

Lesson 1: Parallel Computers

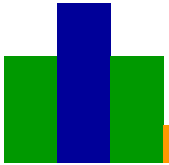
G Stefanescu — National University of Singapore

Parallel & Concurrent Programming
Spring, 2004



The demand for computational speed

- Computationally difficult problems: those that can not be solved in a reasonable amount of time by today's computers.
- Examples: modeling large DNA structures, global weather forecasting, modeling motion of astronomical bodies, etc.
- Detailed example (weather forecasting):
 - Suppose global atmosphere is divided into 5×10^8 cells (each of size 1 mile \times 1 mile \times 1 mile; the height is up to 10 miles) and each calculation requires 200 floating point operations; hence one step requires 10^{11} floating point operations.
 - To forecast the weather over 10 days using 10-minute intervals, a computer operating at 100 Mflops takes:
 $6(\text{iterations for an hour}) \times 24(\text{hours per day}) \times 10(\text{days}) \times 10^{11} : 10^8(\text{Mflops}) \approx 1.5 \times 10^6(\text{sec}) \approx 17\text{days}$ (unacceptable)



Parallel processing

To be successful, *parallel processing* needs to fulfill 3 conditions:

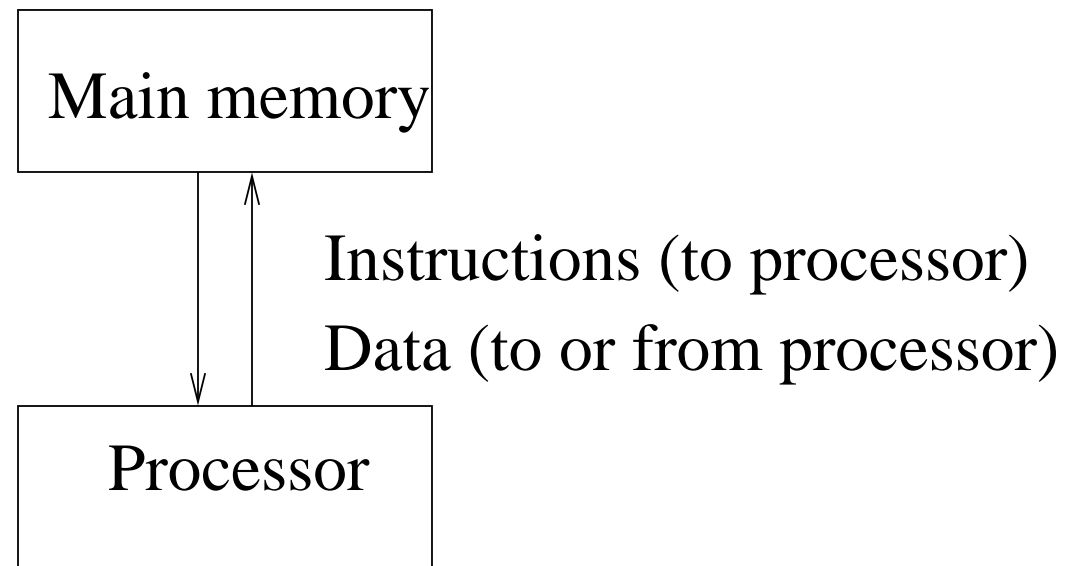
- *algorithm design*: the problem should be good enough to be decomposed in parts which may be solved in parallel
- *programming language*: the programming languages needs specific statements for implementing the parallel algorithm
- *parallel computer*: the computer [or network of computers] needs hardware capabilities for a real parallel running of the program

Our course focuses on the second issue; other SoC courses CS4231 and CS5221 cover in more detail the other aspects of parallel processing.

Parallel computers and programming

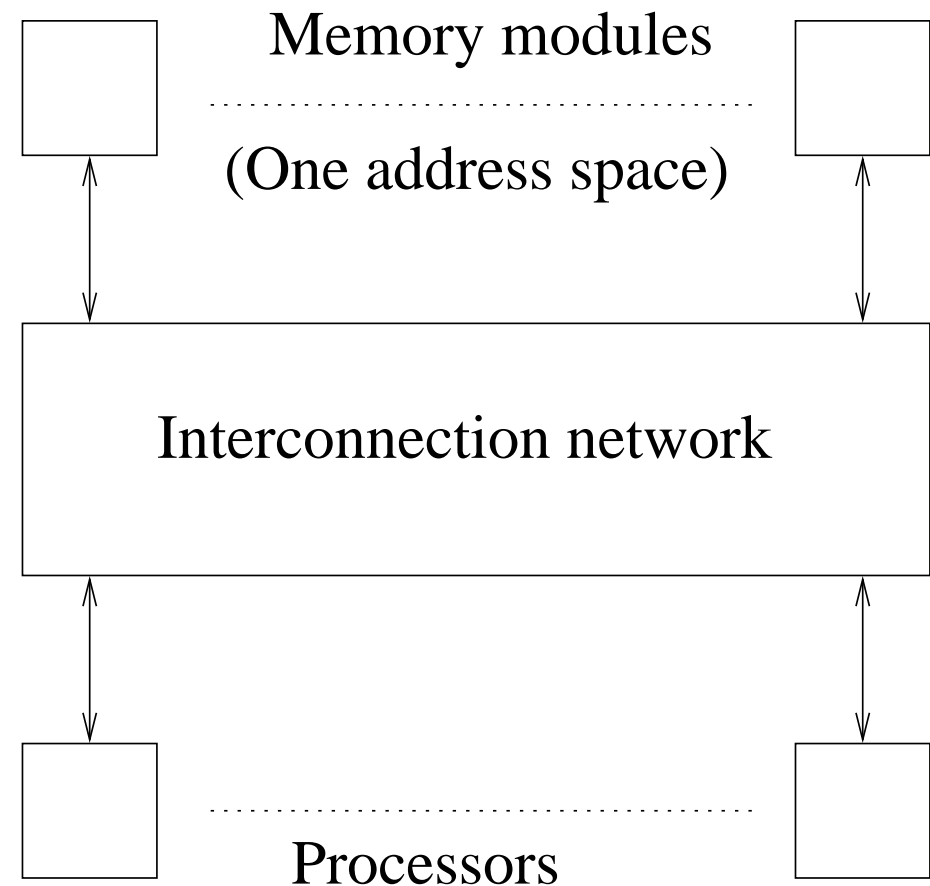
The key point is to use *multiple processors* operating together on a *single problem*. *[It is an old topics, see e.g., Gill, S 1958.]*

A conventional computer has a *processor* executing a program stored in a main *memory*.



