Introduction to High Performance Scientific Computing

Autumn, 2016

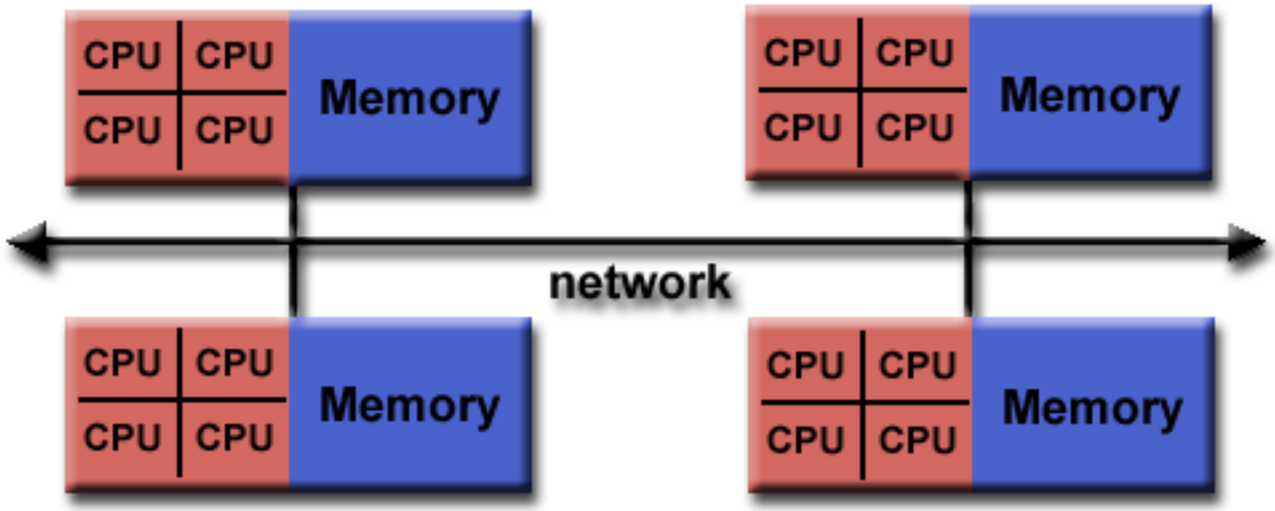
Lecture 15

Imperial College
London

Prasun Ray

28 November 2016

Parallel computing paradigms

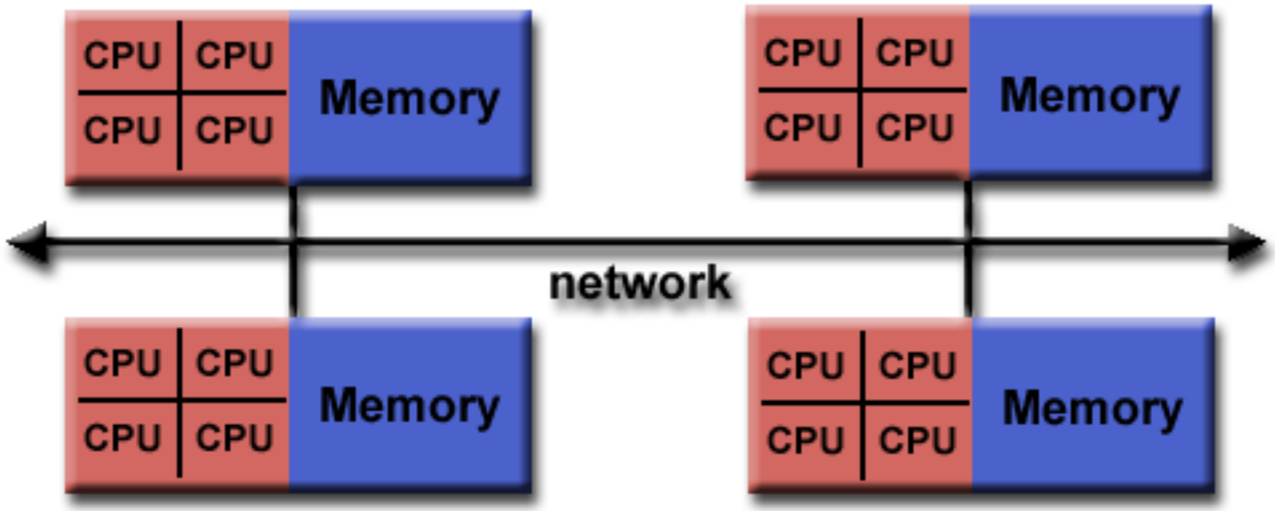


Distributed memory

- Each (4-core) chip has its own memory
- The chips are connected by network ‘cables’
- MPI coordinates communication between two or more CPUs

Imperial College
London

Parallel computing paradigms



Related approaches:

- Hybrid programming: mix of shared-memory (OpenMP) and distributed-memory (MPI) programming
- GPU’s: Shared memory programming (CUDA or OpenCL)
- Coprocessors and co-array programming

MPI intro

- MPI: *Message Passing Interface*
- Standard for exchanging data between processors
- Supports Fortran, c, C++
- Can also be used with Python

