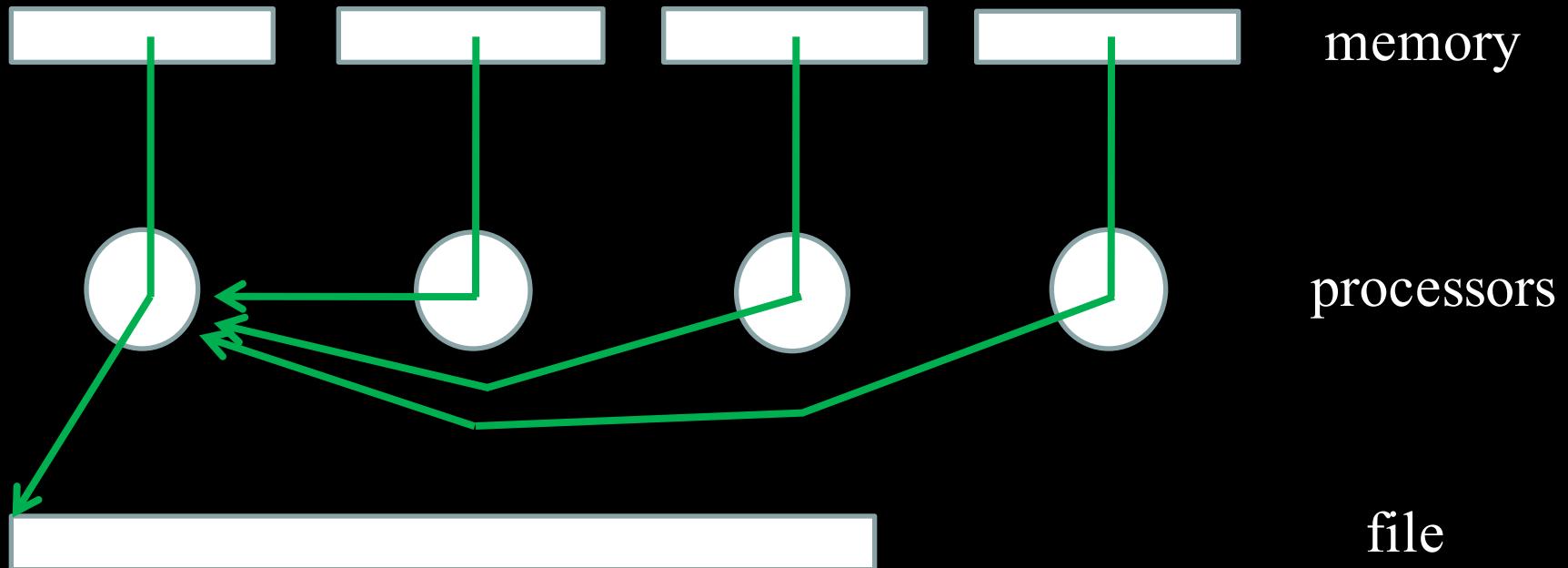
int main (int argc, char **argv)
{
    FILE * pf;
    int i, buffer[SIZE];
    for (i=0; i<SIZE; i++) buffer[i] = i;
    pf = fopen ("/home/guest/jskim/datafile", "wb");
    fwrite (buffer , sizeof(int), sizeof(buffer), pf);
    fclose (pf);
    return 0;
}
```

Parallel input from a file



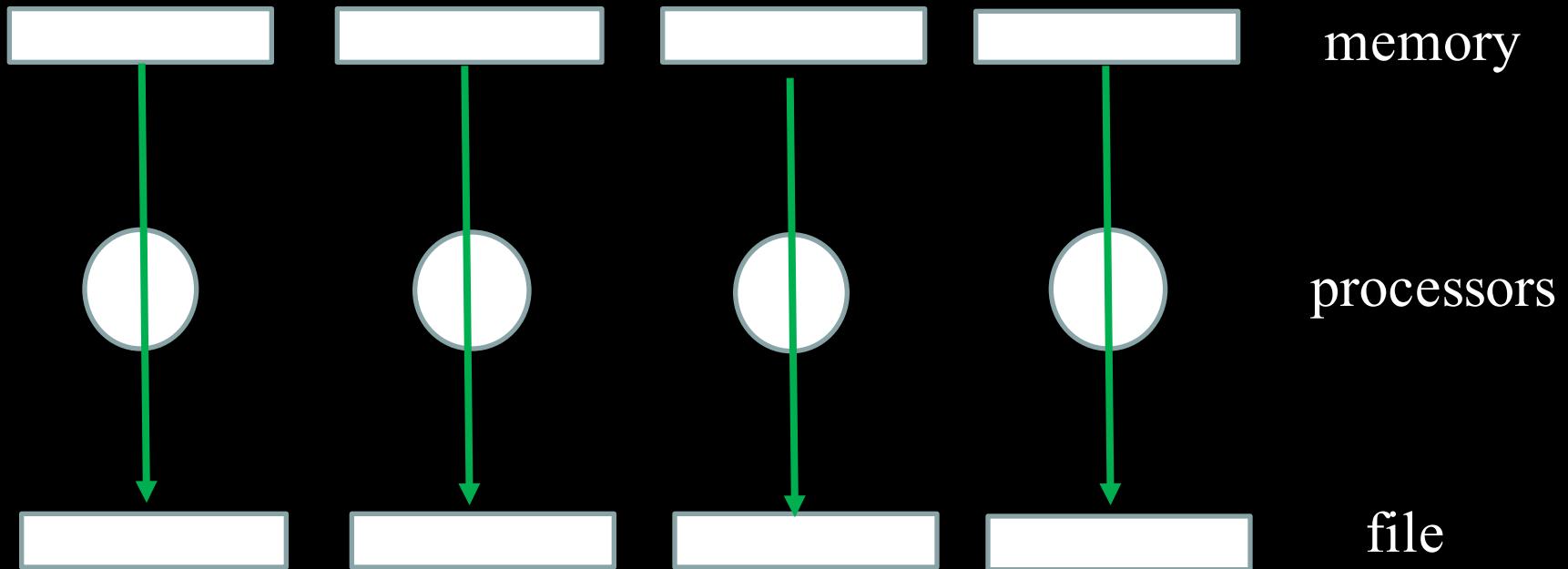
Parallel Output

Sequential I/O from an parallel program



Parallel Output (cont.)

Parallel I/O to multiple files



```
! example of parallel MPI write into multiple files
PROGRAM main
! Fortran 90 users can (and should) use
!   use mpi
