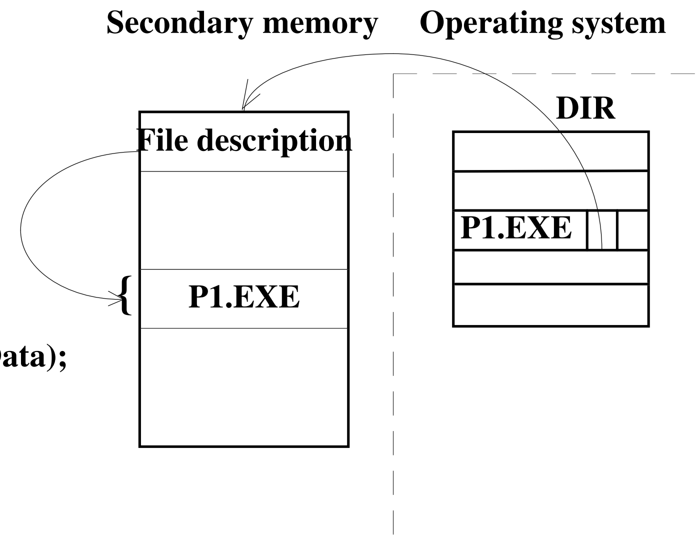
THE CONCEPT OF PROGRAM

```
readDataFromFile( fDataFile, cData);
  while ( cData != EOF ){
            processData( cData );
          readDataFromFile( fDataFile, cData);
              };
```



Programmer's interpretation:

Ordered set of instructions

Operating system's interpretation:

Executable file stored on secondary memory

THE CONCEPT OF PROCESS

PROCESS = CODE + PROCESS CONTROL BLOCK

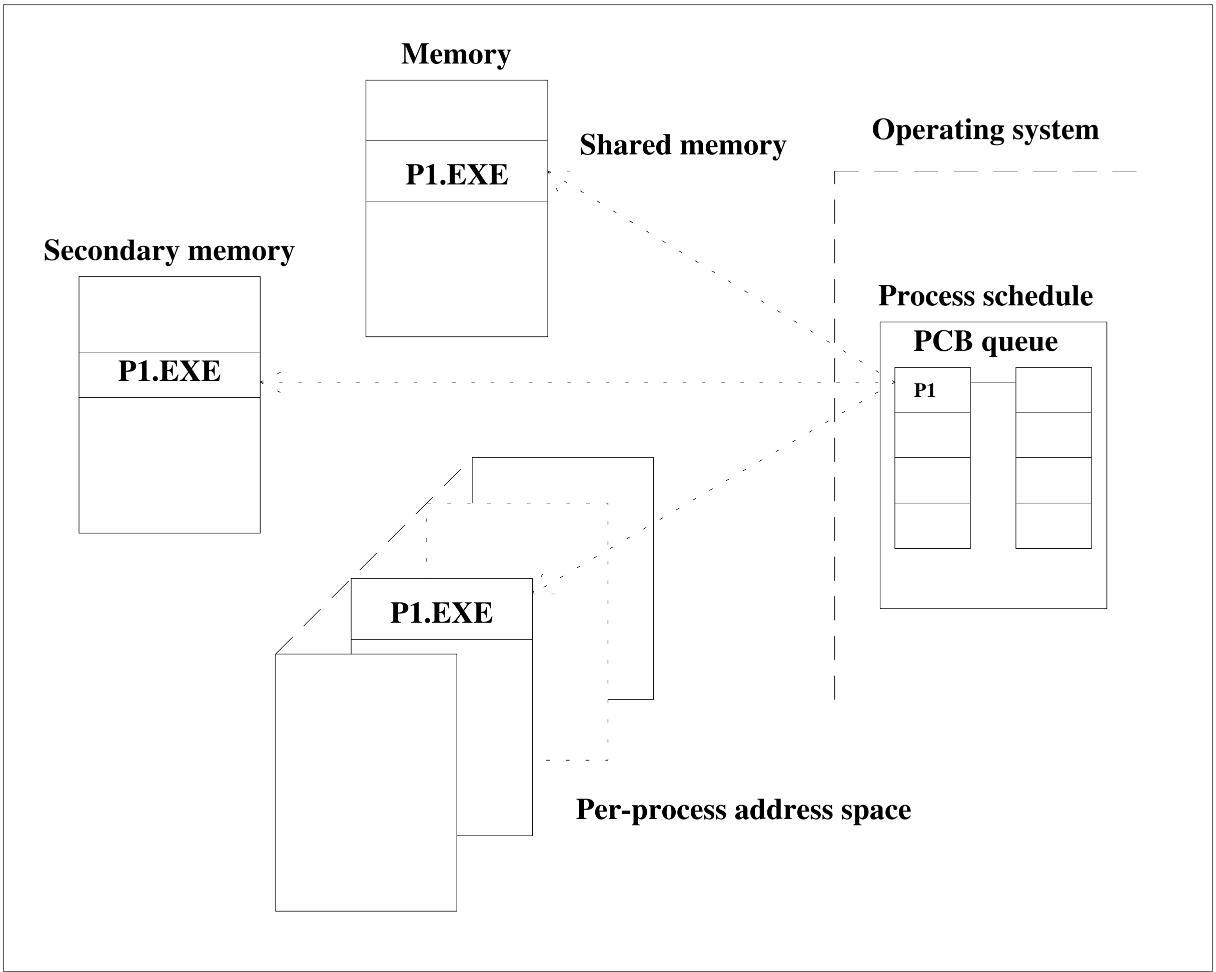
CODE: passive, defines the action to be executed by the process

PROCESS CONTROL BLOCK: describes the state of the process

Process creation involves the following four main actions:

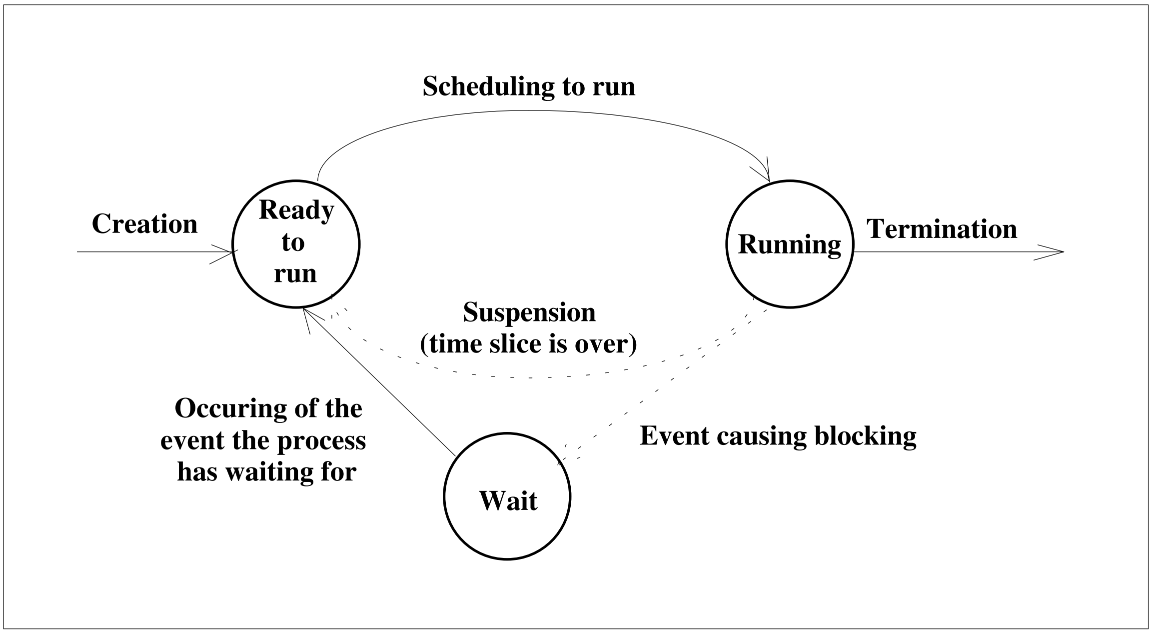
- **setting up the process control block,**
- **allocation of an address space and**
- **loading the program into the allocated address space and**
- **passing on the process control block to the scheduler.**

Description and location of a process after creation:

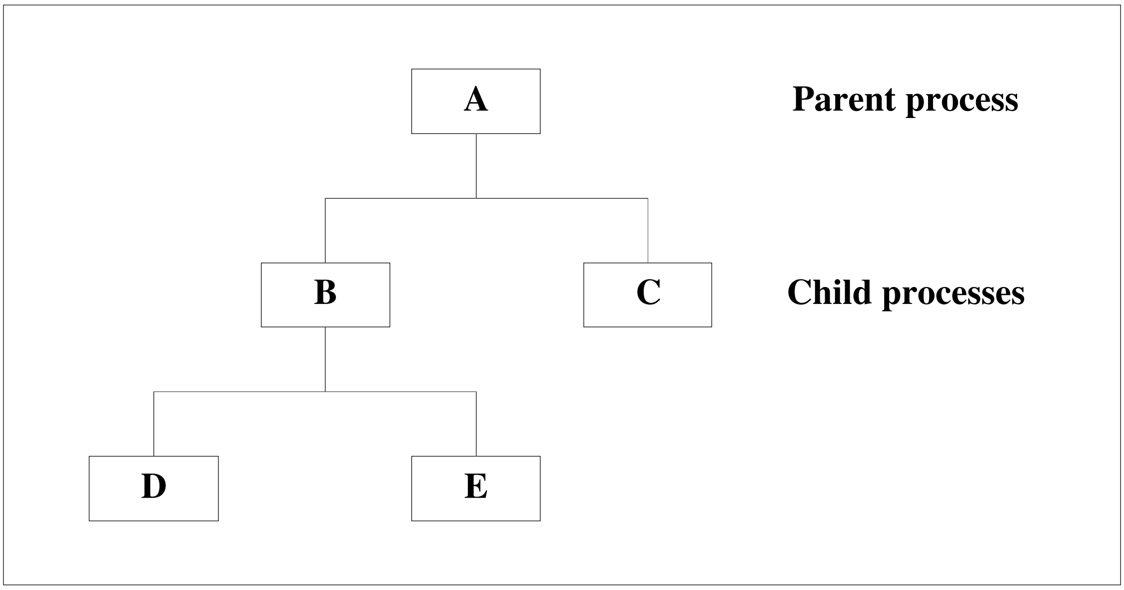


THE CONCEPT OF PROCESS (2)

PROCESS STATES and STATE TRANSITION DIAGRAM:



PROCESS SPAWNING:



THE CONCEPT OF THREAD

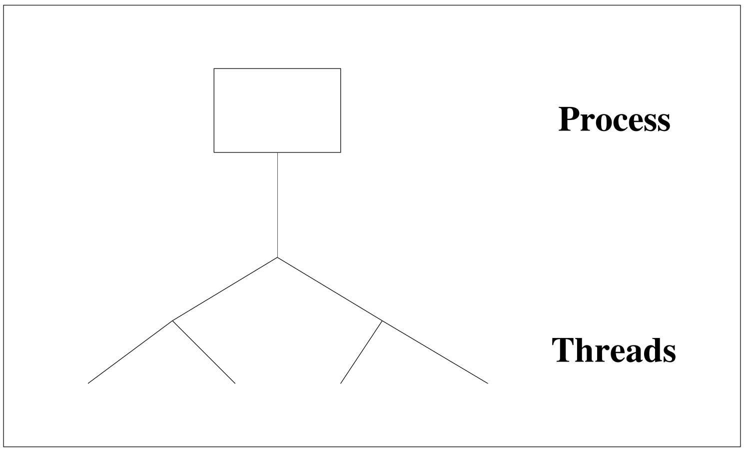
MAIN FEATURES:

- **Threads are created *within and belonging to* processes.**
- **All the threads created within one process share the resources of the process including the address space.**
- **Scheduling is performed on a per-thread basis.**
- **The thread model is a *finer grain scheduling model* than the process model.**
- **Threads have a similar *lifecycle* as the processes and will be managed mainly in the same way as processes are.**

ADVANTAGES:

- **Finer grained entities => more parallelism can be exposed.**
- **The creation of threads or the communication, synchronization or switch among threads are far less expensive operations then those for processes, since all threads belonging to the same process are sharing the same resources.**

THREAD TREE:

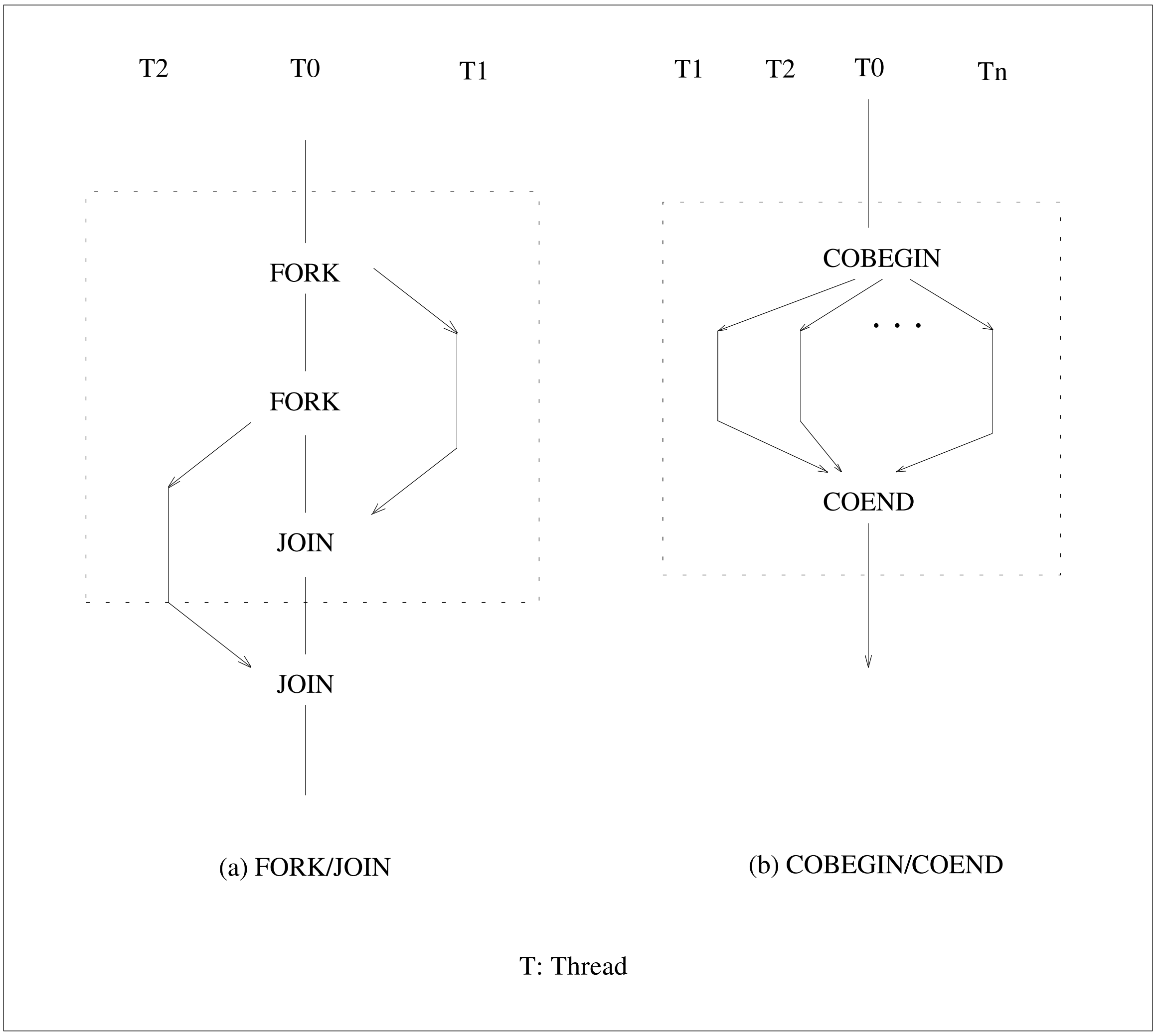


PROCESSES AND THREADS IN LANGUAGES

THREAD CREATION IN PROGRAMMING LANGUAGES

- **unsynchronized creation and termination (based on library functions)**
- **unsynchronized creation and synchronized termination**
  - **FORK and JOIN instructions**
  - **unstructured => error-prone**
- **synchronized creation and termination**
  - **COBEGIN/COEND pairs**
  - **structured method**

