



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# **MPI Programming**

Henrik R. Nagel  
IT Development Section  
IT Division

# Outline

- Introduction
- Basic MPI programming
- Examples
  - Finite Difference Method
  - Finite Element Method
  - LU Factorization
  - Monte Carlo Method
  - Molecular Dynamics
  - MPMD Models

# Introduction

# Acknowledgments

- Thanks to Professor Lasse Natvig and Associate Professor Jørn Amundsen from IDI, NTNU for allowing me to copy from their “Parallel Programming” lecture slides
- Thanks to IBM for allowing me to copy from their MPI redbook

# The Examples

- No exercises, but 7 working examples are explained
  - The last 6 are larger examples
- Course materials:
  - <https://www.hpc.ntnu.no/course-materials/>

# Basic MPI Programming



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# MPI Programs in C

- A C program
  - Has a main() function
  - Includes stdio.h, string.h, etc.
- Need to include **mpi.h** header file
- Identifiers defined by MPI start with “MPI\_”
- First letter following underscore is uppercase
  - For function names and MPI-defined types
  - Helps to avoid confusion

# MPI Programs in Fortran

- include 'mpif.h'
  - No argument checking! Don't use it.
- use mpi
  - Provide explicit interfaces for all MPI routines → compile time argument checking
- use mpi\_f08
  - Fully Fortran 2008 compatible definition of all MPI routines
  - New syntax TYPE(\*), DIMENSION(...) to define choice buffers in a standardized way



# Identifying MPI Processes

- Common practice is to identify processes by non-negative integer ranks
- $p$  processes are numbered  $0, 1, 2, \dots, p-1$
- This can be:
  - $p$  processes distributed over  $p$  processors (“physical parallelism”)
  - $p$  processes running time-multiplexed on a single processor (“logical parallelism”)

# Example 1: Hello, World!

- Change to the directory:  
`$ cd mpi/ex1`
- Compile the code:  
`$ make`  
`cc -O2 mpi_hello.c -o mpi_hello`
- Edit the job script:  
`$ vim run.slurm (change ACCOUNT)`
- Run the job  
`$ sbatch run.slurm`

# MPI Start and End

- `MPI_Init()`
  - Tells MPI to do all the necessary setup

```
int MPI_Init(  
    int*   argc_p  
    char*** argv_p);
```

Pointers to the arguments  
to main: argc & argv

- `MPI_Finalize()`
  - Tells MPI we're done, so clean up anything allocated

```
int MPI_Finalize(void);
```

# Basic Outline

```
...  
#include <mpi.h>  
...  
int main(int argc, char* argv[]) {  
    ...  
    /* No MPI calls before this */  
    MPI_Init(&argc, &argv);  
    ...  
    MPI_Finalize();  
    /* No MPI calls after this */  
    ...  
    return 0;  
}
```

# Communicators

- A collection of processes that can send messages to each other
- **MPI\_Init()** defines a communicator that consists of all the processes created when the program is started
  - **MPI\_COMM\_WORLD**

# Communicators

```
int MPI_Comm_size(  
    MPI_Comm comm /* in */,  
    int* size /* out */);
```

number of processes in the communicator

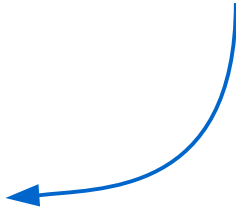
```
int MPI_Comm_rank(  
    MPI_Comm comm /* in */,  
    int* rank /* out */);
```

my rank  
(the process making this call)

# Communication

```
int MPI_Send(  
    void*      buf  
    int        count  
    MPI_Datatype datatype  
    int        dest  
    int        tag  
    MPI_Comm   communicator  
    /* in */,  
    /* in */,  
    /* in */,  
    /* in */,  
    /* in */,  
    /* in */);
```

number of elements in  
the send buffer



# Data Types

MPI datatype

MPI\_CHAR

MPI\_SHORT

MPI\_INT

MPI\_LONG

MPI\_LONG\_LONG

MPI\_UNSIGNED\_CHAR

MPI\_UNSIGNED\_SHORT

MPI\_UNSIGNED

MPI\_UNSIGNED\_LONG

MPI\_FLOAT

MPI\_DOUBLE

MPI\_LONG\_DOUBLE

MPI\_BYTE

MPI\_PACKED

C datatype

signed **char**

signed **short int**

signed **int**

signed **long int**

signed **long long int**

unsigned **char**

unsigned **short int**

unsigned **int**

unsigned **long int**

**float**

**double**

**long double**

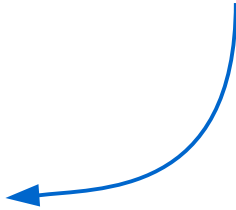




# Communication

max. number of elements  
to receive

```
int MPI_Recv(  
    void*      buf  
    int        count  
    MPI_Datatype datatype  
    int        source  
    int        tag  
    MPI_Comm   communicator  
    MPI_Status* status  
    /* out */,  
    /* in */,  
    /* in */,  
    /* in */,  
    /* in */,  
    /* in */,  
    /* out */);
```



# Message Matching

- Process **q** calls MPI\_Send()

```
MPI_Send(send_buf, send_count, send_type, dest,  
         send_tag, send_comm);
```

- Process **r** calls MPI\_Recv()

```
MPI_Recv(recv_buf, recv_count, recv_type, src,  
         recv_tag, recv_comm, &status);
```

**r****q**

# Receiving Messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message
    - MPI\_ANY\_SOURCE
  - or the tag of the message
    - MPI\_ANY\_TAG

# The Status Argument

- If MPI\_ANY\_SOURCE or MPI\_ANY\_TAG have been used, you can get help from MPI\_Status

```
MPI_Recv(buf, count, datatype, src  
         tag, comm, &status);
```

```
MPI_Status status;
```

```
status.MPI_SOURCE
```

```
status.MPI_TAG
```

# How much data am I receiving?

```
int MPI_Probe(  
    int          source      /* in */,  
    int          tag        /* in */,  
    MPI_Comm     comm       /* in */,  
    MPI_Status*  status     /* out */);
```

```
int MPI_Get_count(  
    MPI_Status*  status      /* in */,  
    MPI_Datatype datatype    /* in */,  
    int*         count       /* out */);
```

# Issues with Send and Receive

- Exact behavior is determined by the MPI implementation
- `MPI_Send()` is blocking as defined in the standard, but is non-blocking up to a certain message size in most implementations
  - `MPI_Ssend()` might be used to force blocking until a receive is posted
  - `MPI_Bsend()` can be used with a user defined send buffer - then always non-blocking
- `MPI_Recv()` always blocks until a matching message is received

# Issues with Send and Receive

- AND, ...
  - MPI programs will easily hang!
    - A receive without corresponding send
    - A send without corresponding receive
    - Or deadlock
      - Circular waiting