(1) Shared memory multiprocessor system

In this model we have *multiple processors* and multiple *memory modules* such that *each processor can access any memory module* (hence we have a *shared-memory configuration*)





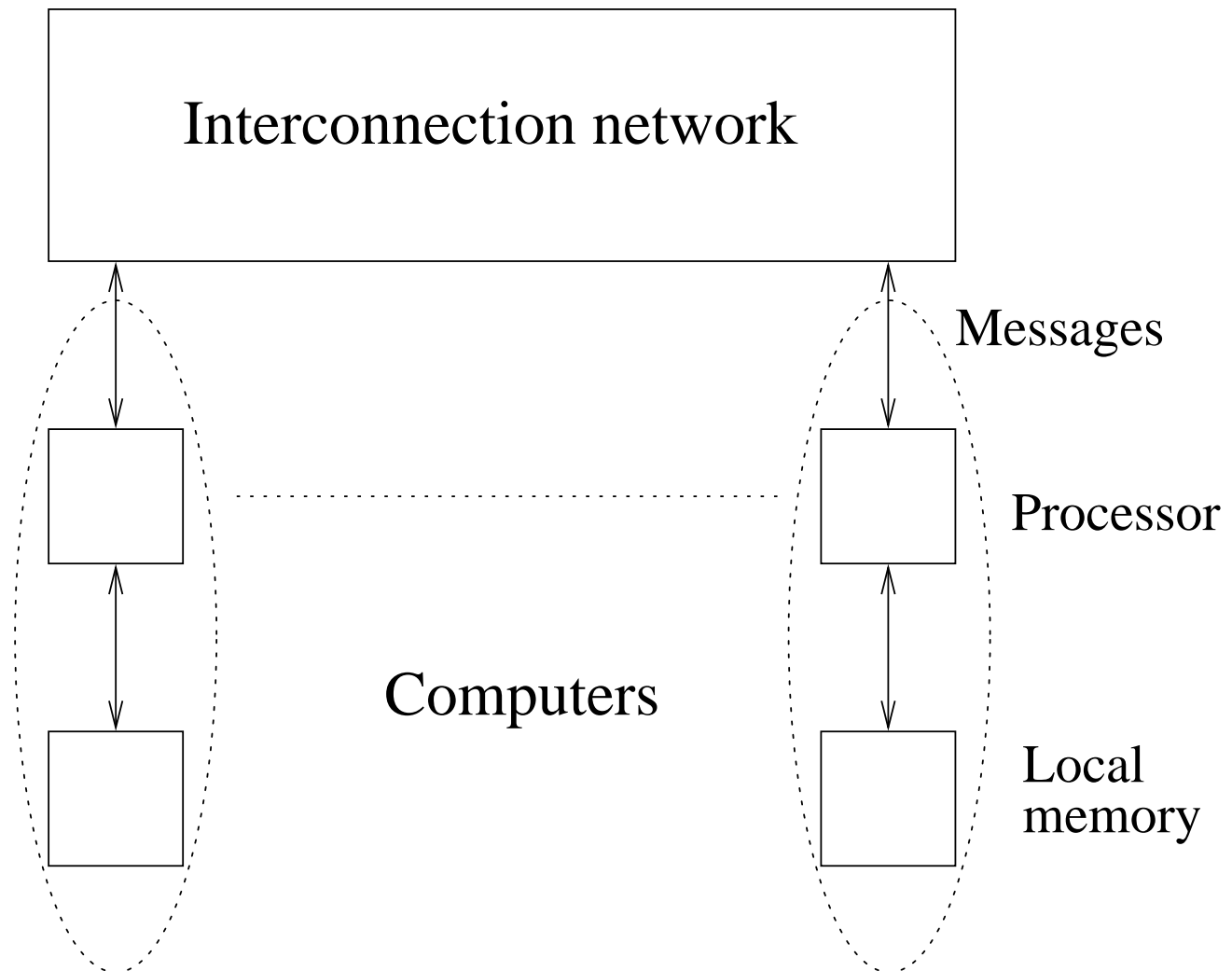
...and programming

Programming in this case can be done in different ways:

- *Special (low level) parallel programming constructs*: The programs use constructs and statements that allow shared variables and parallel code sections to be declared. The *compiler* is responsible for producing the final executable code.
- *Threads*: One can use these high level constructs to describe the code for each processor and to access shared locations.

(2) Message-passing multicomputer (!?)

In this 2nd model we have complete computers connected through an *interconnection network*.





...and programming

- As before, one has to *divide the problem into parts* that are intended to be executed *simultaneously* to solve the problem. (Each part is solved by one of a set of *concurrent processes*.)
- Different from the previous case, the processes will communicate by *sending messages* only.

Notice: *In this case one has a sort of shared memory, too. But now this memory is distributed: Each processor has access to the full memory space using explicit message sending, only. (Sometimes, this access may be done in an automatic way hiding the fact the memory is actually distributed.)*



MIMD and SIMD classification

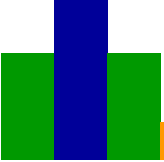
A useful classification of computers takes into account the *number of programs (instruction streams)* and the *number of input data* used for solving a problem.

- *SISD*: Single Instruction-stream Single Data-stream computers are just conventional computers (one program, one input datum);
- *MIMD*: Multiple Instruction-stream - Multiple Data-stream systems have multiple processors, separate programs and data for each processor. (*Both, shared memory and message passing multiprocessors models just described are MIMD systems.*)



..MIMD and SIMD classification

- *SIMD*: Single Instruction-stream - Multiple Data-stream systems have multiple processors, multiple data again, but only one program which is broadcasted to each processor. (*Useful, as there are many important applications.*)
- *MISD*: Not so much used...



..MIMD sub-classification

Within the MIMD class (the main one we are concerned with), one may further identify 2 subclasses:

- *MPMD (Multiple Program Multiple Data)*: Each processor have its own program to execute (e.g., PVM programs).
- *SPMD (Single Program Multiple Data)*: In this case there is a single source program and each processor executes its personal copy of the program, independently. The source program can be constructed such that *parts of the program are executed by certain processors and not by others depending on the identity of the processor*. (This is mainly the case of MPI programs. Notice that SIMD and SPMD are different concepts, leading to different classes.)



Network criteria

The following data may be used to assess the quality of a network:

- *Cost* - given by the number of links in the network (and the difficulty of the network construction, but this is not so easy to be captured as a number)
- *Bandwidth* - number of bits that can be transmitted in unit time (given as bits/sec)
- *Network latency* - the actual time to transfer a message through the network
- *Communication latency* - total time to send a message, including software and interface delays

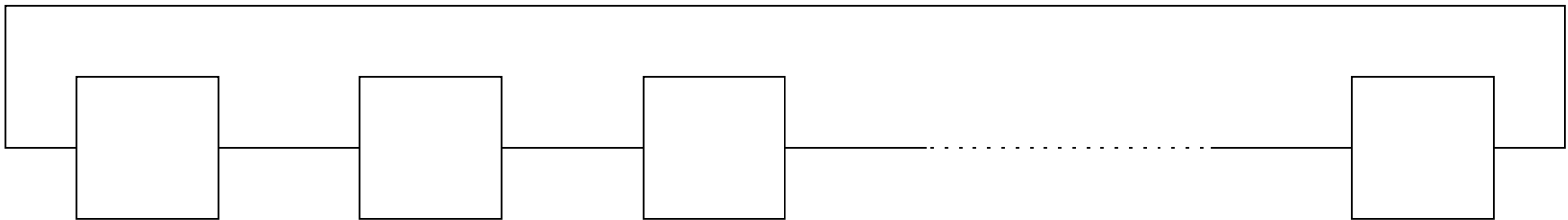


..Network criteria

- *Message latency or startup time* - time to send a zero-length message (basically, the software and hardware overhead in sending messages)
- *Diameter* - the minimum number of links between two farthest nodes in the network (useful to determine the worst case delays)
- *Bisection width* - the number of links that must be cut to divide the network into two equal parts (useful to find lower bounds for sending messages in a parallel algorithm)

Examples of networks

Ring: A network of computers connected in a ring.