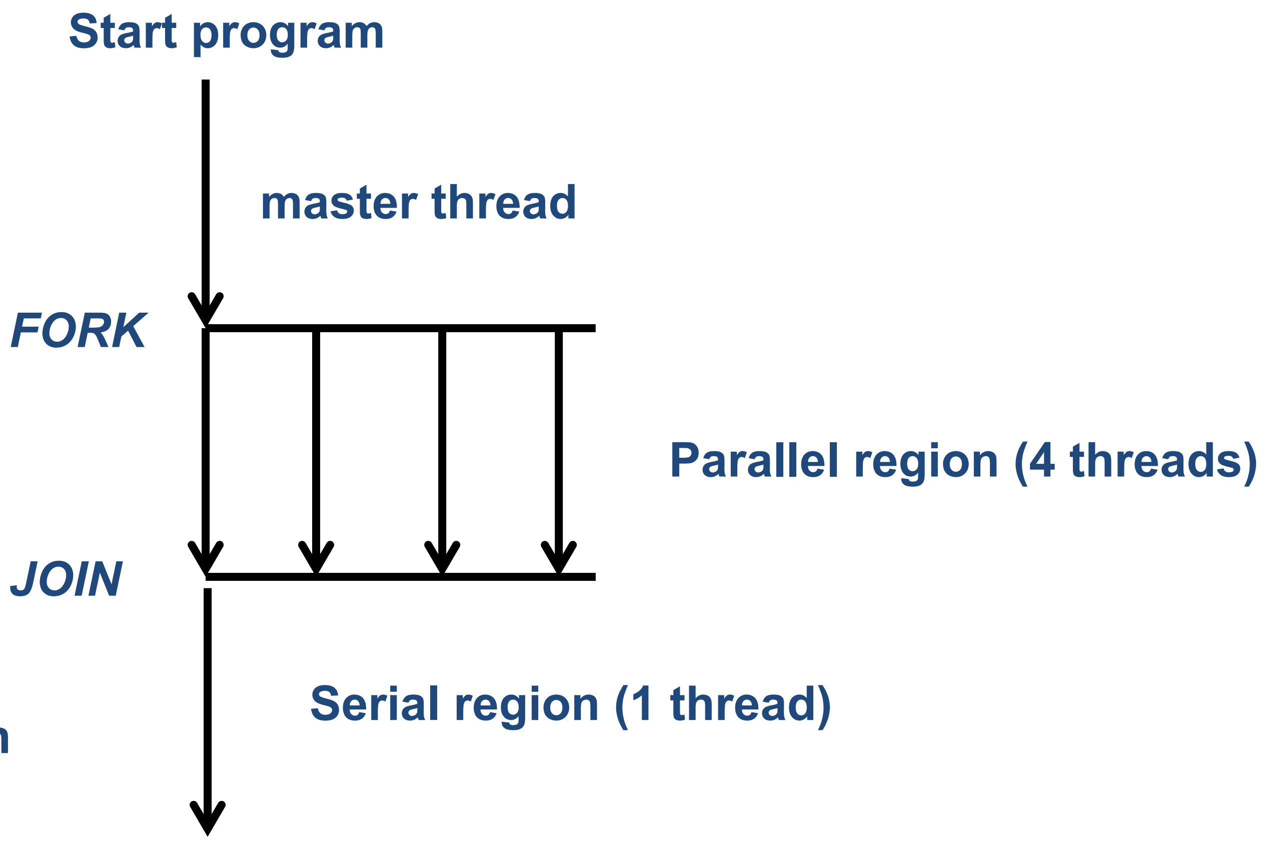
OpenMP schematic

Program starts with single *master thread*

Then, launch parallel region with multiple threads.

Each thread has access to all variables introduced previously

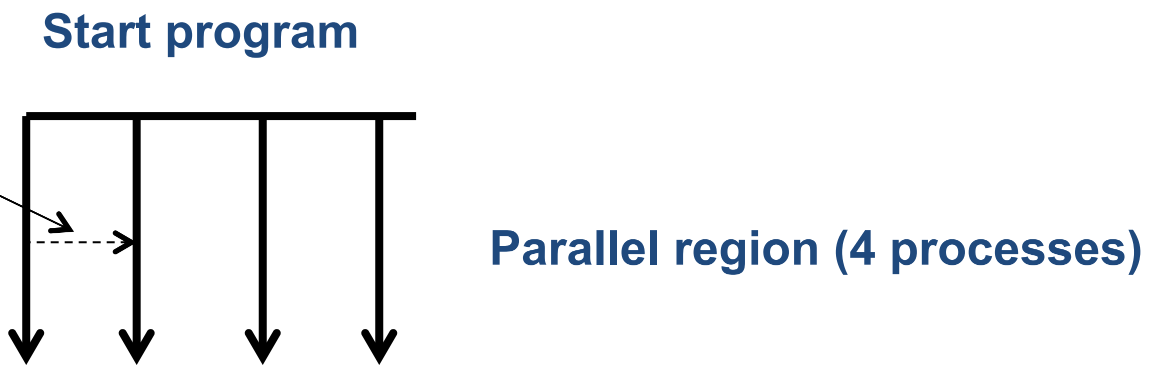
Can end parallel region if/when desired and launch parallel regions again in future as needed