# Non-Blocking Communication

```
int MPI_Isend(void* buffer, int count, MPI_Datatype  
             datatype, int destination, int tag, MPI_Comm comm,  
             MPI_Request* request);
```

```
int MPI_Irecv(void* buffer, int count, MPI_Datatype  
             datatype, int source, int tag, MPI_Comm comm,  
             MPI_Request* request);
```

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
```

```
int MPI_Waitall(int array_size,  
               MPI_Request requests[], MPI_Status statuses[]);
```

I = Immediate



# Non-Blocking Communication

- Immediate-mode **MPI\_Isend()** and **MPI\_Irecv()** only start the data copy operation
- **MPI\_Wait()** and **MPI\_Waitall()** are used to complete the operations
- Useful in complicated send-receive situations (e.g. 2D grid of processes)
- Calculations can take place between those two calls
  - Difficult to make good use of
  - Communication and calculation at the same time is more efficient

# Global Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count  
               MPI_Datatype datatype, MPI_Op op, int root,  
               MPI_Comm comm);
```

- **op** determines which global reduction to perform
- Predefined reductions for the most used types, like **MPI\_MAX**, **MPI\_MIN**, **MPI\_SUM**, **MPI\_PROD**, etc.
- Also possible to specify user defined reduction operations with `MPI_Op_create()`
- **MPI\_IN\_PLACE** specified for sendbuf at rank **root**, makes the receive buffer a send-and-receive buffer

# Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype  
             Datatype, int root, MPI_Comm comm);
```

- Broadcasts a message from the process with rank **root** to all other processes of the communicator

# Scatter and gather

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype  
               sendtype, void *recvbuf, int recvcount,  
               MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype  
              sendtype, void *recvbuf, int recvcount,  
              MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- Routines to spread and collect data from or to **root**
- MPI\_Scatter: if **root** sends 100 numbers to 10 processes, then **sendbuf** on **root** must be 1000 long
- MPI\_Gather: if root receives 100 numbers from 10 processes, then **recvbuf** on **root** must be 1000 long

# Allreduce, allgather and more

- **MPI\_Allreduce()** and **MPI\_Allgather()** are identical to their siblings, except that the end result is made available to all ranks
- As if the operation was followed by a broadcast
- There are also more elaborate combined all-to-all scatter-gather functions, like **MPI\_Alltoall()**, **MPI\_Alltoallv()** and **MPI\_Alltoallw()**
- Use the man pages to get more information, e.g.:  
\$ man MPI\_Allreduce

# Examples

# Writing Larger MPI Programs

- Question:
  - Now that we can write “Hello World!” MPI programs, then what do we need in order to write larger MPI programs for scientific projects?
- Answer:
  - Parallel algorithms
  - Data must be distributed to all proceses so that they all are kept busy during the entire execution of MPI programs

# Data Distribution

- The majority of time is usually spent in DO/FOR loops
- Multiple data distribution methods:
  - Block distribution
    - Column wise
    - Row wise
    - In both dimensions
  - Cyclic distribution
  - Master-worker
- 6 examples



## 2. Finite Difference Method

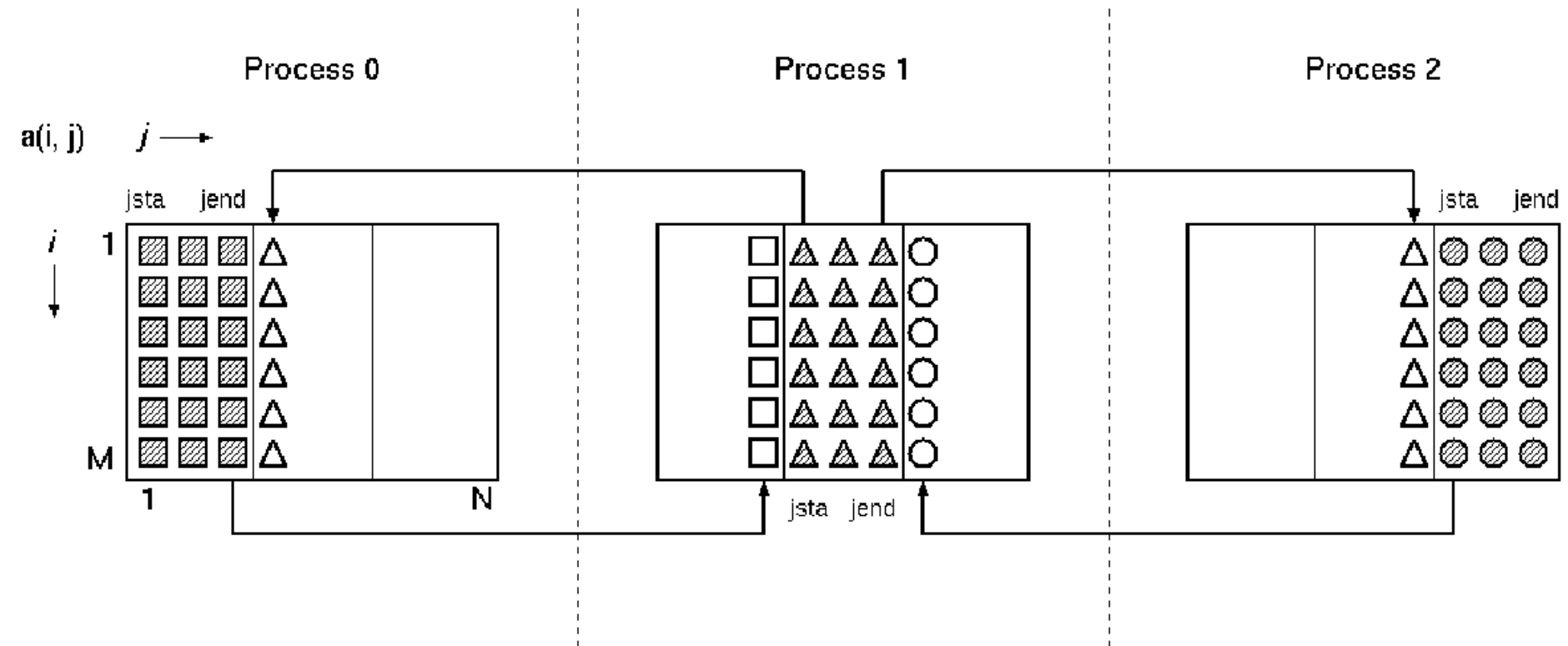
- Wikipedia:  
“Numerical methods for solving differential equations by approximating them with difference equations”
- Only a skeleton 2D FDM program is shown here
- Coefficients and the enclosing loop are omitted
- Data dependencies exist in both dimensions

# 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



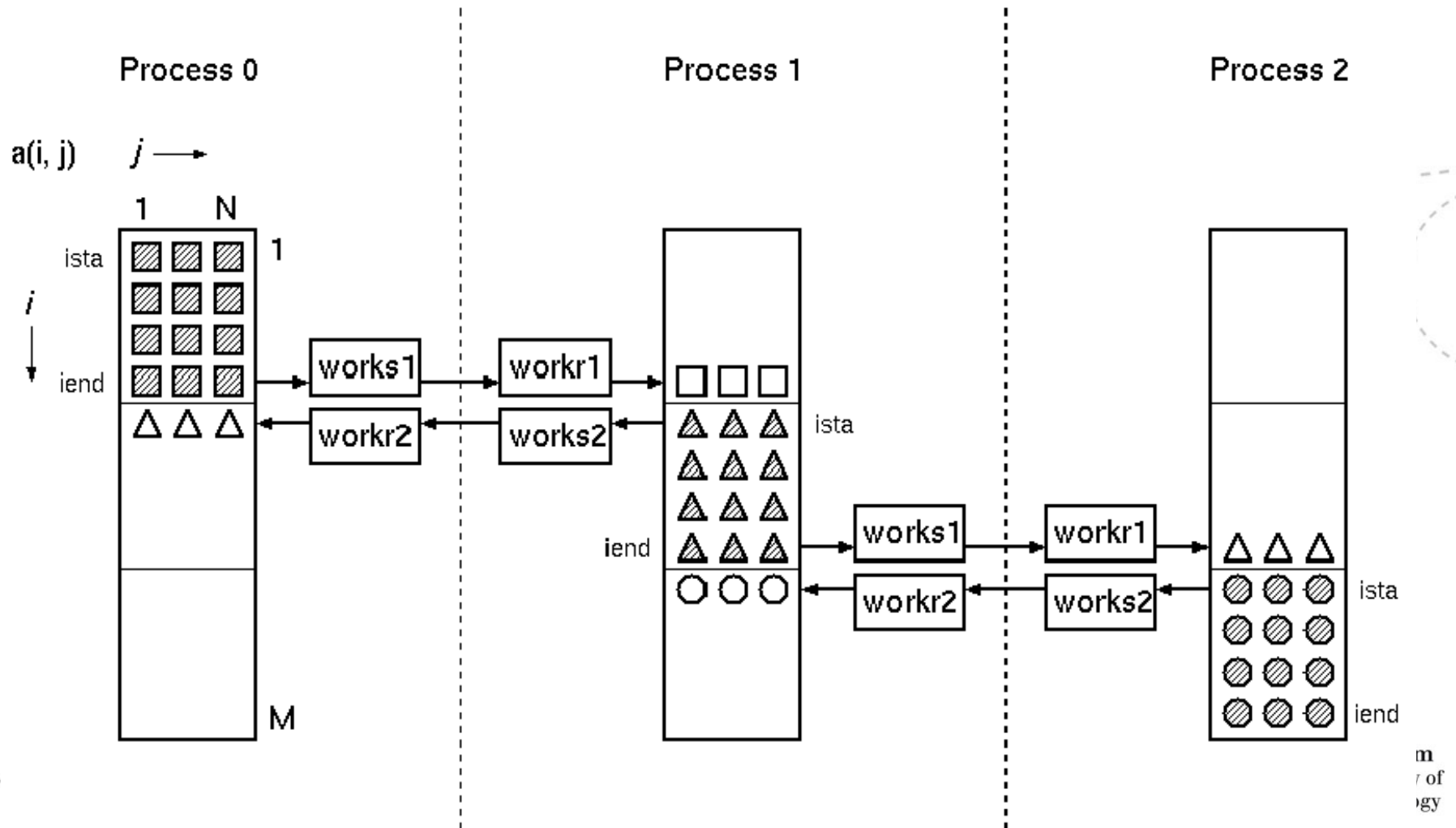
# Column-Wise Block Distribution

- 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

# Example 2

ex2/fdm1.f90

# Row-Wise Block Distribution



# Row-Wise Block Distribution

- 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

# Example 2

ex2/fdm2.f90



# Block Distribution in Both Dim. (1)

- 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)

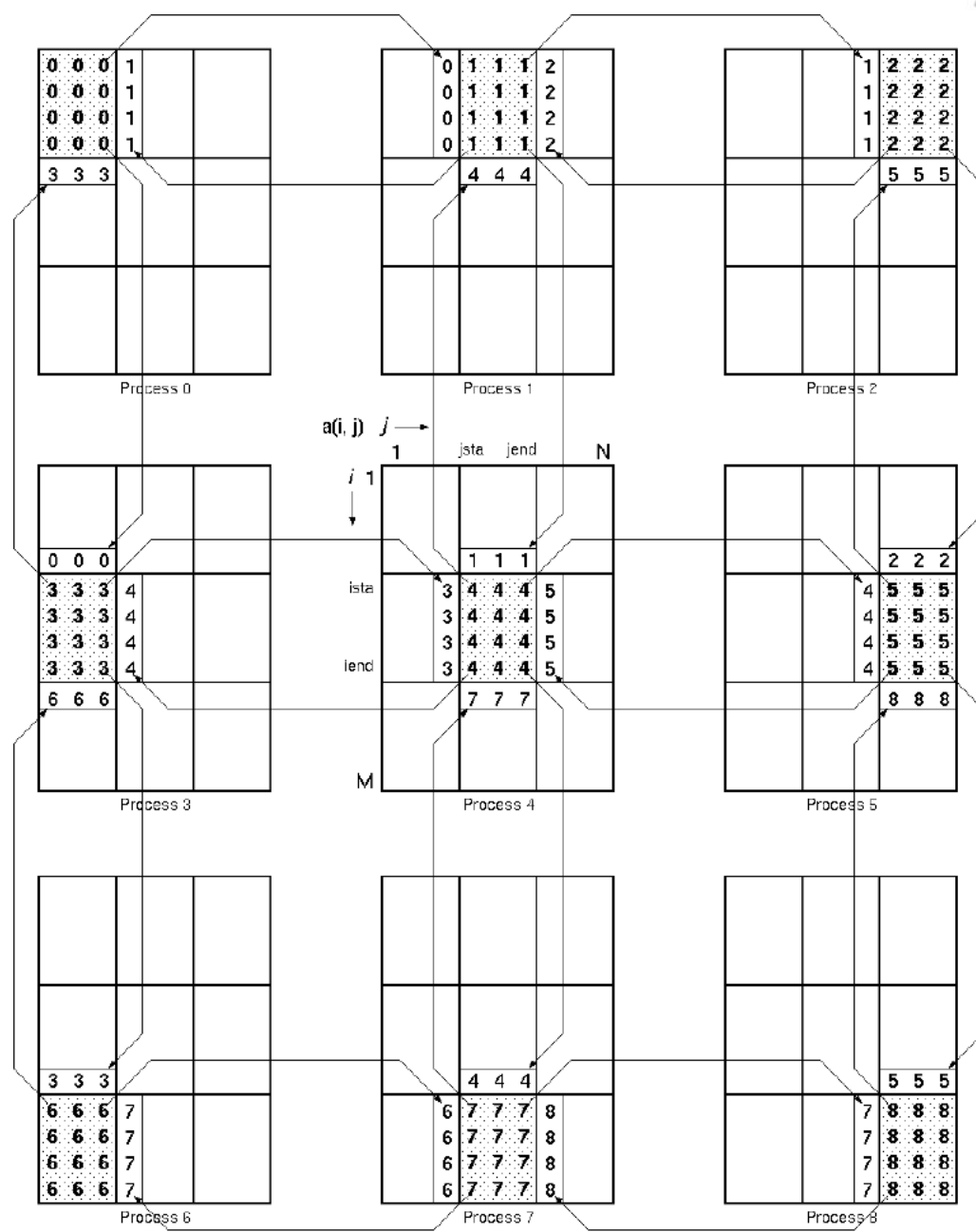
$a(i, j) \quad j \longrightarrow$

		1				N
$i \downarrow$	1	0 0 0	1 1 1	2 2 2	2 2 2	2 2 2
		0 0 0	1 1 1	2 2 2	2 2 2	2 2 2
