INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/474 - 15

COLLEGE ON
"THE DESIGN OF REAL-TIME CONTROL SYSTEMS"
1 - 26 October

PARALLEL PROCESSING

I. WILLERS
Advanced Computing Group
CERN
CH-1211 Geneva 23
Switzerland

Parallel Processing

Ian Willers

College on
The Design of Real-Time
Control Systems

1. Classes of Parallelism
    - Result Parallelism
    - Specialist Parallelism
    - Agenda Parallelism
2. Programming Paradigms
    - Live Data Structures
    - Message Passing
    - Distributed Data Structures
3. Linda
    - Distributed Data Structures
    - Message Passing
    - Live Data Structures

---

1. Classes of Parallelism
    - Result Parallelism
    - Specialist Parallelism
    - Agenda Parallelism

How to write parallel programs -

Choose class of parallelism natural to the problem.

Choose the corresponding programming paradigm

Write a program

– *Result Parallelism*

– Example: building a house

– Start by looking at the final product

– Divide it into its components

front, rear, right, left walls, interior walls, foundation, roof, etc.

– Assign one worker to each component

– Start all workers simultaneously

Each worker is assigned to produce one piece of the result. They all work in parallel up to the natural restrictions imposed by the problem.

– *Specialist Parallelism*

– Example: building a house

– Start by assembling teams of experts to do the job

We need suveyors, excavators, foundation builders, carpenters, roofers, etc.

– Each skill is represented by a specialist worker

– Start all workers simultaneously

Each worker works on his skill. They all work in parallel up to the natural restrictions imposed by the problem.

– Agenda Parallelism

– Example: building a house

– Write an agenda of activities that must be completed in order to build the house

1. **Build foundation**
2. **Erect external walls**
3. **Construct roof**
4. **Erect internal walls**
5. **Do plastering**
6. **etc**

– Select a team that can do the whole job

– Start all workers on item 1. When some finish they start item 2 and so on.

Each worker helps with the current item on the agenda and they all work in parallel up to the natural restrictions imposed by the problem.

– Result Parallelism

– focuses on the shape of the finished product;

– Specialist Parallelism

– focuses on the makeup of the workers;

– Agenda Parallelism

– focuses on the list of tasks to be performed.

– How does this map onto software?

– Result Parallelism

– plan a parallel application around the data structure yielded as the ultimate result, and get parallelism by computing all elements of the result simultaneously;

– Specialist Parallelism

– plan an application around an ensemble of specialists connected into a logical network of some kind; parallelism results from all nodes of the network being active simultaneously;

– Agenda Parallelism

– plan an application around a particular agenda of activities and then assign many workers to each step.

– Building a House

– What conceptual class of parallelism do we use?

– Result Parallelism

The factory built house –
The walls, roof assembly, staircases etc. are all built in parallel in the factory and assembled on site.

– Specialist Parallelism

The standard house –
The specialists each do their own task. Sometimes in parallel as the plumber and the electrician may work at the same time.

– Agenda Parallelism

Barn raising –
One group turns its attention to one of a list of tasks in turn.

– *Programming using Result Parallelism*

– The programmer asks

Is my program intended to produce some muliple–element data structure as its result (or can it be conceived in these terms)?

If so, can I specify exactly how each element of the resulting structure depends on the rest and on the input?

– The program then looks like

Build a data structure to represent the result

Determine the value of each element of this structure simultaneously by specifying the computation required

Terminate when all values are known.

– *Programming using Result Parallelism*

– *A simple example*

– Compute the sum S of two n–element vectors A and B

Construct an n–element vector S

To determine the i–th element of S, add the i–th element of A to the i–th element of B

When all additions done terminate

– Since the elements of S are independent, all additions can start simultaneously and can be done in parallel

- *Programming using Result Parallelism*

- *When is this applicable?*

In any problem whose goal is to produce a series of values with predictable organisation and interdependencies.

- *Programming using Specialist Parallelism*

- **The programmer asks**

Can my problem be expressed as a network in which each node executes a relatively autonomous computation and internode communication follows predictable paths

The program then looks like

Model my problem onto processes that perform according to items in my model and communicate via well defined channels.

– Programming using Specialist Parallelism

– Examples:

– A Circuit Emulator

Each circuit element is represented by a process and communication takes place according to the circuit

– Europe wide transport estimates for travel time between two points, given current estimates of road conditions, weather and traffic.

Each country could be represented by a node in the network knowing conditions in that country.

A lorry going on a route from Trieste to London could be passed from Italy, via France to England. The England node would then print the travel time.