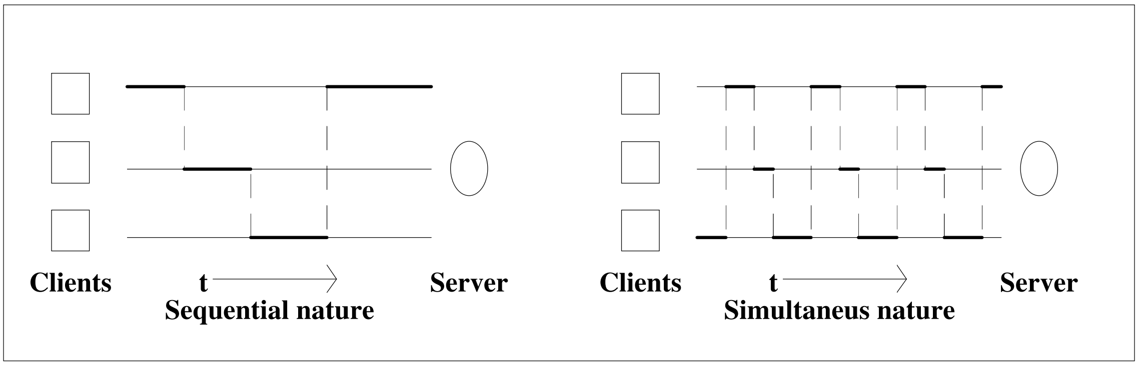
Comparison of FORK/JOIN and COBEGIN/COEND methods:



THE CONCEPTS OF
CONCURRENT AND PARALLEL EXECUTION

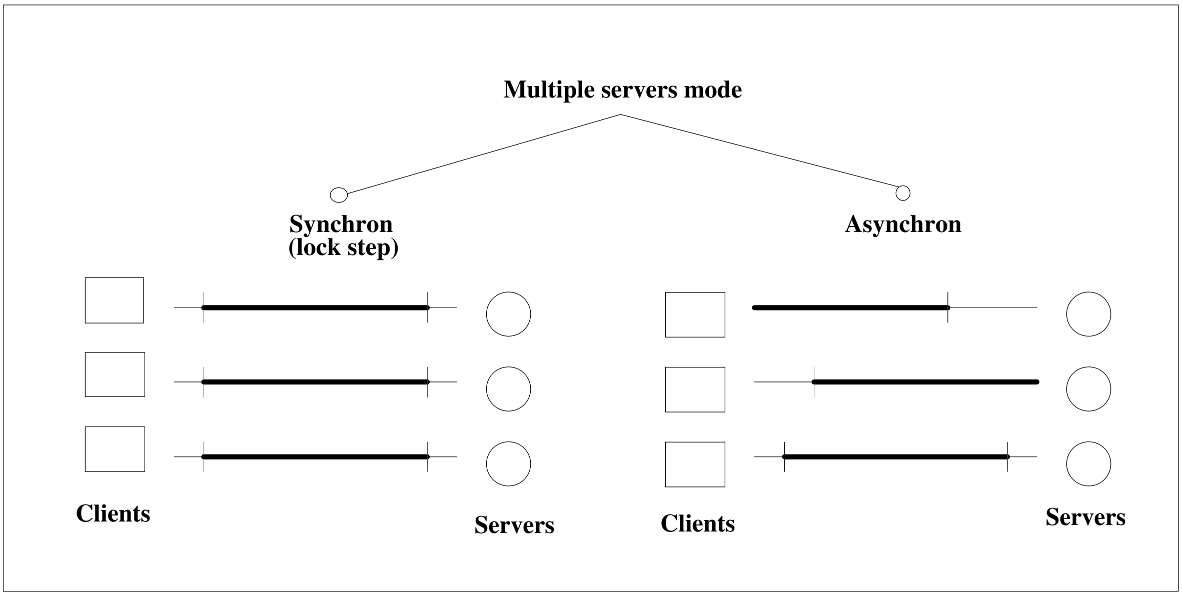
Concurrent execution

is the temporal behaviour of the *multiple clients* (requesters) *single server model* (see figure below) where one single client is served at any given moment. This model has a dual nature, it is *sequential* in a diminutive time scale, but *simultaneous* in a rather large time scale.



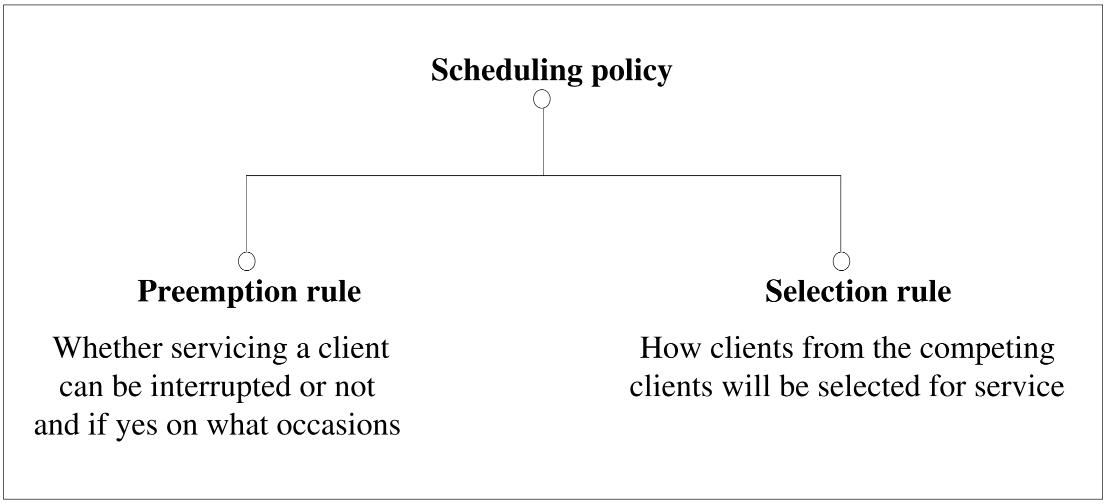
Parallel execution

is associated with the *multiple clients multiple servers model* (see figure below). Having more than one server (let us say processor) allows the servicing of more than one client (processes or threads) at the same time, which is called parallel execution.



SCHEDULING POLICY

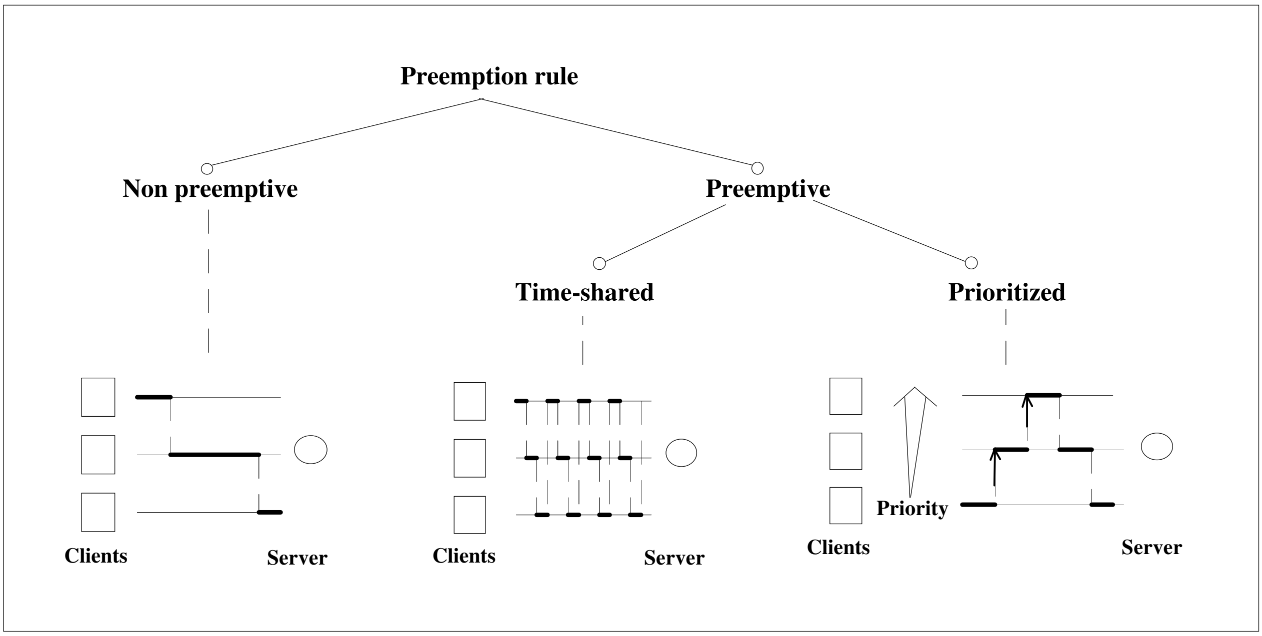
MAIN ASPECTS OF THE SCHEDULING POLICY



Selection rule

specifies an algorithm to determine a numeric value, which we will call *rank*, from the parameters like priority, time of arrival etc... The client with the *highest rank* will be scheduled for service.

BASIC PREEMPTION SCHEMES



CONCURRENT AND PARALLEL PROGRAMMING LANGUAGES

Classification of programming languages:

Languages	1-client 1-server model	N-client 1-server model	1-client N-server model	N-client N-server model
Sequential	+	-	-	-
Concurrent	+	+	-	-
Data parallel	+	-	+	-
Parallel	+	+	-	+

Languages can be classified according to the available language constructs:

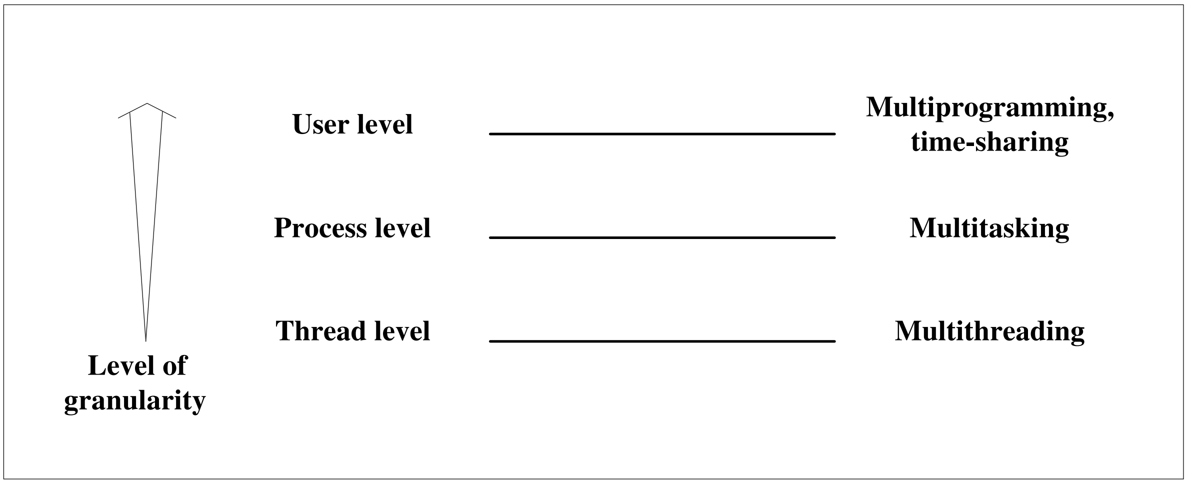
- Languages that do not contain any constructs to support the N-client model belong to the class of **sequential** (or traditional) **languages** (For example: C, Pascal, FORTRAN, Prolog, Lisp).

- **Concurrent languages** employ constructs to specify the N-client model by specifying parallel threads and processes but miss language constructs to describe the N-server model (For example: Ada, Concurrent Pascal, Modula-2, Concurrent Prolog).

- **Data parallel languages** introduce special data structures that are processed in parallel, element by element. They also apply special mapping directives to help the compiler in the optimal distribution of parallel data structures among processors. (For example: High Performance FORTRAN, DAP FORTRAN, DAP Prolog, Connection Machine C and Lisp). Finally,

- **Parallel languages** extend the specification of the N-client model of concurrent languages with processor allocation language constructs that enable the use of the N-server model (For example: Occam-2, 3L Parallel C, Strand-88).

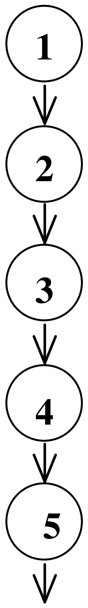
CONCURRENT EXECUTION MODELS

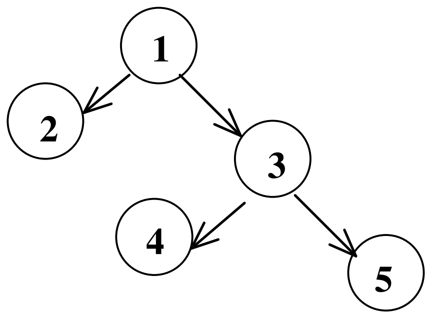


- Thread level concurrent execution is termed as **multithreading**. In this case multiple threads can be generated to each process, and these threads will be executed concurrently on a single processor under the control of the operating system.
- **Process level concurrent** execution is usually called *multitasking*. Each widely used present operating system supports this concept. **Multitasking** refers to concurrent execution of processes. Multiple ready to run processes can be created either by a single user if process spawning is feasible, or by multiple users participating in multiprogramming or in time-sharing.
- User level concurrent or parallel execution can be either *multiprogramming* or *time-sharing*.
    - **Multiprogramming** aims at the effective utilization of the processor by creating multiple ready-to-run processes, each belonging **to different users**. If the actually running process becomes blocked, because it has to wait for a particular event, such as completition of I/O, the processor will be scheduled to another ready to run process (if any). Notice that multiprogramming is *internally* implemented as *multitasking* for *independent tasks* arising from different users.
    - **time-sharing** has the objective to offer *adequate quality* computer *services* to a number of different users through terminals. This mode of operation intends to guarantee a *short access time* to all users instead of striving for the effective utilization of the processor.

SEQUENTIAL VERSUS PARALLEL COMPUTATION

Tasks





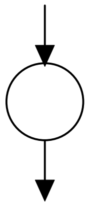
Parallel Computation
Two or more processors

Sequential computation
A single processor

✓**Real Dependences" Communication between tasks.**

✓**Parallel Processing is becoming feasible as hardware becomes cheaper.**

TASK GRANULARITY



Type of Task

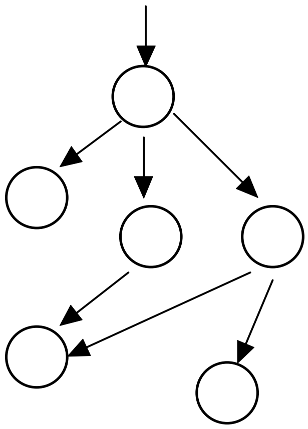
- ✓ **Bit level operation**
- ✓ **Machine language instruction**
- ✓ **Function level operation**

Fine granularity

↕

Coarse granularity

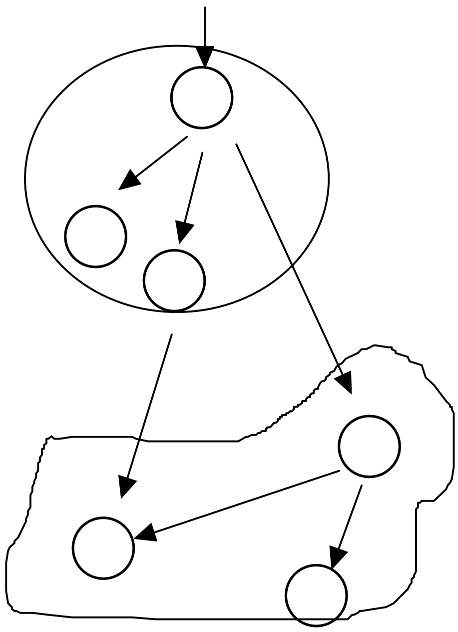
Fine granularity



✓***More*** **communication** ***between*** **tasks**

✓**Higher level of parallelism**

Coarse granularity



✓**Less communication between tasks**

✓**Lower level of parallelism**

TYPES OF AVAILABLE PARALLELISM

Data and Computation Decomposition

a) How ?