		0 0 0	1 1 1	2 2 2	2 2 2	2 2 2
		0 0 0	1 1 1	2 2 2	2 2 2	2 2 2
		3 3 3	4 4 4	5 5 5	5 5 5	5 5 5
		3 3 3	4 4 4	5 5 5	5 5 5	5 5 5
		3 3 3	4 4 4	5 5 5	5 5 5	5 5 5
		3 3 3	4 4 4	5 5 5	5 5 5	5 5 5
		6 6 6	7 7 7	8 8 8	8 8 8	8 8 8
		6 6 6	7 7 7	8 8 8	8 8 8	8 8 8
		6 6 6	7 7 7	8 8 8	8 8 8	8 8 8
	M	6 6 6	7 7 7	8 8 8	8 8 8	8 8 8

(a) The distribution of  $a()$

		$j \rightarrow$				
itable(i, j)		-1	0	1	2	3
$i \downarrow$	-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



# Example 2

ex2/fdm3.f90

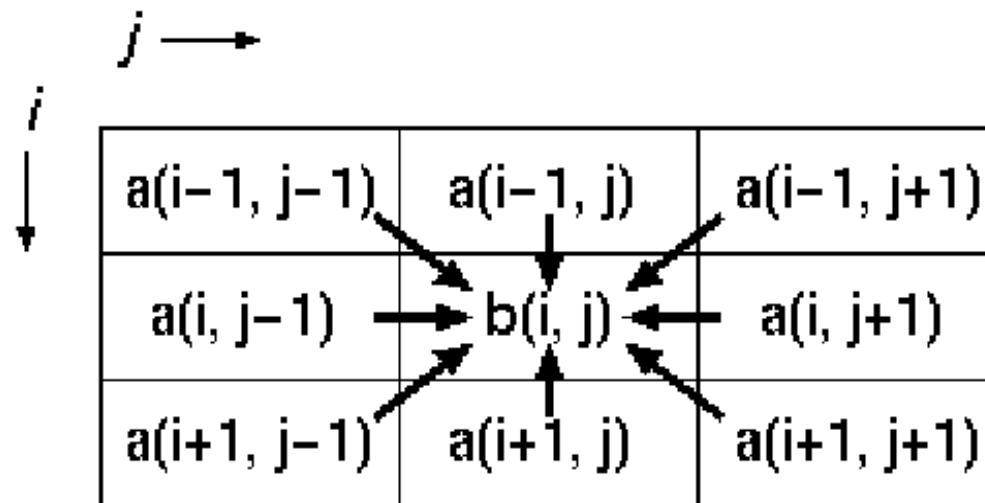
# Block Distribution in Both Dim. (2)

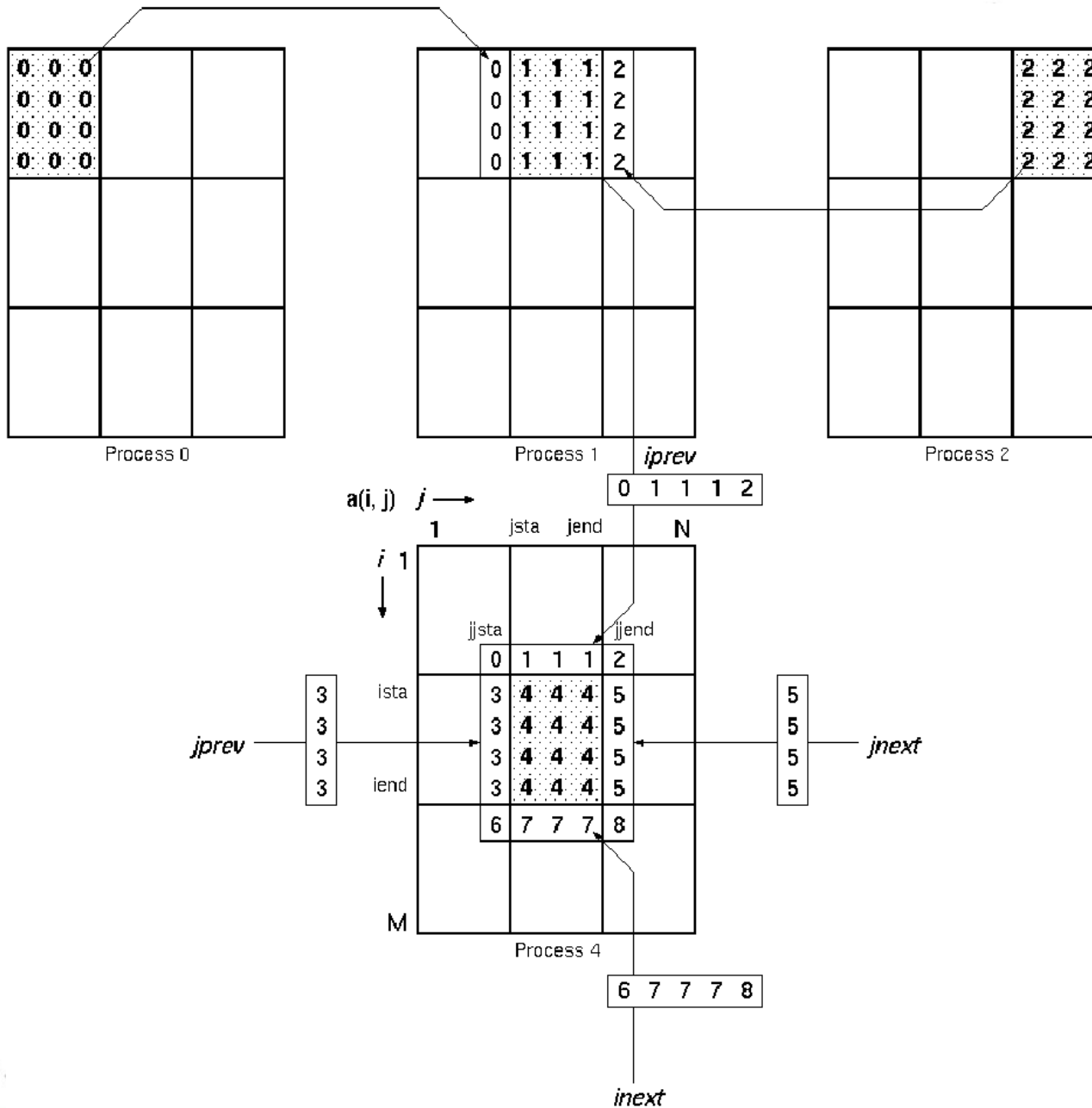
- The corner elements are now included
- The data dependencies are therefore more complex

# 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







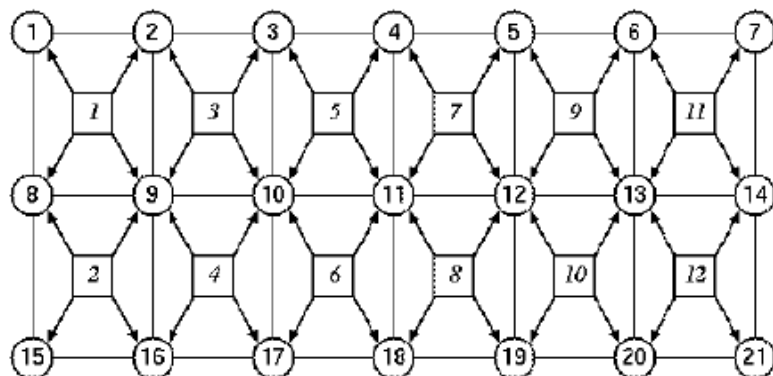
# Example 2

ex2/fdm4.f90

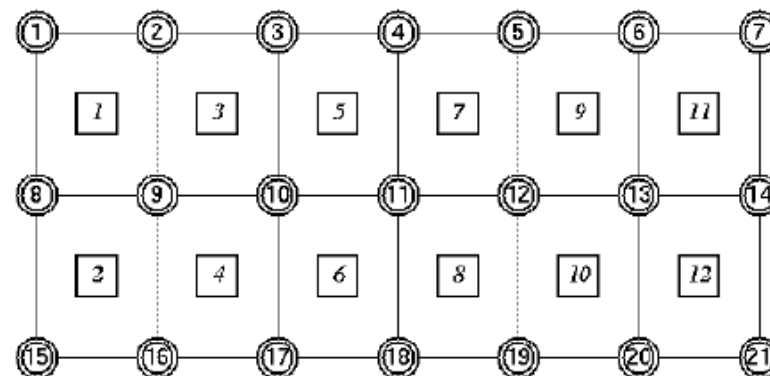
# 3. Finite Element Method

- Wikipedia:  
“Numerical technique for finding approximate solutions to boundary value problems for partial differential equations”
- A more complete example that produces a result

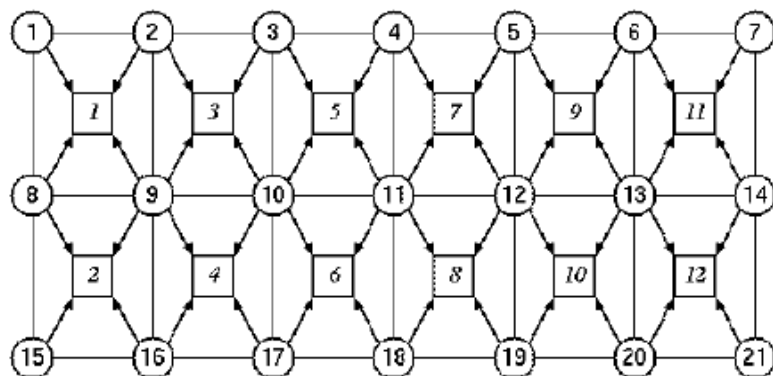
# Finite Element Method



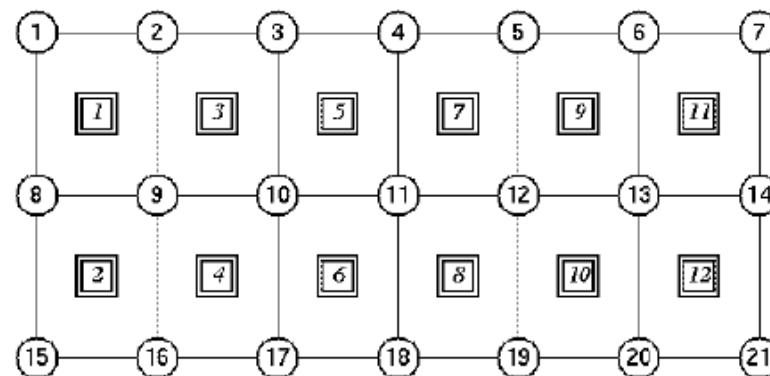
(a) Elements  $\rightarrow$  Nodes



(b) Update Nodes



(c) Nodes  $\rightarrow$  Elements



(d) Update Elements

eim  
sity of  
ology

# 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