– Programming using Agenda Parallelism

– The programmer asks

Can my problem be expressed as a series of transformations to be applied to some set of data in parallel

– The program then looks like

The master–worker paradigm is a good example.

The master initialises the computation and creates a collection of identical worker processes

Each worker process is capable of performing any step in the computation

Workers repeatedly seek a task to perform, do the task and repeat

When no tasks remain terminate.

– Programming using Agenda Parallelism

– The master–worker paradigm advantages

The same program can run with 1, 10 or 1000 workers in three consecutive runs.

By distributing tasks on the fly, this is naturally load balancing. While one worker is tied up with a time consuming task, another might execute a dozen shorter task assignments

– Programming using Agenda Parallelism

– An example

– Identify the employer with the lowest ratio of salary to dependents in a database of employee records

The master obtains the employee records and puts each record in a "bag" where the workers can find those records

Each worker repeatedly draws a record from the "bag", computes the ratio of salary to dependents and sends the result to the master

The master keeps a record of the lowest so far and, when all tasks are complete, report the answer.

– Programming using Agenda Parallelism

– Data Parallelism

– The transformations take place on all elements of a data structure simultaneously

This is usually associated with SIMD machines where all transformations happen concurrently and synchronously.

– Programming using Agenda Parallelism

– Speculative Parallelism

– A collection of parallel activities is undertaken with the understanding that some may ultimately prove to be unnecessary to the final result

An example is in logic programming

x or y

We can work on x and y in parallel the first one that is true determines the result.

Many workers can work on a list of tasks, yet only one of the results contributes to the final result

1. Classes of Parallelism
    - Result Parallelism
    - Specialist Parallelism
    - Agenda Parallelism
2. **Programming Paradigms**
    - **Live Data Structures**
    - **Message Passing**
    - **Distributed Data Structures**
3. Linda
    - Distributed Data Structures
    - Message Passing
    - Live Data Structures

2. **Programming Paradigms**
    - **Live Data Structures**
    - **Message Passing**
    - **Distributed Data Structures**

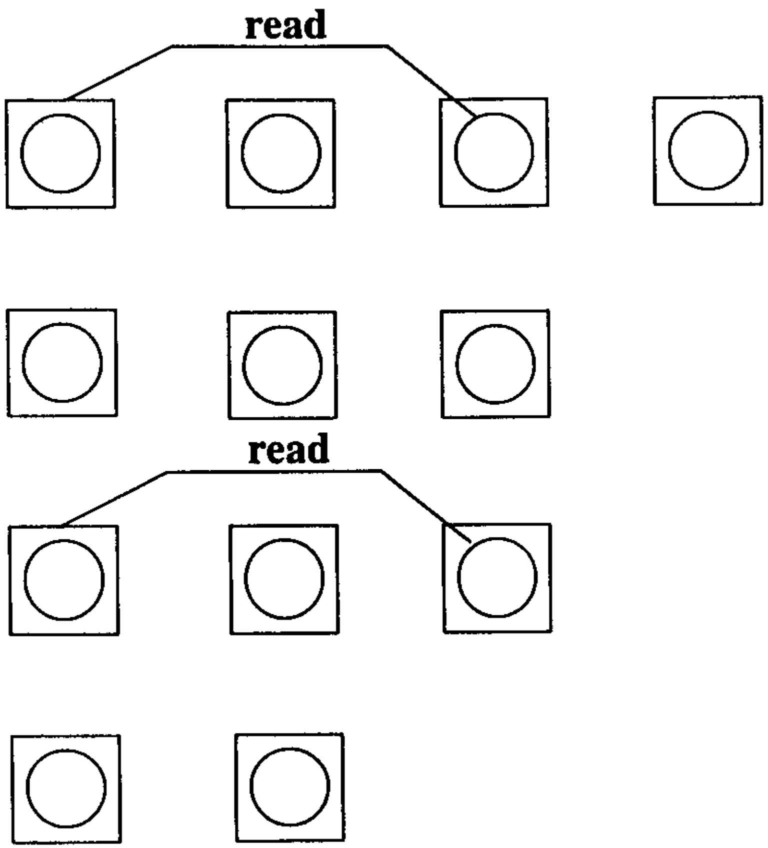
How to write parallel programs –

Choose class of parallelism natural to the problem.

