NTNU – Trondheim
Norwegian University of
Science and Technology

MPI Programming

Henrik R. Nagel
Scientific Computing
IT Division

Outline

- Introduction
- Finite Difference Method
- Finite Element Method
- LU Factorization
- SOR Method
- Monte Carlo Method
- Molecular Dynamics
- MPMD Models

Introduction

- OpenMP
  - can “easily” be added to existing serial programs
  - 1 node only
  - Semi-automatic parallelization
    - High communication bandwidth (memory)
    - Critical variables slow down performance
  - Limited amount of memory available (GB)
- MPI
  - should be use from the start
  - as many nodes as needed
  - Manual parallelization
    - Slow communication bandwidth (network)
    - Programmer has full control
  - All memory is available (TB)

Used MPI Calls

- MPI_SEND, MPI_RECV
- MPI_ISEND, MPI_IRECV, MPI_WAITALL
- MPI_GATHER
- MPI_BCAST, MPI_REDUCE, MPI_ALLREDUCE

Getting Started

- Based on:
  - http://www.redbooks.ibm.com/abstracts/sg245380.html
- Kongull:
  - ssh -Y kongull.hpc.ntnu.no
  - cp -r /home/hrn/Kurs/mpi .
  - module load intelcomp
- Vilje:
  - ssh -Y vilje.hpc.ntnu.no
  - cp -r /home/ntnu/hrn/Kurs/mpi .
  - module load intelcomp

Finite Difference Method

- Example 1 in the source code
- Only a skeleton of a 2D FDM is shown here
- Coefficients and the enclosing loop are omitted
- Data dependencies exist in both dimensions

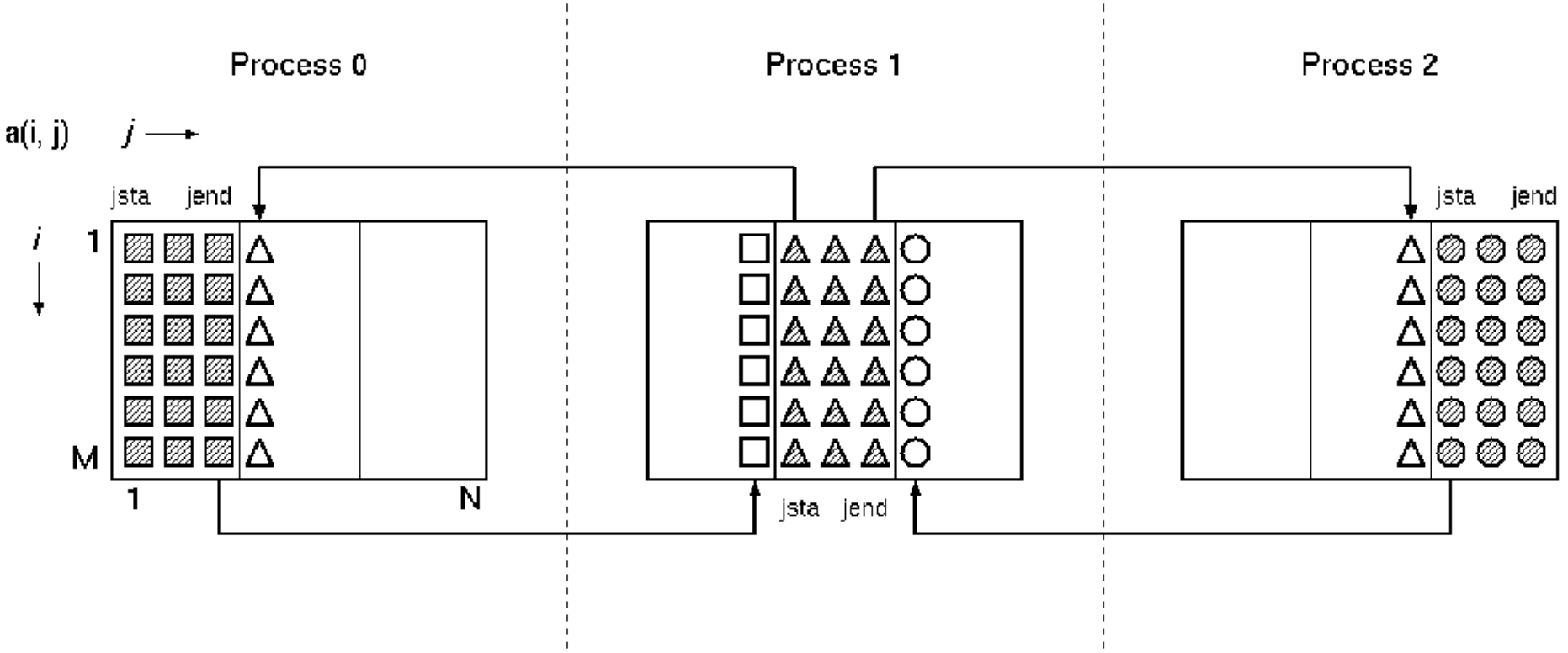
The Sequential Algorithm

```
PROGRAM main
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 6, n = 9)
DIMENSION a(m,n), b(m,n)
DO j = 1, n
  DO i = 1, m
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
DO j = 2, n - 1
  DO i = 2, m - 1
    b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
  ENDDO
ENDDO
END
```

Column-Wise Block Distribution

- Code example: ex1/fdm1.f90
- We must distribute a 2D matrix onto the processes
- Fortran stores arrays in column-major order
- Boundary elements between processes are contiguous in memory
- There are no problems with using MPI_SEND and MPI_RECV

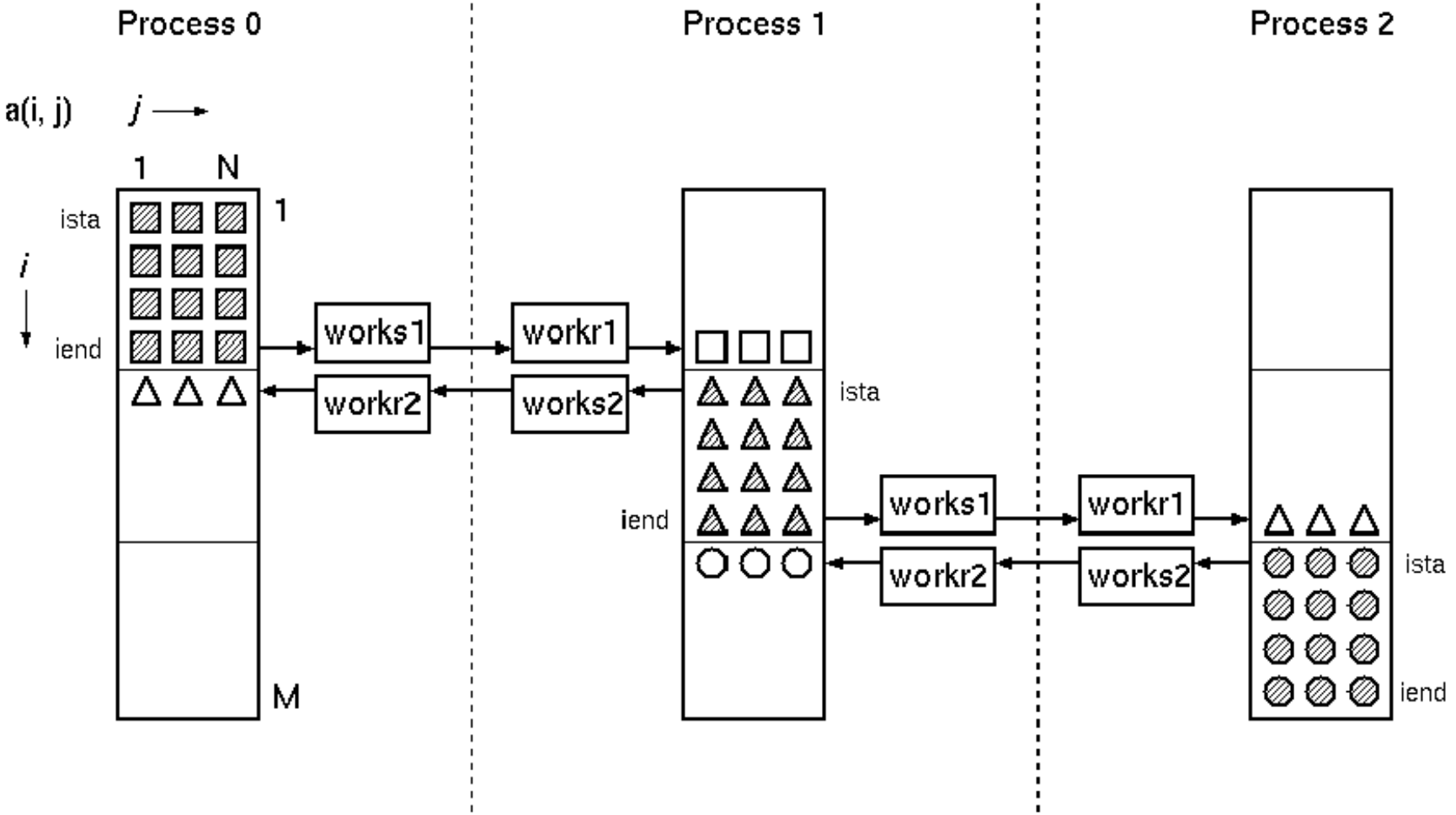
Column-Wise Block Distribution



Row-Wise Block Distribution

- Code example: ex1/fdm2.f90
- Fortran stores arrays in column-major order
- Boundary elements between processes are not contiguous in memory
- Boundary elements can be copied by:
  - Using derived data types
  - Writing code for packing data, sending/receiving it, and then unpacking it

Row-Wise Block Distribution



Block Distribution in Both Dim. (1)

- Code example: ex1/fdm3.f90
- The amount of data transferred might be minimized
  - Depends upon the matrix size and the number of processes
- A process grid itable is prepared for looking up processes quickly