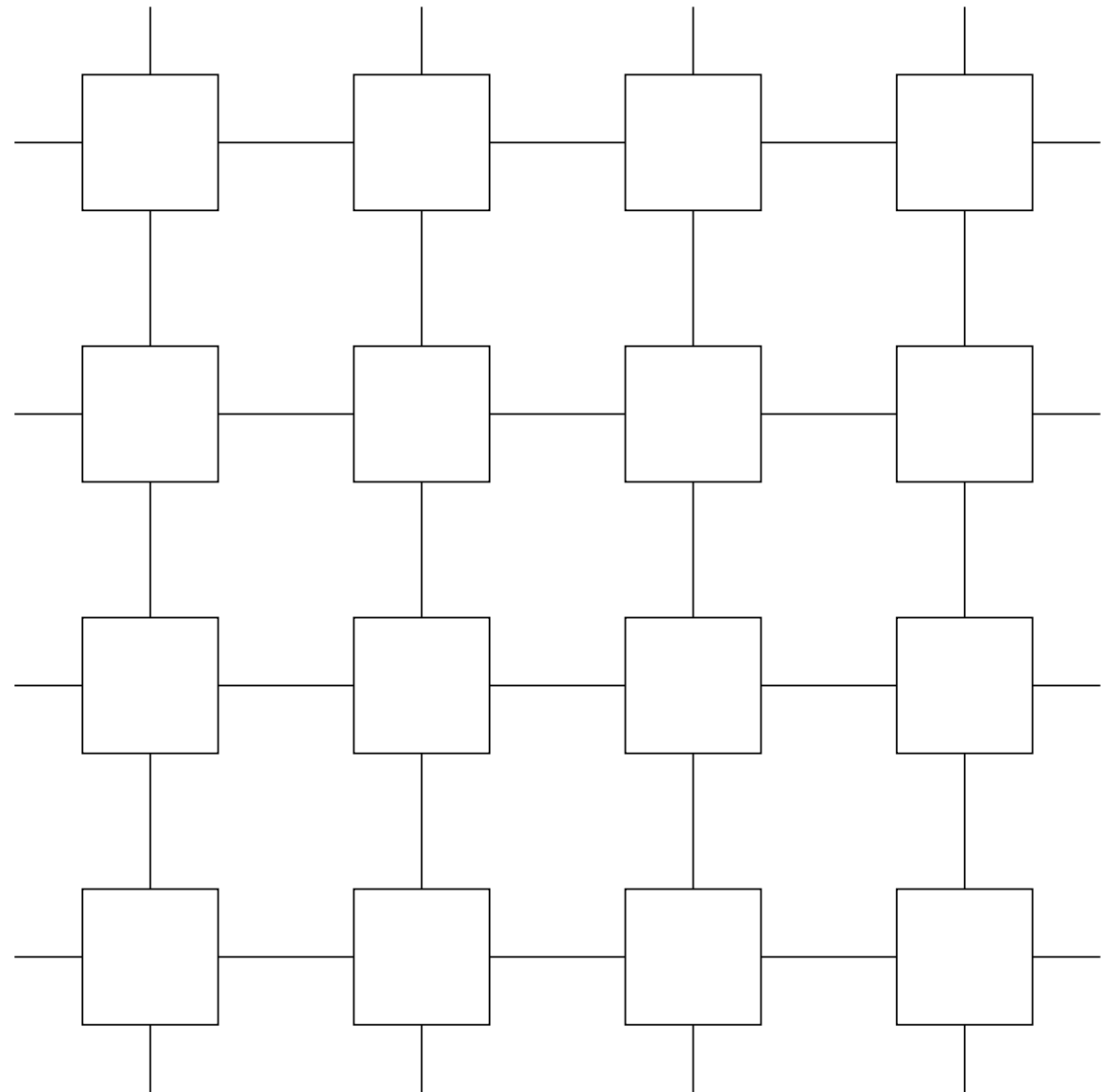
Example of measure: The diameter D is

- $n - 1$ (n is the number of nodes) if the links are in one direction only;
- $\lfloor n/2 \rfloor$, if the links are bidirectional.

..Examples of networks

Mash: A two-dimensional array of computers.

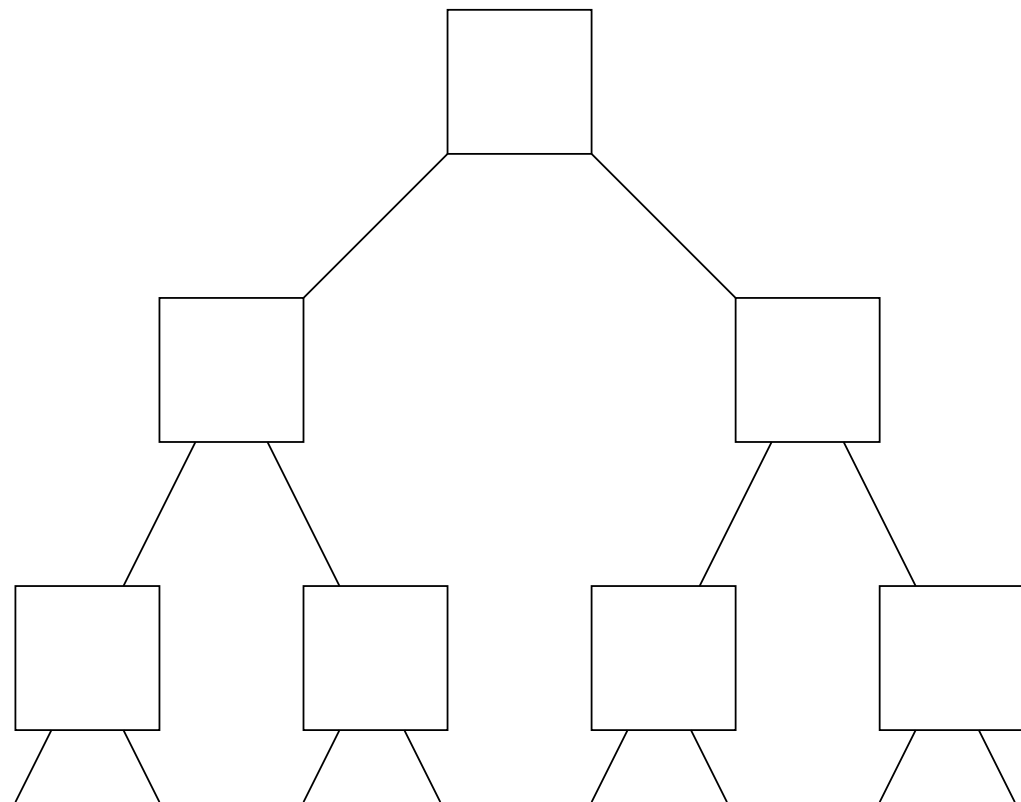
Example of measure:
The diameter D is
 $2(\sqrt{n} - 1)$ (the links are
bidirectional)



..Examples of networks

Tree: A network of computers connected as a binary tree.