❒ **Data Parallelism**

Decompose and distribute data. All the processors execute the same code but operating on local data

❒ **Function Parallelism**

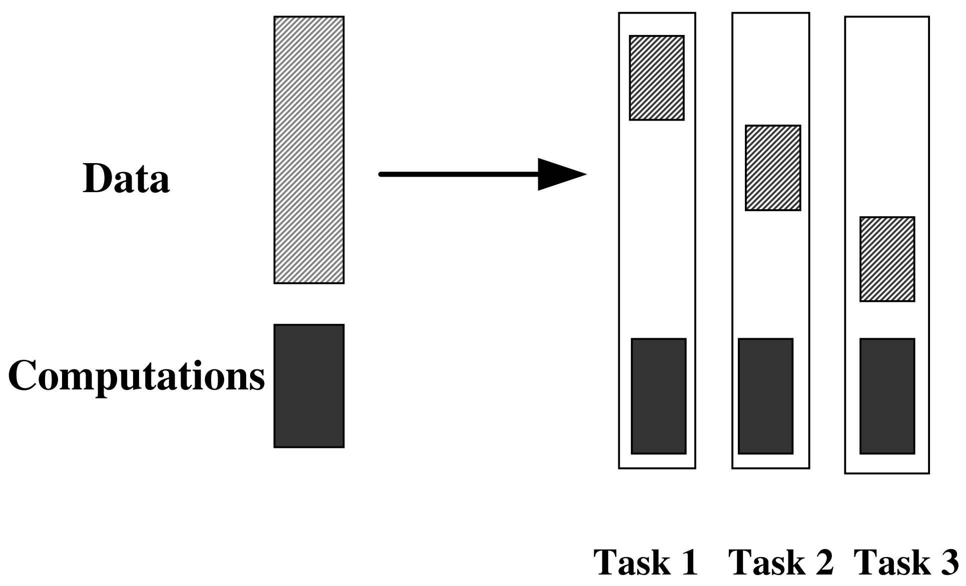
Decompose COMPUTATIONS into pieces (functions).
Data are taken to the tasks where they are needed.

❒ **Pipelining**

A given (complex) computation must be performed on a large set of data. Decompose the COMPUTATION into functions.

DATA PARALLELISM

Decompose DATA into pieces. All the tasks perform the same computation operating with their own piece of data.



✓ The level of parallelism is proportional to the amount of data

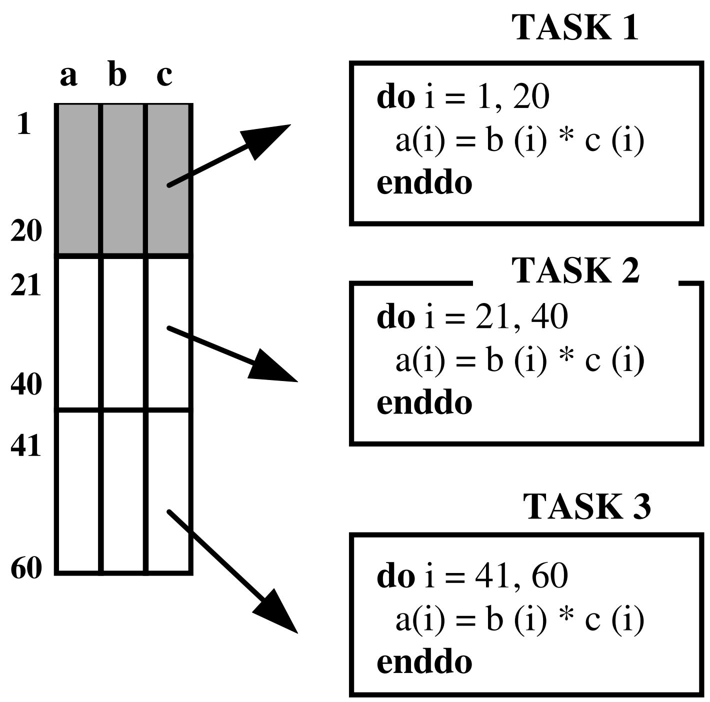
✓ Suitable for scientific computations on large and regular data structures (vectors, matrices, etc.).

DATA PARALLELISM:

A SIMPLE EXAMPLE

A typical example to exploit data parallelism appears in do-loops:

```
do  i = 1,  60
   a (i)  =  b  (i) * c (i)
enddo
```

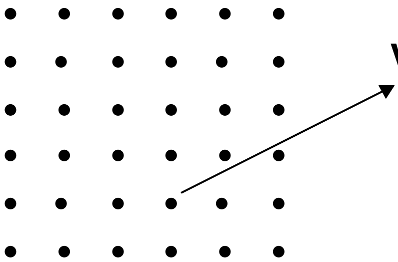


DATA PARALLELISM:

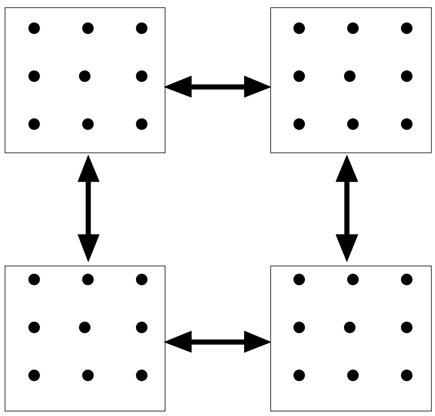
ANOTHER EXAMPLE

- **Another typical form of data parallelism is applied in solving Laplace's equations on a square based on the Jacobi iterative method.**

- **In such neighbourhood-oriented computations we get processes with a regular communication pattern.**

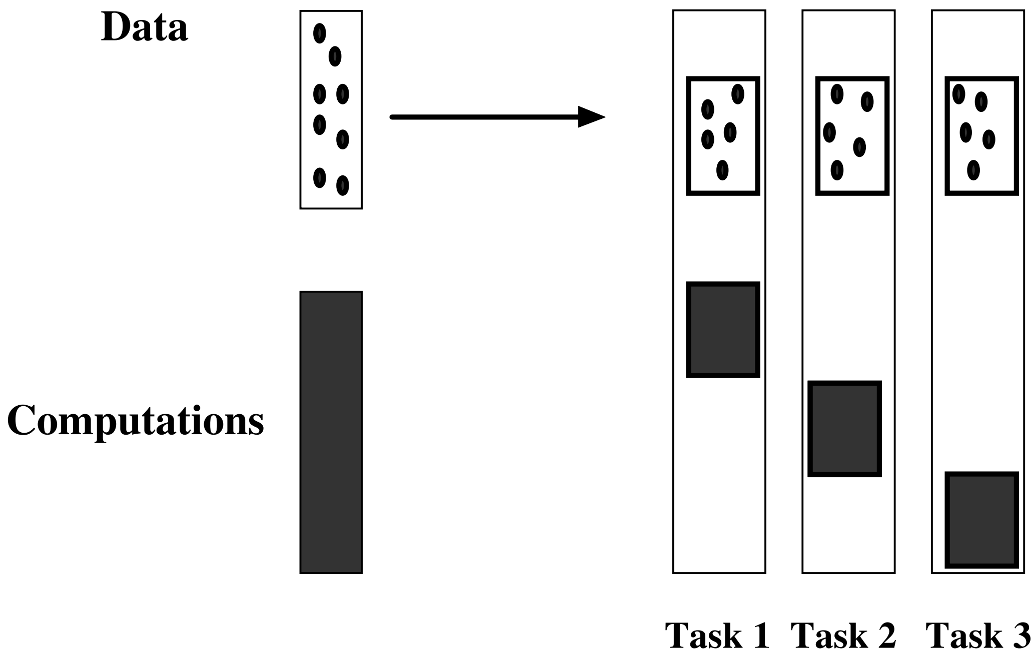


$$V_{ij}^{t+1} = \frac{1}{4}\left(V_{i-1j}^{t} + V_{i+1j}^{t} + V_{ij-1}^{t} + V_{ij+1}^{t}\right)$$



FUNCTION PARALLELISM

- **Decompose COMPUTATIONS into pieces (functions).**
- **Data are taken to the tasks where they are needed.**

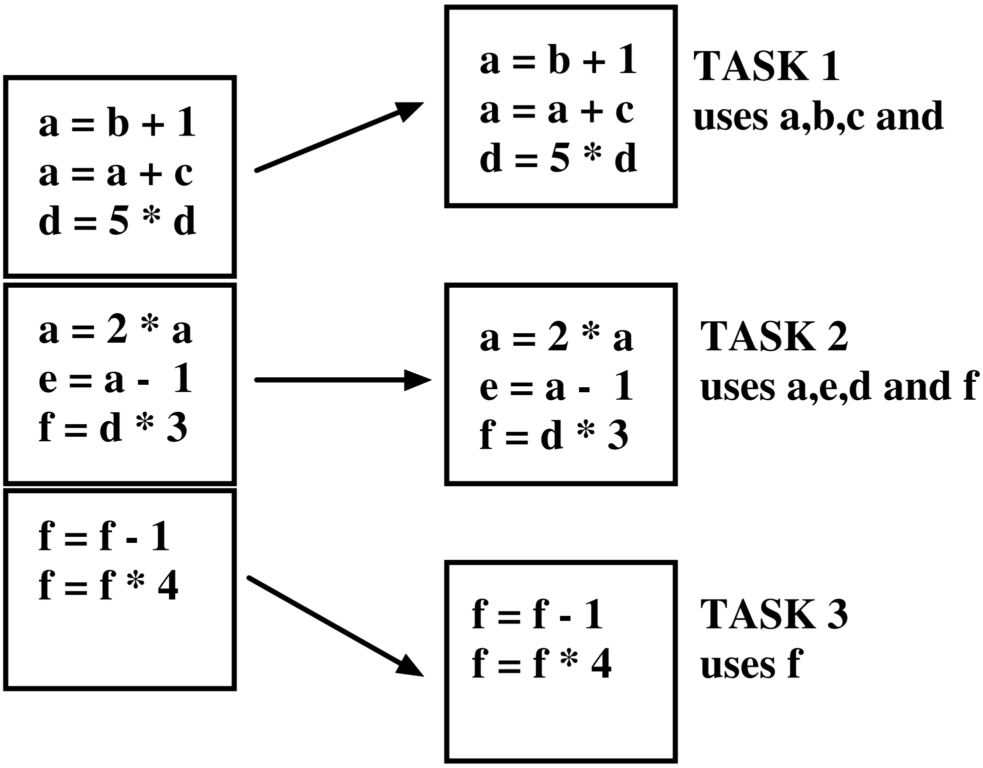


✓ The level of parallelism is limitted (number of functions).

✓ Suitable for non regular computations (non regular data).

FUNCTION PARALLELISM:

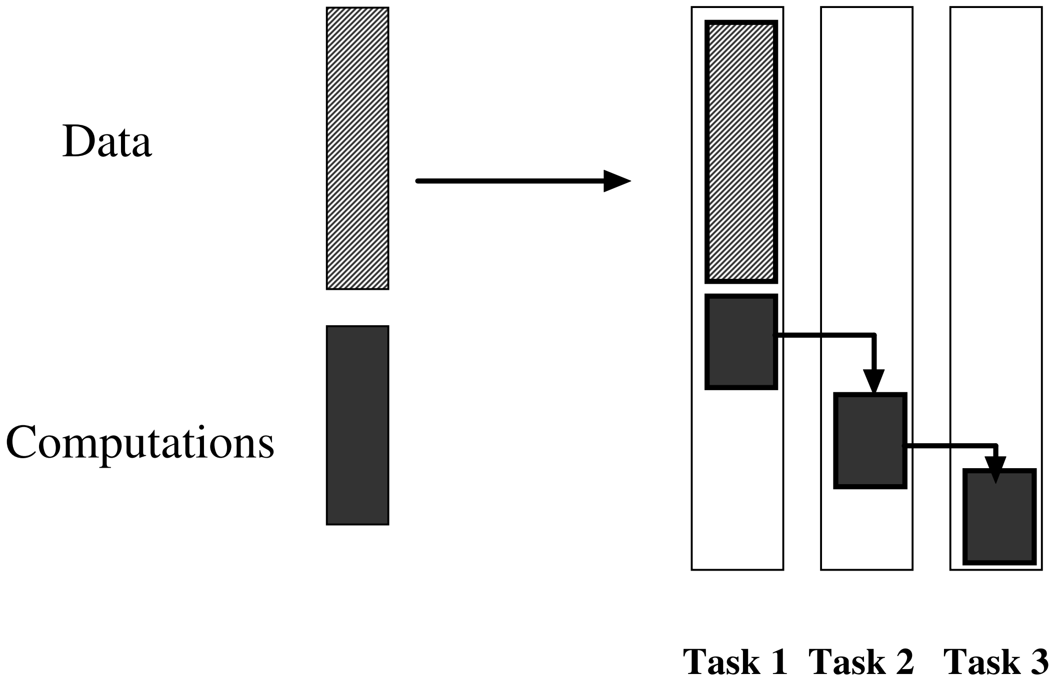
A SIMPLE EXAMPLE



✓ **Usually, we get non regular communication pattern.**

PIPELINING

A given (complex) computation must be performed on a large set of data. Decompose the COMPUTATION into functions.