! instead of include 'mpif.h' if their MPI implementation provides a
! mpi module.
include 'mpif.h'

integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
character*12 ofname

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

do i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo

write(ofname,'(a8,i4.4)') 'testfile',myrank

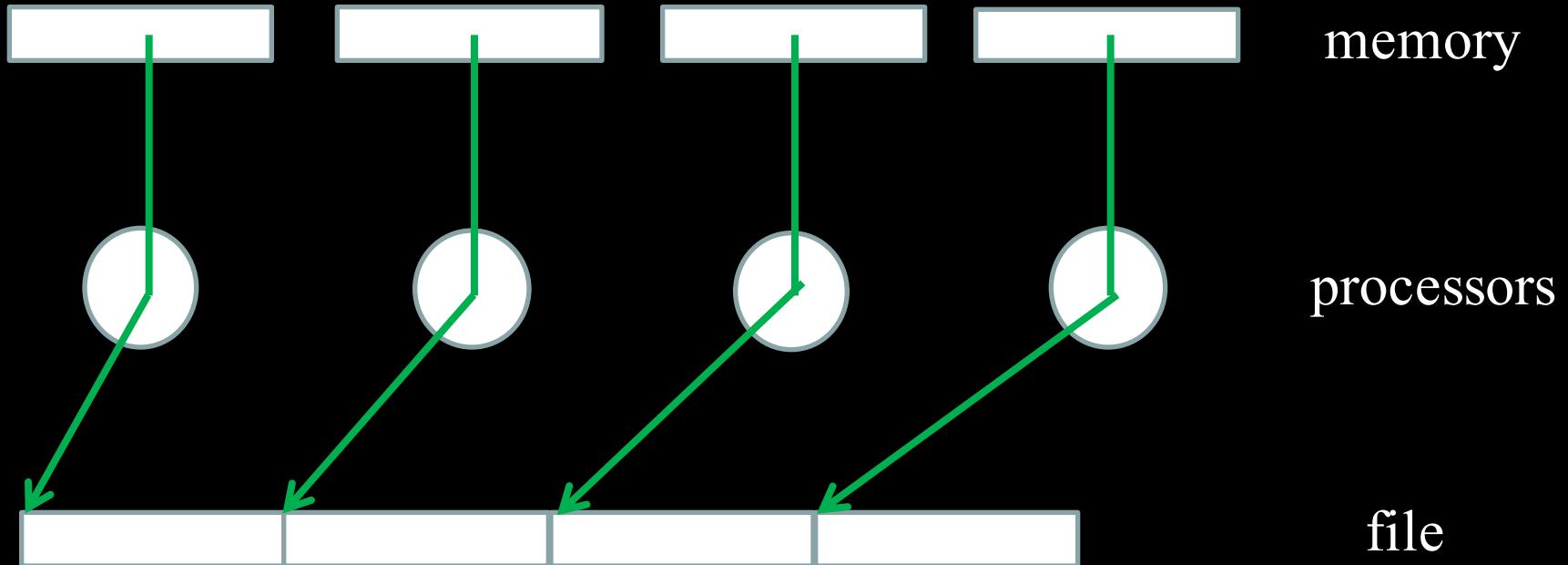
open(unit=11,file=ofname,form='unformatted')
write(11) buf

call MPI_FINALIZE(ierr)

END PROGRAM main
```

Parallel Output (cont.)

Parallel I/O to a single file



```
! example of parallel MPI write into a single file