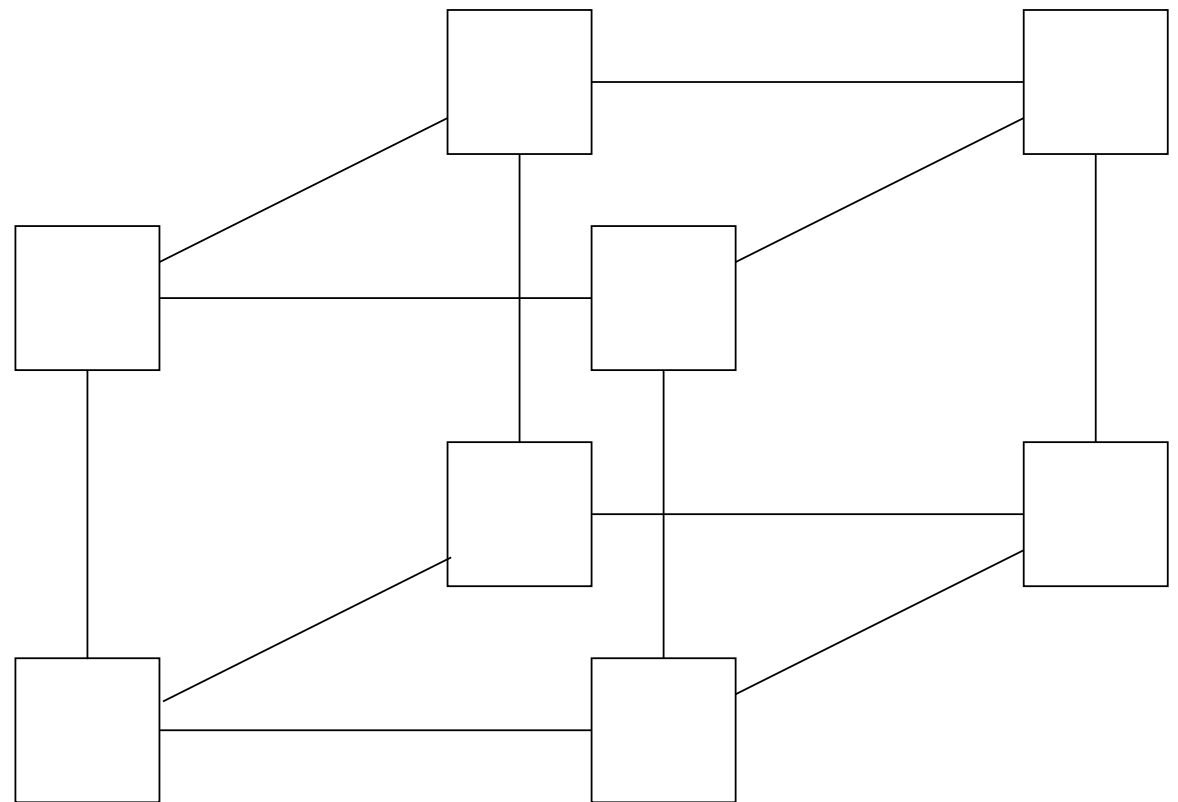
Example of measure:
The diameter D is
 $2 \log_2(n+1) - 2$
(the links are bidirectional)

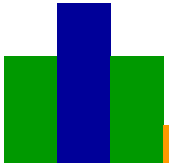


..Examples of networks

Hypercube: A network of computers connected as a three-dimensional hypercube.

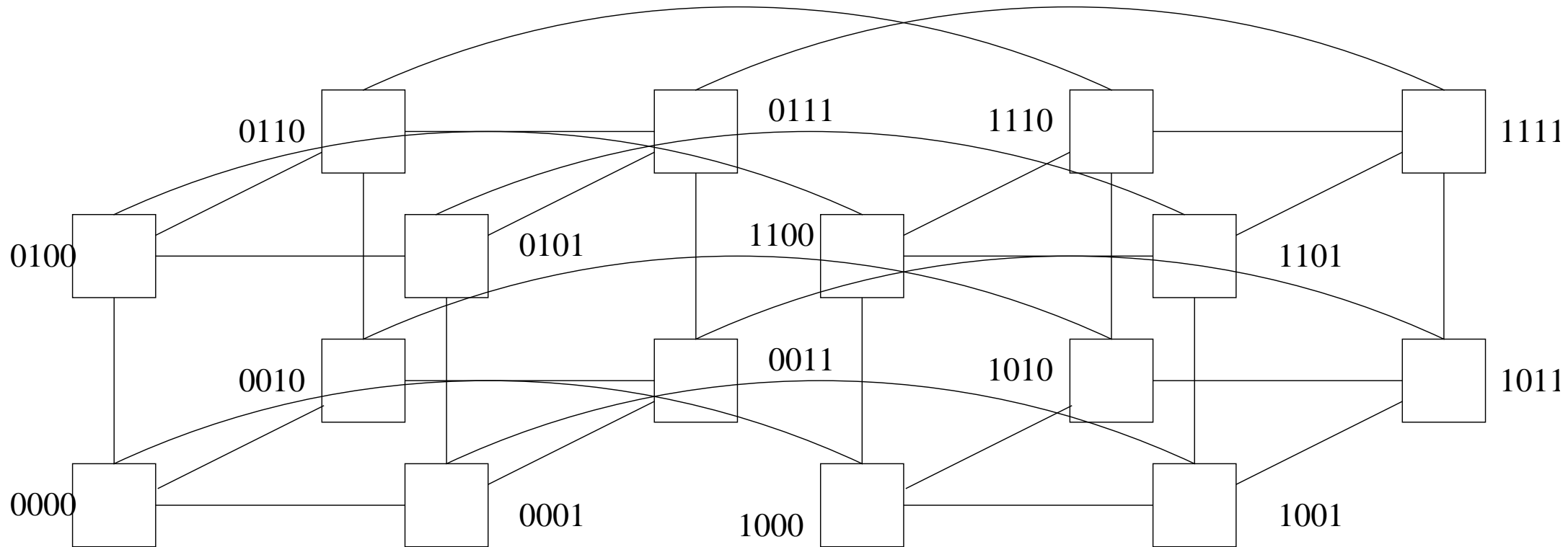
Example of measure:
The diameter D is $\log_2 n$
(the links are bidirectional)