Block Distribution in Both Dim. (1)

a(i, j) $j \longrightarrow$

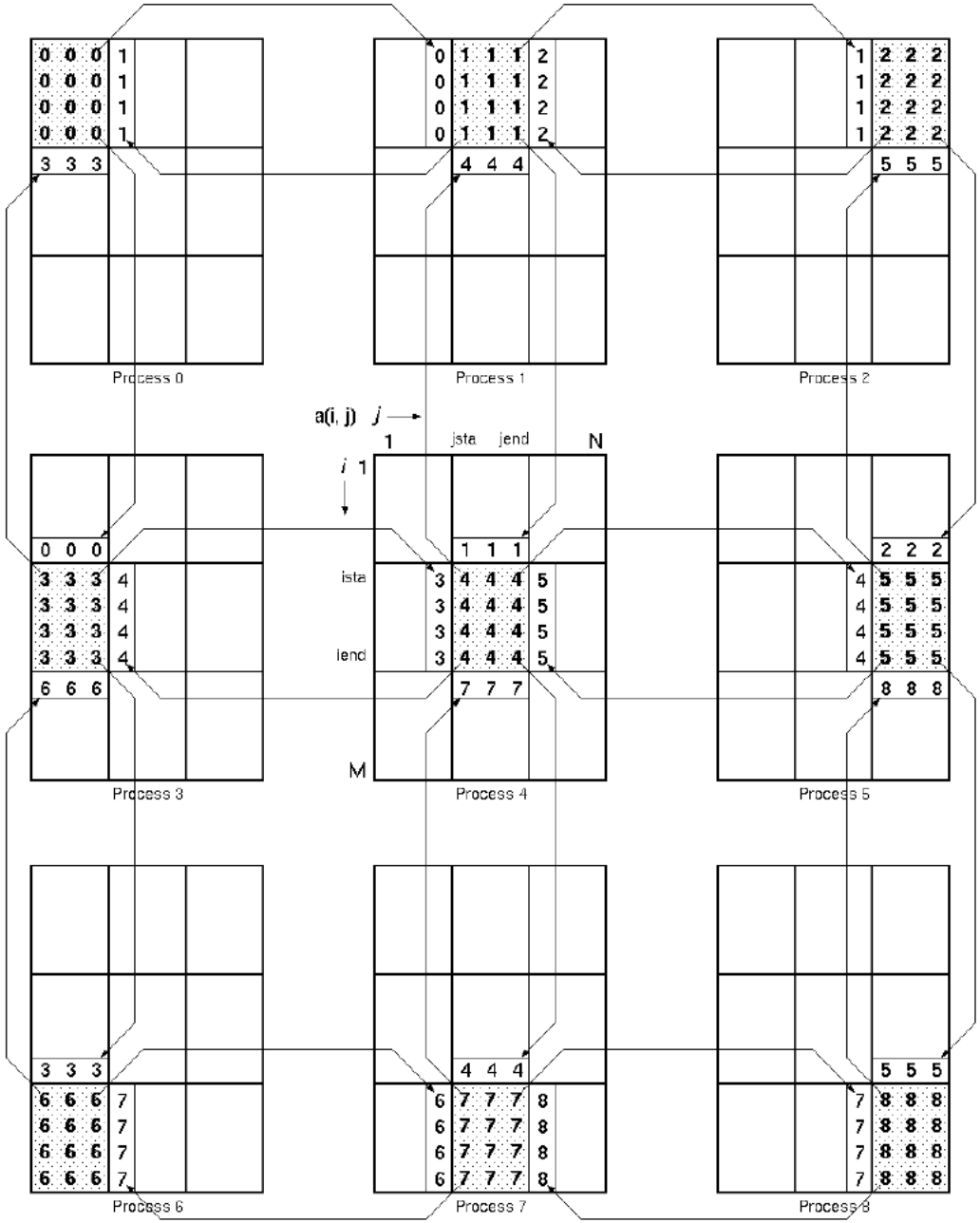
i ↓	1								N
1	0	0	0	1	1	1	2	2	2
	0	0	0	1	1	1	2	2	2
	0	0	0	1	1	1	2	2	2
	0	0	0	1	1	1	2	2	2
	3	3	3	4	4	4	5	5	5
	3	3	3	4	4	4	5	5	5
	3	3	3	4	4	4	5	5	5
	3	3	3	4	4	4	5	5	5
	6	6	6	7	7	7	8	8	8
	6	6	6	7	7	7	8	8	8
	6	6	6	7	7	7	8	8	8
M	6	6	6	7	7	7	8	8	8

(a) The distribution of a()

itable(i, j) $j \longrightarrow$

i ↓	−1	0	1	2	3
−1	null	null	null	null	null
0	null	0	1	2	null
1	null	3	4	5	null
2	null	6	7	8	null
3	null	null	null	null	null

(b) The process grid


Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
a(i, j)
j
i
1
jsta
jend
N
M
ista
iend

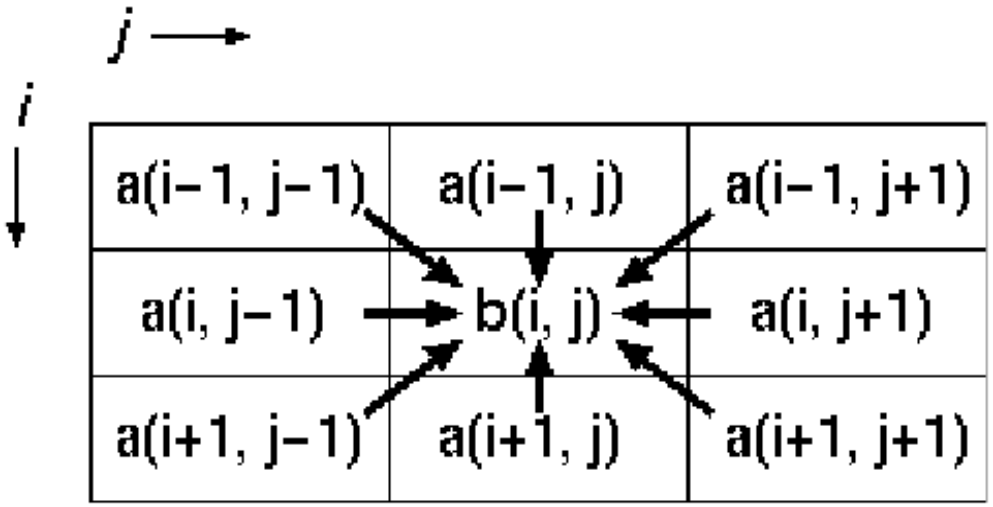
Block Distribution in Both Dim. (2)

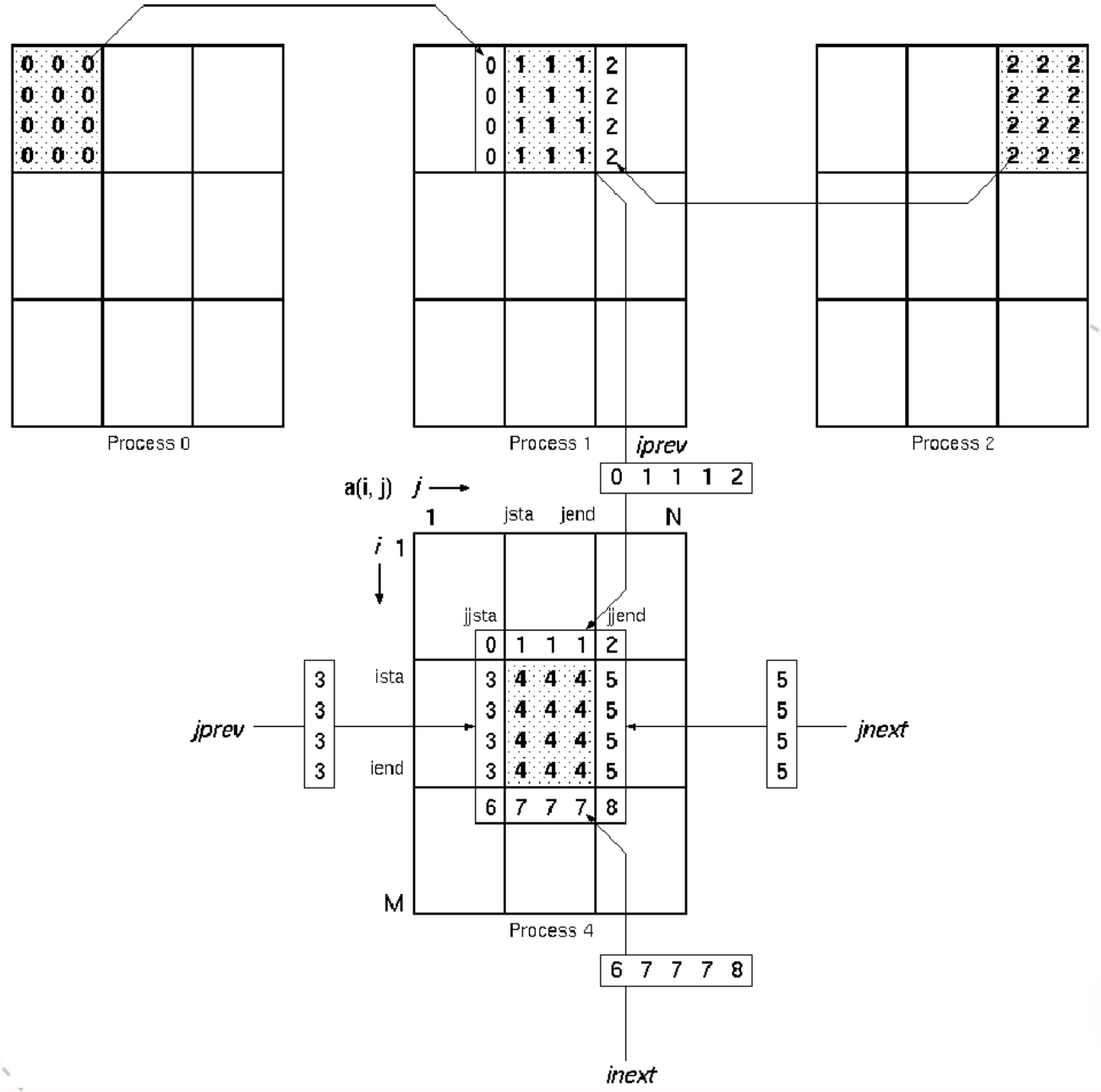
- Code example: ex1/fdm4.f90
- The corner elements are now included
- The data dependencies are therefore more complex

The Sequential Algorithm

```
PROGRAM main
IMPLICIT REAL*8 (a-h,o-z)
PARAMETER (m = 12, n = 9)
DIMENSION a(m,n), b(m,n)
DO j = 1, n
  DO i = 1, m
    a(i,j) = i + 10.0 * j
  ENDDO
ENDDO
DO j = 2, n - 1
  DO i = 2, m - 1
    b(i,j) = a(i-1,j ) + a(i, j-1) + a(i, j+1) + a(i+1,j ) + &
             a(i-1,j-1) + a(i+1,j-1) + a(i-1,j+1) + a(i+1,j+1)
  ENDDO
ENDDO
END
```

The Data Dependency

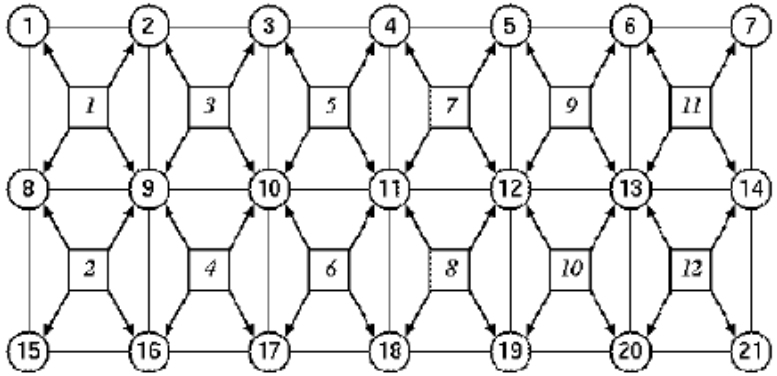



Process 0
Process 1
Process 2
iprev
0 1 1 1 2
a(i, j)
j
1
jsta
jend
N
i 1
jjsta
jjend
ista
iend
jprev
jnext
M
Process 4
6 7 7 7 8
inext