PROGRAM main
! Fortran 90 users can (and should) use
!   use mpi
! instead of include 'mpif.h' if their MPI implementation provides a
! mpi module.
include 'mpif.h'

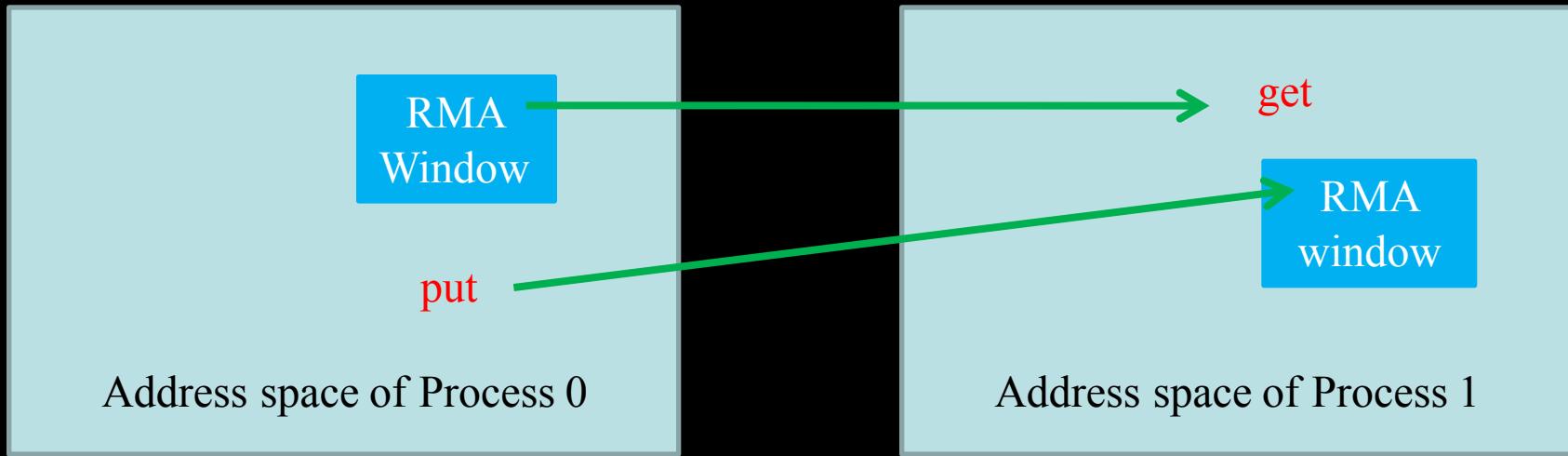
integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

do i = 1, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
                  MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                  MPI_INFO_NULL, thefile, ierr)
! assume 4-byte integers
disp = myrank * BUFSIZE * 4
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                      MPI_INTEGER, 'native', &