..Examples of networks

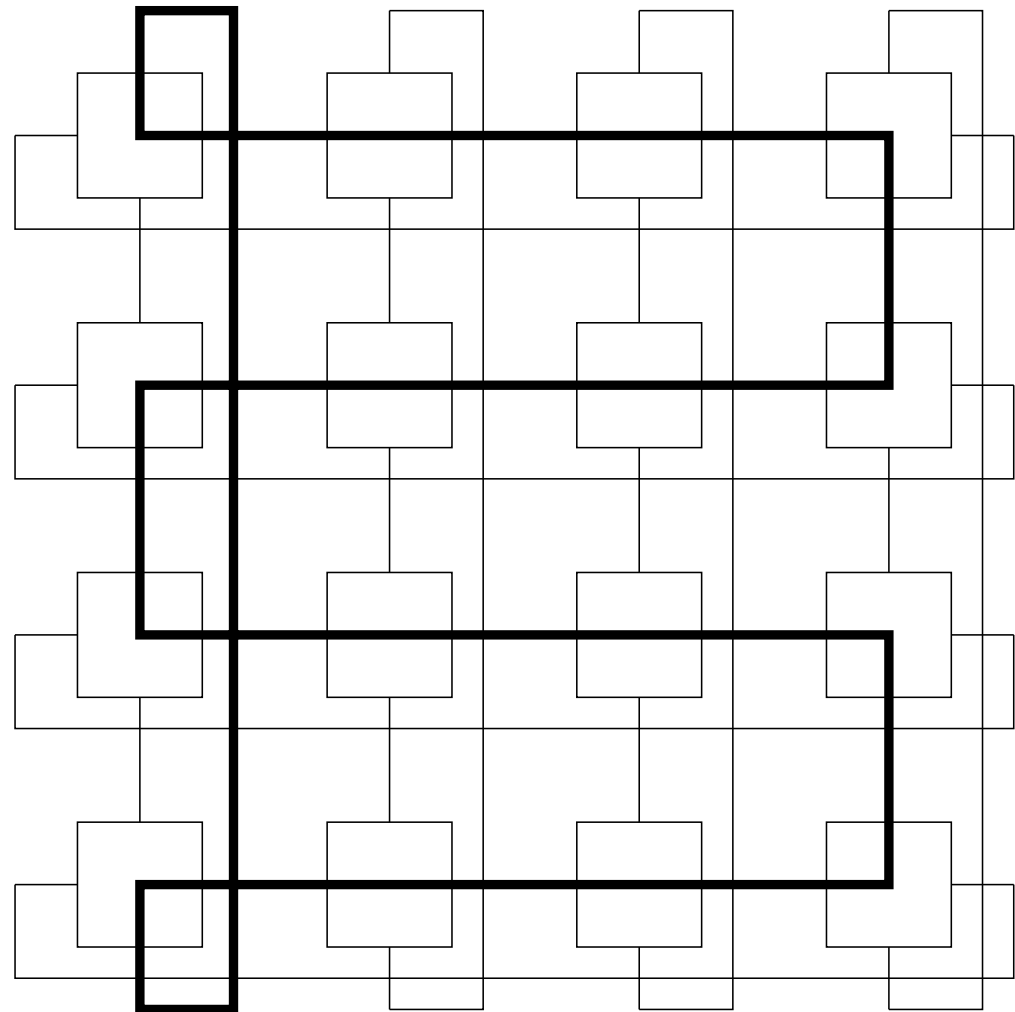
A four-dimensional hypercube

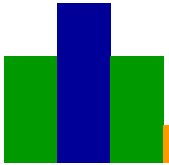


Embedding

An *embedding* is an injective mapping of the nodes of one network to the nodes of another network. It is a *perfect embedding* if the mapping is also surjective.