MPI schematic

Program starts with all processes running

MPI controls communication between processes



MPI intro

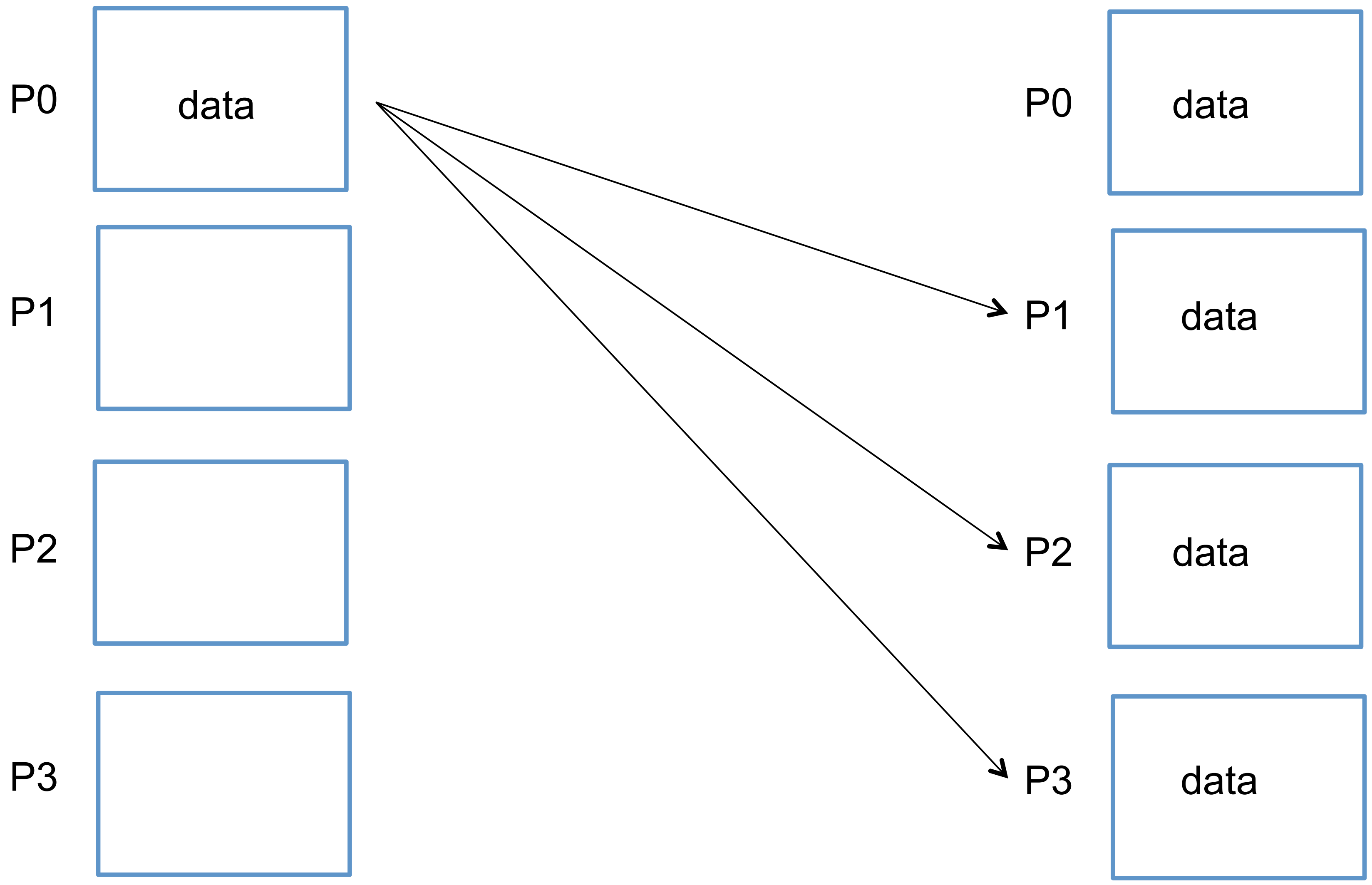
- **Basic idea: calls to MPI subroutines control data exchange between processors**

- **Example:**

```
call MPI_BCAST(n, 1, MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

This will send the integer n which has size 1 from processor 0 to all of the other processors.

MPI broadcast



Imperial College
London

MPI intro

- **Basic idea: calls to MPI subroutines control data exchange between processors**

- **Example:**

```
call MPI_BCAST(n, 1, MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

This will send the integer n which has size 1 from processor 0 to all of the other processors.

Generally, need to specify:

- **source and/or destination of message**
- **size of data contained in message**
- **type of data contained in message (integer, double precision, …)**
- **the data itself (or its location)**

Fortran code structure

```
! Basic Fortran 90 code structure

!1. Header
program template

    !2. Variable declarations (e.g. integers, real numbers,...)

    !3. basic code: input, loops, if-statements, subroutine calls
    print *, 'template code'


!4. End program
end program template

! To compile this code:
! $ gfortran -o f90template.exe f90template.f90
! To run the resulting executable: $ ./f90template.exe
```

MPI intro

See *mpif90template.f90*

```
! Basic MPI + Fortran 90 code structure

!1. Header
program template
    use mpi

    !2a. Variable declarations (e.g. integers, real numbers,...)
    integer :: myid, numprocs, ierr

    !2b. Initialize MPI
    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

    !3. basic code: input, loops, if-statements, subroutine calls
    print *, 'this is proc # ',myid, 'of ', numprocs

!4. End program
    call MPI_FINALIZE(ierr)
end program template

! To compile this code:
! $ mpif90 -o mpitemplate.exe mpif90template.f90
! To run the resulting executable with 4 processes:$ mpiexec -n 4 mpitemplate.exe
```