                      MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
                   MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)

END PROGRAM main
```

Remote Memory Access

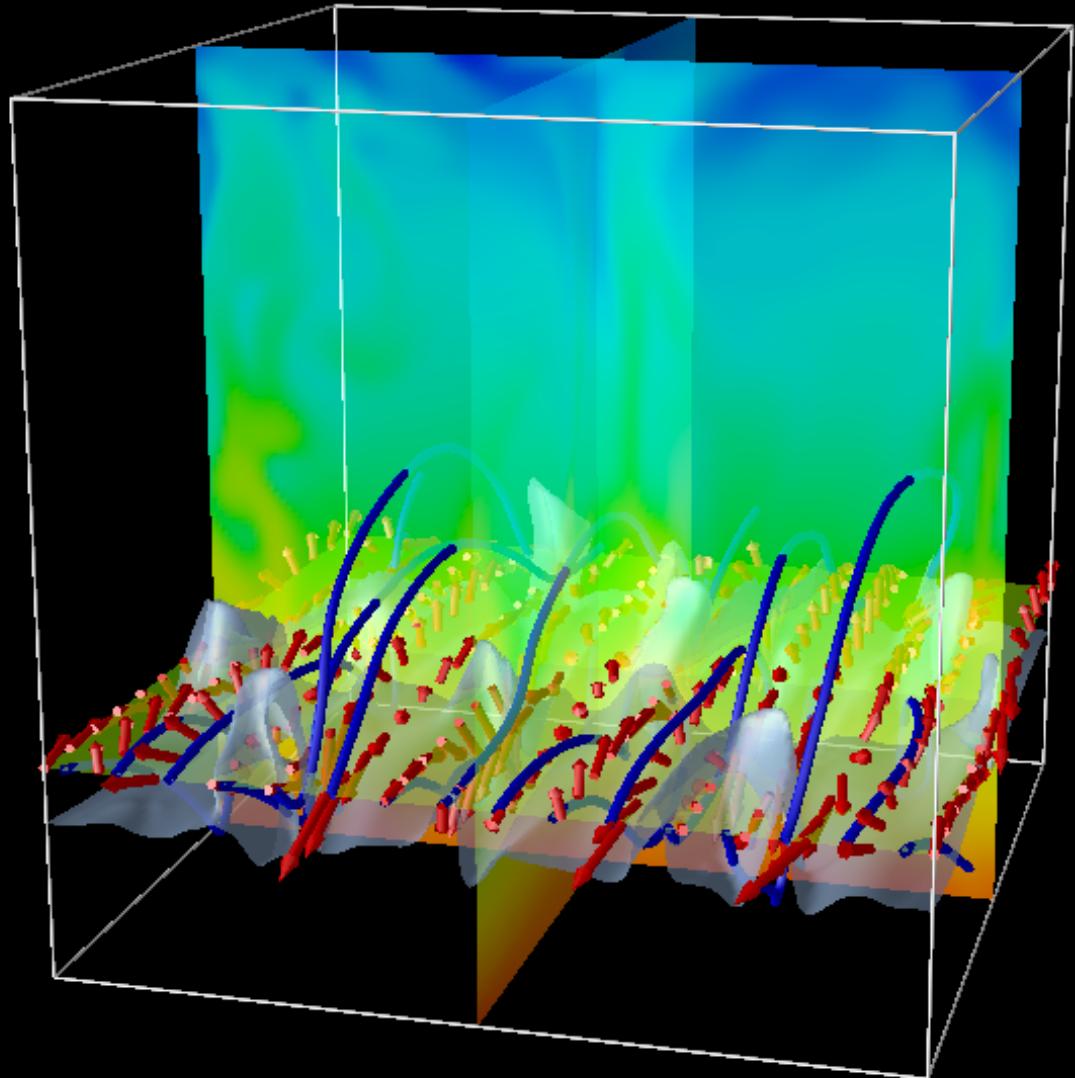


Dynamic Process Management

- Intercommunicator
 - spawning; connecting

3D MHD Simulation Using MPI

- 256^3 cells
- Cray T3E-900
- 64 PEs, 8000 hrs
- Kim+ 1998, ApJL



Isothermal HD equations

- Isothermal HD equations

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \quad \rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) + \nabla (\rho a^2) = 0$$

- Conservative Form