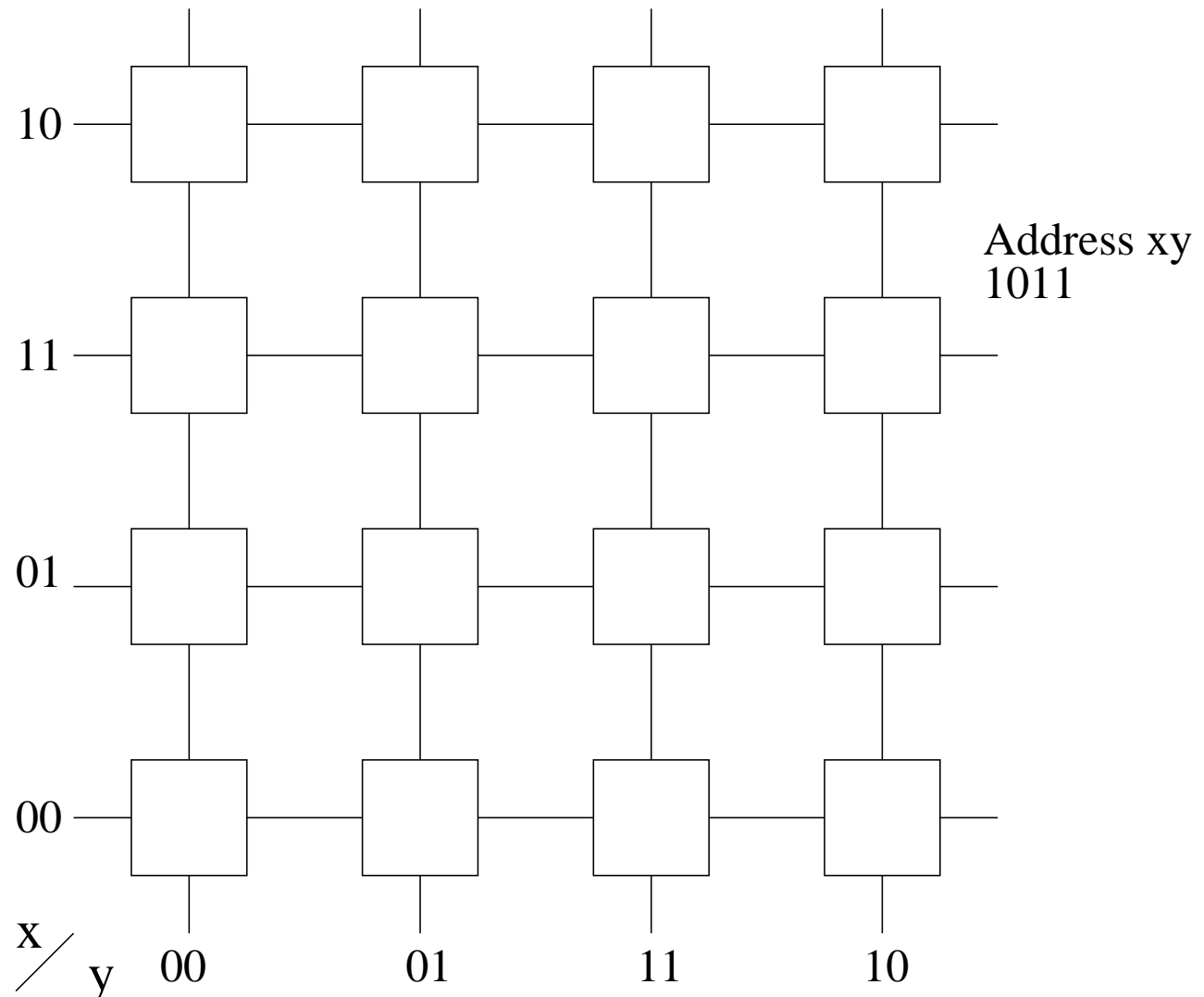
Example: Embedding a ring in a *torus*



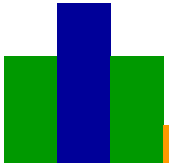


..Embedding

Example: Embedding a 2-dim mash (or torus) in a 4-dim hypercube

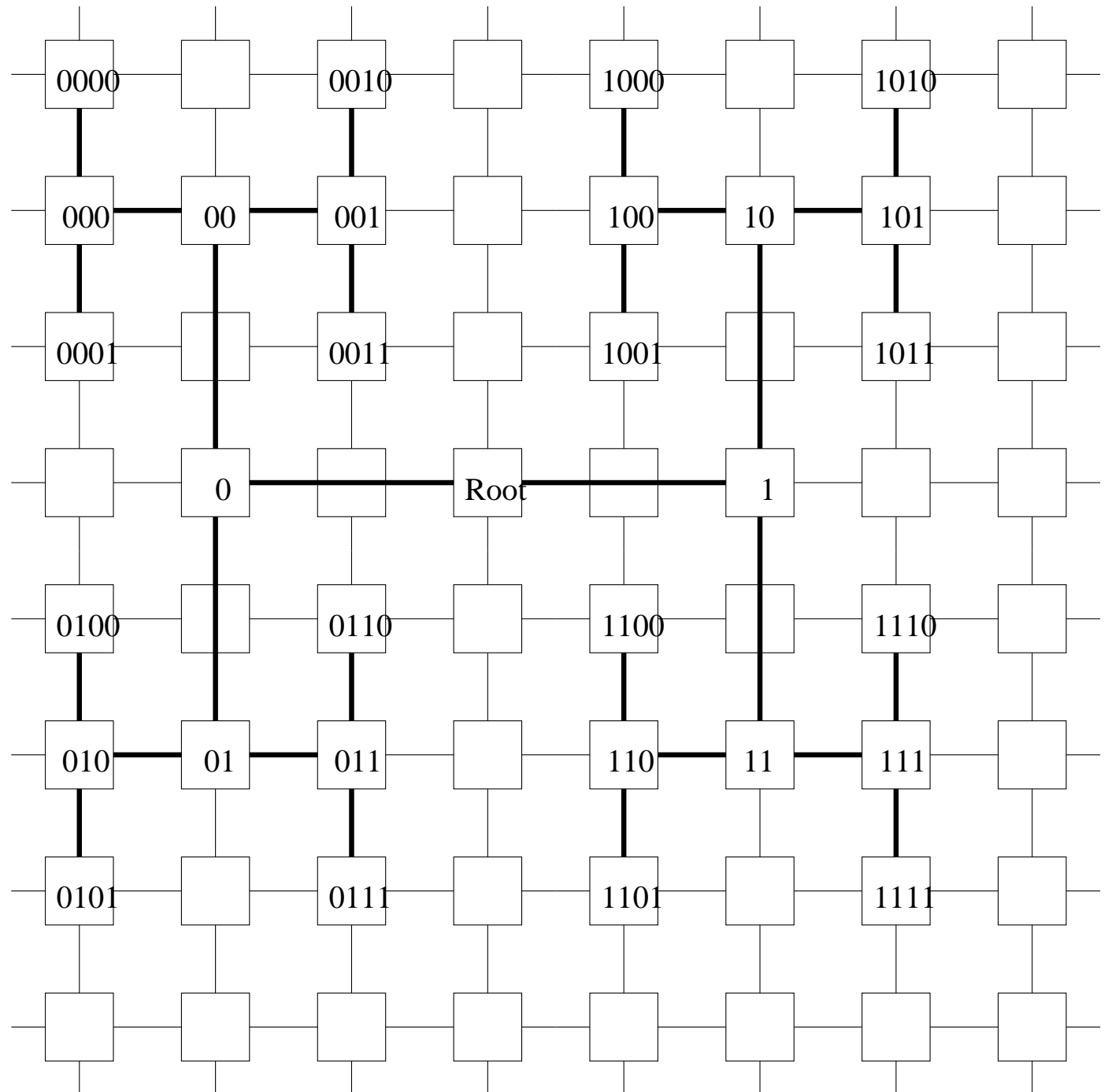


Notice: The embedding in the hypercube is based on one-bit change for each link.



..Embedding

Example: Embedding a binary tree into a 2-dim mash





Dilation

- The *dilation of an embedding A to B* is the maximum number of links in the embedding network B corresponding to a link of the embedded network A .
- It is used to measure the quality of the embedding.
- The dilation of the embedding of a tree into a mesh may be arbitrarily large [given by the height of the tree].

Circuit switching:

- In this case one has to establish a connection path between the source and the destination and to maintain all the links in the path till the message is completely sent.
- Example: Conventional (simple) telephone systems.
- Main problem: It is not too efficient as it forces all the links in the path to be reserved till the complete transfer.



..Communication methods

Packet switching: A *store-and-forward* method may be used:

- Messages are divided into *packets* of information, each including source and destination addresses for routing.
- One puts a limit on the *maximum size* of a packet, say 1000 bytes; if a message is longer, then it is split into more packets.
- Each node has a *buffer* to hold the packets before transferring them onward to the next node.

The usual mail system is based on this method. It may be more efficient than circuit switching as links may be used by a greater number of messages. However, it has a significant latency since packets must first be stored in buffers even in the case when the outgoing link is available.



..Communication methods

Virtual cut-through: This is an improvement of the previous method aiming to eliminate storage latency:

- If the outgoing link is available, then the message is immediately passed forward without being stored in the nodal buffer.

If a complete path is available, then the message is passed immediately through to the destination. If no such path is available, then storage is needed for a successful transmission.



..Communication methods

Wormhole routing: The messages are divided into smaller units called *flits (flow control digits)*. Only the head of the message is initially transmitted from source node to next node when connecting link is available. Subsequent flits of message are transmitted when links become available.

A *request/acknowledge system* is used to *pull flits along*. It uses an (extra) R/A wire between the sending node and the receiving node:

- R/A is reset to 0 by receiving node when it is ready to receive a flit (i.e., when its flit buffer is empty)
- R/A is set to 1 by sending node when the sending node is about to send a flit

Sending node must wait for R/A to become 0 before setting it to 1 and sending the flit. It knows that the data has been received when receiving node resets R/A to 0.



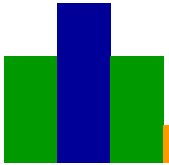
Deadlock

A tricky problem in network routing is the *deadlock*: it may occur when *packets cannot be forwarded because they are blocked by other packets waiting to be forwarded and these packets are blocked in a similar way such that actually none of the packets can move*.

Example:

- node 1 wishes to send to 3 via node 2;
- node 2 wishes to send to 4 via node 3;
- node 3 wishes to send to 1 via node 4;
- node 4 wishes to send to 2 via node 1;

A famous example was introduced by Dijkstra: *5 philosophers* may think or eat spaghetti using a circular table with 5 plates and 5 forks. Eating spaghetti requires two forks (!?). A deadlock appears when all philosophers are present and all held the right-hand side fork.



..Deadlock

A general solution to deadlock is to use *virtual channels*:

- multiple virtual channels are associated with a physical channel and
- they are time-multiplexed onto the physical channel.



Networked computers as a multicomputer

A *cluster of workstations* or a *network of workstations* offers a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing. The key advantages are:

- high performance workstations and PCs available at low cost
- the latest processors can be easily incorporated into the system
- one may use existing (or slightly modified) software

Parallel programming tools for clusters

- *PVM (Parallel Virtual Machine)* - developed in late 1980s and very popular
- *MPI (Message-Passing Interface)* - standard defined in 1990s



Real networks of workstations

Ethernet

- common communication network for workstations
- it consists of a single wire to which all computer are attached
- an ethernet frame consists of : preamble (64b), destination address (48b), source address (48b), type (16b), data (variable length), frame check sequence (32b)

Main problems: collisions and significant message latency.



..Real networks of workstations

Ring structures:

- a collection of workstations connected in a ring. Example: Token rings

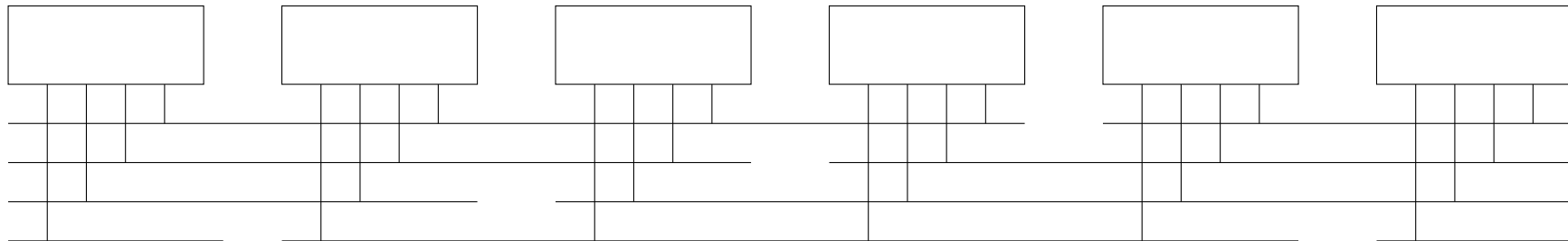
Point-to-point communication:

- this provides the highest intercommunication bandwidth;
- various point-to-point communications can be created using hubs and switches
- examples: HIPPI (High Performance Parallel Interface), Fast (100 MHz) and Gigabit Ethernet, and fiber optics.