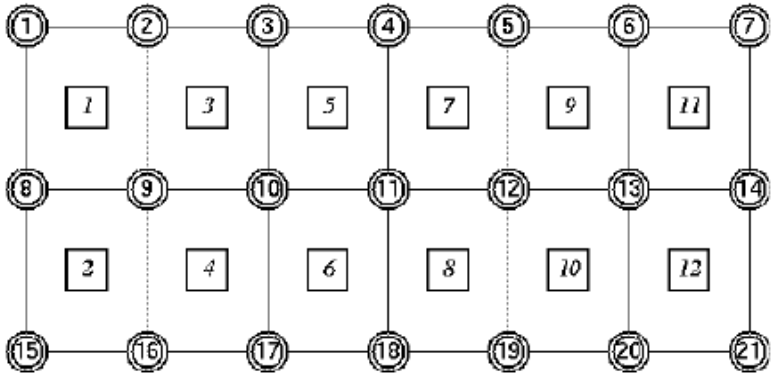
Finite Element Method

- Example 2 in the source code
- A more complete example that produces a result

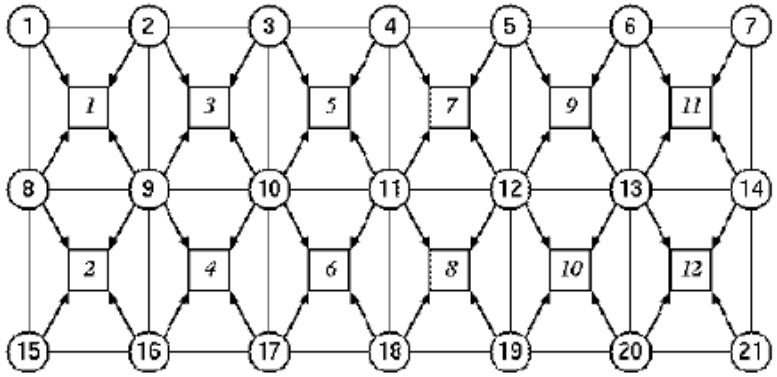
Finite Element Method



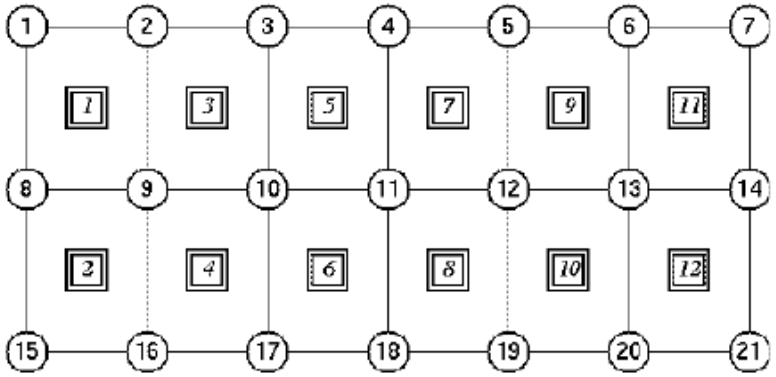
(a) Elements -> Nodes



(b) Update Nodes



(c) Nodes -> Elements

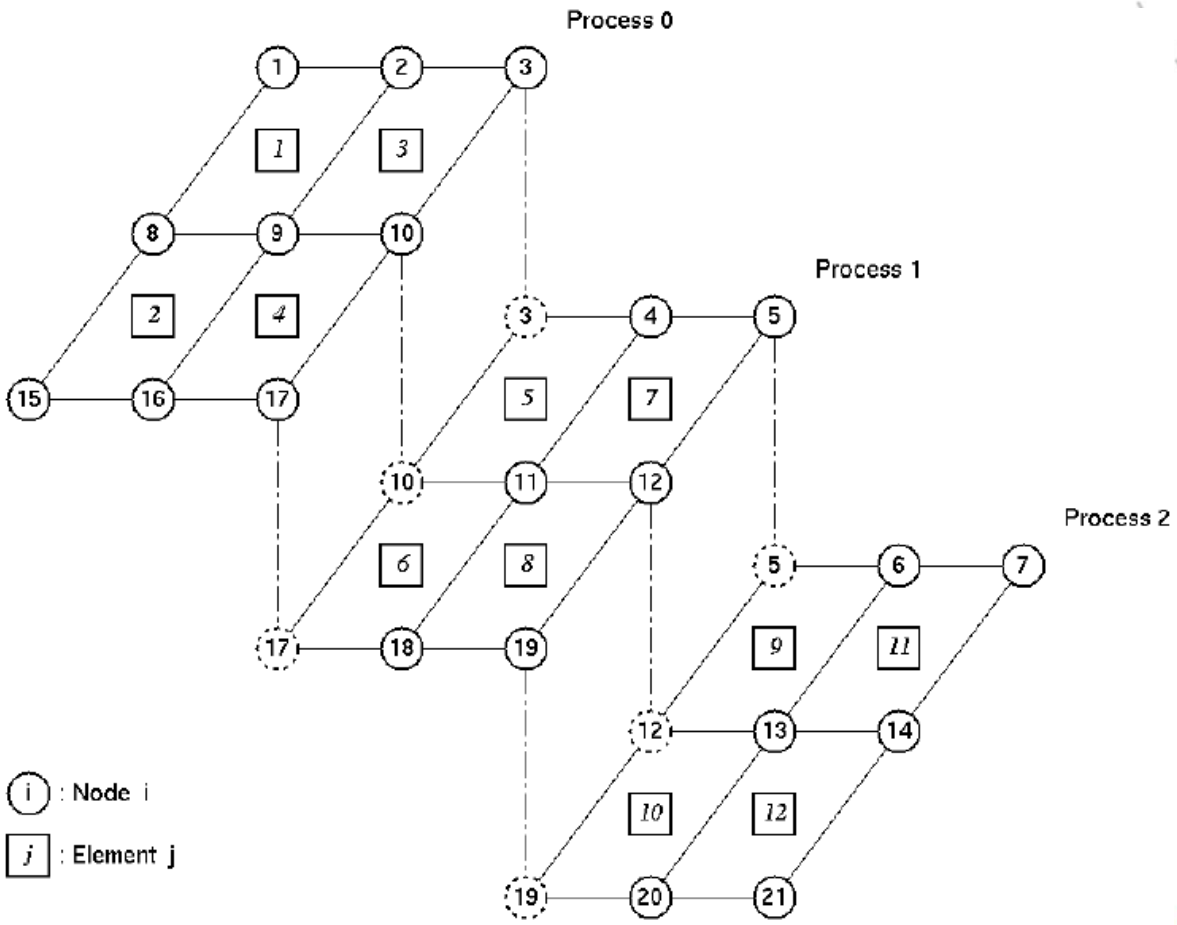


(d) Update Elements

The Sequential Algorithm

```
PARAMETER(iemax = 12, inmax = 21)
REAL*8 ve(iemax), vn(inmax)
INTEGER index(4,iemax)
...
DO ie = 1, iemax
  ve(ie) = ie * 10.0
ENDDO
DO in = 1, inmax
  vn(in) = in * 100.0
ENDDO
DO itime = 1, 10
  DO ie = 1, iemax
    DO j = 1, 4
      vn(index(j,ie)) = vn(index(j,ie)) + ve(ie)
    ENDDO
  ENDDO
  DO in = 1, inmax
    vn(in) = vn(in) * 0.25
  ENDDO
  DO ie = 1, iemax
    DO j = 1, 4
      ve(ie) = ve(ie) + vn(index(j,ie))
    ENDDO
  ENDDO
  DO ie = 1, iemax
    ve(ie) = ve(ie) * 0.25
  ENDDO
ENDDO
PRINT *,'Result',vn,ve
```

Distributing the Data



The Parallel Code

- Example ex2 in the source code
- Differences from original IBM version
  - 2D enumeration (row, column) is used instead of 1D enumeration
  - The amount of memory allocated by each process is minimized
  - A node column is sent to the right
  - An element column is sent to the left

LU Factorization

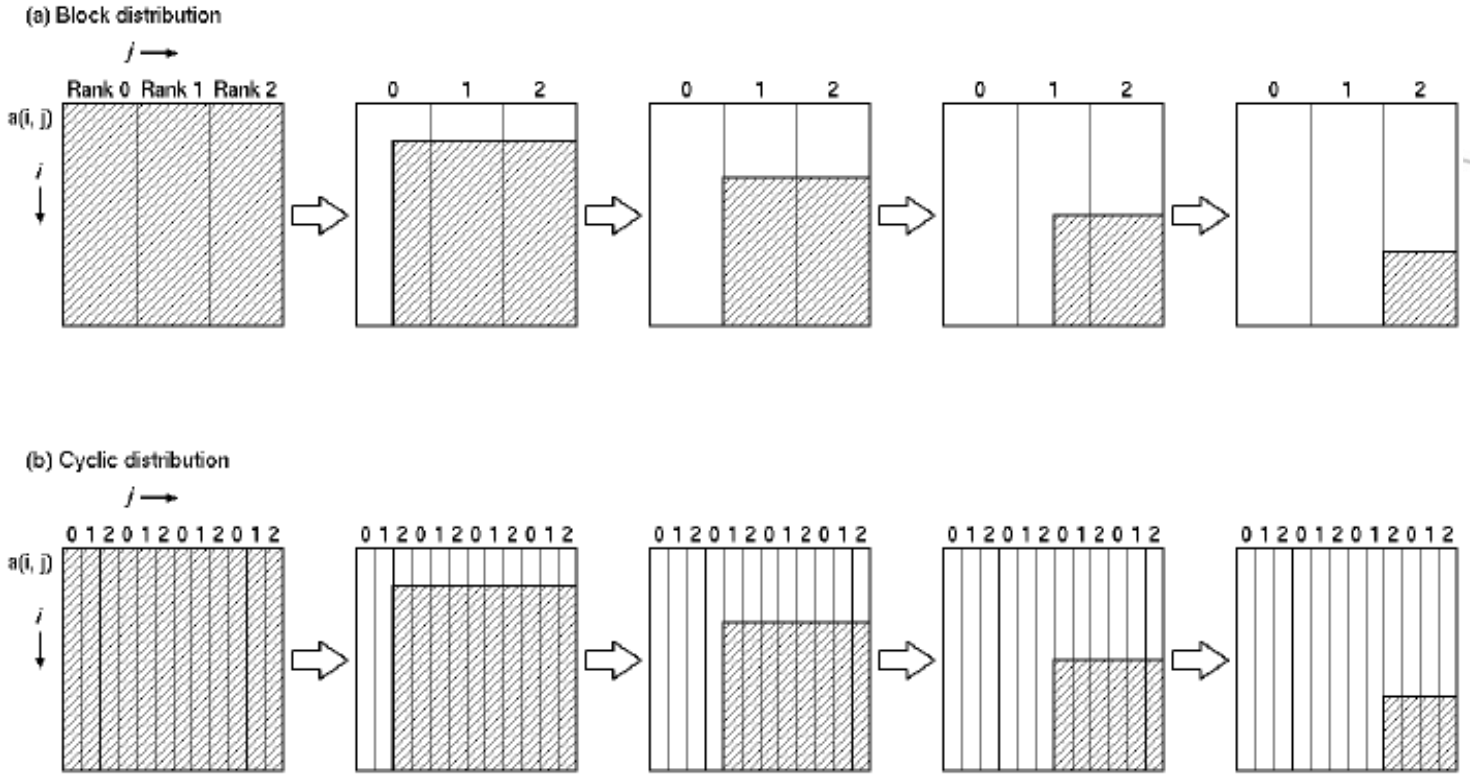
- Example 3 in the source code
- Actually just Gaussian Elimination
- Solving a system of linear equations: Ax = b
- Parallel ESSL has subroutines for LU factorization (outside the scope of this MPI course)
- Pivoting and loop-unrolling is not considered