MPI intro

See *mpif90template.f90*

```
! Basic MPI + Fortran 90 code structure

!1. Header
program template
    use mpi

    !2a. Variable declarations (e.g. integers, real numbers,...)
    integer :: myid, numprocs, ierr

    !2b. Initialize MPI
    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

    !3. basic code: input, loops, if-statements, subroutine calls
    print *, 'this is proc # ',myid, 'of ', numprocs

!4. End program
    call MPI_FINALIZE(ierr)
end program template

! To compile this code:
! $ mpif90 -o mpitemplate.exe mpif90template.f90
! To run the resulting executable with 4 processes:$ mpiexec -n 4 mpitemplate.exe
```

MPI intro

See *mpif90template.f90*

```
! Basic MPI + Fortran 90 code structure

!1. Header
program template
    use mpi

    !2a. Variable declarations (e.g. integers, real numbers,...)
    integer :: myid, numprocs, ierr

    !2b. Initialize MPI
    call MPI_INIT(ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

    !3. basic code: input, loops, if-statements, subroutine calls
    print *, 'this is proc # ',myid, 'of ', numprocs

!4. End program
    call MPI_FINALIZE(ierr)
end program template

! To compile this code:
! $ mpif90 -o mpitemplate.exe mpif90template.f90
! To run the resulting executable with 4 processes:$ mpiexec -n 4 mpitemplate.exe
```

MPI intro

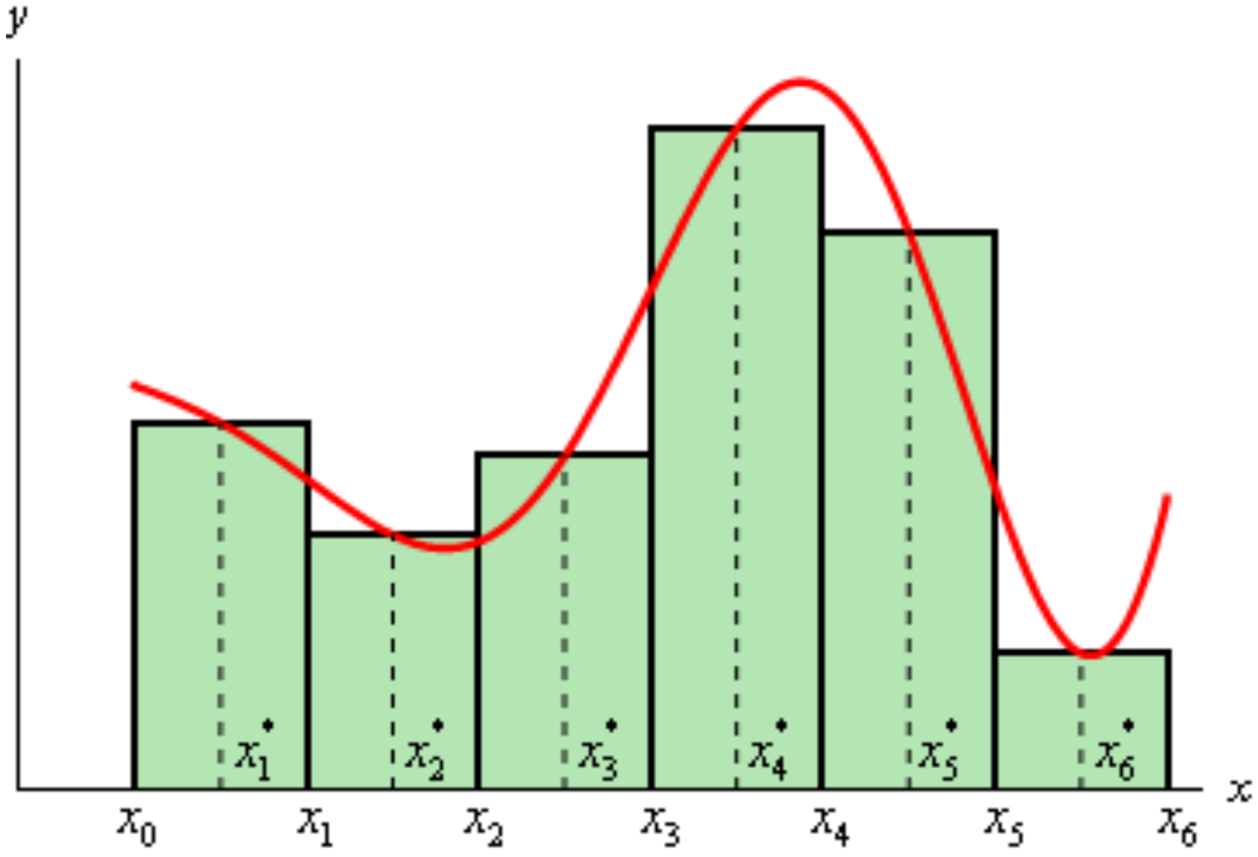
- **Compile + run:**

```
$ mpif90 -o mpif90template.exe mpif90template.f90

$ mpiexec -n 4 mpif90template.exe
 this is proc #            0 of            4
 this is proc #            3 of            4
 this is proc #            1 of            4
 this is proc #            2 of            4
```

Note: The number of processes specified with mpiexec can be larger than the number of cores on your machine, but then tasks are run sequentially.