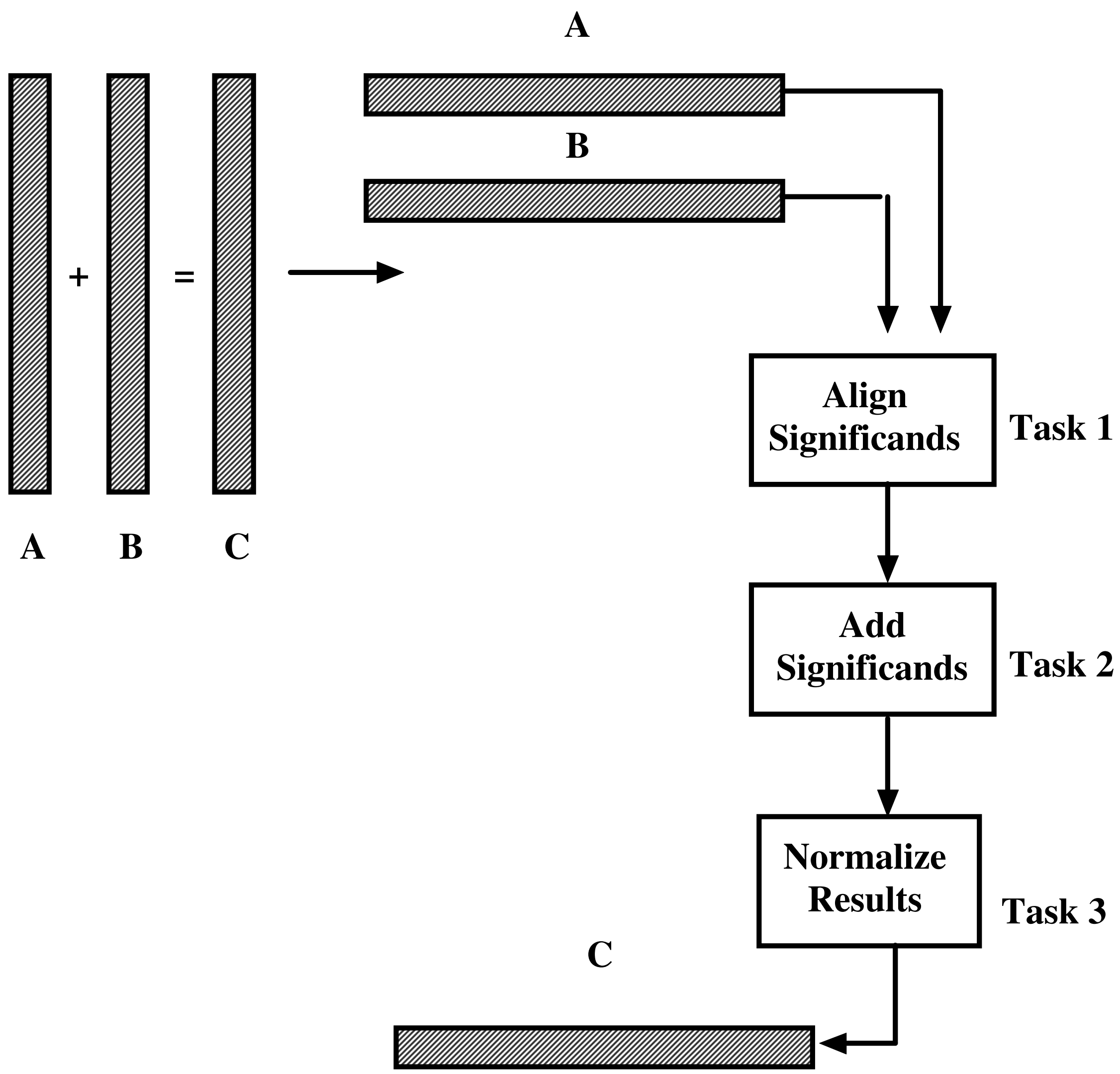


- ✓ **The level of parallelism is limited (number of functions)**
- ✓ **Also suitable for regular computations on regular data.**

PIPELINING:

AN EXAMPLE

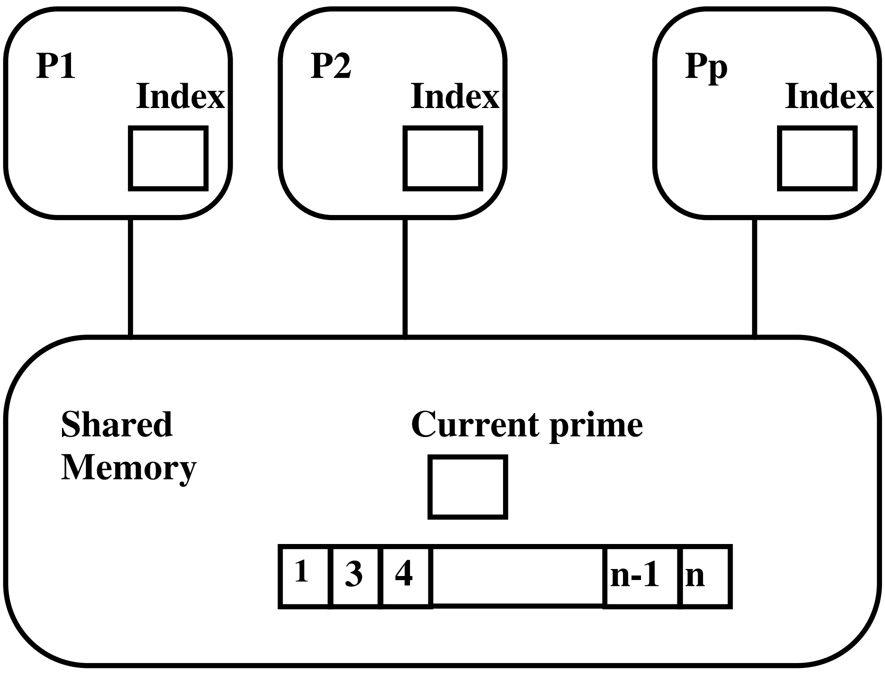
A typical pipelining example used in vector computers to add vectors:



FUNCTION PARALLEL SOLUTION
OF THE SIEVE OF ERATOSTHENES

Shared memory model
for function parallel Sieve of Erathostenes algorithm

Each processor has its own private loop index and shares access to other variables with all the other processors.



LIMITATIONS OF THE FUNCTION PARALLEL SOLUTION

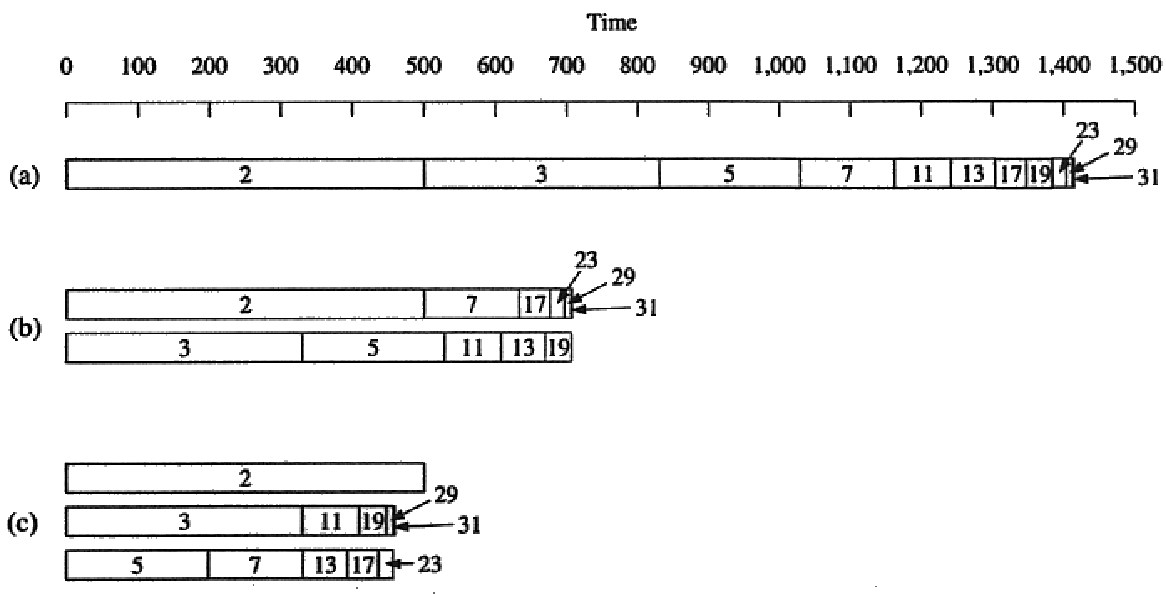
- **Study of how adding processors reduces the execution time of the function parallel Sieve of Eratosthenes algorithm when n = 1,000.**

- **The number in the bar represents the prime whose multiples are being marked. The length of the bar is the time needed to strike these multiples.**

  a) Single processor strikes out all composite numbers in 1,411 units of line.

  b) With two processors execution time drops to 706 time units.

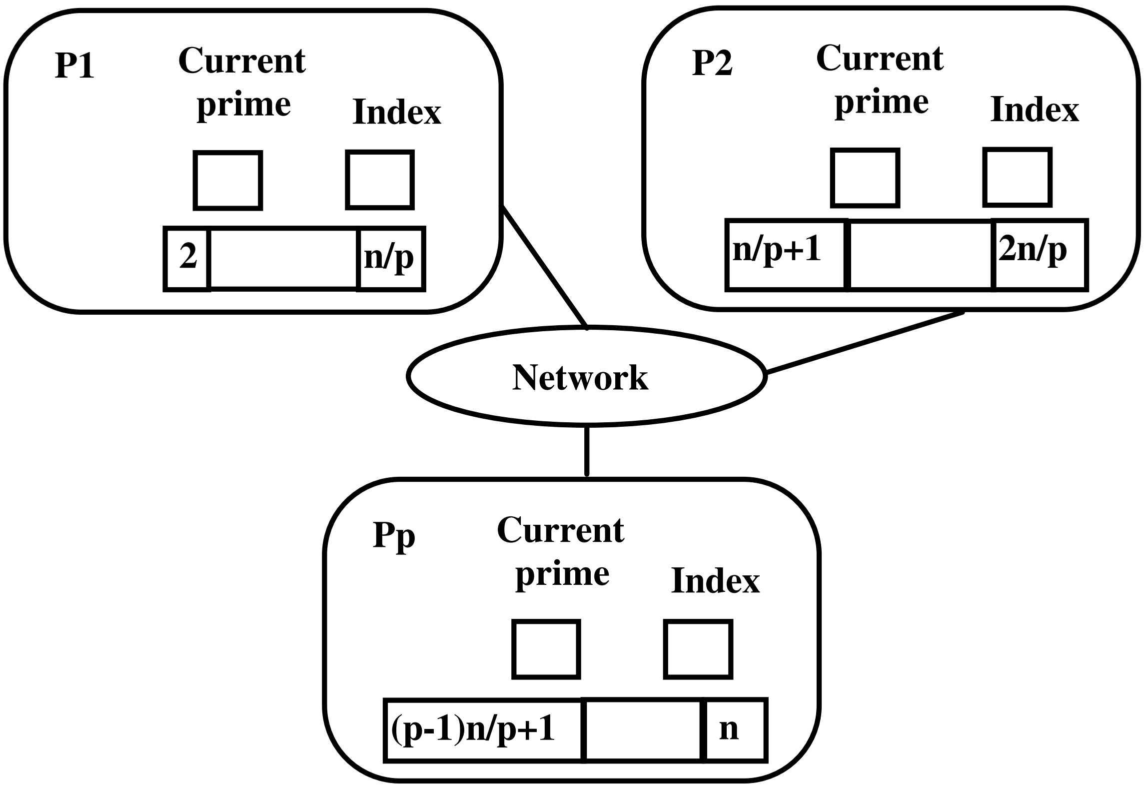
  c) With three or more processors execution time is 499 time units, the time needed for a processor to strike all multiples of 2.



DATA PARALLEL SOLUTION
OF THE SIEVE OF ERATOSTHENES

Distributed memory model
for data parallel Sieve of Erathostenes algorithm

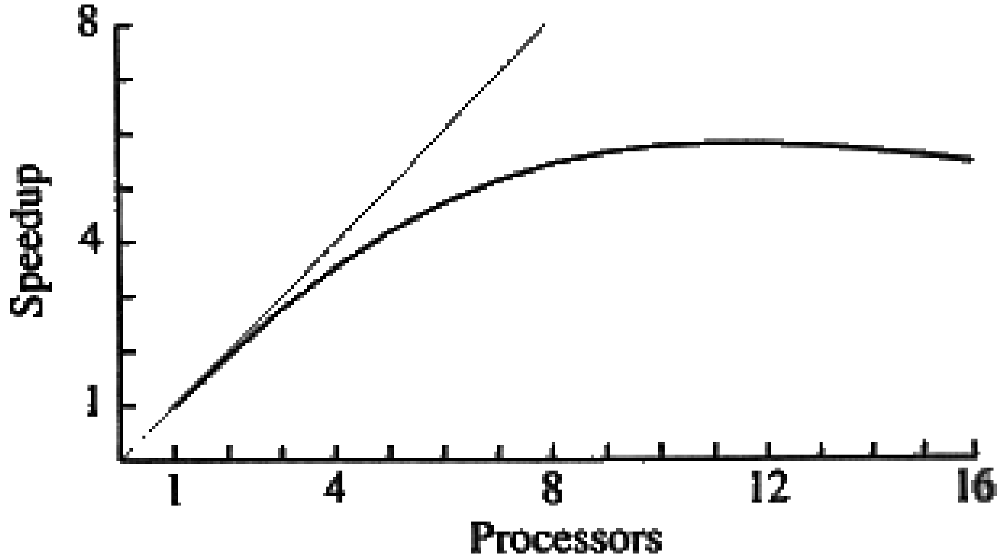
- **Each processor has its own copy of the variables containing the current prime and the loop index.**
- **Processor 1 finds primes and communicates them to the other processors.**
- **Each processor iterates through its own portion of the array of natural numbers, marking multiples of the prime.**



PERFORMANCE ESTIMATION OF THE DATA PARALLEL DISTRIBUTED MEMORY SOLUTION

Estimated speedup of the data parallel Sieve of Eratosthenes algorithm assuming that n = 1,000,000 and $\lambda = 100\chi$ where

- **it takes χ time units for a processor to mark a multiple of a prime as being a composite number**
- **processor 1 spends λ time units each time it passes a number to another processor**

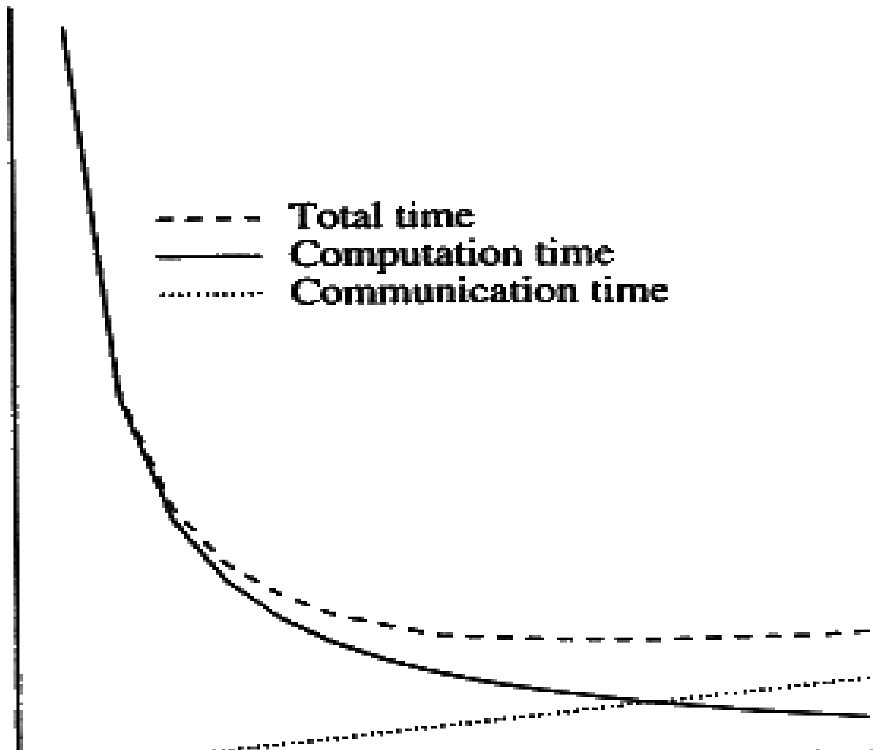


PERFORMANCE ESTIMATION OF THE DATA PARALLEL DISTRIBUTED MEMORY SOLUTION

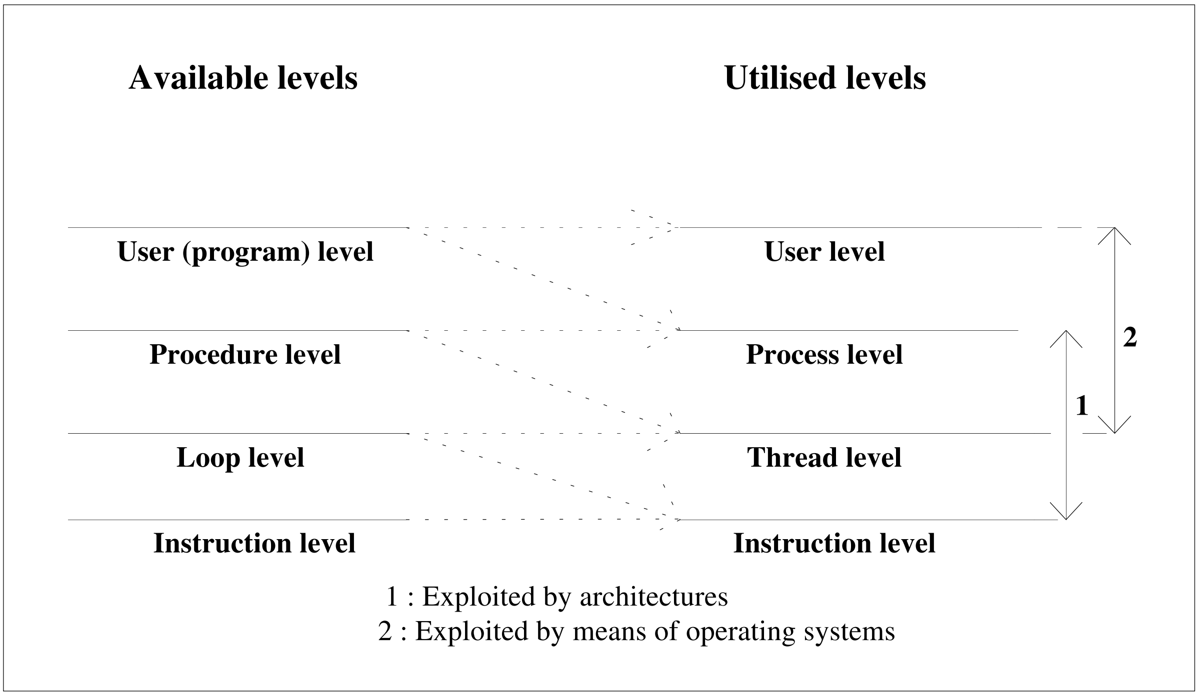
Total execution time of the data-parallel Sieve of Eratosthenes algorithm is the sum of

- **the time spent computing and**
- **the time spent communicating.**

- **Computation time is inversely proportional to the number of processors;**
- **Communication time is directly proportional to the number of processors**



LEVELS OF AVAILABLE FUNCTIONAL PARALLELISM



- **Available instruction-level parallelism means that particular instructions of a program may be executed in parallel.**
- **Parallelism may also be available at the loop-level. Here subsequent loop iterations are candidates for parallel execution.**
- **There is also parallelism available at the procedure-level in form of parallel executable procedures.**
- **Parallelism is also available at the user-level (which we consider to be of coarse-grained parallelism). Multiple, independent users are a key source of parallelism occurring in computing scenarios.**

Evidently, in a problem solution different levels of parallelism are not exclusive but, they may coexist at the same time.