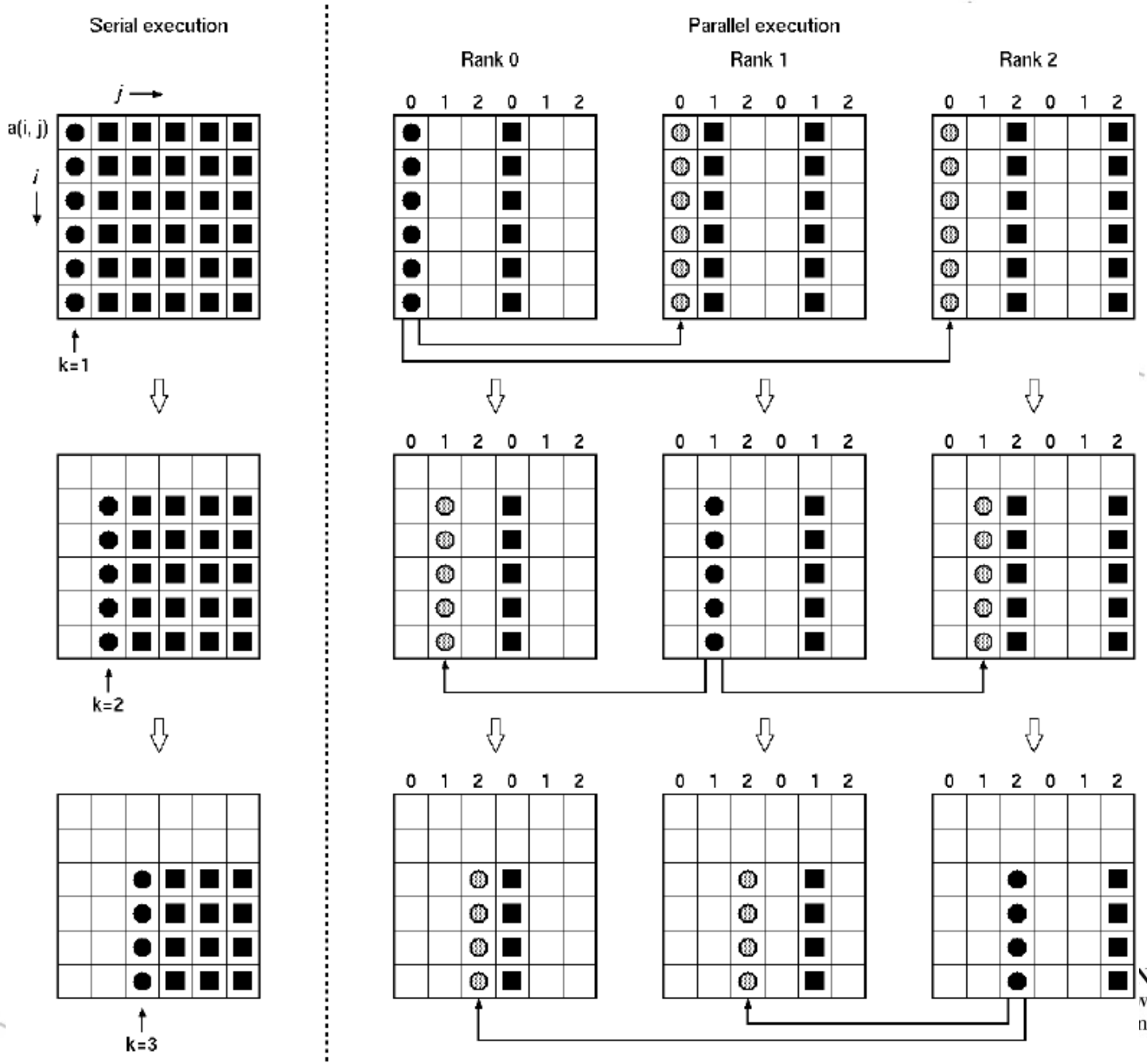
The Sequential Algorithm

```
PROGRAM main
PARAMETER (n = ...)
REAL a(n,n), b(n)
...
! LU factorization
DO k = 1, n-1
  DO i = k+1, n
    a(i,k) = a(i,k) / a(k,k)
  ENDDO
  DO j = k+1, n
    DO i = k+1, n
      a(i,j) = a(i,j) - a(i,k) * a(k,j)
    ENDDO
  ENDDO
ENDDO
```

```
! Forward elimination
DO i = 2, n
  DO j = 1, i - 1
    b(i) = b(i) - a(i,j) * b(j)
  ENDDO
ENDDO
! Backward substitution
DO i = n, 1, -1
  DO j = i + 1, n
    b(i) = b(i) - a(i, j) * b(j)
  ENDDO
  b(i) = b(i) / a(i,i)
ENDDO
...
END
```

Distributing the Data

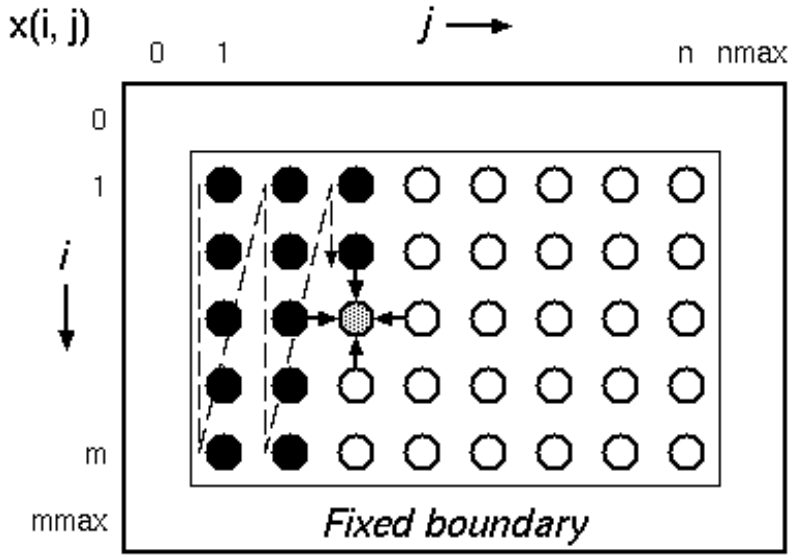



Serial execution
Parallel execution
Rank 0
Rank 1
Rank 2
a(i, j)
j
i
k=1
k=2
k=3
0 1 2 0 1 2

The SOR Method

- Example 4 in the source code
- Solving the 2D Laplace equation using the Successive Over-Relaxation (SOR) method