MPI+Fortran example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

MPI+Fortran quadrature

Two most important tasks:

1. **Decide how many intervals per processor**

2. **Each processor will compute its own partial sum,** sum_proc, **how do we compute** sum(sum_proc)?

MPI+Fortran quadrature

Two most important tasks:

1. **Decide how many intervals per processor**

2. **Each processor will compute its own partial sum,** sum_proc, **how do we compute** sum(sum_proc)**?**

- N **= number of intervals**

- numprocs **= number of processors**

- **Need to compute** Nper_proc: **intervals per processor**

MPI+Fortran quadrature

- N **= number of intervals**
- numprocs **= number of processors**
- **Need to compute** Nper_proc: **intervals per processor**
  - **Basic idea: if** N = 8 * numprocs**,** Nper_proc = 8
  - **But, if** N <= numprocs**, N/numprocs = 0**

$\boxed{\text{Nper\_proc = (N + numprocs – 1)/numprocs}}$

MPI+Fortran quadrature

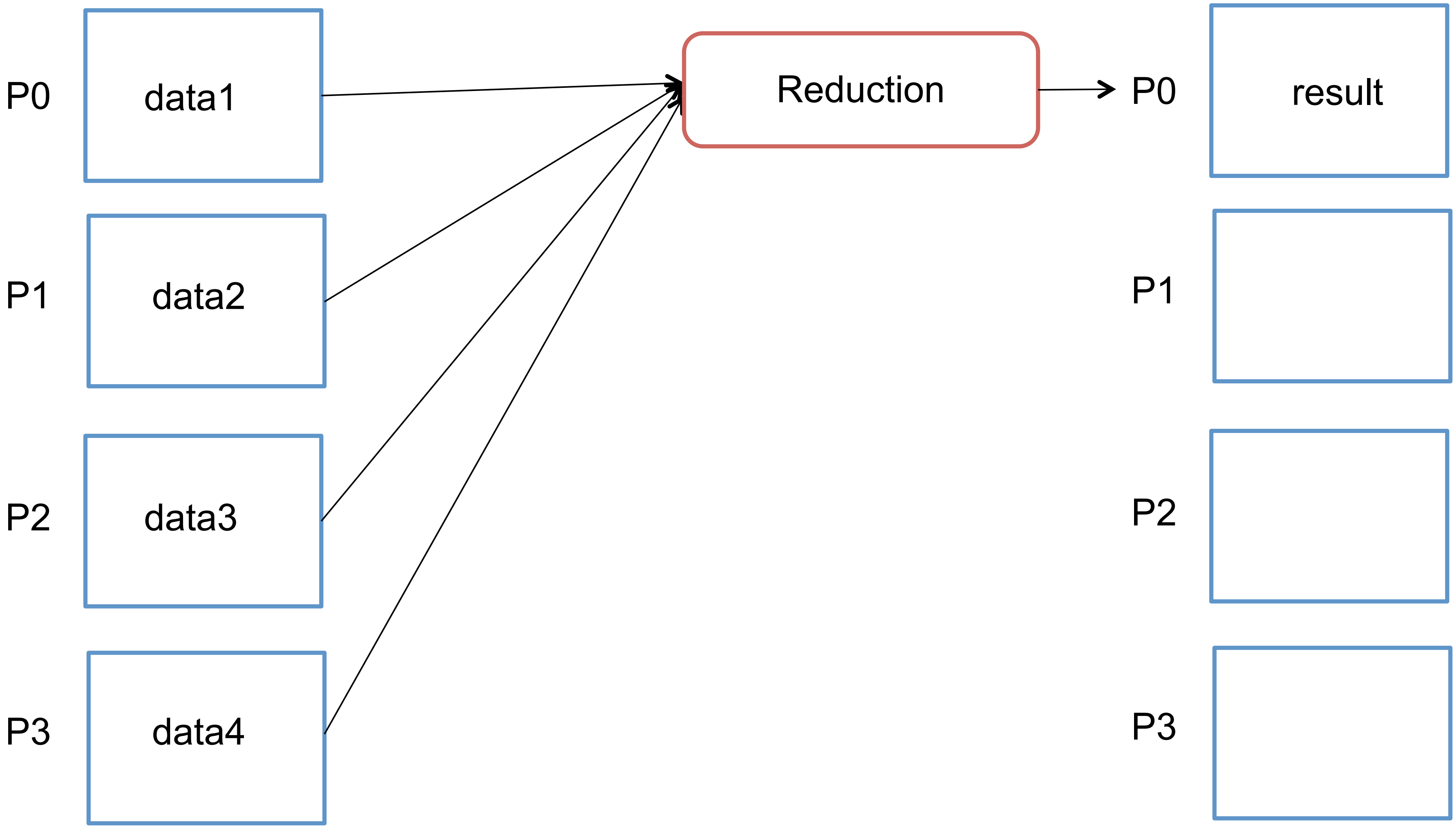
Two most important tasks:

1. **Decide how many intervals per processor**

2. **Each processor will compute its own partial sum,** sum_proc, **how do we compute** sum(sum_proc)?