Choose the corresponding programming paradigm

Write a program

– Live Data Structures



○ = process □ = data element

– Live Data Structures

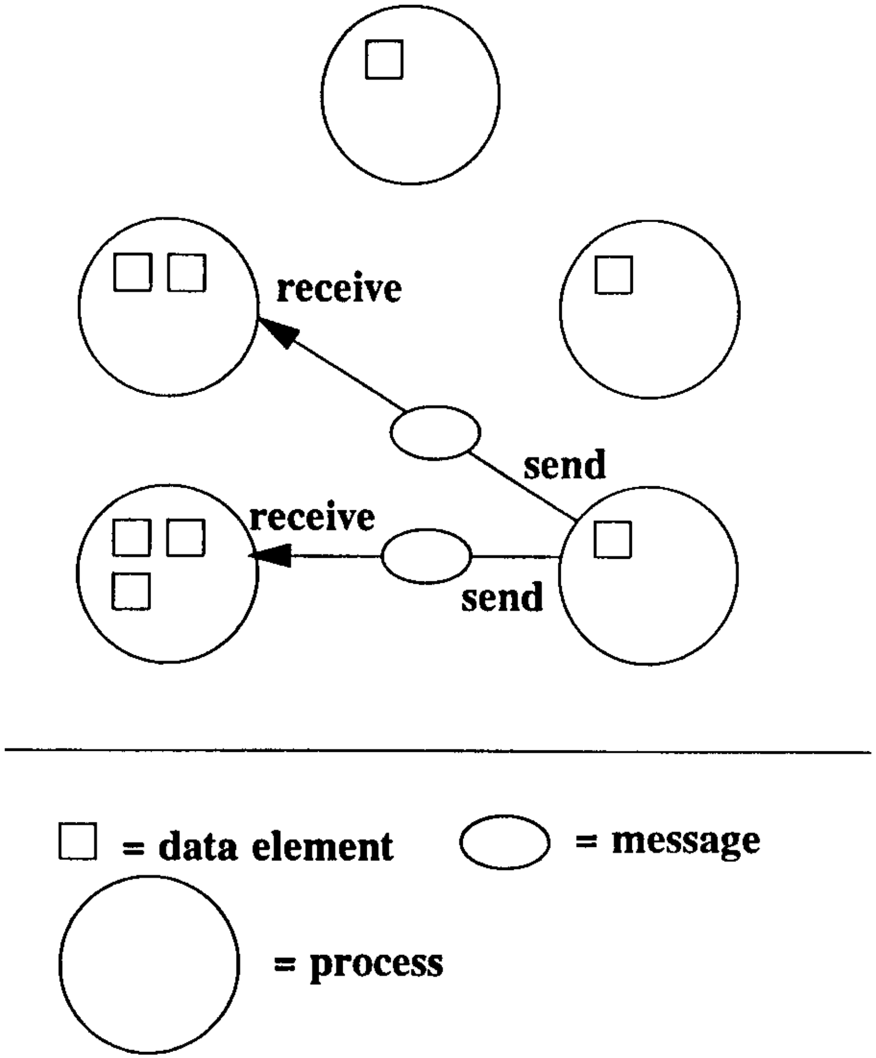
The result data structure – the number of its elements and their relationship – determines the program structure