The Basic SOR Method



● : Updated in the current iteration

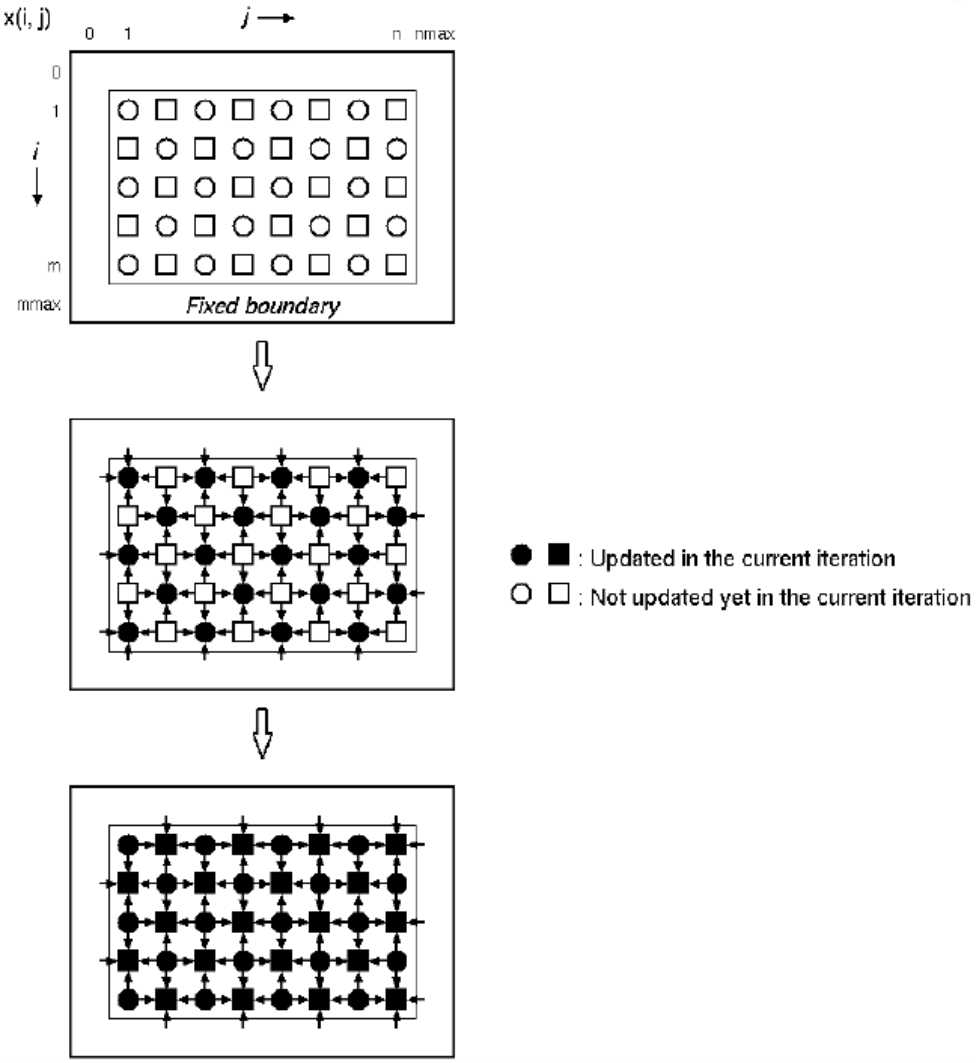
◍ : About to be updated

○ : Not updated yet in the current iteration

The Sequential Algorithm

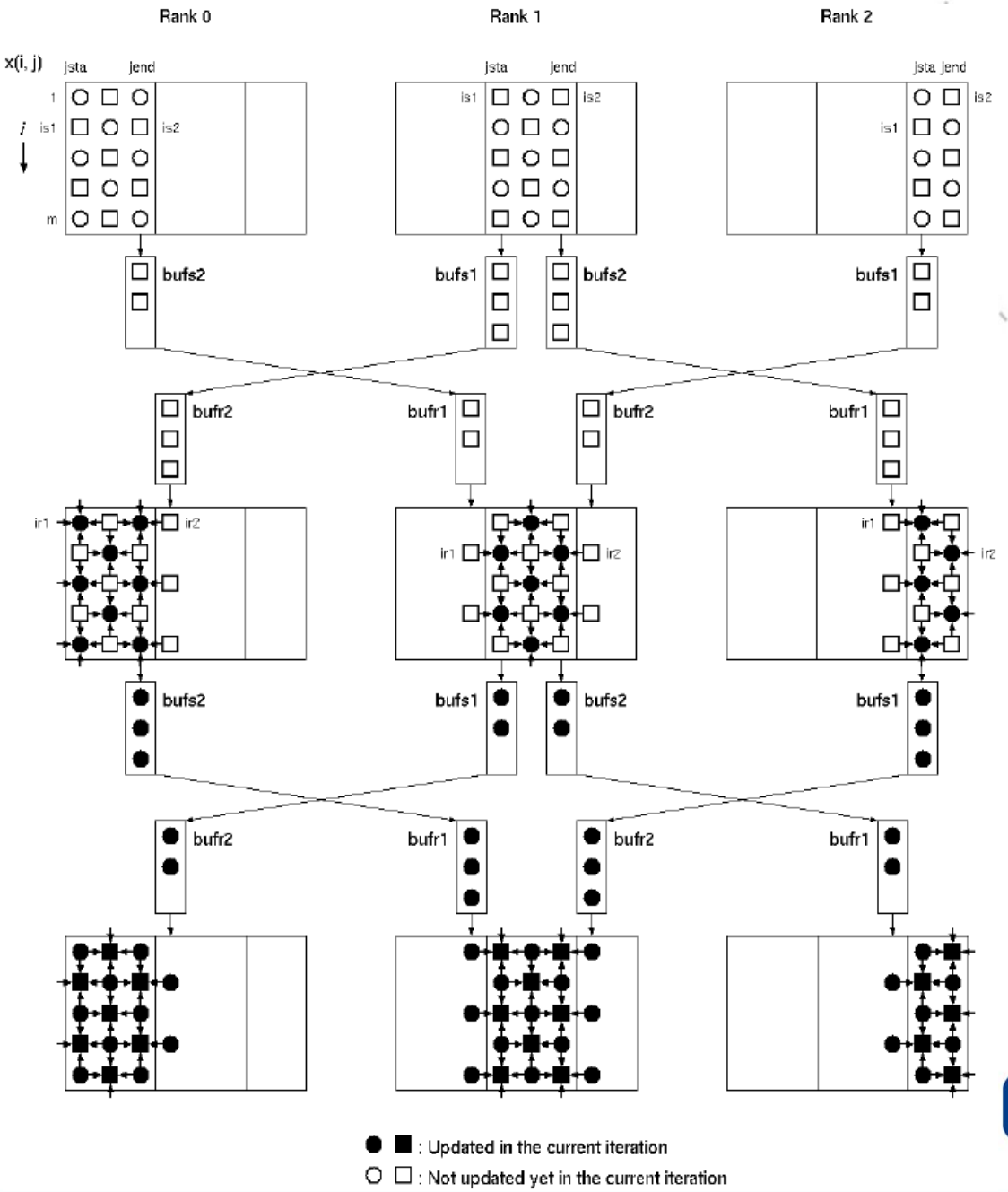
```
PROGRAM sor
PARAMETER (mmax = 6, nmax = 9)
PARAMETER (m = mmax - 1, n = nmax - 1)
REAL x(0:mmax, 0:nmax)
...
DO k = 1, 300    ! iterations
  err1 = 0.0
  DO j = 1, n     ! columns
    DO i = 1, m   ! rows
      temp = 0.25 * (x(i,j-1) + x(i-1,j) + x(i+1,j) + x(i,j+1)) - x(i,j)
      x(i,j) = x(i,j) + omega * temp
      IF (abs(temp) > err1) err1 = abs(temp)
    ENDDO
  ENDDO
  IF (err1 <= eps) exit
ENDDO
...
END
```

The Red-Black SOR Method



The Sequential Algorithm

```
DO k = 1, 300    ! iterations
  err1 = 0.0
  DO j = 1, n    ! columns
    DO i = 1 + MOD(j+1,2), m, 2    ! circles on rows
      temp = 0.25 * (x(i,j-1) + x(i-1,j) + x(i+1,j) + x(i,j+1)) - x(i,j)
      x(i,j) = x(i,j) + omega * temp
      IF (abs(temp) > err1) err1 = abs(temp)
    ENDDO
  ENDDO
  DO j = 1, n    ! columns
    DO i = 1 + MOD(j,2), m, 2    ! squares on rows
      temp = 0.25 * (x(i,j-1) + x(i-1,j) + x(i+1,j) + x(i,j+1)) - x(i,j)
      x(i,j) = x(i,j) + omega * temp
      IF (abs(temp) > err1) err1 = abs(temp)
    ENDDO
  ENDDO
  IF (err1 <= eps) exit
ENDDO
```

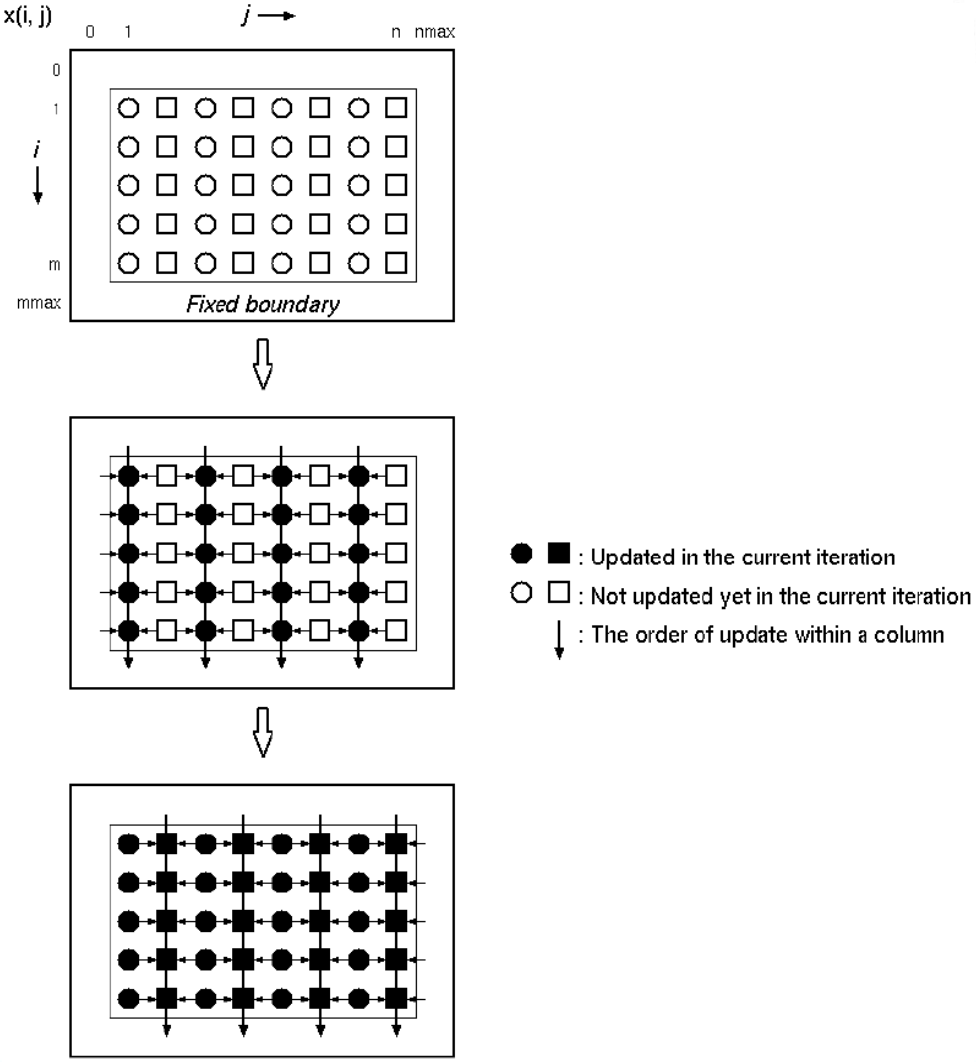

Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
i
1
m
is1
is2
bufs2
bufs1
bufr2
bufr1
ir1
ir2
● ■ : Updated in the current iteration
○ □ : Not updated yet in the current iteration