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{F}_x}{\partial x} + \frac{\partial \mathbf{F}_y}{\partial y} + \frac{\partial \mathbf{F}_z}{\partial z} = 0$$

$$\mathbf{q} = \begin{bmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \end{bmatrix}$$

$$\mathbf{F}_x = \begin{bmatrix} \rho v_x \\ \rho v_x^2 + \rho a^2 \\ \rho v_x v_y \\ \rho v_x v_z \end{bmatrix}$$

$$\mathbf{F}_y = \begin{bmatrix} \rho v_y \\ \rho v_x v_y \\ \rho v_y^2 + \rho a^2 \\ \rho v_y v_z \end{bmatrix}$$

$$\mathbf{F}_z = \begin{bmatrix} \rho v_z \\ \rho v_x v_z \\ \rho v_y v_z \\ \rho v_z^2 + \rho a^2 \end{bmatrix}$$

Strang-type Directional Splitting

- Reduce the multi-dimensional problem to one-dimensional one.

$$\mathbf{q}^{n+1} = L_z L_y L_x q^n$$

$$L_x : \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{F}_x}{\partial x} = 0$$

$$L_y : \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{F}_y}{\partial y} = 0$$

$$L_z : \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{F}_z}{\partial z} = 0$$

$$\mathbf{q}^{n+6} = (L_x L_z L_y)(L_y L_x L_z)(L_y L_z L_x)(L_x L_z L_y)(L_x L_y L_z)(L_z L_y L_x)q^n$$

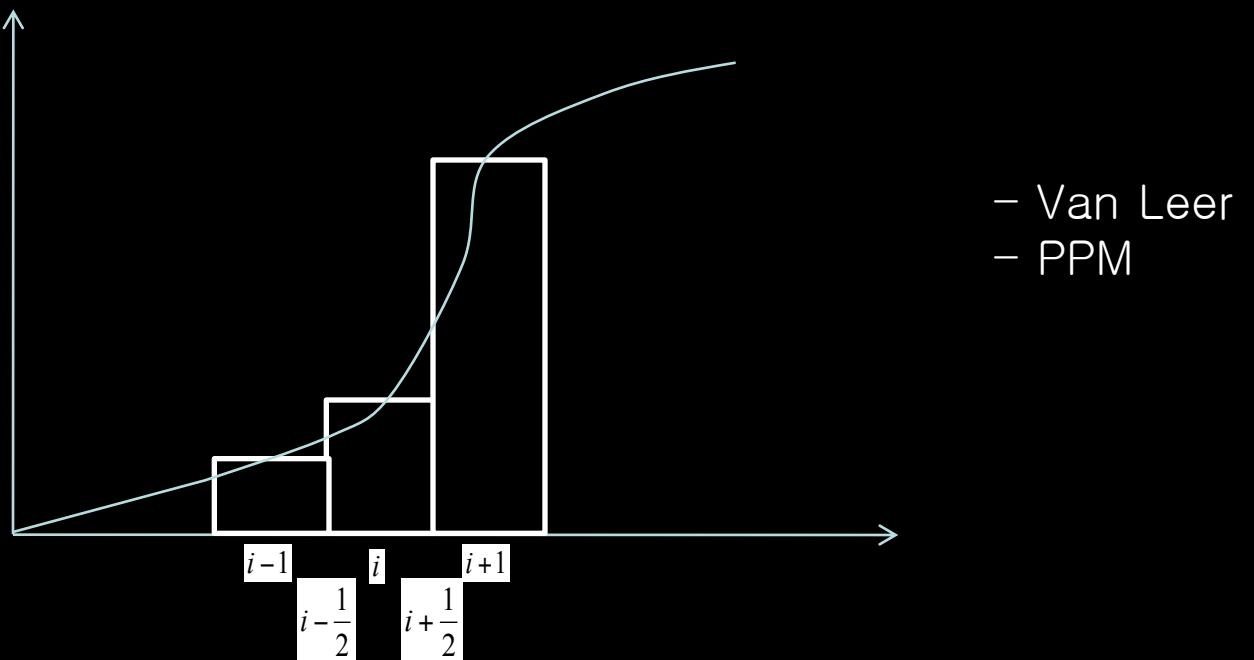
Godunov's Scheme

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = 0$$