Use MPI_REDUCE

MPI reduce


P0
data1
P1
data2
P2
data3
P3
data4
Reduction
P0
result
P1
P2
P3

MPI+Fortran quadrature

Two most important tasks:

1. **Decide how many intervals per processor**
2. **Each processor will compute its own partial sum,** sum_proc, **how do we compute** sum(sum_proc)**?**

- **Use MPI_REDUCE**
- **Reduction options: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD**

MPI+Fortran quadrature

Two most important tasks:

1. **Decide how many intervals per processor**

2. **Each processor will compute its own partial sum,** sum_proc, **how do we compute** sum(sum_proc)**?**

- **Use MPI_REDUCE**
- **Reduction options: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD**
- **For quadrature, we need MPI_SUM**

MPI+Fortran quadrature

For quadrature, we need MPI_SUM:

```
call MPI_REDUCE(data, result, 1, MPI_DOUBLE_PRECISION,
                                  MPI_SUM,0,MPI_COMM_WORLD,ierr)
```

This will:

1. **Collect the** double precision **variable** data **which has size** 1 **from each processor.**

2. **Compute the sum (because we have chosen** MPI_SUM) **and store the value in** result **on processor** 0.

   Note: Only processor 0 will have the final sum. With MPI_ALLREDUCE, **the result will be on every processor.**

MPI+Fortran quadrature

midpoint_p.f90: distribute data

```
 !set number of intervals per processor
   Nper_proc =  (N + numprocs - 1)/numprocs

!starting and ending points for processor
    istart = myid * Nper_proc + 1
    iend = (myid+1) * Nper_proc
    if (iend>N) iend = N
```

MPI+Fortran quadrature

***midpoint_p.f90:* 1. distribute data, 2. compute** sum_proc

```
 !set number of intervals per processor
   Nper_proc =  (N + numprocs - 1)/numprocs

!starting and ending points for processor
    istart = myid * Nper_proc + 1
    iend = (myid+1) * Nper_proc
    if (iend>N) iend = N

!loop over intervals computing each interval's contribution to 
integral
    do i1 = istart,iend
        xm = dx*(i1-0.5) !midpoint of interval i1
        call integrand(xm,f)
        sum_i = dx*f
        sum_proc = sum_proc + sum_i !add contribution from interval 
to total integral
    end do
```

Imperial College
London

MPI+Fortran quadrature

***midpoint_p.f90:* 1. distribute data, 2. compute** sum_proc, **3. reduction**

```
 !set number of intervals per processor
   Nper_proc =  (N + numprocs - 1)/numprocs

!starting and ending points for processor
    istart = myid * Nper_proc + 1
    iend = (myid+1) * Nper_proc
    if (iend>N) iend = N

!loop over intervals computing each interval's contribution to integral
    do i1 = istart,iend
        xm = dx*(i1-0.5) !midpoint of interval i1
        call integrand(xm,f)
        sum_i = dx*f
        sum_proc = sum_proc + sum_i !add contribution from interval to 
total integral
    end do
!collect double precision variable, sum, with size 1 on process 0 using 
the MPI_SUM option
    call MPI_REDUCE(sum_proc,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,
0,MPI_COMM_WORLD,ierr)
```

MPI+Fortran quadrature

Compile and run:

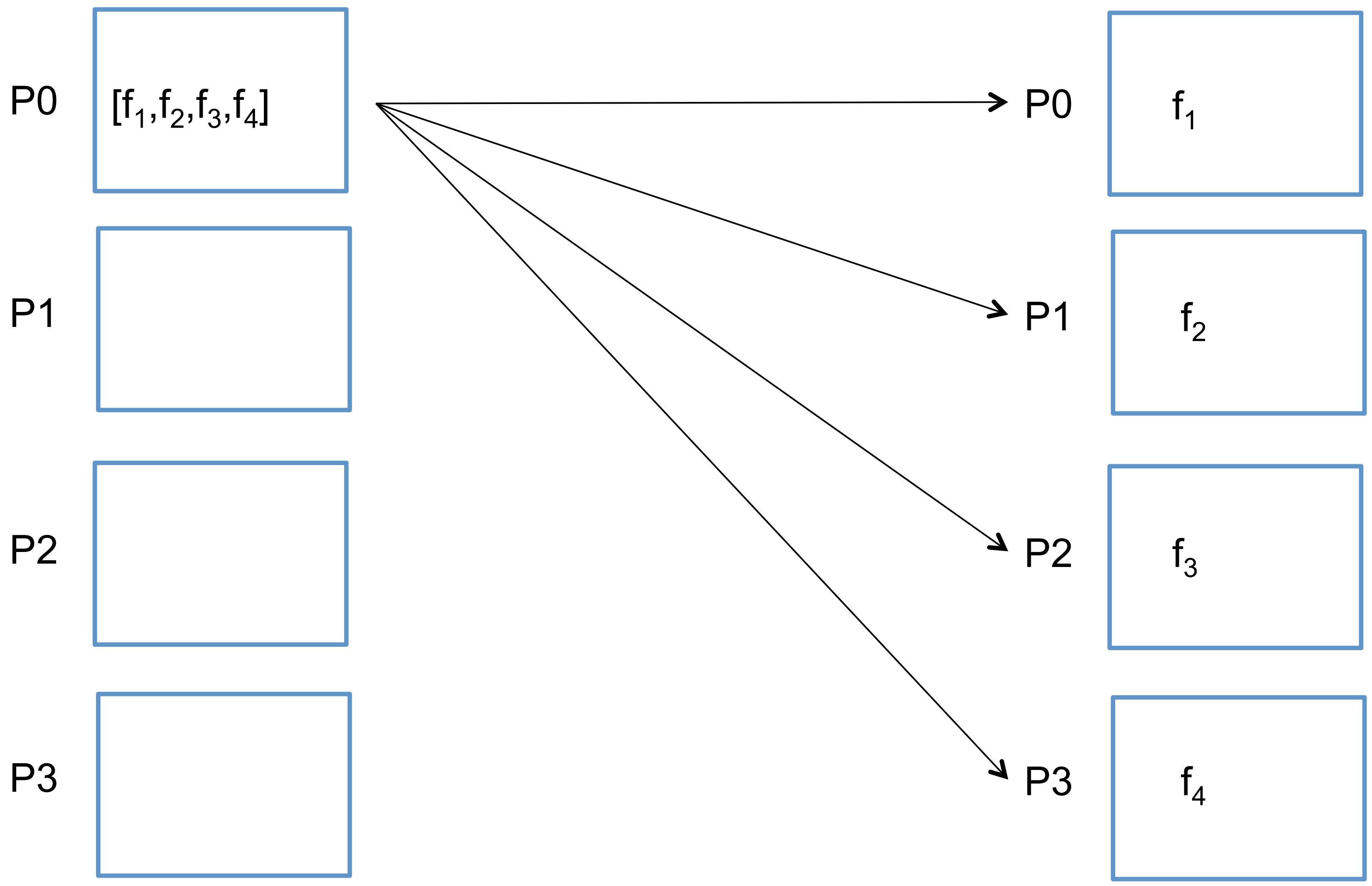
```
$ mpif90 -o midpoint_p.exe midpoint_p.f90

$ mpiexec -n 2 midpoint_p.exe
 number of intervals =         1000
 number of procs =              2
 Nper_proc=          500
 The partial sum on proc #           0 is:   1.8545905426699112
 The partial sum on proc #           1 is:   1.2870021942532193
 N=        1000
 sum=   3.1415927369231307
 error=   8.3333337563828991E-008
```