Every concurrent process is locked inside a data object

The job of the process is to produce the value for that data element

A process may read another data object

– Message Passing



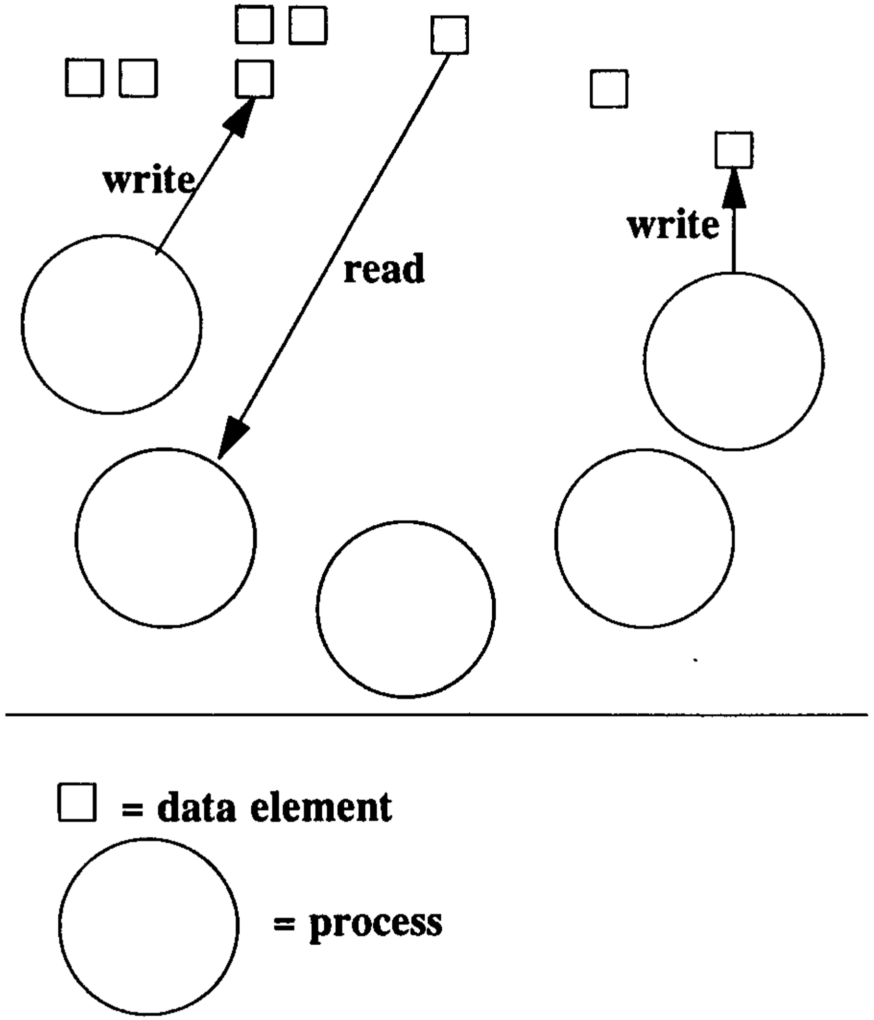
– Message Passing

The process structure – the number of processes and their relationships – determines the program structure

A collection of concurrent processes communicate by exchanging messages

Every data object is locked inside some process

– *Distributed Data Structures*



– *Match between Parallel Classes and Programming Paradigms*

Result Parallelism

↔

Live Data Structures

Specialist Parallelism

↔

Message Passing

Agenda Parallelism

↔

Distributed Data Structures

Result Parallelism

↔

Live Data Structures

Example: Calculate the sum of two vectors A and B and put the result in S.

The live structure is the vector S

Trapped inside each element of S is a process that computes A(i) + B(i) for the appropiate i.

When a process is complete it vanishes leaving behind the value it was charged to compute.

Specialist Parallelism

↔

Message Passing

Example: The travel time program for lorries in Europe.

Lorries and routes are represented by messages

To introduce a lorry into the French road system we send a message describing the lorry and its route to France.

France calculates an estimated transit time and passes the message on

– *Match between Parallel Classes and Programming Paradigms*

Agenda Parallelism

↔

Distributed Data Structures

Example: master–worker using a "bag"

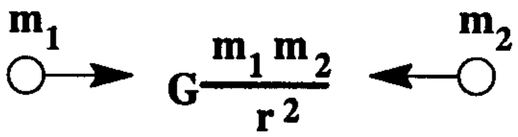
The master puts tasks into the "bag"

The worker takes tasks from the "bag" and returns the result to the "bag"

The master picks up its results from the "bag"

– *The n–body problem*

The 2–body problem



$$G\frac{m_1 m_2}{r^2}$$

The n–body problem

- **Simulate the n–bodies**
- **Calculate the forces between all bodies**
- **Update each body's position**
- **Do this for a number of time steps**

– *Result Parallelism / Live Data Structures*

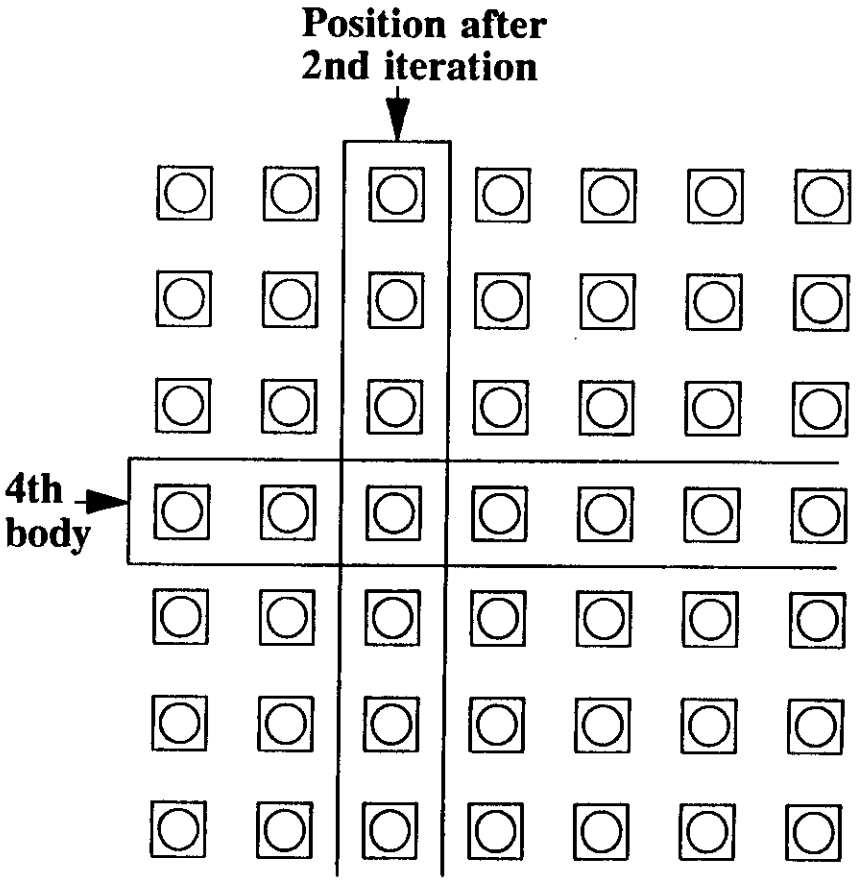
Use a matrix M(i , j) to describe the problem