..Real networks of workstations

Overlapping connectivity networks:

- there are regions of connectivity and they may overlap
- an example for Ethernet is





Performance analysis

Speedup factor:

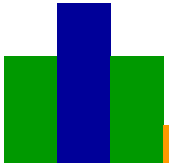
- Denote by t_s the execution time on a single processor and by $t_p(n)$ the execution time on a n processor system. The speedup factor $S(n)$ is

$$S(n) = \frac{t_s}{t_p(n)}$$

Notice: The sequential and parallel algorithms may be (and usually are) different.

- A different approach is to use the number of *computational steps*: if cs_s is the number of computational steps on a single processor and $cs_p(n)$ is the number of parallel computational steps on a n processor system, then $S_{cs}(n)$ is

$$S_{cs}(n) = \frac{cs_s}{cs_p(n)}$$



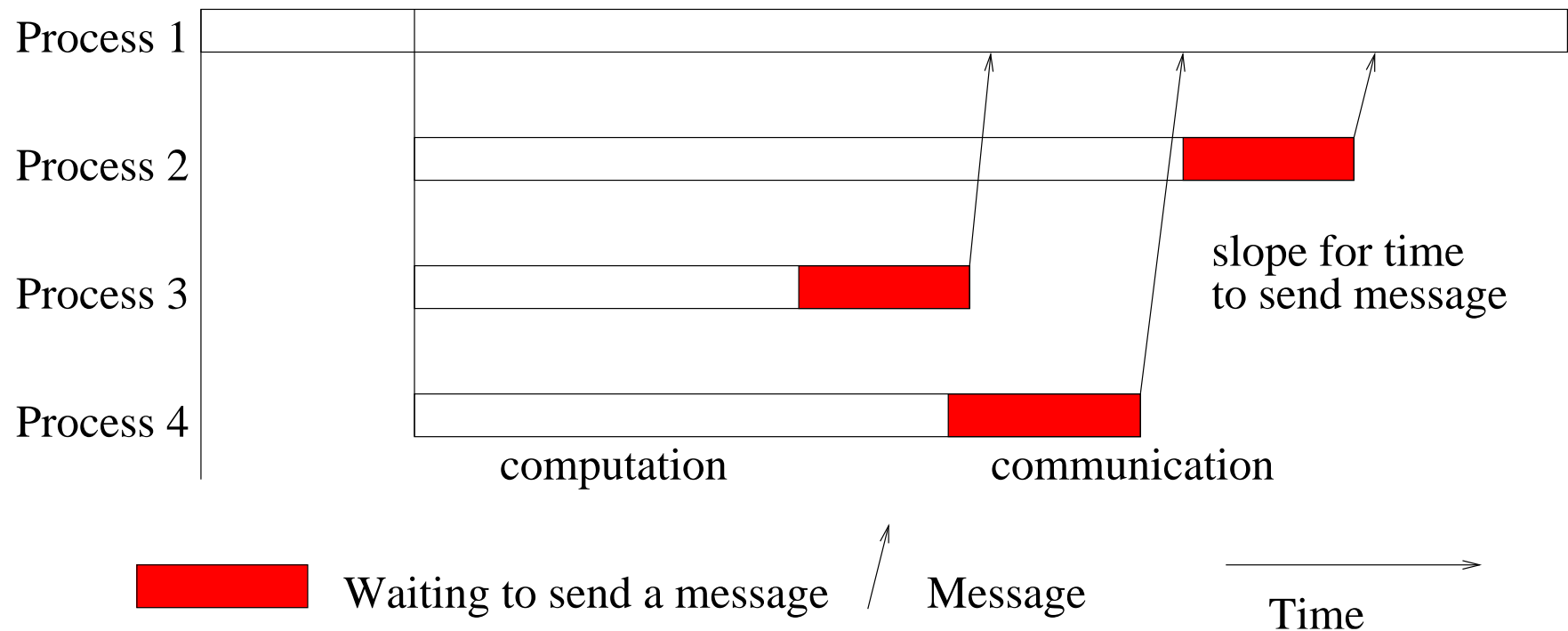
..Performance analysis

Superlinear speedup:

- the maximum speedup $S(n)$ is n , i.e., a *linear speedup*.
- sometimes there are cases when $S(n) > n$, known as “superlinear speedup” (e.g., in some search algorithms), but usually this is due to the use of an suboptimal sequential algorithm, or the use of some feature of the architecture that favors the parallel algorithm, or the use of extra memory in the multiprocessor system

..Performance analysis

Space-time diagram of a message passing program



An useful measure is *communication overhead*

$$\text{Communication overhead} = \frac{\text{Computation time}}{\text{Communication time}}$$



..Performance analysis

Amdahl's law: It allows to compute the maximum speedup for a problem having a serial factor f which can not be parallelized

- the speedup is

$$S(n) = \frac{t_s}{ft_s + \frac{(1-f)t_s}{n}} = \frac{n}{1 + (n-1)f}$$

With an infinite number of processors the maximum speedup is limited to $\frac{1}{f}$. E.g., if 5% is serial, then the maximum speedup is 20, irrespective of the number of processors.

Notice: Take it with some care. It was used in 1960s to promote single processor systems.



..Performance analysis

Efficiency: it gives the fraction of time the processors are being used on computation.

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{Number of processors}}$$

In symbols,

$$E(n) = \frac{t_s}{t_p(n) \times n} = \frac{S(n)}{n}$$

Notice: Usually it is given as a percentage, hence multiply the above result by 100.



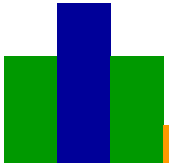
..Performance analysis

Cost:

Cost = Execution time \times Number of processors used

- In the sequential case the cost is $C_s = t_s \times 1 = t_s$
- in the parallel case, $C_p(n) = t_p(n) \times n = \frac{nt_s}{S(n)} = \frac{t_s}{E(n)}$

Cost-optimal parallel algorithm: one in which the cost to solve the problem on a multiprocessor is proportional to the cost of solving it on a single processor system. (In other words, $E(n)$ is constant.)



..Performance analysis

Scalability: used to capture the situation when the increasing of the size of the system leads to a proportional increasing of the result.

- *hardware scalability* - larger hardware leads to proportional better performance
- *algorithmic scalability* - the parallel algorithm is such that increasing the input data leads to a bounded increase of computational steps



..Performance analysis

Gustafson's law: rather than assuming the problem size is fixed, we now assume that the parallel time is fixed and the serial part of the problem does not increase along with the problem size

- *Scaled speedup factor (or Gustafson's law)*

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$

(here $0 < s, p < 1$ and $s + p = 1$)

Notice: For 5% and 20 processors this gives a better result than Amdahl's law (10.05 vs. 10.26) due to different assumptions.