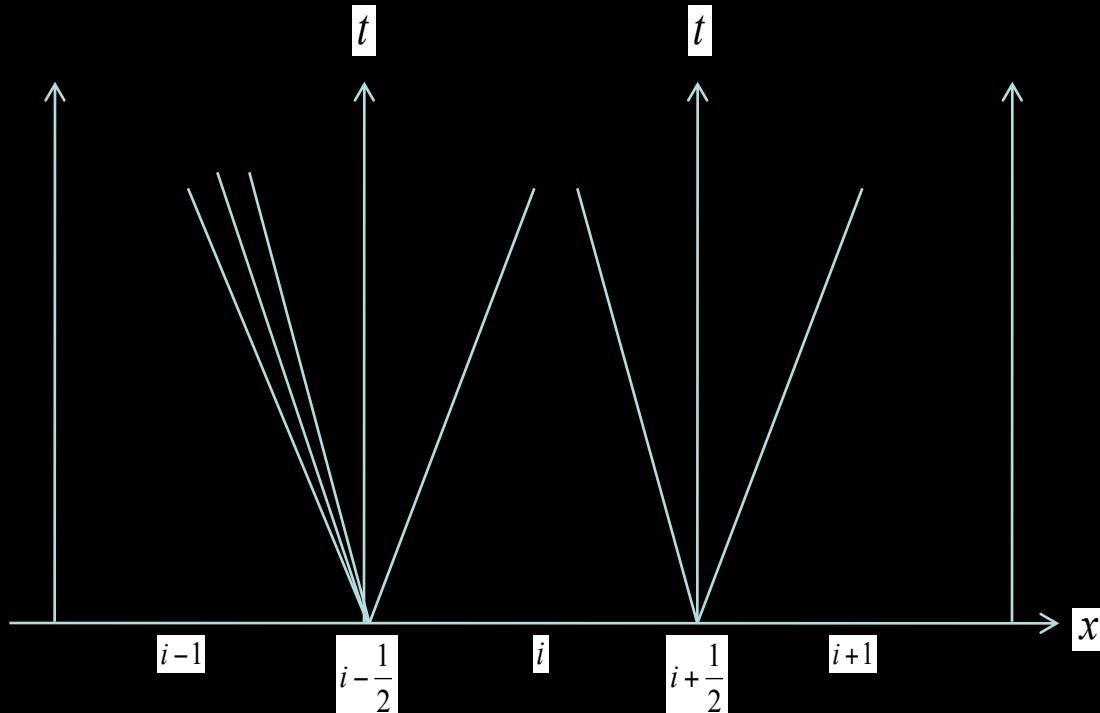
$$\mathbf{q}_i^{n+1} = \mathbf{q}_i^n + \frac{\Delta t}{\Delta x} \left(\mathbf{F}_{i-\frac{1}{2}} - \mathbf{F}_{i+\frac{1}{2}} \right)$$

$$\mathbf{q}_i^{n+1} = \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \mathbf{q}(x, t^n) dx$$

$$\Delta t = \frac{\Delta x}{S_{\max}^n}$$



Riemann problems



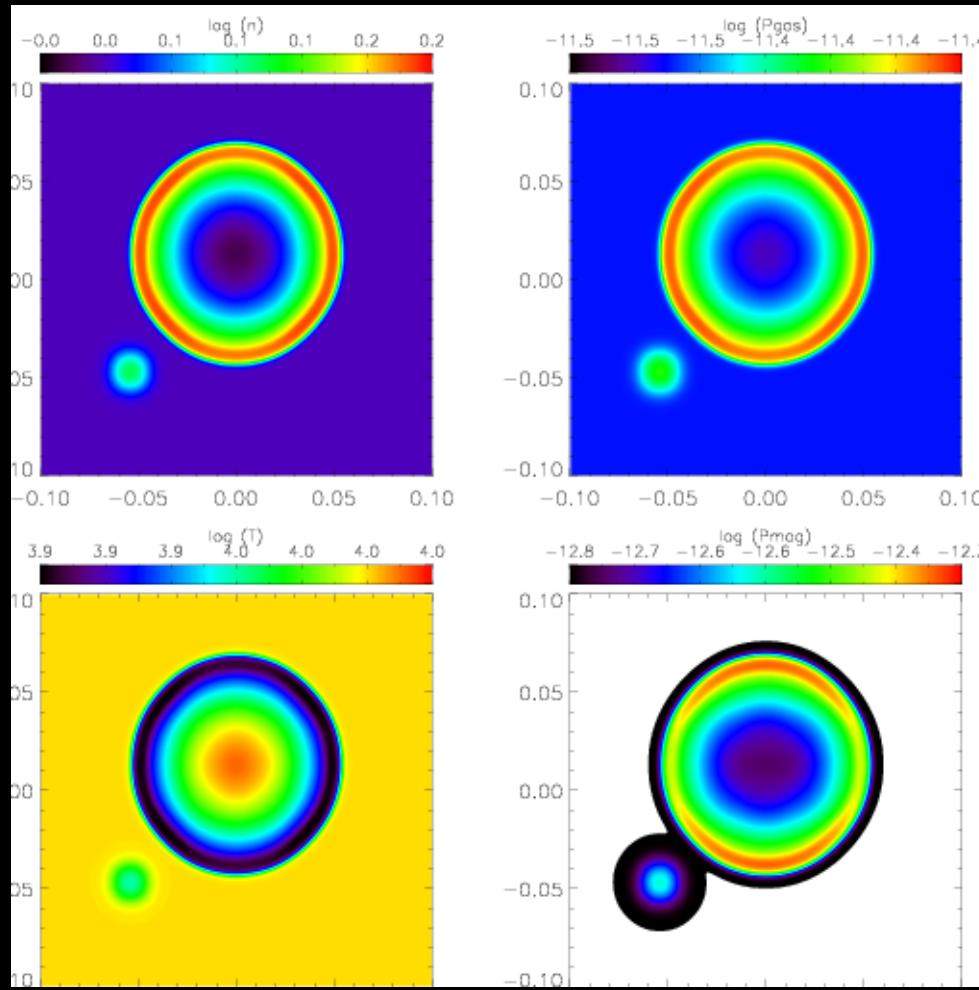
- Exact Riemann solver
- Roe’s Scheme
- HLL Scheme

KASI-ARCSEC PC CLUSTER



- 128 CPUs
- 128 GB Mem.
- 6TB Disk

Test problem for a benchmark for the PC cluster



Benchmark test of the MHD code