M(i , j) is the position of the i–th body after j time steps

M(i , 0) gives the starting position and the last column gives the final position of each body.

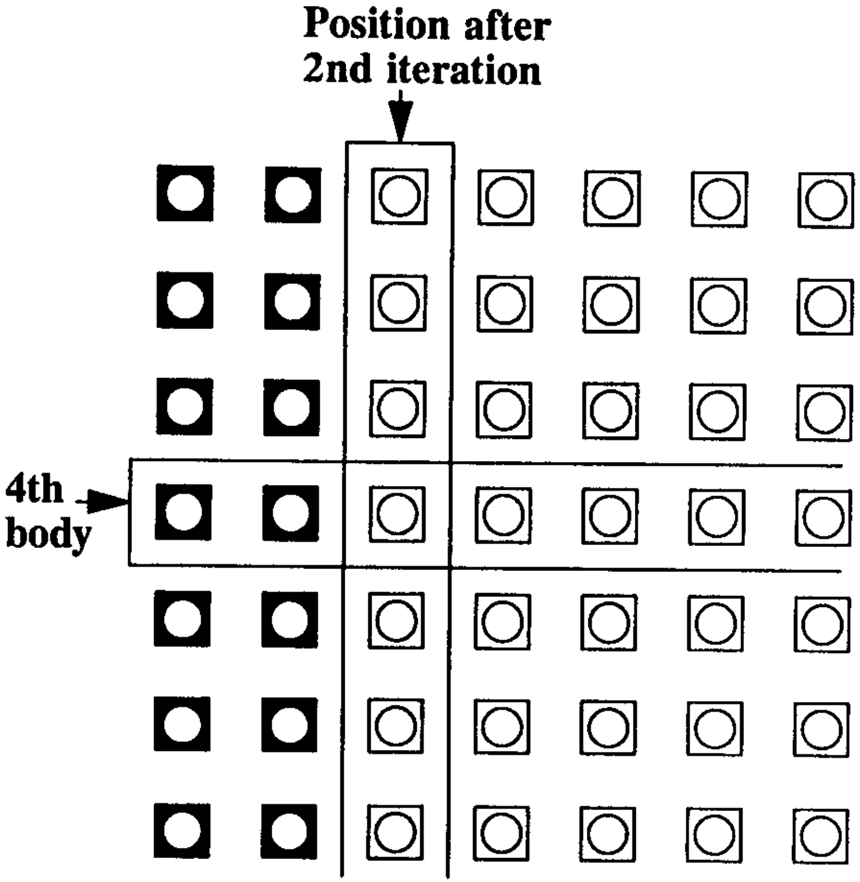
Now define a function
position (i , j)
that calculates the position of body i for the j–th iteration.