```

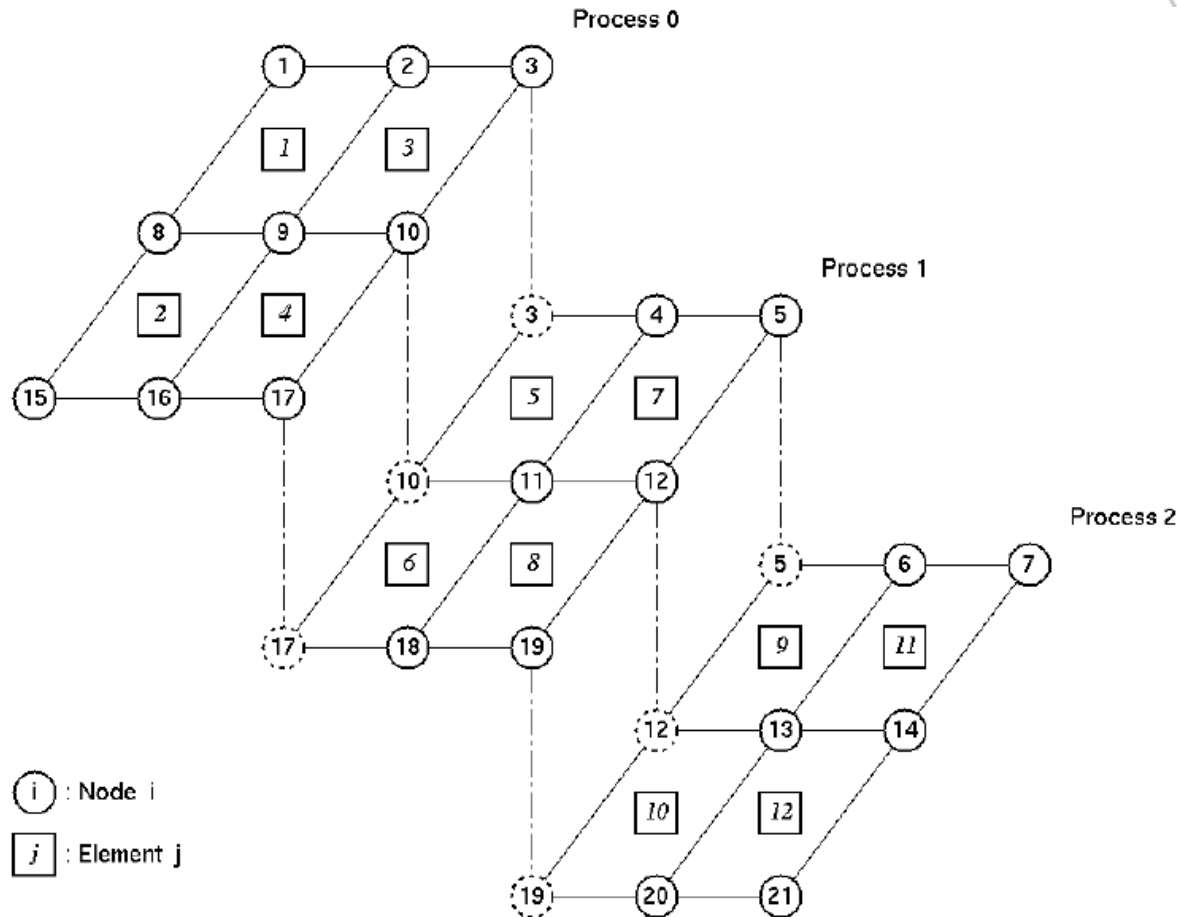
```

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



# Differences from 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

# Example 3

ex3/main.f90 and ex3/grid.f90

## 4. LU Factorization

- Wikipedia:
  - “Factors a matrix as the product of a lower triangular matrix and an upper triangular matrix”
  - Used for solving square systems of linear equations:  
 $Ax = b$
- ScaLAPACK and Intel's MKL library have optimized subroutines for this (outside the scope of this 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