The Parallel Code

- Go to the ex4/version1 directory
- Look at main.f90 and grid.f90

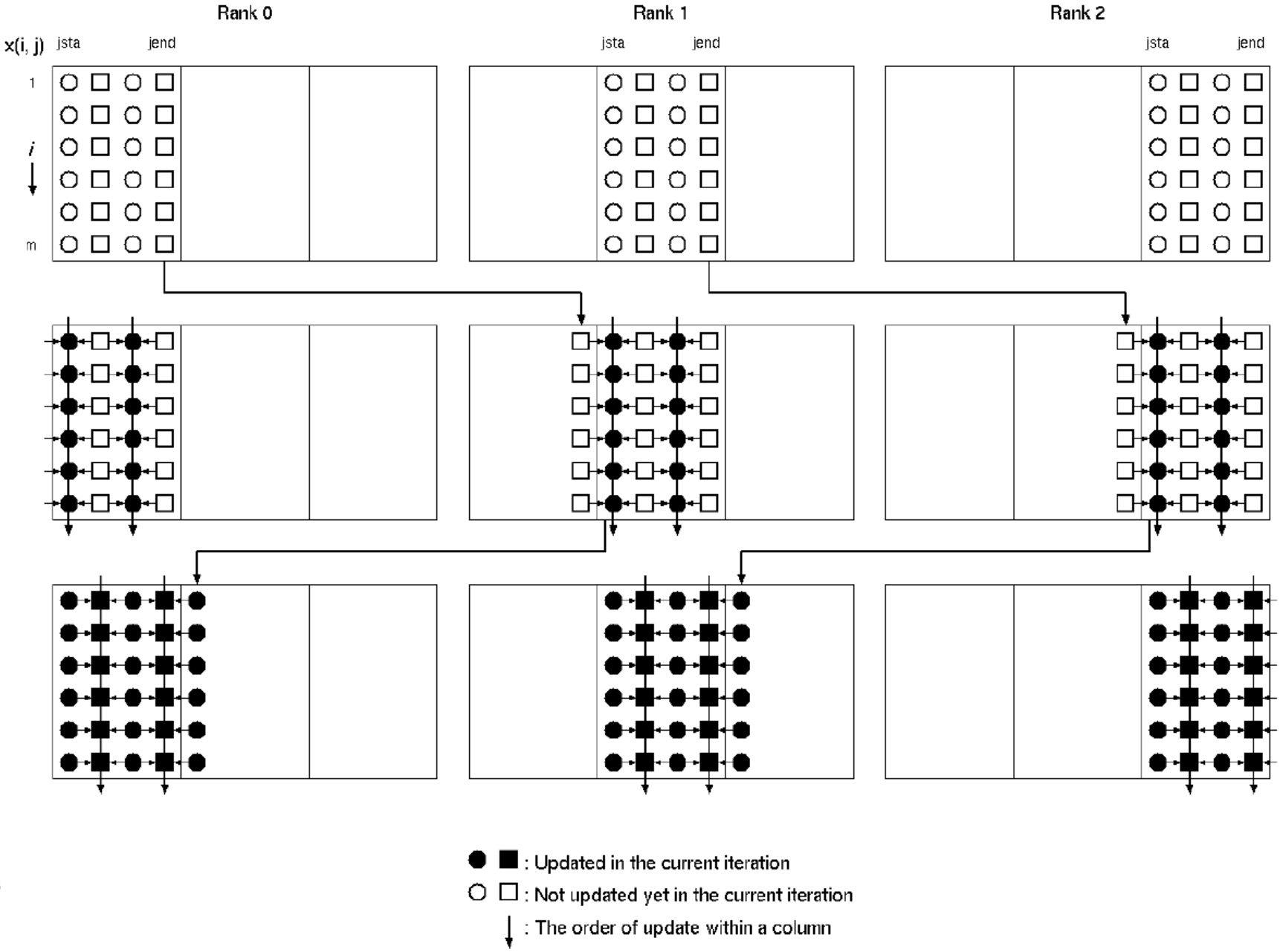
- The data is distributed among the processes
- It is not clear whether the local block of data starts with a circle or a square
  - The variables **even** and **odd** keep track of whether the global column number of the first column in the local data is odd or even

The Zebra SOR Method



The Sequential Algorithm

```
DO k = 1, 300
  err1 = 0.0
  DO j = 1, n, 2
    DO i = 1, m
      Update x(i,j) and err1
    ENDDO
  ENDDO
  DO j = 2, n, 2
    DO i = 1, m
      Update x(i,j) and err1
    ENDDO
  ENDDO
  IF (err1 <= eps) EXIT
ENDDO
```

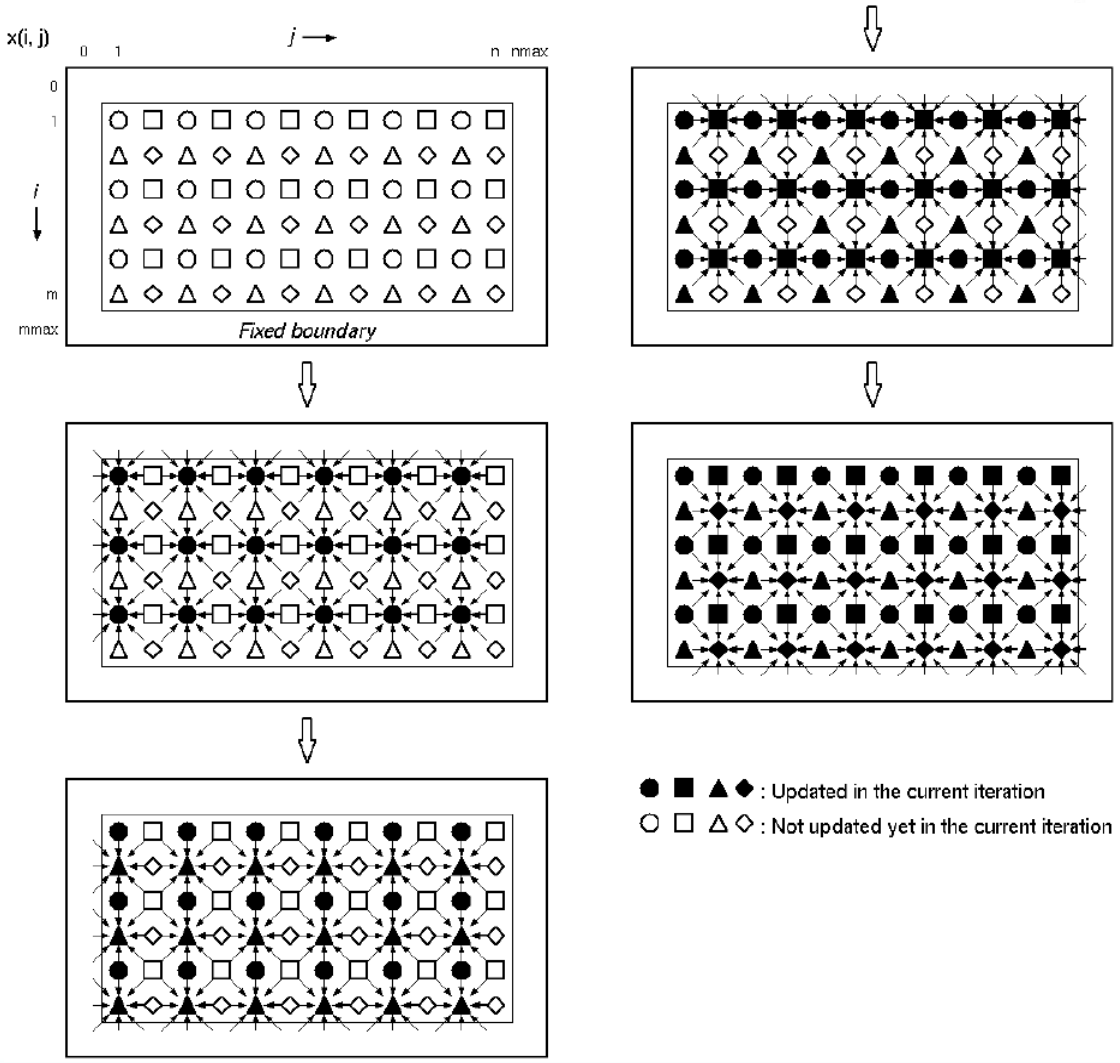

Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
1
i
m
: Updated in the current iteration
: Not updated yet in the current iteration
: The order of update within a column

n
of
gy

The Parallel Code

- Go to the ex4/version2 directory
- Look at main.f90 and grid.f90

The Four-Colour SOR Method

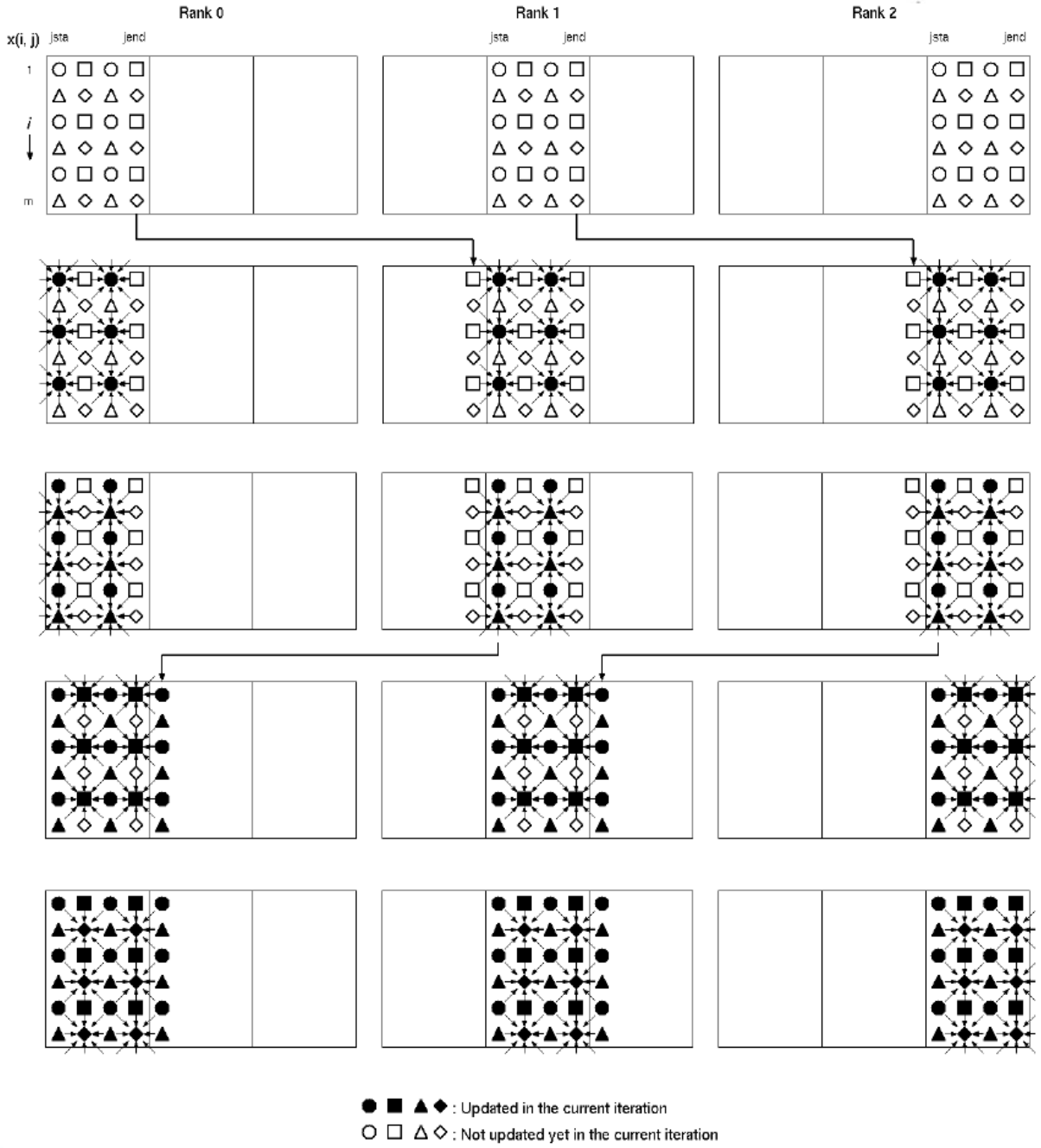


The Sequential Algorithm

```
PROGRAM fourcolor
PARAMETER (mmax = ..., nmax = ...)
PARAMETER (m = mmax - 1, n = nmax - 1)
DIMENSION x(0:mmax, 0:nmax)
...
DO k = 1, 300
  err1 = 0.0
  DO j = 1, n, 2
    DO i = 1, m, 2
      Update x(i,j) and err1
    ENDDO
  ENDDO
  DO j = 1, n, 2
    DO i = 2, m, 2
      Update x(i,j) and err1
    ENDDO
  ENDDO
  DO j = 2, n, 2
    DO i = 1, m, 2
      Update x(i,j) and err1
    ENDDO
  ENDDO
  DO j = 2, n, 2
    DO i = 2, m, 2
      Update x(i,j) and err1
    ENDDO
  ENDDO
  IF (err1 <= eps) EXIT
ENDDO
...
END
```