– *Result Parallelism / Live Data Structures*



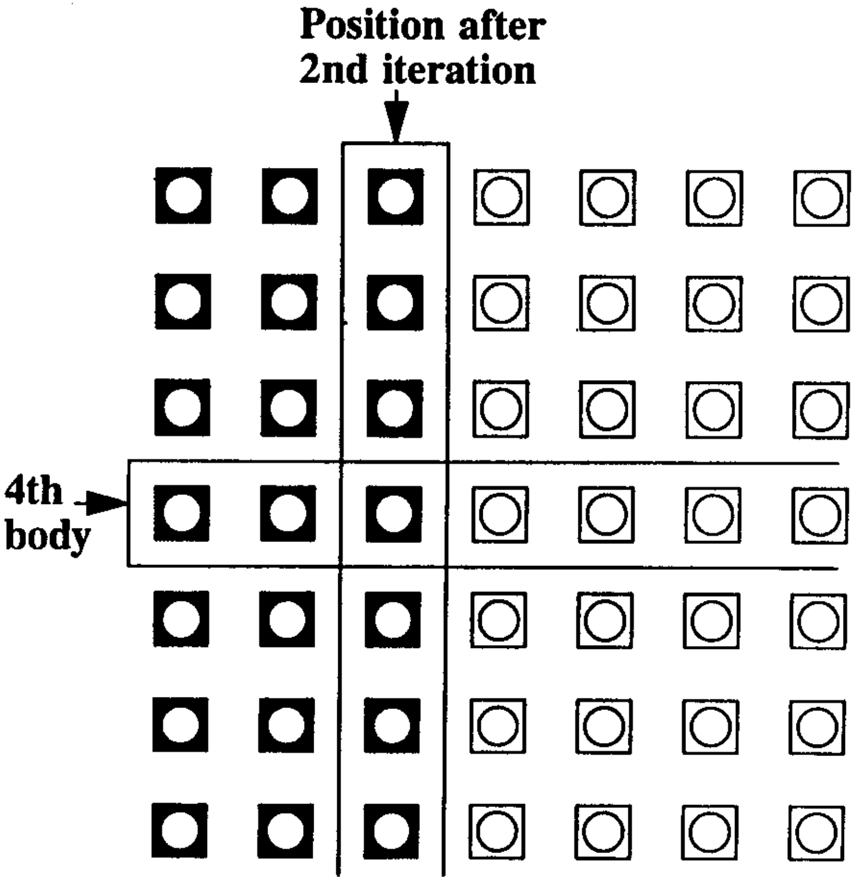
– *The n–body problem*

– *Result Parallelism / Live Data Structures*



– *The n–body problem*

– *Result Parallelism / Live Data Structures*



– The n–body problem

– Specialist Parallelism / Message Passing

Create a series of processes each one specialising in a single body

That process is responsible for calculating a single body's current position throughout the simulation

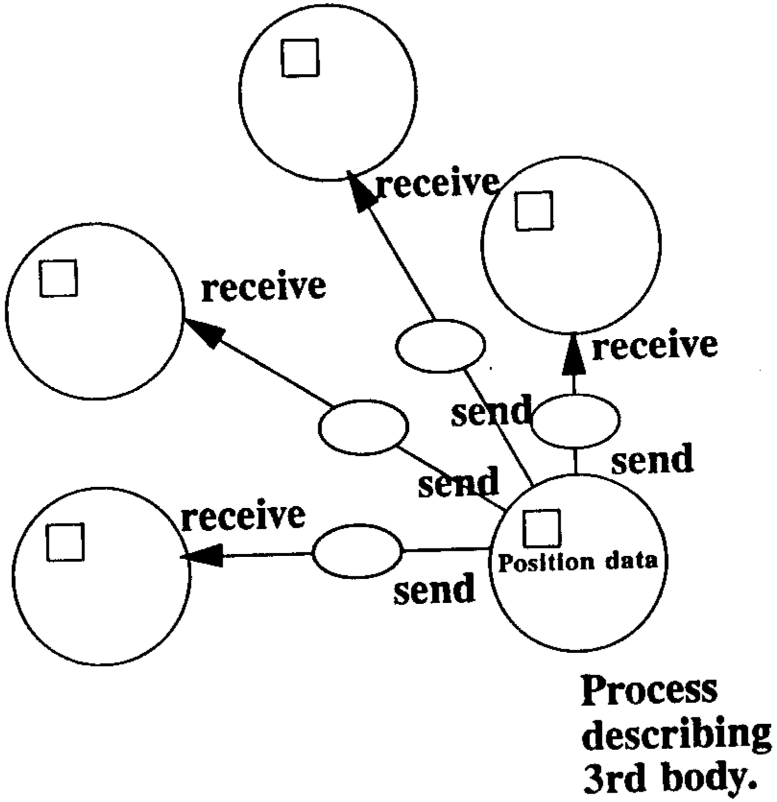
At the start of each iteration, each process informs each other process by message of the current position of its body

Each process receives the position of all the other bodies

The process calculates its new position

Then iterates

– Message Passing



– *The n–body problem*

– *Agenda Parallelism / Distributed Data Structure*

Repeatedly apply the transformation
compute next position
to all bodies in the set

Create a master process that creates n initial task descriptions, one for each body

On the first iteration, each worker in a group of identical workers processes repeatedly grabs a task descriptor and computes the next position of the corresponding body