```

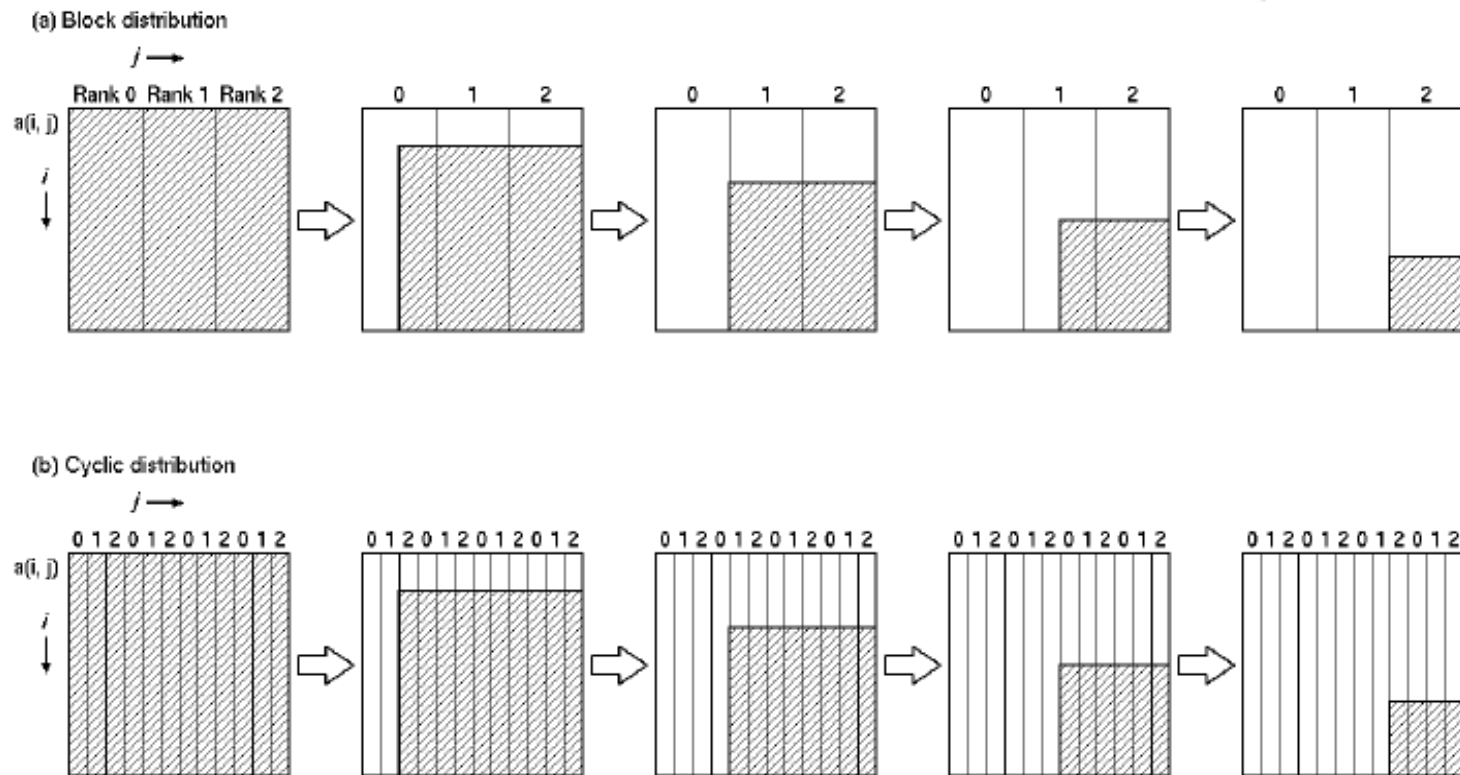
```

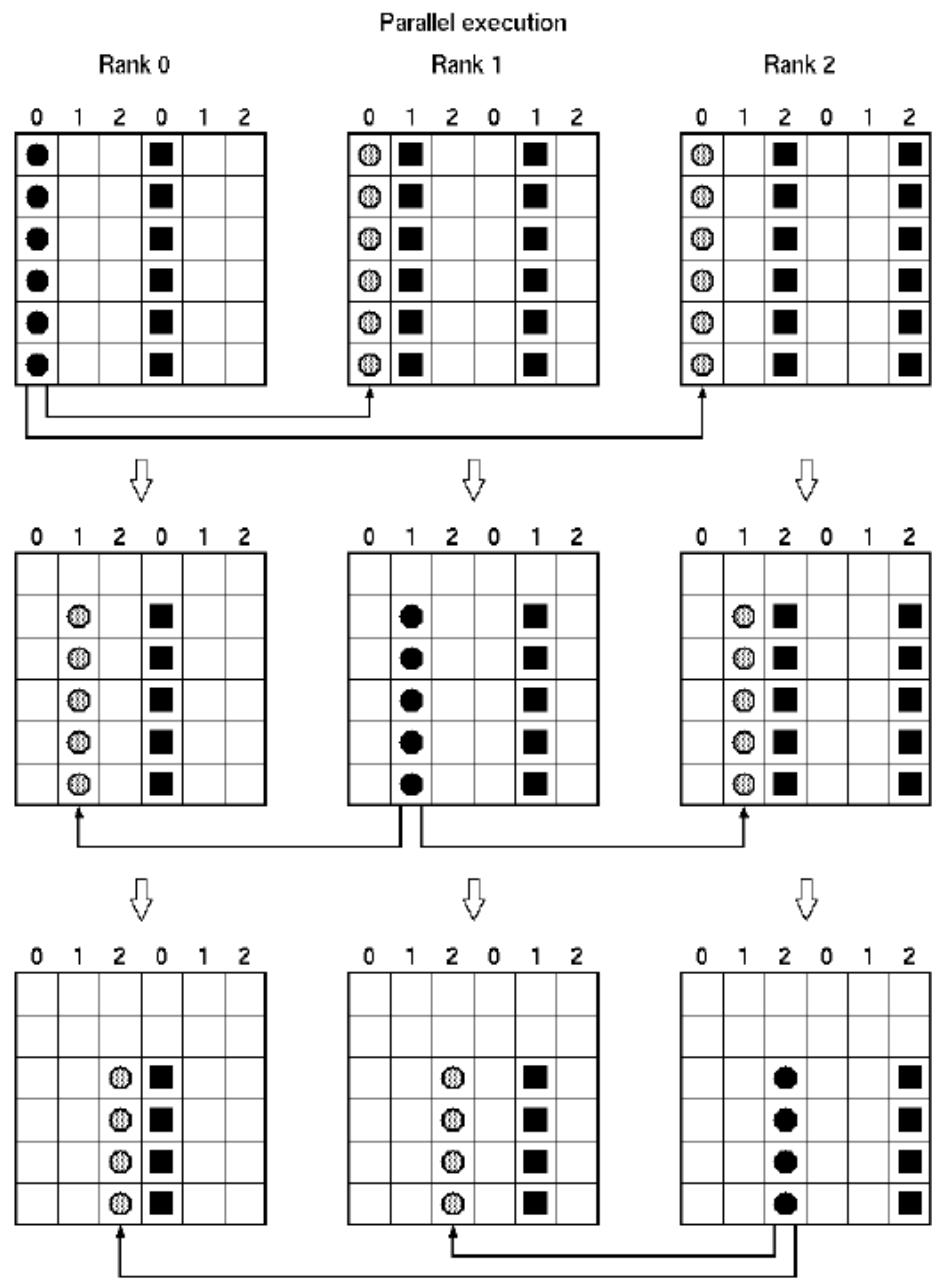
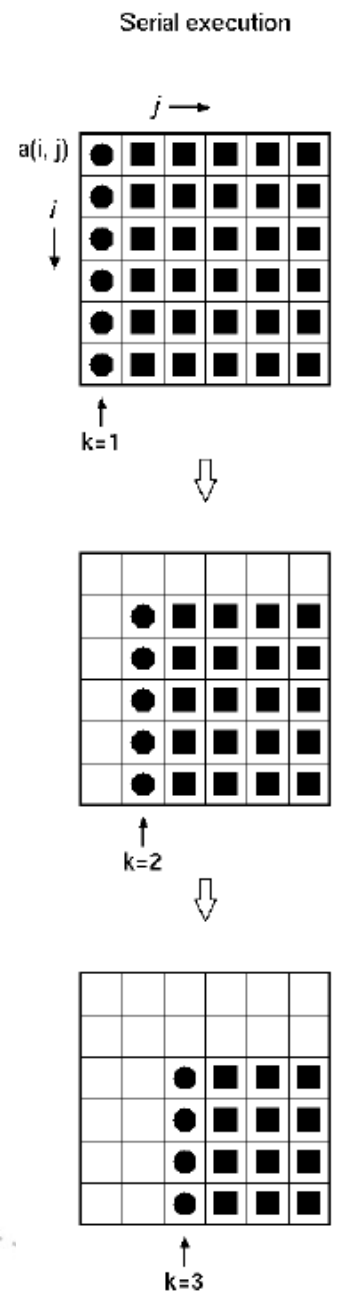
! Forward elimination
DO i = 2, n
  DO j = 1, i - 1
    b(i) = b(i) - a(i,j)*b(j)
  ENDDO
ENDDO
! Backward substitution
DO i = n, 1, -1
  DO j = i + 1, n
    b(i) = b(i) - a(i,j)*b(j)
  ENDDO
  b(i) = b(i) / a(i,i)
ENDDO
...
END