8 Neighbours are Involved

```
DO j = 1, n, 2
  DO i = 1, m, 2
    temp = 0.125 * (x(i, j-1) + x(i-1,j) + x(i+1,j) + x(i, j+1) + &
                    x(i-1,j-1) + x(i+1,j-1) + x(i-1,j+1) + x(i+1,j+1)) - x(i,j)
    x(i,j) = x(i,j) + omega * temp
    IF (abs(temp) > err1) err1 = abs(temp)
  ENDDO
ENDDO
```


Rank 0
Rank 1
Rank 2
x(i, j)
jsta
jend
1
i
m
: Updated in the current iteration
: Not updated yet in the current iteration

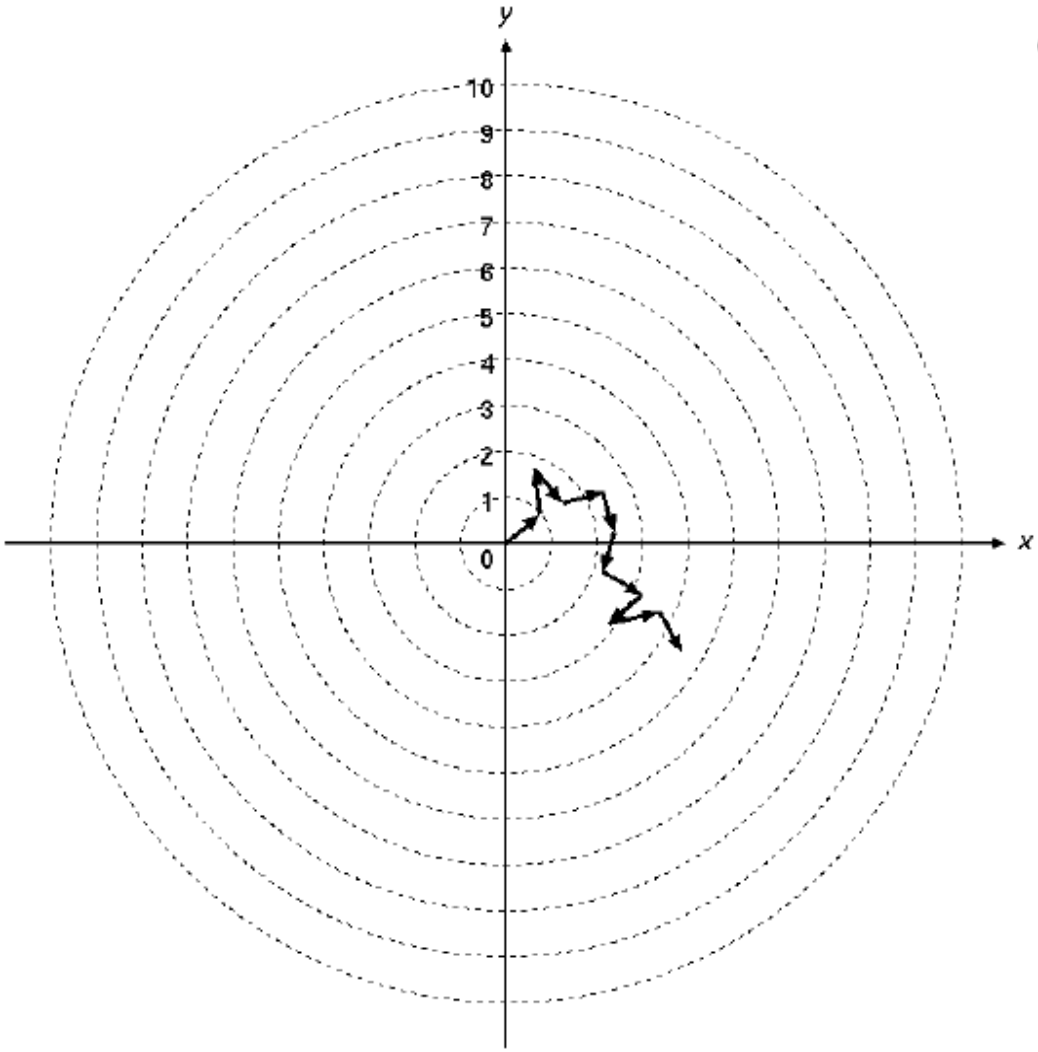
The Parallel Code

- Go to the ex4/version3 directory
- Look at main.f90 and grid.f90

The Monte Carlo Method

- Example 5 in the source code
- A random walk in 2D
- 100,000 particles
- 10 steps

A Sample Trajectory



The Sequential Algorithm

```
PROGRAM main
PARAMETER (n = 100000)
INTEGER itotal(0:9)
REAL seed
pi = 3.1415926
DO i = 0, 9
   itotal(i) = 0
ENDDO
seed = 0.5
CALL srand(seed)
DO i = 1, n
   x = 0.0
   y = 0.0
   DO istep = 1, 10
      angle = 2.0 * pi * rand()
      x = x + cos(angle)
      y = y + sin(angle)
   ENDDO
   itemp = sqrt(x**2 + y**2)
   itotal(itemp) = itotal(itemp) + 1
ENDDO
PRINT *,'total =',itotal
END
```

The Parallel Code

- Go to the directory ex5
- The sequential version is in directory version1
- The parallel version is in directory version2

Molecular Dynamics

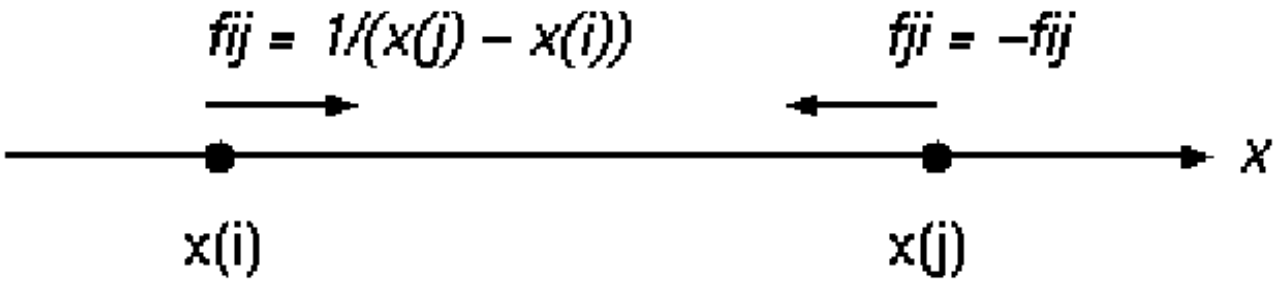
- Example 6 in the source code
- N particles interact in 1 dimension
- The force on particle i from particle j is given by

  $f_{ij} = 1/(x_j - x_i)$

- The law of action and reaction applies:

  $f_{ij} = f_{ji}$

Forces in 1D



Forces Acting on 7 Particles

f(1) =		+f12	+f13	+f14	+f15	+f16	+f17
f(2) =	−f12		+f23	+f24	+f25	+f26	+f27
f(3) =	−f13	−f23		+f34	+f35	+f36	+f37
f(4) =	−f14	−f24	−f34		+f45	+f46	+f47
f(5) =	−f15	−f25	−f35	−f45		+f56	+f57
f(6) =	−f16	−f26	−f36	−f46	−f56		+f67
f(7) =	−f17	−f27	−f37	−f47	−f57	−f67	

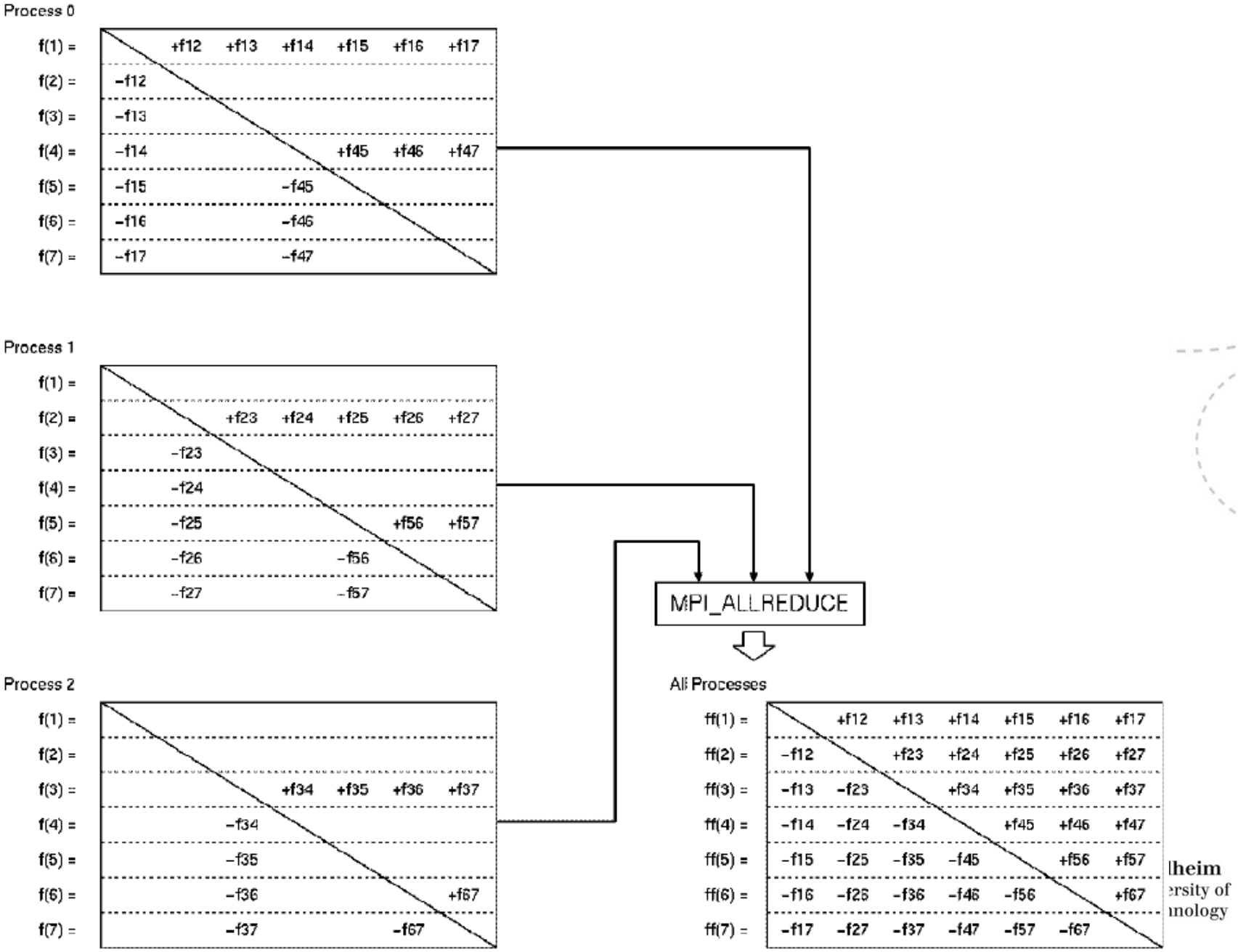
The Sequential Algorithm

```
PARAMETER (n = ...)
REAL f(n), x(n)
...
DO itime = 1, 100
  DO i = 1, n
    f(i) = 0.0
  ENDDO
  DO i = 1, n-1
    DO j = i+1, n
      fij = 1.0 / (x(j)-x(i))
      f(i) = f(i) + fij
      f(j) = f(j) - fij
    ENDDO
  ENDDO
  DO i = 1, n
    x(i) = x(i) + f(i)
  ENDDO
ENDDO
```