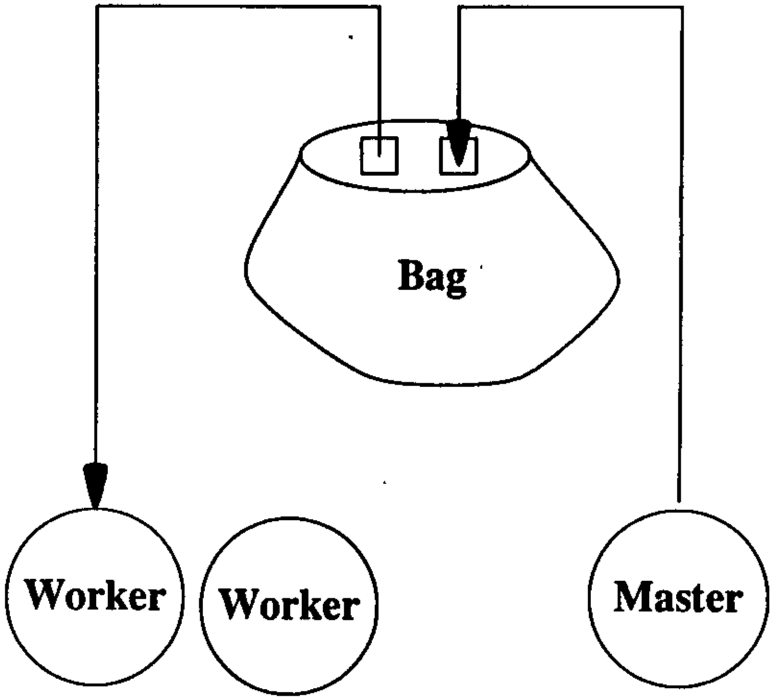
When the pile of task descriptors are used up the tasks stop and the bodies have advanced to the new position

Store the current position in a distributed table so all tasks can refer to it

then iterate

– *The n–body problem*

– *Agenda Parallelism / Distributed Data Structure*



1. Classes of Parallelism
    - Result Parallelism
    - Specialist Parallelism
    - Agenda Parallelism
2. Programming Paradigms
    - Live Data Structures
    - Message Passing
    - Distributed Data Structures
3. **Linda**
    - **Distributed Data Structures**
    - **Message Passing**
    - **Live Data Structures**

3. **Linda**
    - **Distributed Data Structures**
    - **Message Passing**
    - **Live Data Structures**

How to write parallel programs –

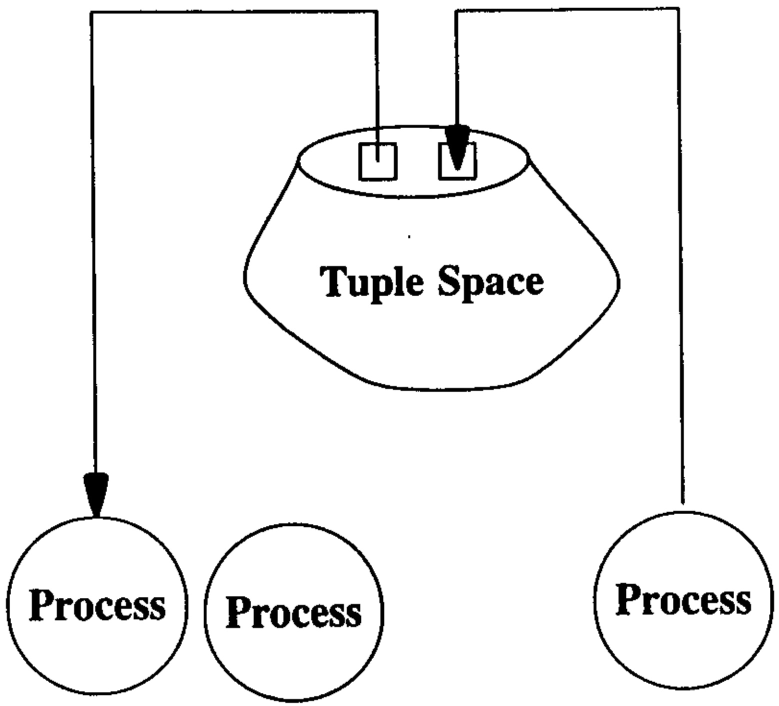
Choose class of parallelism natural to the problem.