UTILISATION OF FUNCTIONAL PARALLELISM

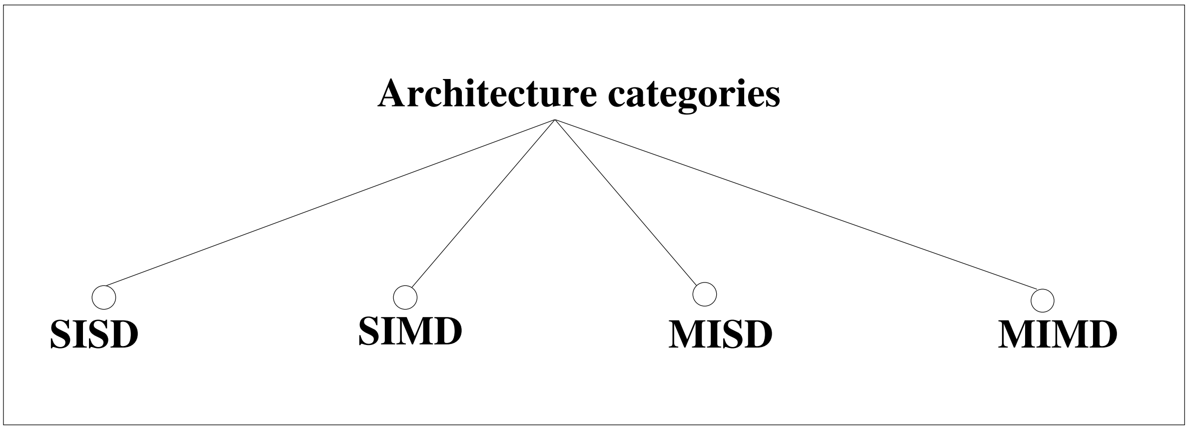
Available parallelism can be utilised by architectures, compilers and operating systems conjointly for speeding up computation:

- **Instruction-level parallelism**
  - It is **utilised by** architectures capable of parallel instruction execution. Such architectures are referred to as instruction-level function-parallel or simply instruction-level parallel architectures (**ILP-architectures**).
  - It must be **detected** either by a dedicated compiler (called usually parallel optimising compiler) or by the ILP-architecture itself.
- **Available loop-, procedure- and program-level parallelism can be utilised at the thread and/or at the process-level.**
  - Threads and processes can be created either **by the programmer** using parallel languages or by operating systems that support multithreading or multitasking.
  - They can also automatically be generated **by parallel compilers** during compilation of high-level language programs.
- **There are two different ways to execute threads and processes.**
  - They can be executed in parallel by specialised architectures referred to as **multithreaded and MIMD architectures**, respectively. Multithreaded architectures are typically specialised processors able to perform very fast context switch.
  - The other way to execute threads and processes concurrently is the use of architectures that run threads or processes in sequence, under the supervision of a **multithreaded or multitasking operating system**.

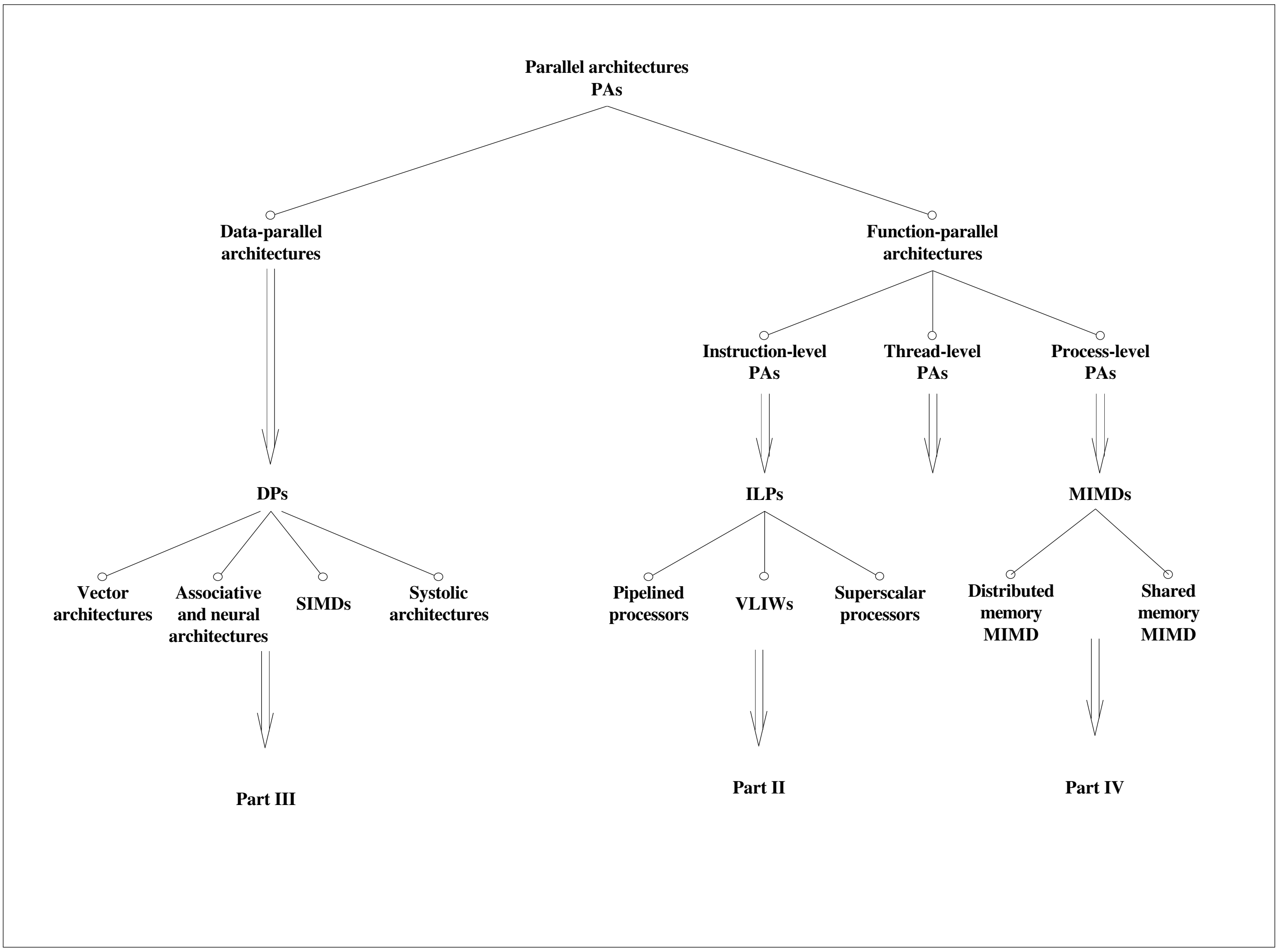
CLASSIFICATION OF PARALLEL ARCHITECTURES

Flynn's classic taxonomy is based on the number of the control units as well as of the processors available in a computer. Accordingly, he introduced the notions of

- **single instruction stream (i.e. the architecture has one single control unit producing one single stream of instructions), abbreviated as SI,**
- **multiple instruction streams (i.e. the architecture has multiple control units, each of them producing a distinct stream of instructions), abbreviated as MI,**
- **single data stream (i.e. one single processor is available which executes one single stream of data), abbreviated as SD and**
- **multiple data streams (i.e. multiple processors are available each of them executing a distinct stream of data), abbreviated as MD.**



PROPOSED CLASSIFICATION



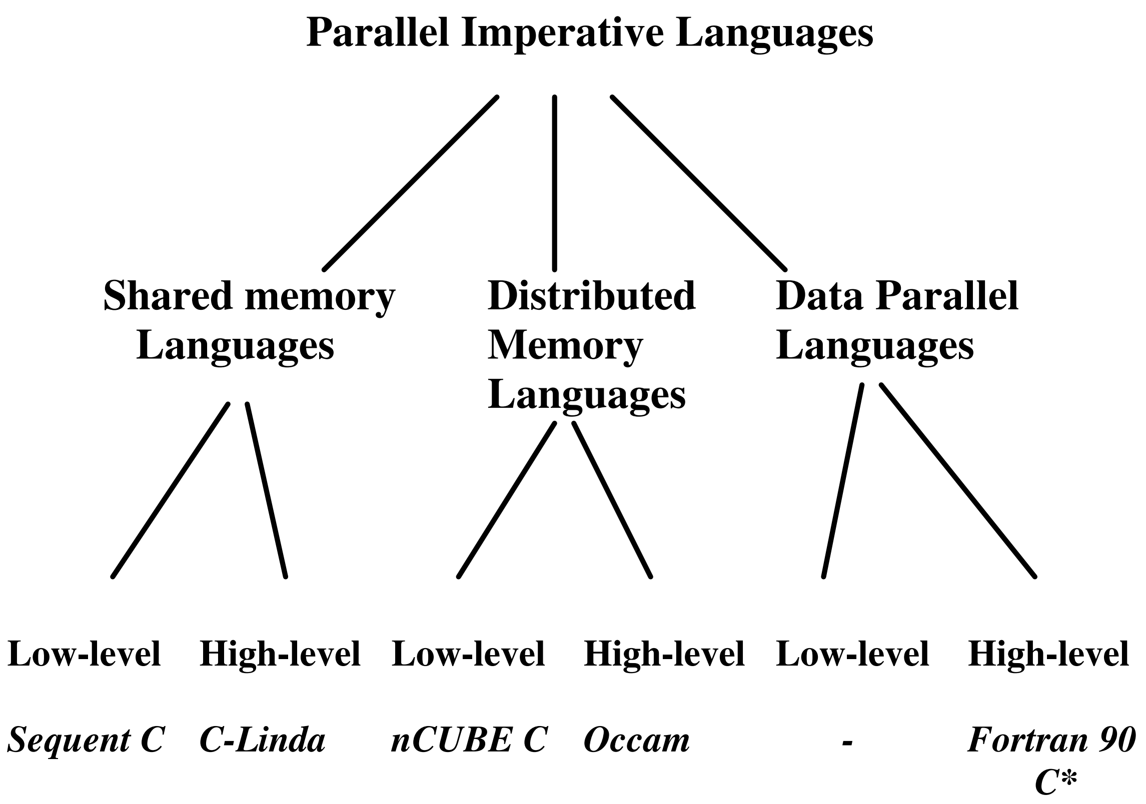
PARALLEL TECHNIQUES IN COMPUTER ARCHITECTURES

There are two basic forms of exploiting parallelism by parallel computer architectures:

1. **Pipelining: a number of functional units are employed in sequence to perform a single computation.**
2. **Replication: replicated functional units can execute the same operation simultaneously on as many data elements as many replicated computational recources are available.**

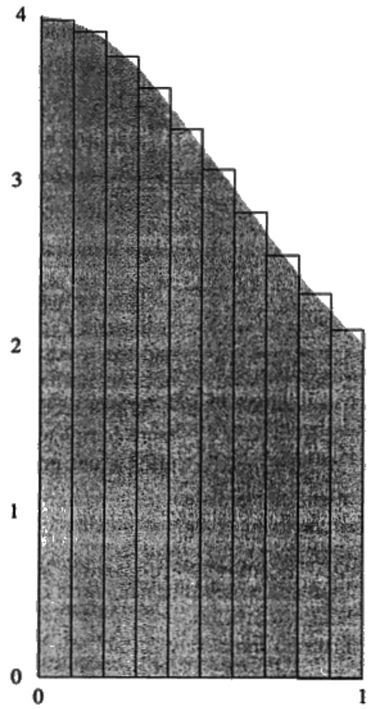
	Pipelining	Replication
Vector processors	+	
Systolic arrays	+	+
SIMD (array) proc.		+
Associative proc.		+
Pipelined proc.	+	
VLIW proc.		+
Superscalar proc.	+	+
Multithreaded machines	+	+
Multicomputers	+	+
Multiprocessors		+

CLASSIFICATION OF PARALLEL IMPERATIVE LANGUAGES



A SIMPLE APPLICATION

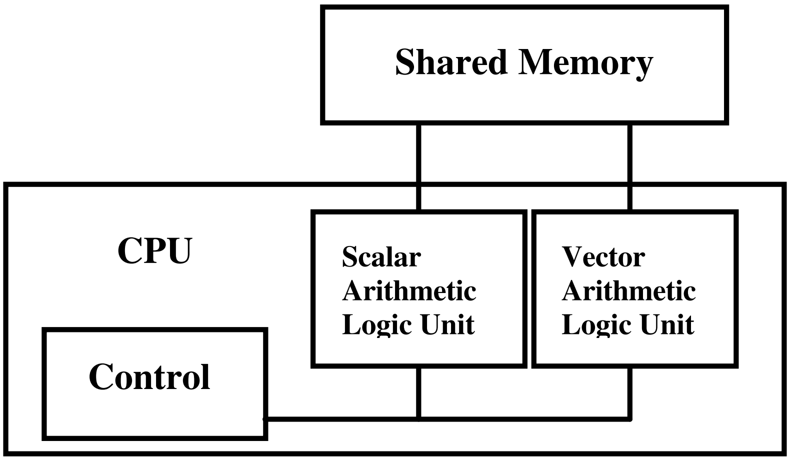
- **A simple numerical integration problem to be solved by each language.**
- **The parallel program must compute an approximation to π by using numerical integration to find the area under the curve $4/(1+x^2)$ between 0 and 1.**