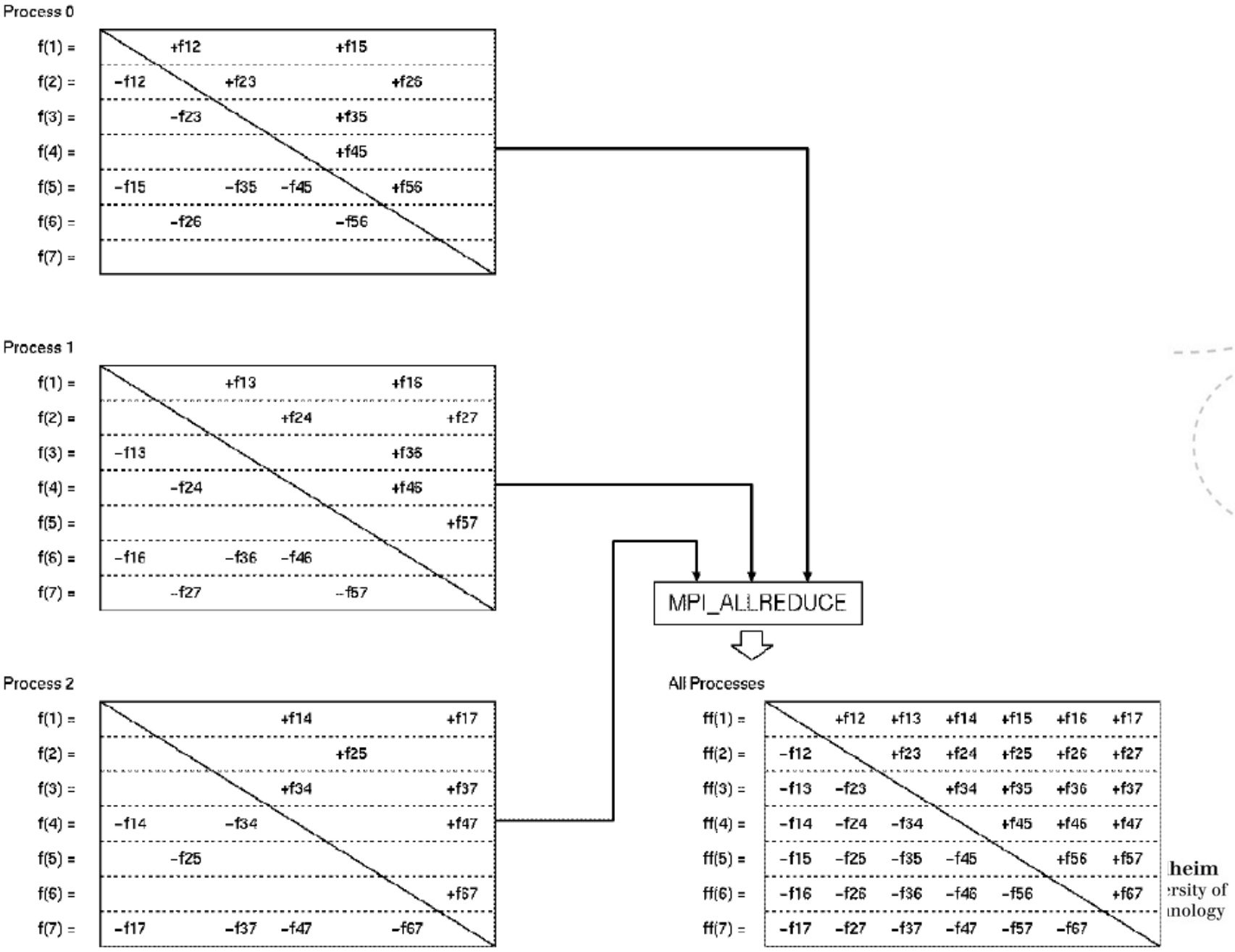
Two Parallelisation Methods

- Cyclic distribution of the outer loop
- Cyclic distribution of the inner loop

```
DO i = 1, n-1
  DO j = i+1, n
    fij = 1.0 / (x(j)-x(i))
    f(i) = f(i) + fij
    f(j) = f(j) - fij
  ENDDO
ENDDO
```



lheim
ersity of
nology

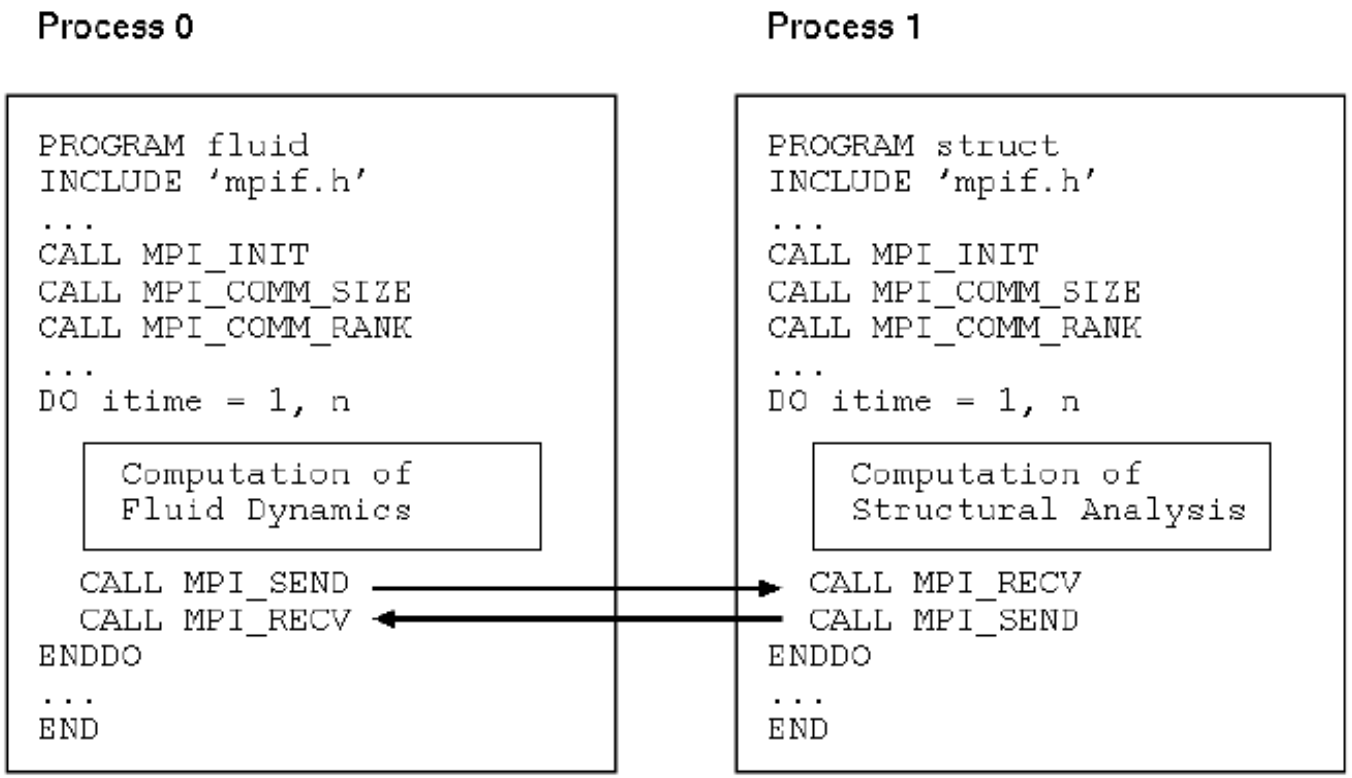


heim
ersity of
nology

MPMD Models

- Example 7 in the source code
- Multiple Programs Multiple Data
- Different programs run in parallel and communicate with each other

MPMD Model


Process 0
PROGRAM fluid
INCLUDE 'mpif.h'
...
CALL MPI_INIT
CALL MPI_COMM_SIZE
CALL MPI_COMM_RANK
...
DO itime = 1, n
Computation of
Fluid Dynamics
CALL MPI_SEND
CALL MPI_RECV
ENDDO
...
END
Process 1
PROGRAM struct
INCLUDE 'mpif.h'
...
CALL MPI_INIT
CALL MPI_COMM_SIZE
CALL MPI_COMM_RANK
...
DO itime = 1, n
Computation of
Structural Analysis
CALL MPI_RECV
CALL MPI_SEND
ENDDO
...
END

Master/Worker Programs

- The master coordinates the execution of all the other processes
- The master has a list of jobs that must be processed
- Suitable if:
  - The processing time varies greatly from job to job
  - Neither block nor cyclic distribution gives a good load balancing
  - A heterogeneous environment where the performance of the machines is not uniform

More Information

- All examples are based on:
  - http://www.redbooks.ibm.com/abstracts/sg245380.html
- Our Web-site:
  - http://www.hpc.ntnu.no/
- Send an e-mail to:
  - support-ntnu@notur.no
  - support-kongull@hpc.ntnu.no