```



# Cyclic Data Distributing





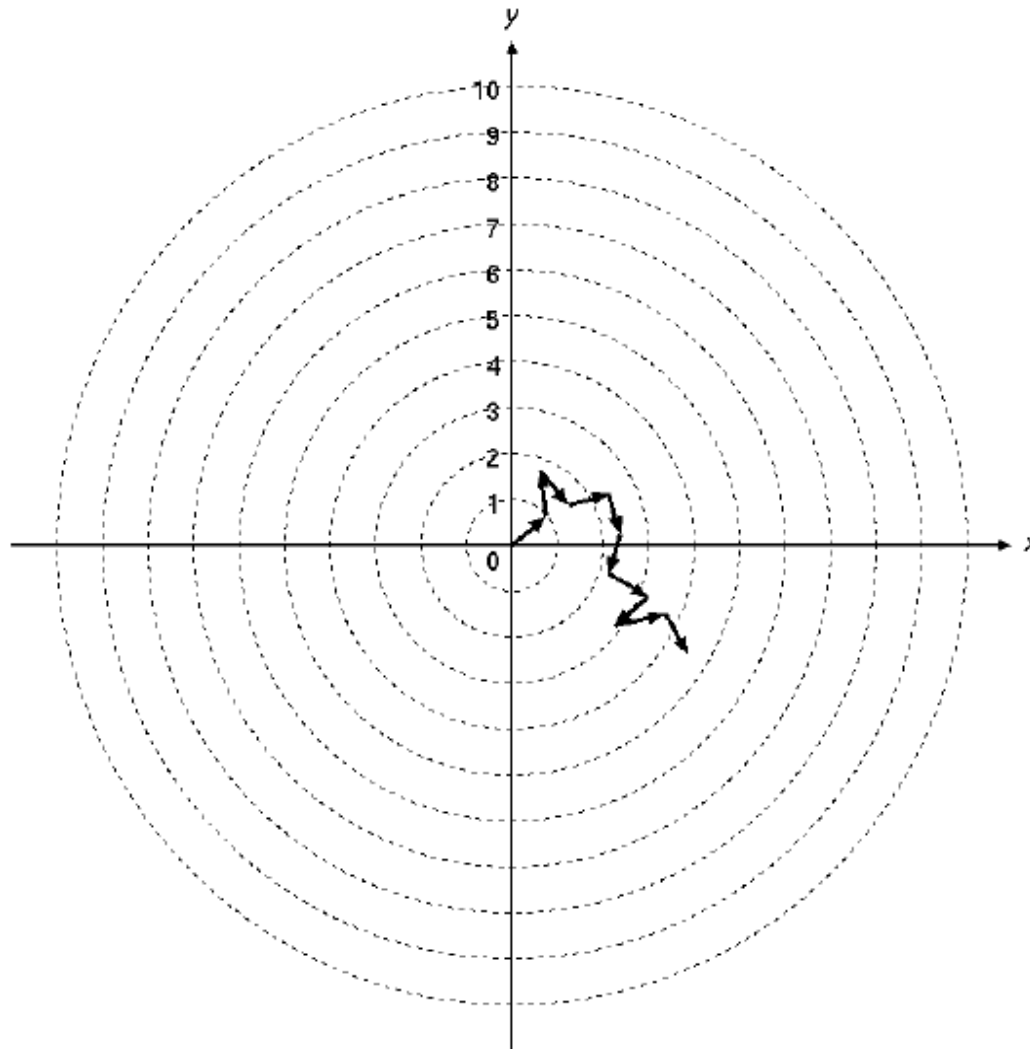
# Example 4

ex4/lu.f90

# 5. The Monte Carlo Method

- Wikipedia:  
“A broad class of computational algorithms that rely on repeated random sampling to obtain numerical results”
- 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
```

# Example 5

ex5/mc.f90



## 6. Molecular Dynamics

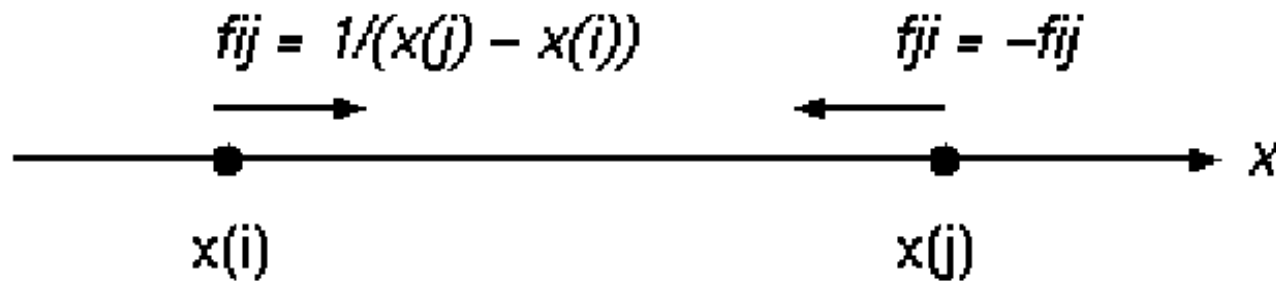
- Wikipedia:  
“a computer simulation of physical movements of atoms and molecules”
- N particles interact in 1 dimension
- The force on particle i from particle j is given by

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

- The law of action and reaction applies:

$$f_{ij} = -f_{ji}$$

# Forces in 1D



# Forces Acting on 7 Particles

$f(1) =$		$+f_{12}$	$+f_{13}$	$+f_{14}$	$+f_{15}$	$+f_{16}$	$+f_{17}$
$f(2) =$	$-f_{12}$		$+f_{23}$	$+f_{24}$	$+f_{25}$	$+f_{26}$	$+f_{27}$
$f(3) =$	$-f_{13}$	$-f_{23}$		$+f_{34}$	$+f_{35}$	$+f_{36}$	$+f_{37}$
$f(4) =$	$-f_{14}$	$-f_{24}$	$-f_{34}$		$+f_{45}$	$+f_{46}$	$+f_{47}$
$f(5) =$	$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$		$+f_{56}$	$+f_{57}$
$f(6) =$	$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$		$+f_{67}$
$f(7) =$	$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$	



# 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

- Most of the time is spent in the calculation loop:

```
D0 i = 1, n-1
  D0 j = i+1, n
    fij = 1.0/(x(j) - x(i))
    f(i) = f(i) + fij
    f(j) = f(j) - fij
  ENDDO
ENDDO
```

- Two parallelization methods:
  - Cyclic distribution of the outer loop
  - Cyclic distribution of the inner loop

Process 0

f(1) =		+f12	+f13	+f14	+f15	+f16	+f17
f(2) =	-f12						
f(3) =	-f13						
f(4) =	-f14			+f45	+f46	+f47	
f(5) =	-f15		-f45				
f(6) =	-f16		-f46				
f(7) =	-f17		-f47				

Process 1

f(1) =			+f23	+f24	+f25	+f26	+f27
f(2) =							
f(3) =		-f23					
f(4) =		-f24					
f(5) =		-f25		+f56	+f57		
f(6) =		-f26		-f56			
f(7) =		-f27		-f57			

Process 2

f(1) =							
f(2) =							
f(3) =			+f34	+f35	+f36	+f37	
f(4) =			-f34				
f(5) =			-f35				
f(6) =			-f36			+f67	
f(7) =			-f37			-f67	

MPI\_ALLREDUCE

All Processes

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

# Example 6

ex6/md1.f90

Process 0

f(1) =  
f(2) =  
f(3) =  
f(4) =  
f(5) =  
f(6) =  
f(7) =

	+f12		+f15			
	-f12	+f23		+f25		
		-f23		+f35		
			+f45			
	-f15	-f35	-f45	+f55		
	-f26		-f56			

Process 1

f(1) =  
f(2) =  
f(3) =  
f(4) =  
f(5) =  
f(6) =  
f(7) =

	+f13		+f16			
		+f24		+f27		
	-f13		+f35			
	-f24		+f45			
			+f57			
	-f16	-f36	-f46			
	-f27		-f57			

Process 2

f(1) =  
f(2) =  
f(3) =  
f(4) =  
f(5) =  
f(6) =  
f(7) =

	+f14		+f17			
		+f25				
		+f34		+f37		
	-f14	-f34		+f47		
		-f25				
				+f57		
	-f17	-f37	-f47	-f67		

MPI\_ALLREDUCE

All Processes

ff(1) =  
ff(2) =  
ff(3) =  
ff(4) =  
ff(5) =  
ff(6) =  
ff(7) =

	+f12	+f13	+f14	+f15	+f16	+f17
	-f12	+f23	+f24	+f25	+f26	+f27
	-f13	-f23	+f34	+f35	+f36	+f37
	-f14	-f24	-f34	+f45	+f46	+f47
	-f15	-f25	-f35	-f45	+f56	+f57
	-f16	-f26	-f36	-f46	-f56	+f67
	-f17	-f27	-f37	-f47	-f57	-f67



# Example 6

ex6/md2.f90

## 7. MPMD Models

- Wikipedia:  
“Multiple Program, Multiple Data: multiple autonomous processors simultaneously operating at least 2 independent programs”
- Different programs run in parallel and communicate with each other

# An example

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

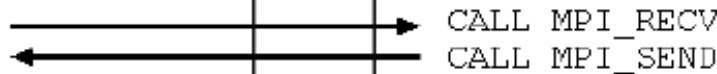

    CALL MPI_SEND
    CALL MPI_RECV
ENDDO
...
END
```

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

# Example 7

ex7/master.f90 and ex7/worker.f90

# More Information

- Sigma2:
  - <https://www.sigma2.no/documentation/>
- Our Web-site:
  - <http://www.hpc.ntnu.no/>