Choose the corresponding programming paradigm

Write a program

– Linda

– Tuple Space (Distributed Data Structure)



□ = a tuple

– Linda

– The four basic operations

out(t) causes tuple 't' to be inserted into tuple space

in(s) causes some tuple 't' that matches the template 's' to be withdrawn from tuple space

If no matching tuple 't' exists when in(s) executes, the process suspends until one is, then proceeds as before

rd(s) is the same as in(s) except that the matching tuple remains in tuple space

eval(t) is the same as out(t) except that the tuple 't' is evaluated after it enters tuple space

– Linda

– Simple Examples

```
out( “a string”, 15.01, 17, “another string” )

out( 0 , 1 )

in( “a string”, ? f , ? i , “another string” )

rd( “a string”, ? f , ? i , “another string” )

eval( “e” , 7 , exp( 7 ) )

rd( “e” , 7 , ? value )
```

– Linda

– Semaphores

A semaphore is used as a lock to protect critical code.

```
                 out( “sem” )
                      |
in( “sem” )           |     in( “sem” )
<critical code 1>     |     <critical code 2>
out( “sem” )          |     out( “sem” )
```

A counting semaphore limits the access to devices by ‘n’ processes

```
for ( i = 1 ; i < n ; ++i ) (
  out( “sem” )
)
```

– Bags

A bag is a data structure that defines two operations

'add an element'

and

'withdraw an element'

Master–worker paradigm

Add a task to the bag (master):

```
out( "task", taskdescription )
```

Withdraw a task (worker):

```
in( "task", newtask )
```

– Make a sequential loop parallel using a bag

The sequential loop is:

```
for ( <loop control> ) (
   <something>
)
```

Then the parallel loop is:

```
for ( <loop control> ) (
   eval( "this loop", something( ) )
)

for ( <loop control> ) (
   in("this loop", 1 )
)
```

*** <something> is re-written to be a function something( ) returning the value '1'**

– Named–Access Structures

This is often used in real-time monitors:

Use the tuple (name , value)

Then to read the value corresponding to a device use:

```
rd ( name , ? value )
```

To update the value:

```
in( name , ? old )
out( name , new )
```

– Barrier Synchronization

Each process in some group waits at a barrier until all processes in the group have reached the barrier

Then all can proceed

If the group contains ‘n’ processes we set up the barrier by executing:

```
out( “barrier”, n )
```

Then each process executes:

```
in( “barrier”, ? val )

out( “barrier”, val – 1 )

rd( “barrier” , 0 )
```

– *Position Accessed Structures*

This is a distributed array.

To set the 14th. element of vector 'V'

```
out( "V" , 14 , 123.5 )
```

For an array containing prime numbers

```
out( "primes" , 1 , 2 )
out( "primes" , 2 , 3 )
out( "primes" , 3 , 5 )
etc.
```

If some processes are calculating primes while others want to read:

```
rd( "primes" , 100 , ? val )
```

will block until the 100–th prime has been calculated.

– *Generalized Streams*

A stream is an ordered sequence of elements to which processes may apper elements

and processes may, at any time, remov the stream's head element

Must keep track of the head and tail so we need the tuples:

```
( "strm" , "head" , value )
( "strm" , "tail" , value)
```

– Linda

– Generalized Streams

To append a new element

```
in( “strm” , “tail” , ? index )
out( “strm” , “tail” , index + 1 )
out( “strm” , index , newelement )
```

To remove an element from the stream

```
in( “strm” , “head” , ? index )
out( “strm” , “head”, index + 1 )
in( “strm” , index , ? element )
```

– Linda

– Live Data Structures

To get the live version of a data structure we use ‘eval’ instead of ‘out’

Suppose we need a stream of processes then we write

```
eval( “live stream” , i , f( i ) )
```

if f is the factorial then this resolves to the stream

```
( “live stream” , 1 , 1 )
( “live stream” , 2 , 2 )
( “live stream” , 3 , 6 )
etc.
```

Note that

```
rd( “live stream” , 1 , ? x )
```

blocks until the evaluation of f(1) ends.

– Linda

– A Complete Program

Here we will use result parallelism:

We define an n–element vector whose i–th value is the i–th prime

We build this structure in tuple space and associate with each element with a process ‘isprime(i)’

We set this up with the loop

```
for ( i = 2 ; i < limit ; ++i ) (
   eval( “primes” , i , isprime( i ) )
)
```

We then read the j–th element with

```
rd( “primes” , j , ? ok )
```

```
lmain( ) (
   int i,ok

   for( i = 2; i < limit ; ++i ) (
      eval( “primes” , i , isprime( i ) )
   )

   for( i = 2; i < limit; ++i ) (
      rd( “primes” , i , ? ok )
      if ( ok ) printf( “%d\n”, i )
   )
)

isprime( me ) int me;
(
   int i, limit, ok;

   limit = sqrt( (double) me ) + 1;

   for ( i = 2; i < limit; ++i) (
      rd( “primes”, i , ? ok )
      if ( ok && ( me % i == 0 ) ) return 0
   )
   return 1
)
```