- **The interval [0, 1] is divided into n subintervals of width l/n. For each of these intervals the algorithm computes the area of a rectangle whose height is such that the curve $4/(1+x^2)$ intersects the top of the rectangle at its midpoint. The sum of the areas of the n rectangles approximates the area under the curve. Increasing n reduces the difference between the sum of the rectangle's area and the area under the curve.**
- **This algorithm is data-parallel, since the areas of all the rectangles can be computed simultaneously. Computing the area of each rectangle requires the same amount of work: hence load balancing is insignificant.**

FORTRAN 90 PROGRAMMER´S MODEL

- **The model of parallel computation is similar to a PRAM.**
- **A CPU and a vector unit share a single memory.**
- **The CPU executes sequential instructions, accessing variables stored in the shared memory.**
- **To execute parallel operations, the CPU controls the vector unit, which also stores and fetches data to and from the shared memory.**



FORTRAN 90 LANGUAGE FEATURES

- **Fortran 90 introduces the notion of an array constant:**

  (/ 2, 4, 6, 8, 10 /)

- **An implied DO notation can simplify the specification of array constants:**

  (/ (I, I = 2, 10, 2) /)

- **Fortran 90 also allows operations on arrays.**

- **When applied to an array, the unary intrinsic operators + and - return an array of the same dimensions, where the elements in the result array are found by applying the operator to the corresponding elements in the operand array. For example, given the array declarations**

```
REAL, DIMENSION(100, 50) :: X, Y
REAL, DIMENSION(100) :: Z
```

- **the following are examples of legal array expressions:**

```
X + Y            ! Array of shape (100, 50), elements X(I,J) + Y(I,J)
X + 1.0          ! Array of shape (100, 50), elements X(I,J) + 1.0
X .EQ. Y         ! Array of shape (100, 50), elements having value
                 ! .TRUE. if  X(I,J) .EQ. Y(I,J) otherwise .FALSE.
X(1:100,3) + Z   ! Array of shape (100), elements X(I,3) + Z(I)
```

- **The WHERE statement allows the programmer to specify which array elements are to be active:**

```
WHERE (A > 0.0) A = SQRT(A)
```

- **The syntax of the most general WHERE statement is**

```
WHERE (logical-array-expression)
        array-assignment-statements
ELSEWHERE
        array-assignment-statements
END WHERE
```

Fortran 90 program
to compute π using numerical integration

```
INTEGER, PARAMETER :: N = 131072
INTEGER, PARAMETER :: LONG = SELECTED_REAL_KIND(13,99)
REAL(KIND=LONG) PI, WIDTH
INTEGER, DIMENSION(N) :: ID
REAL(KIND=LONG), DIMENSION(N) :: X, Y

WIDTH = 1.0-LONG / N
ID = (/ (I, I = 1, N) /)
X = (ID - 0.5) * WIDTH
Y = 4.0 / (1.0 + X * X)
PI = SUM(Y) * WIDTH
10 FORMAT (  'ESTIMATION OF PI WITH ',I6, &
             'INTERVALS IS ',F14.12)
PRINT 10, N, PI
END
```

C* PROGRAMMER'S MODEL

- **C* programmers imagine they are programming a SIMD computer consisting of a frontend uniprocessor attached to an adaptable back-end parallel processor.**

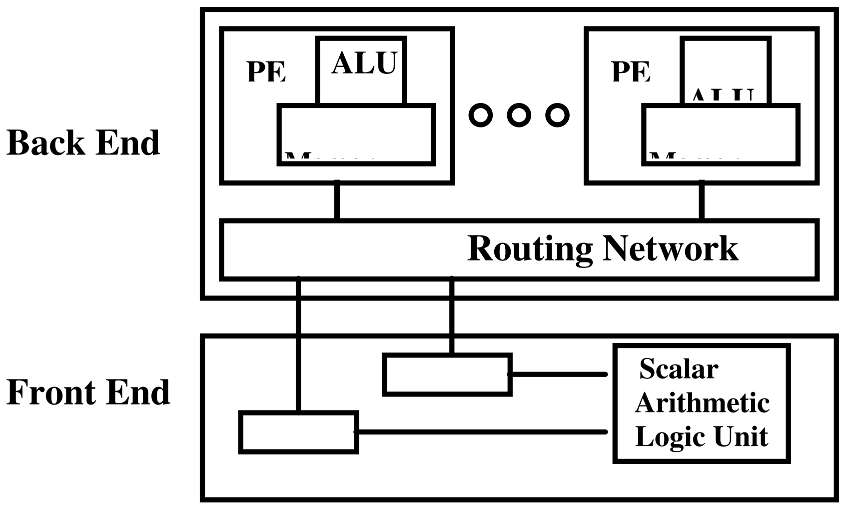
- **The front-end processor stores the sequential variables and executes the sequential code.**

- **The back-end processors store the parallel variables and execute the parallel portions of the program.**

- **Each processing element in the back end has its own local memory.**

- **There is a single flow of control; at any one time either the front-end processor is executing a sequential operation, or the back-end processors are executing a parallel operation.**

- **The back-end processor array is adaptable—that is, programmers can select the size and shape of the processing elements they want to activate at different points in the same program.**



C* LANGUAGE FEATURES

- **C* introduces the notion of a shape, which specifies the way in which parallel data are organised. By declaring a variable to be of some shape type, programmers indicate that they want the ability to perform parallel operations on that variable:**

```
shape [128] [128] foo;
```

- **The *with* statement activates the positions of a shape, setting up a context in which variables of that type can be manipulated in parallel:**

```
with (foo) ( aa = bb + cc; }
```

performs an element-wise addition on each component of *bb* and *cc* and stores the resulting values in the corresponding components of *aa*.

- **C* has a set of built-in reduction operators that reduce parallel values into sequential values:**

```
main() {
        real sum;
        with (foo) { sum = ( += aa ); }
}
```

- **The where statement gives C* programmers the ability to perform operations on a subset of the elements of a parallel variable:**

```
with (foo) {
        where (aa > 0.0 ) {
                aa = bb + cc;
        }
}
```

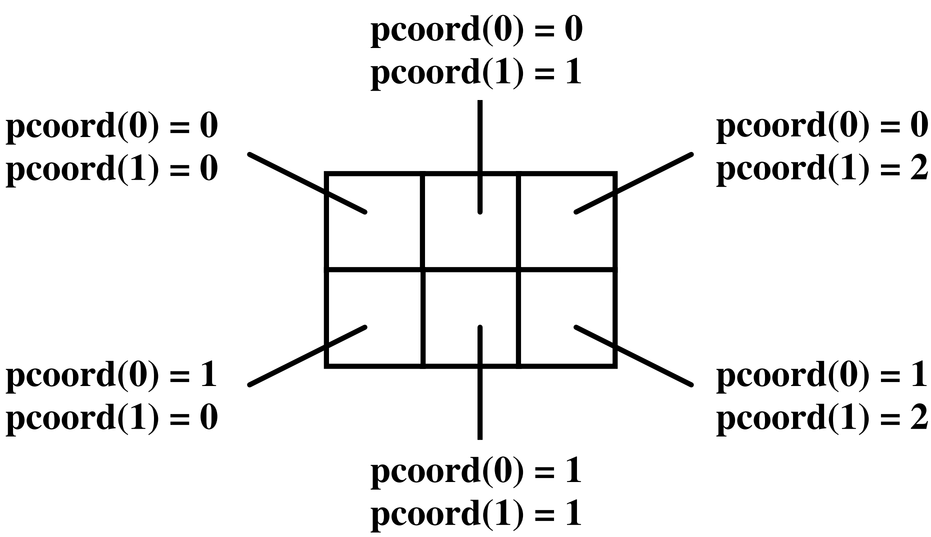
C* LANGUAGE FEATURES (CONT.)

- **The with statement selects the current shape. If an operation is to be performed on certain elements of a parallel variable, the where statement can be used to "set the context":**

```
shape [65536]students:
float:students credits, grade_points, gpa;
int:students count;
float upper_gpa, lower gpa;
with (students) {
        where (credits >= 64.0) {
                gpa = grade points / credits;
                count = 1;
                upper gpa = (+= gpa) / (+= count);
        }
}
```

computes the mean grade-point average of upper-division students (those who have at least 64 credits).

- **Sometimes the value of each element of a parallel variable depends upon its relative location within the shape. The function *pcoord*, passed a dimension number, returns each element of a parallel variable to its position within that dimension of the shape. The figure below illustrates the workings of function *pcoord*.**

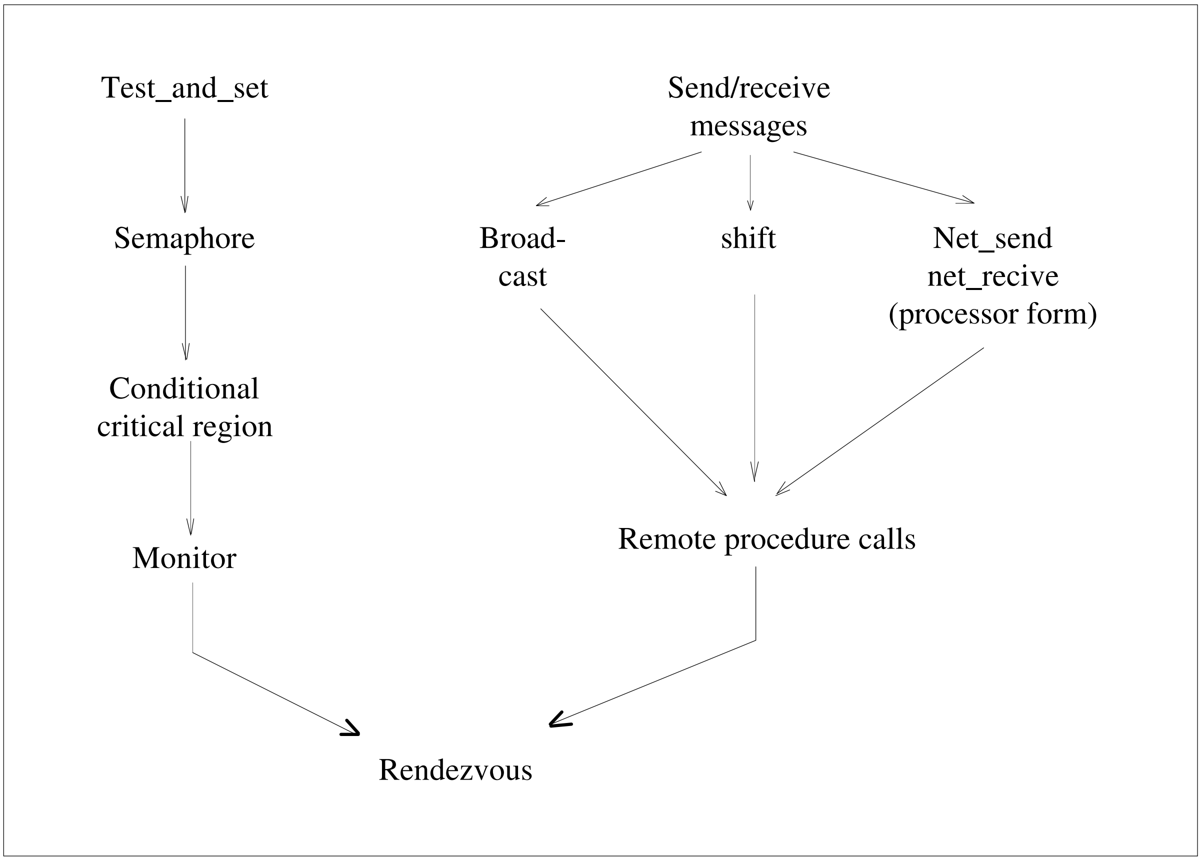


Pi computation program written in C*, version 6.0

```
#define INTERVALS (8192 * 16)
shape [INTERVALS] span;

main () {
    double sum;                      /* Sum of areas */
    double width=l.0/INTERVALS;      /* Width of rectangle */
    with(span) {
        double:span x;               /* Midpoint of rectangle
                                        on x axis */
        x = (pcoord(0) + 0.5) * width;
        sum = (+= (4.0/(l.0+x*x)));
    }
    sum *= width;
    printf ("Estimation of pi is %14.12f\n", sum) ;
}
```

PROCESS-LEVEL FUNCTION-PARALLEL ARCHITECTURES

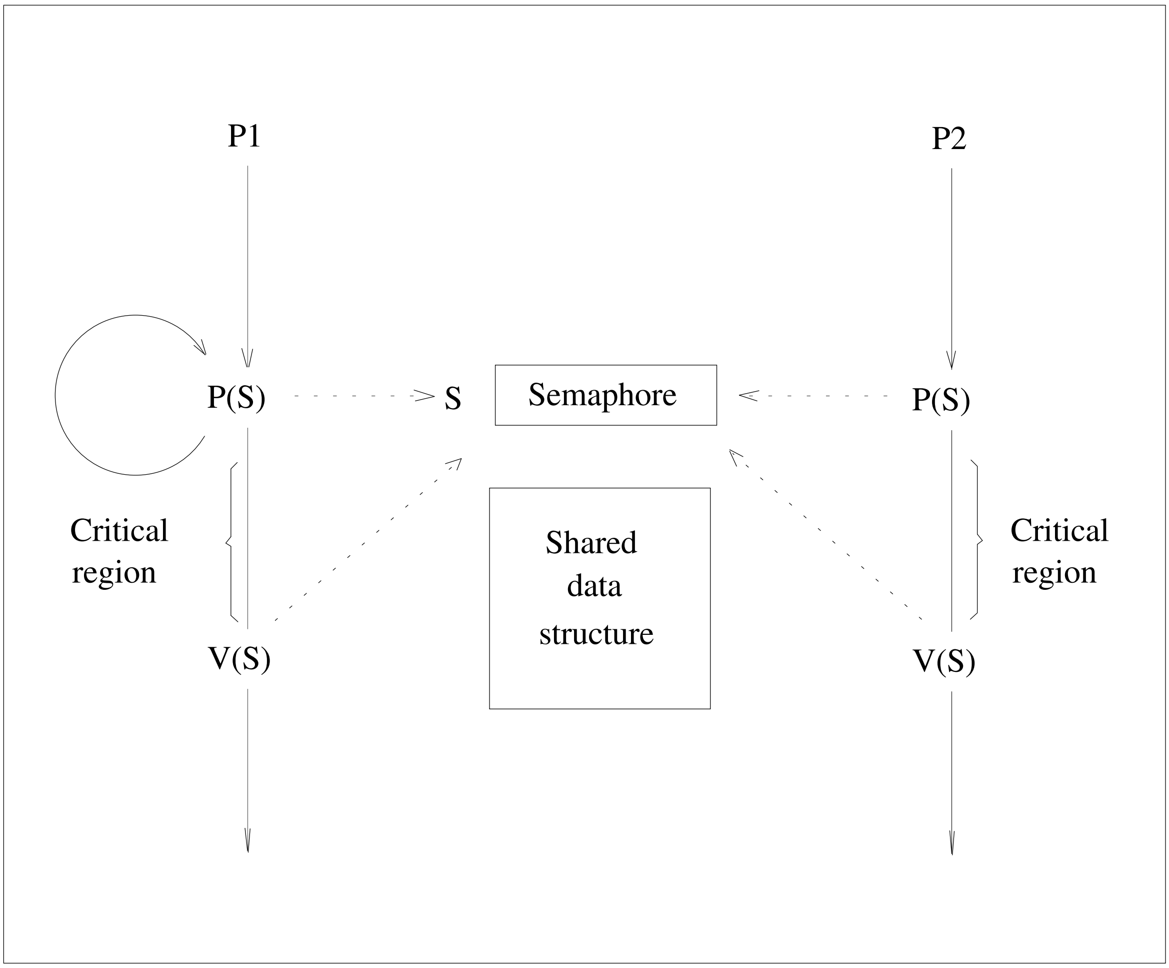
Progress of language constructs used for syncronization:



- **The first column represents shared memory architectures.**
- **The second column represents distributed memory architectures.**

SHARED MEMORY MIMD ARCHITECTURES

Using semaphore to solve the mutual exclusion problem:

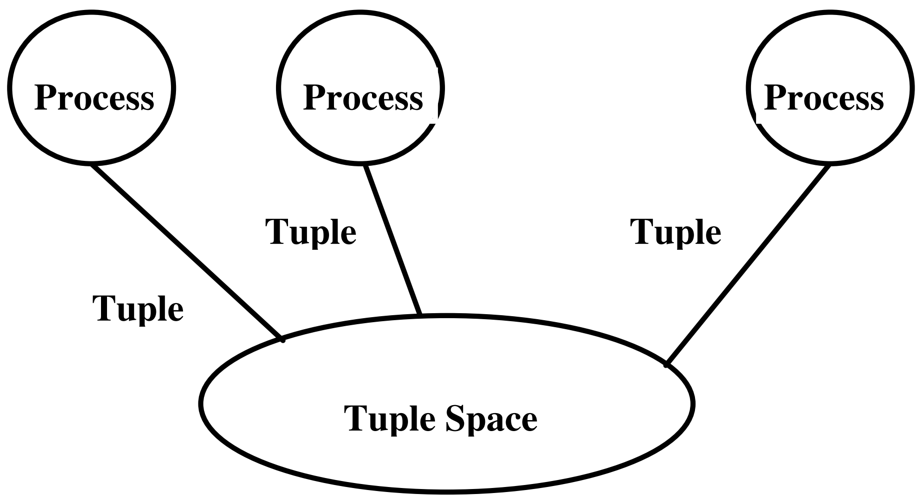


C-LINDA

- Linda consists of several operations that work on tuple space, a shared associative memory.
- Incorporating Linda operations into a sequential base language yields a parallel programming language. For example, in C-Linda, Linda primitives have been added to C.