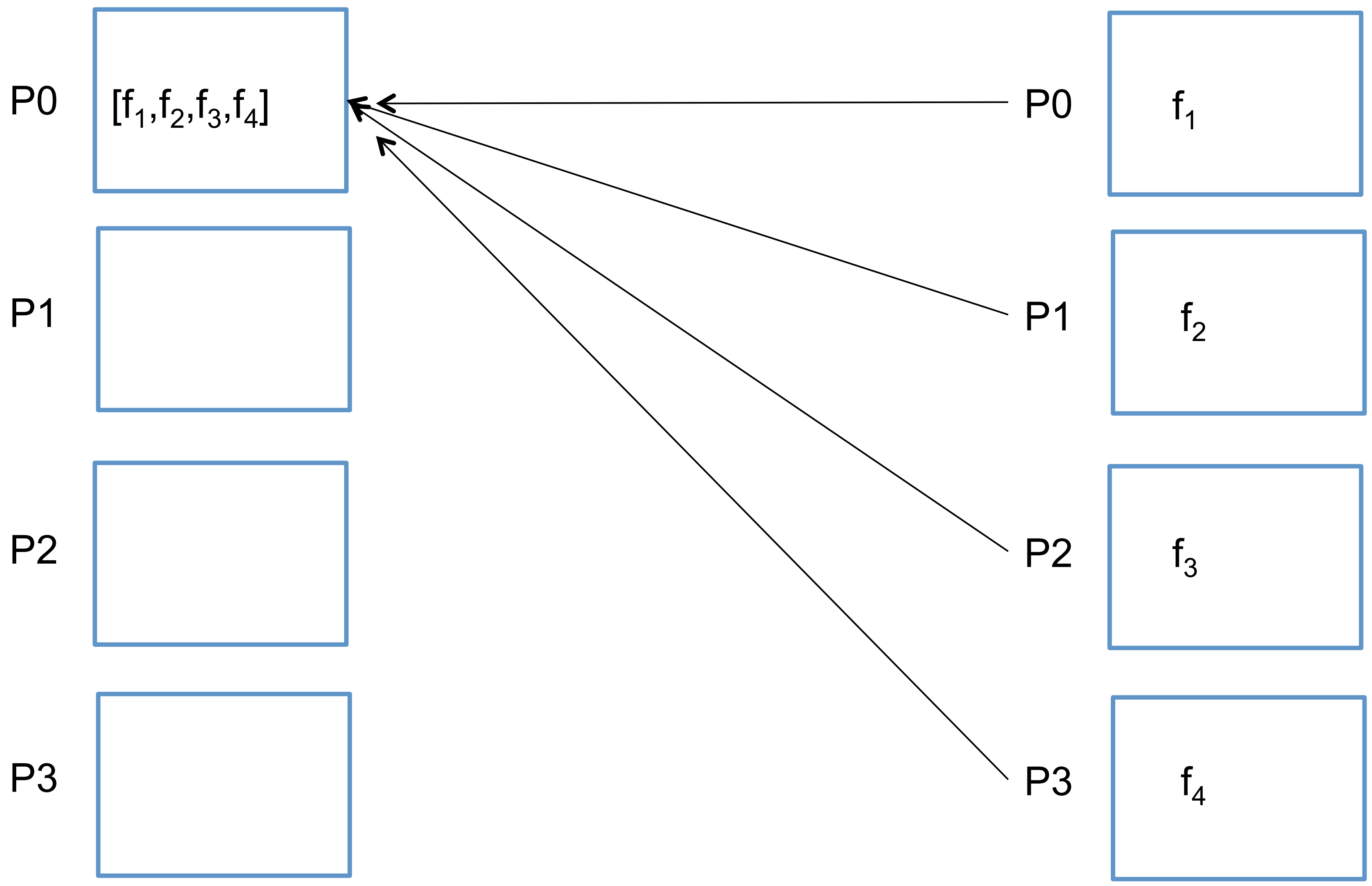
Other collective operations

- Scatter and gather

MPI scatter



MPI gather



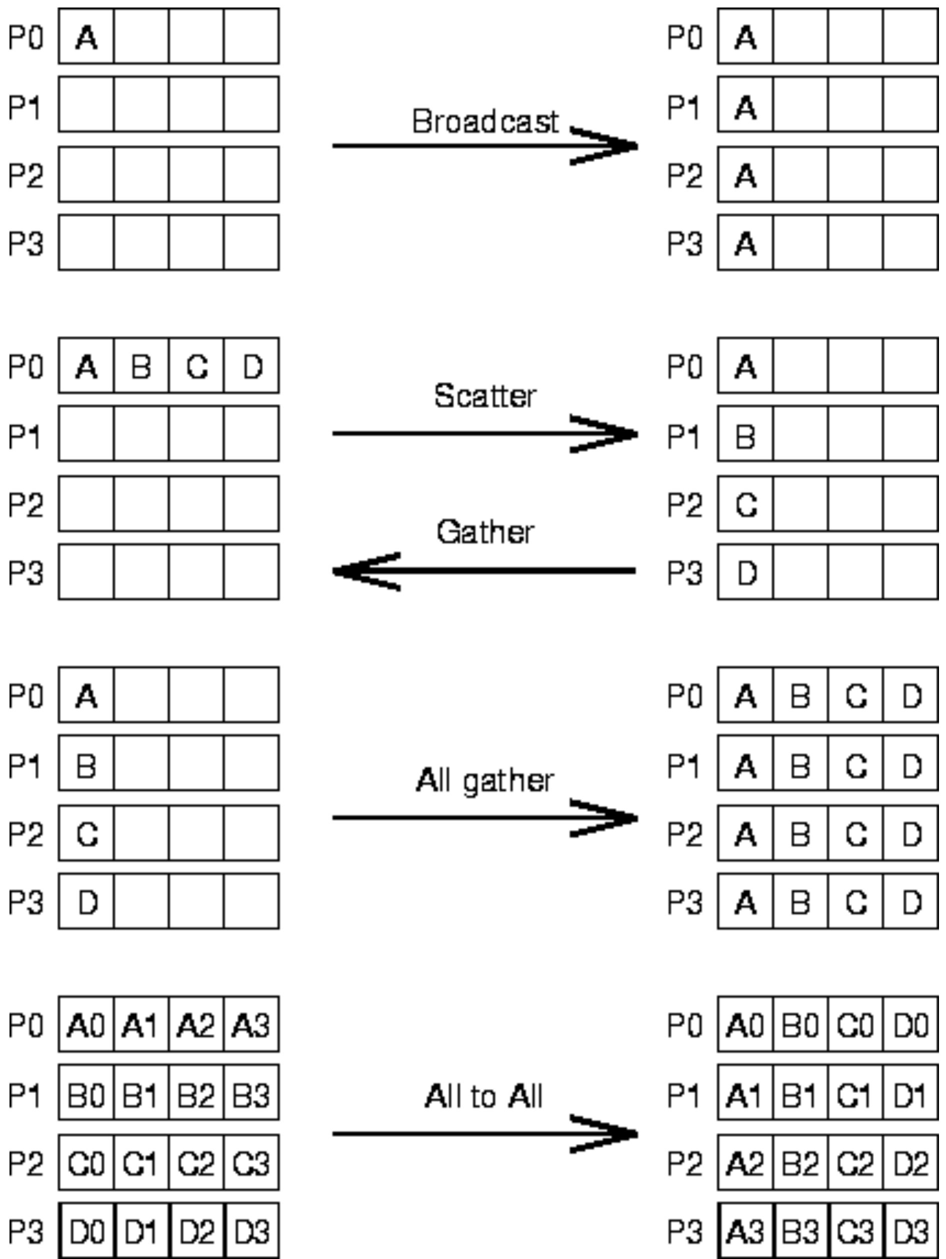
Other collective operations

- Scatter and gather
  - *Gather* all particles on processor
  - Compute interaction forces for particles on that processor

$$\frac{d^2\mathbf{x}_i}{dt^2} = \sum_{j=1}^{N} f(|\mathbf{x_i} - \mathbf{x_j}|),\ i = 1, 2, ..., N$$

- Avoid for big problems (why?)

MPI collective data movement



From *Using MPI*