Programmer’s Model

- Linda is a MIMD model of parallel computation.
- The Linda programmer envisions an asynchronously executing group of processes that interact by means of a tuple space consisting of a collection of logical tuples.
- Parallelism is achieved by creating process tuples, which are evaluated by processors needing work.



- Parallel processes interact by sharing data tuples.
- After a process tuple has finished execution, it returns to tuple space as a data tuple.

C-Linda Language Constructs

Six functions enable a process to interact with tuple space.

<u>Function out</u>, passed tuple t as an argument, causes t to be added to tuple space. The process executing the out operation continues immediately.

<u>Function in</u>, passed template s, removes from tuple space a tuple t that matches template s. The values of the actual parameters in t are assigned to the formal parameters in s. The in operation blocks the process until a suitable matching tuple is found. If multiple tuples match, an arbitrary one is chosen.

<u>Function rd</u>, passed template s, behaves in the same way as function in, except that the matching tuple is not removed from tuple space.

<u>Function eval</u>, passed tuple t, forks a new process to evaluate t. Once t has been evaluated, it is placed in tuple space as an ordinary data tuple.

<u>Function inp</u> is a nonblocking version of in. It returns a 1 if a matching tuple is retrieved, 0 if not.

<u>Function rdp</u> is a nonblocking version of rd. It returns a 1 if a matching tuple is found, 0 if not.

Master/worker-style C-Linda program to compute π

```
/* Compute the area under the curve 4/(1 + x*x) between 0 and 1 */

#define INTERVALS 4000000
#define NUMLWORKERS 7
double work (start, finish, width)
        int start, finish;
        double width;              /* Width of interval */
{
        int i;
        double sum;                /* Sum of areas */
        double x;                  /* Midpoint of rectangle on x axis */
        sum = 0.0;
        for (i = start; i < finish; i ++) {
                x = ( i+0.5 ) *width;
                sum += 4.0/ ( 1.0 + x*x);
        }
        return (sum * width);
}

real_main() {
        int i, left_overs, len, w;
        double result, sum;
        len = INTERVALS/NUM_WORKERS + l;
        left_overs = lNTERVALS%NUM_WORKERS;
        /* This process will do some work too, so start the worker
         count/id variable, w, at 1 rather than at 0. */
        for (i = 0, w = 1; w < NUM_WORKERS; i += len, ++w) {
                if (i == left_overs) - -len;
                eval("worker", work(i, i+len, 1.0/INTERVALS));
        }
        sum = work(i, INTERVALS, 1.0/INTERVALS);
        for (w = 1; w < NUM_WORKERS; ++w) {
                in ("worker", ? result ) ;
                sum += result ;
        }
        printf ("Estimation of pi is %14.12f\n", sum) ;
}
```

Divide-and-conquer-style C-Linda program to compute π

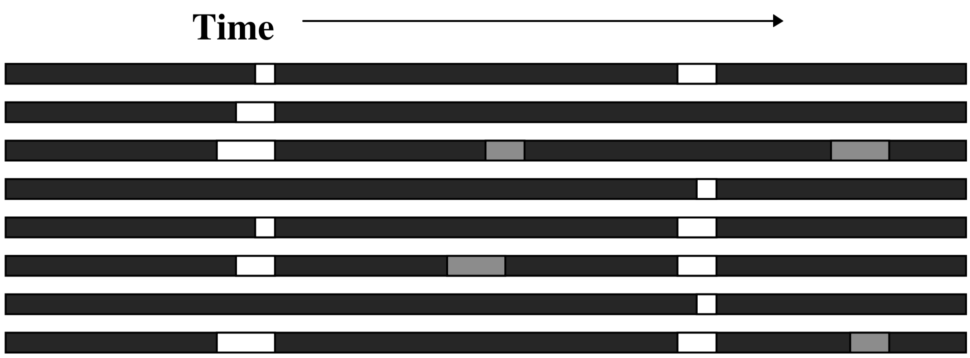
```
/* Compute the area under the curve 4/(1 + x*x) between 0 and 1 */

#define INTERVAL_LIM      50000
#define INTERVALS         400000

double work (id, start, finish, width)
        int id, start, finish;
        double width;                    /* Width of interval */
{
        int i;
        int length = finish - start;
        double sum, sum2;
        double x;

        if (length > INTERVAL_LIM) {
                eval("worker", id, work(id< < 1, start,
                        start+(length > > l), width) );
                eval("worker", id, work((id< < 1)+1,
                        start+(length >> 1), finish, width) );
                in("worker", id, ?sum);
                in("worker", id, ?sum2);
                return (sum+sum2);
        }
        for (i = start; i < finish; i ++) {
                x = ( i+0.5 ) *width;
                sum += 4.0/ ( 1.0 + x*x);
        }
        return (sum * width);
}

real main() {
        printf("Estimation of pi is %14.12f\n",
        work(1, 0, INTERVALS, 1.0/INTERVALS) );
}
```

DISTRIBUTED MEMORY MIMD ARCHITECTURES

NCUBE C

The Run-Time Model

- **SPMD (Single Program, Multiple Data) style programming.**
- **The same program executes on every node processor.**
- **Each node program begins execution as soon as the operating system loads it into the node.**
- **Processors work on their own local data until they reach a point in the computation when they need to interact with other processors to: swap data items, communicate results, perform a combining operation on partial results, etc.**
- **If a processor initiates a communication with a partner that has not finished its own computation, then the initiator must wait for its partner to catch up.**
- **Once the processors have performed the necessary communications, they can resume work on local data.**
- **One way to view the entire computation is to consider the activities of the processors, as they cycle between computing and communicating, with occasional periods of waiting.**



This figure gives an abstract view of the nCUBE C run-time model. Every processor is represented by a horizontal line. Black represents computing on the processor, white represents time spent on communicating with other processors, and grey shows time spent waiting for a message.

Extensions to the C Language

- The C programs running on the nodes are inherently parallel, in the sense that every node is active from the time its program is loaded.

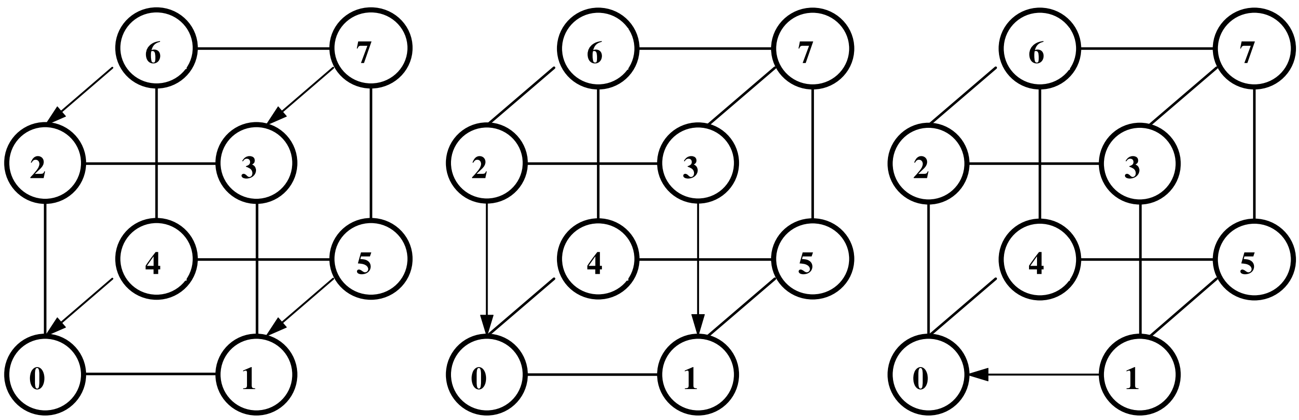
- There is no need for a process creation mechanism in nCUBE C. In fact, three new functions are sufficient for us to run our sample application on the nCUBE:

  - whoami: Returns information about the node process, including the dimension of the allocated hypercube and the position of the node in the allocated hypercube.

  - pwrite: Sends a message to another node. Arguments to this function include the destination node number, the address of the message to be sent, the length of the message, and the message type, an integer. Function nwrite is nonblocking.

  - nread: Receives a message from another node. Arguments to this function include the source node number, the message type, the address where the message is to be stored, and the maximum number of bytes to be read. The value returned from the function is the number of bytes actually read. The user has the option of specifying that the message to be read can be of any type or from any source, or both. Function nread blocks until a message fitting the source and type criteria has been read.

Typical communication pattern to realize reduction operations:



First part of nCUBE C program to compute π using numerical integration

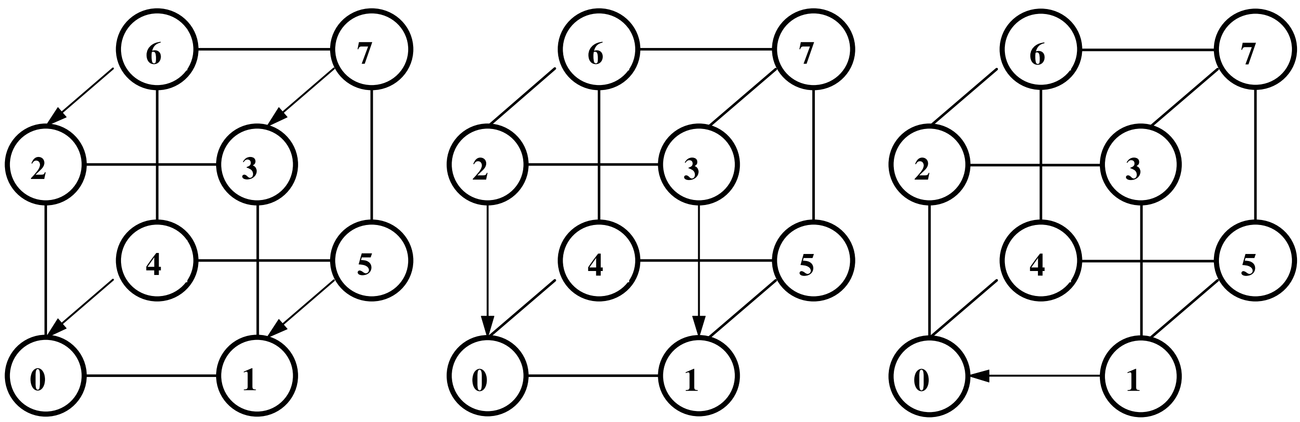
```
/* Node program */
#include < special.h>
#include < stdio.h>

main (argc, argv)
        int argc; char *argv[];
{
        void fan_in();
        int cube_dim;       / * Dimension of allocated hypercube * /
        int i;
        int intervals;      / * Number of intervals * /
        int nodenum;        / * Position of node in allocated hypercube * /
        int not_used;       / * Place holder * /
        int num_procs;      / * Number of active processors * /
        double sum;         / * Sum of areas * /
        double width;       / * Width of interval * /
        double x;           / * Midpoint of rectangle on x axis * /

/* argv[0] is name of file containing host program */
/* argv[l] is name of file containing node program */
/* argv[2] is dimensional of allocated hypercube */
/* argv[3] is number of intervals */

        intervals = atoi ( argv [ 3 ] );
        width = 1.0 / intervals;
        whoami (&nodenum, &not_used, &not_used, &cube_dim);
        num_procs = 1 < < cube_dim;
        sum = 0.0;
        for (i = nodenum; i < intervals; i += num_procs) {
                x = ( i+0.5 ) *width;
                sum += width* ( 4.0/ ( 1.0 + x*x) );
        }
        fan_in (&sum, nodenum, cube_dim);

        i f ( ! nodenum )
                printf ('Estimation of pi is %14.12f\n", sum);
}
```

Second part of nCUBE C program to compute π using numerical integration

nCUBE C source code for function fan_in, which accumulates each node's subtotal into a sum stored on node 0:

```
void fan_in (value, nodenum, cube_dim)
        double *value; int nodenum; cube_dim;
{
        int dest, i, source, type;
        double tmp;
        type = FANIN;
        for (i = cube_dim-1; i >= 0; i- -)
                if (nodenum < (1 << i) ) {
                        source = nodenum ^ (1 << i);
                        nread (&tmp, sizeof (double), &source, &type);
                        *value += tmp;
                }else if (nodenum < (1 << i (i+1) ) ) {
                        dest = nodenum ^ (1 << i );
                        nwrite (value, sizeof (double), dest, type);
                }
}
```

Function fan_in computes the sum of values stored on the hypercube nodes using the communication pattern shown by the arrows. Together, the arrows form a binomial tree:



OCCAM-2

- **The design of Occam-2 is based on the communicating sequential processes (Hoare 1978).**

- **Occam-2 has been closely associated with the evolution of the Transputer**

Programmer´s Model

- **An Occam-2 program is a collection of asynchronously executing processes**

- **Processes communicate via a synchronous message-passing protocol.**

- **An Occam-2 program is built from three kinds of primitive process:**

    - **assignment**
    - **input**
    - **output.**

- **The programmer can assemble more complicated processes by specifying when they must execute sequentially and when they can execute in parallel.**

- **The assignment process, one of the three primitive processes in an Occam-2 program, assigns the value of an expression to a variable.**

- **An input process assigns to a variable the value read from a channel.**

- **An output process evaluates an expression and writes the result to a channel.**

- **A channel is a one-way, point-to-point, synchronous link from one process to another process.**

- **An Occam-2 channel is a logical entity independent of the underlying hardware.**

Structure of a hypothetical Occam-2 program

- **At the global level the program is one Occam-2 process, indicated by the ellipse E around the graph.**

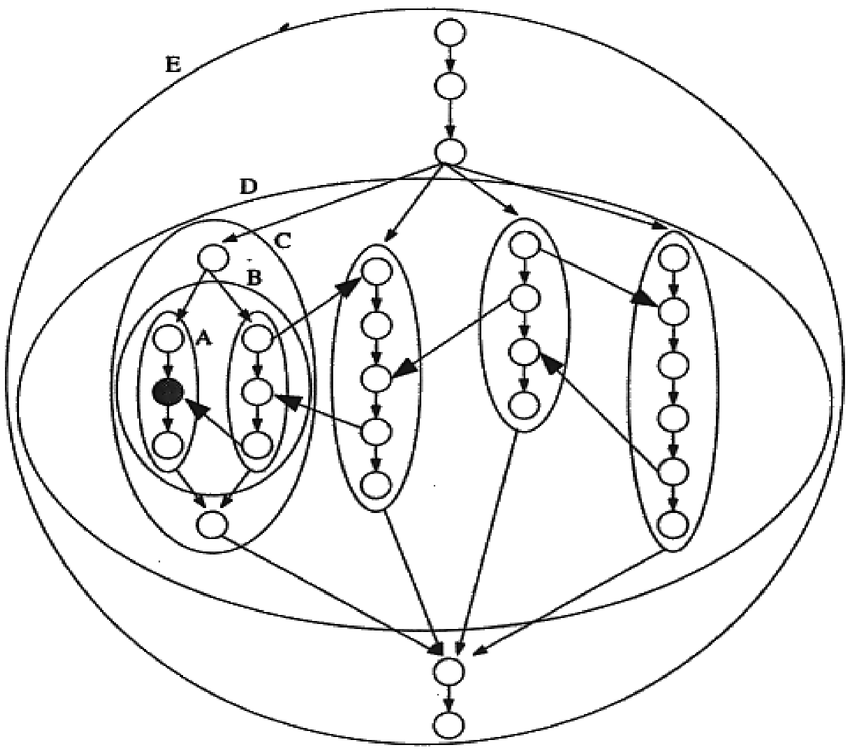
- **This process has six subprocesses that must execute sequentially.**

- **All but the fourth process are primitive processes, indicated by small circles.**

- **The fourth process is a built-up process, as indicated by ellipse D.**

- **Built-up processes can represent either a sequential or a parallel collection of simpler processes.**
  - **inside a sequential collection of processes (A)**
  - **inside a parallel collection of processes (B)**
  - **inside a sequential collection of processes (C)**
  - **inside a parallel collection of processes (D)**
  - **inside a sequential collection of processes (E)**

- **The primitive process represented by a filled grey circle is an input process.**



Language Constructs

- **The assignment process:**

```
sum := partial1 + partial2
```

- **The input process reads a value from a channel and assigns that value to a variable:**

```
channel1 ? partial1
```

- **The output process writes a value to a channel:**

```
channel3 ! sum
```

- **The SEQ (pronounced "seek") construction:**

```
SEQ
    channel1 ? partial1
    channel2 ? partial2
    sum := partial1 + partial2
    channel3 ! sum
```

- **The PAR construction specifies that a number of processes may execute in:**

```
SEQ
    PAR
        channel1 ? partial1
        channe12 ? partial2
    sum := partial1 + partial2
```

- **A PAR construction terminates when its last constituent process terminates.**

Language Constructs (Cont.)

- **The IF construct:**

```
IF
  aa > 10
    aa := 10
  aa < 0
    aa := 0
  TRUE
    SKIP
```

- **The ALT construct introduces nondeterministic behaviour:**

```
ALT
  temperature.channel ? value
    IF
      value > high.value
        high.value := value
      TRUE
        SKIP
  enable report.channel ? check
    SEQ
      result.channel ! high.value
        high.value := -100
```

- **The replicator construct allows the constructs SEQ, PAR, IF, and ALT to execute some number of times:**

  PAR i = [0 FOR 10]

  creates 10 replicated parallel processes, each with a uniques value of i in the range from 0 through 9.

An Occam-2 program
to compute π using numerical integration

```
DEF N = 400000 :
DEF PROCESSES = 8 :
DEF CHUNK = N / PROCESSES :
CHAN sum [PROCESSES] :
PAR
  PAR i = [0 FOR PROCESSES]
  REAL64 x, localsum, width :
  SEQ
    localsum := 0.0
    width := 1.0 / N
    x := ((i * CHUNK) + 0.5) * width
    SEQ i = [0 FOR CHUNK]
      SEQ
        localsum := localsum + (4.0 / (1.0 + (x * x)))
        x := x + width
      localsum := localsum * width
      sum[i] ! localsum
  REAL64 pi :
  INT got[PROCESSES] :
  SEQ
    pi := 0.0
    SEQ i = [0 FOR PROCESSES]
      got[i] := FALSE
    SEQ i = [0 FOR PROCESSES]
      REAL64 y :
      SEQ
        ALT i = [0 FOR PROCESSES]
          (got[i] = FALSE) & sum[i] ? y
          got[i] := TRUE
        pi :=pi +y
  output ! "Approximation to pi is "; pi
```

SUMMARY OF FORMS OF PARALLELISM

Specification of parallelism	Execution model	Language	Parallel architecture
Loops	Vectorization	Conventional procedural	Vector computers
Loops	SPMD	Conventional procedural	MIMD architectures with barrier synchr.
1 Explicit declaration of parallel data structures 2. Explicit allocation of parallel data structures to processors	SIMD	Data parallel	SIMD (array processors)
No explicit specification	Instruction-level functional parallel	Any	ILP architectures
No explicit specification	Data driven	Applicative	Multithreaded architectures
1. Explicit partitioning of program into parallel processes 2. Explicit synchronization among processes	Processes communicating through shared data	Concurrent	Shared memory MIMD machines
1. Explicit partitioning of program into parallel processes 2. Explicit messages among processes 3. Explicit allocation of processes to processors	Processes communicating by message passing	Parallel	Distributed memory MIMD machines

PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS

❒ Parallel Performance Measures

Speedup

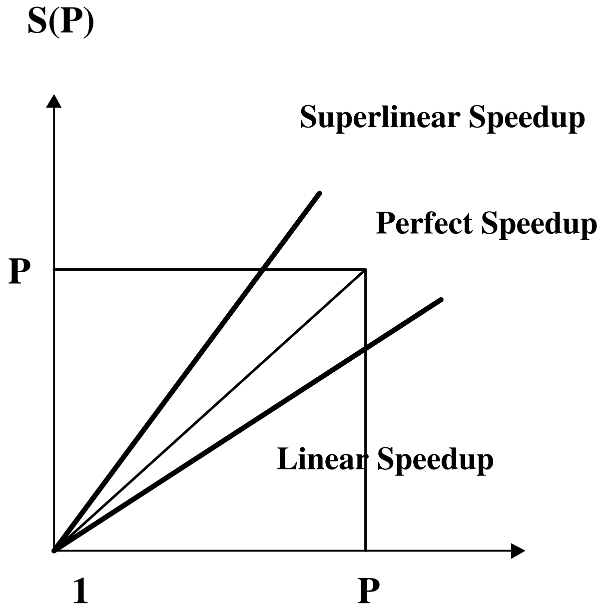
$$S(P) = \frac{T(1)}{T(P)}$$

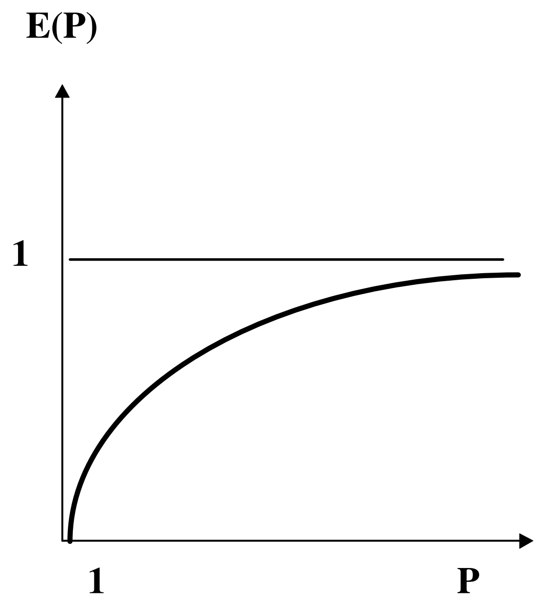
P is the number of processors

T(i) is the execution time using i processors

Efficiency

$$E(P) = \frac{S(P)}{P}$$





PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS

❒ The Amdahl Law

$$T(1) = Ts + Tp$$

Ts **is the time spent performing computations which cannot be parallelized.**
Tp **is the time spent performing computations which can be perfectly parallelized among any number of processors.**

$$T(P) = Ts + \frac{Tp}{P}$$

$$S(P) = \frac{T(1)}{T(P)} = \frac{Ts + Tp}{Ts\,\frac{Tp}{p}}$$

$$\lim_{P \to \infty} S(P) = 1 + \frac{Tp}{Ts}$$

The limit for the Speedup (no matter the number of procvessors) is:

$$1 + \frac{Tp}{Ts}$$

For instance, if Ts = Tp then the maximum reachable Speedup is 2.
Parallelism is not very promising, particularly if we are thinking about machines with a large number of processors.

PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS

	definition	optimal case	practical case
speed -up	$S_N = \frac{T_{seq}}{T_N}$	$S_N = N$	$S_N < N$
efficiency (speed-up/node)	$\varepsilon = \frac{S_N}{N} = \frac{T_{seq}}{N.T_N}$	$\varepsilon = 1$	$\varepsilon < 1$
(overhead	$f = \frac{1}{\varepsilon} - 1$	**f = 1**	**f > 0)**
communication overhead	$f_c = \frac{T_{com}}{T_{Nq}}$	$f_c = 0$	$f_c > 0$

Reasons of inefficiency:

1. **Nonoptimal algorithm or algorithmic overhead**
2. **Software overhead**
3. **Load balancing**
4. **Communication overhead**

Amdahl's law:

If an inherently sequential component of the problem takes fraction α of the time on a single node, then one can never realize a speed-up factor greater than α^{-1}.

$$S \leq \frac{1}{\alpha}$$

COMMUNICATION OVERHEAD

Notations

t_{calc}: typical time to perform generic calculation
t_{comm}: typical time to comm. a single word
t_{calc}: total calc. time for a node
t_{comm}: total comm. time for a node

Generic nearest-neighbour algorithm for image processing

$T_{calc} \cong 9n * t_{calc}$ each node holds n pixels.

$T_{comm} \cong 4(\sqrt{n} + 1) * t_{comm}$

$\Downarrow$

$$(1) \quad fc = \frac{T_{comm}}{T_{calc}} = \frac{4(\sqrt{n}+1)t_{comm}}{9n * t_{calc}} \approx \frac{const}{\sqrt{n}} * \underbrace{\frac{t_{comm}}{t_{calc}}}_{\text{depends on hardware}}$$

Conclusions:

fc depends on: - hardware (tipically 1.5 - 3)
- grain size n

fc independent of: - number of processors

Generalization:

$$(2) \quad fc \cong \frac{c}{\sqrt[d]{n}} * \frac{t_{comm}}{t_{calc}}$$

for d dimensional nearest-neighbour algorithms (for example partial differential equations solved in d dimensions)

COMM. OVERHEAD (CONT.)

Long-range comm. (in image processing)

the updated value of the pixel at any point is a function of all other pixels in the image

$\Downarrow$

$T_{calc} = n*nN*t_{calc}$ **N is the number of nodes**

$T_{comm} = (nN-n) * t_{comm}$

(3) $fc = \frac{T_{comm}}{T_{calc}} \approx \frac{1}{n} * \frac{t_{comm}}{t_{calc}}$

corresponds to d = 1

$\Downarrow$

n calculations for each communicated pixels

Comparison of (1) and (3): $f_c^{(3)} < f_c^{(u)}$

fc depends on the ratio of comm. to calc. and not simply on the amount of comm.

in (1) min. amount of comm. but also min. calc.

in (3) max. " " " max. "

Irregular problems and comm.

Empirical studies demonstrate that (2) can be applied for these problems as well. However, d is different for different cases.

for example in simulation of integrated circuit

$$d_{circuit \approx 3}$$

$$\Downarrow$$

$$f_{c,circuit} \approx \frac{c}{\sqrt[3]{n}} * \frac{t_{comm}}{t_{calc}}$$

COMMUNICATION OVERHEAD AND GRANULARITY

Assume we have only comm. overhead

$\Downarrow$

$$T_N = \frac{T_{seq} + T_{com}}{N} = \frac{T_{seq}}{N} + \frac{T_{com}}{T_{seq}} * \frac{T_{seq}}{N} = \frac{T_{seq}}{N}(1 + fc)$$

$\Downarrow$

$$S_N = \frac{T_{seq}}{T_N} = \frac{N}{1 + fc}$$

$$\varepsilon = \frac{S_N}{N} = \frac{1}{1 + fc} \approx 1 - fc \qquad \text{if } fc \ll 1$$

It can be shown that

$$fc \approx \frac{c}{\sqrt[d]{k}} \frac{t_{com}}{t_{calc}}$$

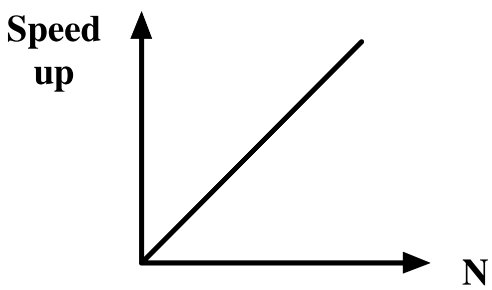
where

k	is the	amount of data/m
t_{com}		comm/node
t_{calc}		calc/node

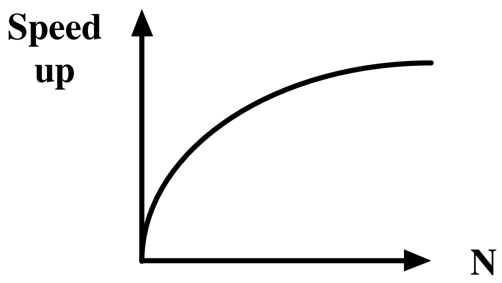
Efficiency (ε) does not depend on N in distributed mem. system.

$\Downarrow$

Parallel computers with <u>large number of nodes</u> can be efficienly used.



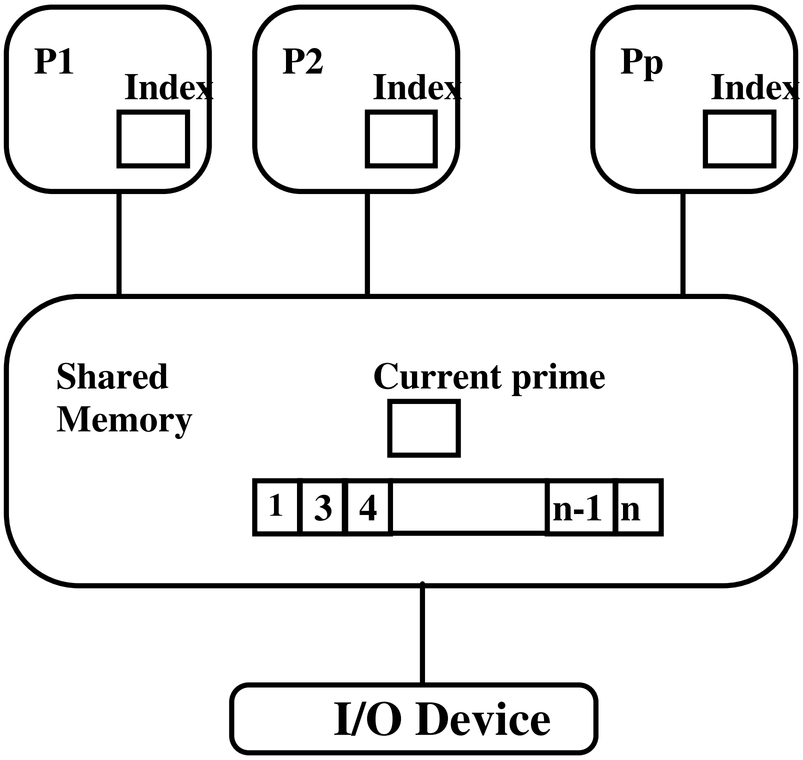
fixed grain size
fc << 1



fixed problem size:
as N increases k decreases

DATA PARALLEL APPROACH WITH I/O

Shared-memory parallel model incorporating a seuential I/O device:



Total execution time as a function of its two components. This graph is for the case when n = 1,000,000 and $\beta = \chi$, where β denotes the time of transmitting one prime number to the I/O device.