Lesson 2: Message-passing computing

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming
Spring, 2004



Parallel programming options

Programming a message-passing multicomputer can be achieved by

- Designing a *special* parallel programming language (e.g., OCCAM for transputers)
- *Extending* the syntax/reserved words of an existing sequential high-level language to handle message passing (e.g., CC+, FORTRAN M)
- Using an existing sequential high-level language and providing a *library* of external procedures for message passing (e.g., MPI, PVM)

Another option will be to write a sequential program and to use a *parallelizing compiler* to produce a parallel program to be executed by multicomputer.



..Parallel programming options

We will concentrate on the third option. In such a case we have to say explicitly:

- *what processes* are to be executed
- *when to pass messages* between concurrent processes
- *what to pass* in the messages

Two methods are needed for this form of message-passing systems:

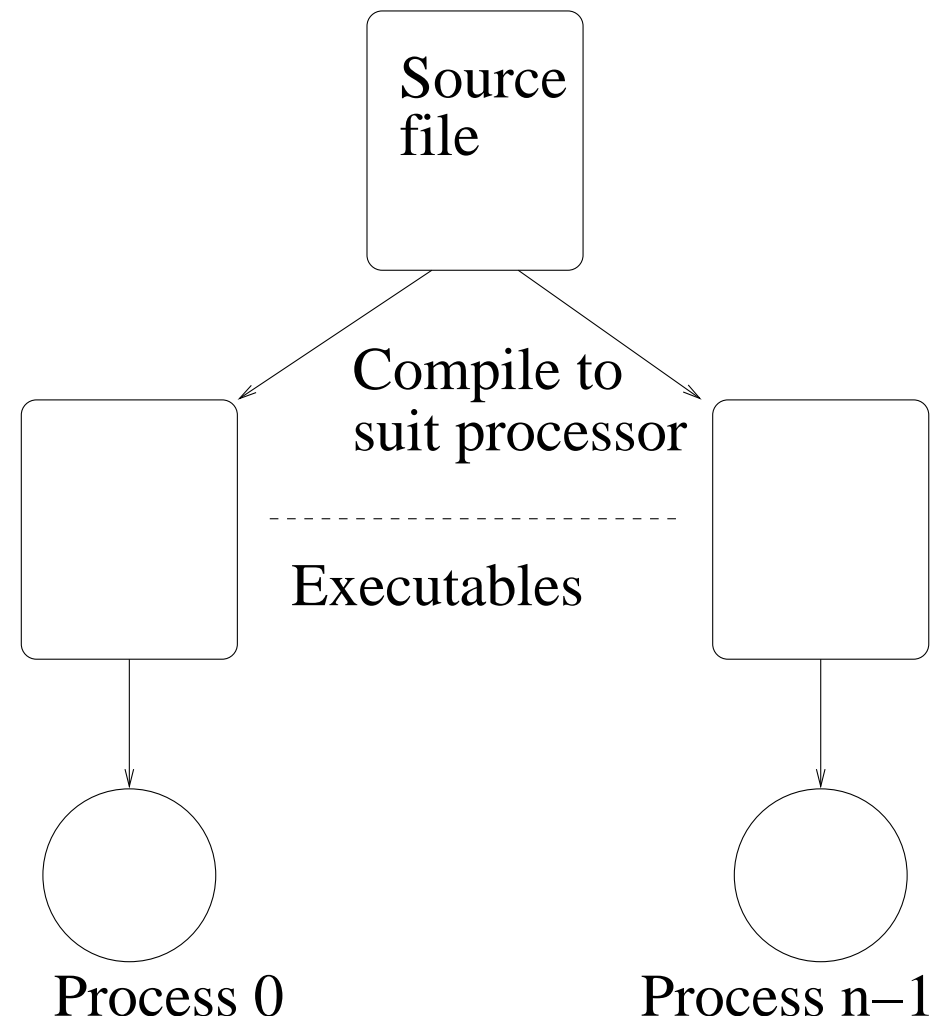
- a method of *creating separate processes* for execution on different computers
- a method for *sending and receiving messages*

SPMD (Single Program Multiple Data) model

In this case *different processes are merged into one program*. Within the program there are control statements that will customize the code, i.e., select different parts for each process.

Basic features:

- usually *static* process creation
- a basic model is MPI

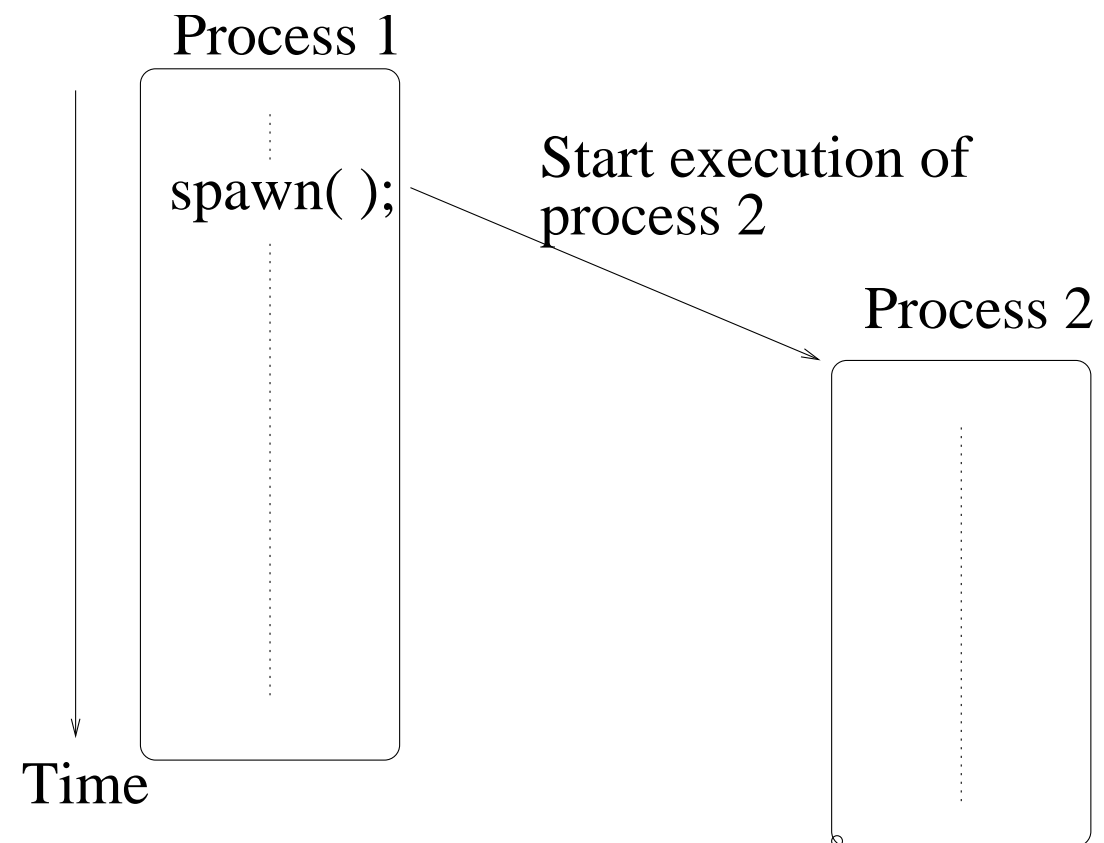


MPMD (Multiple Program Multiple Data) model

In this case *separate programs are written for each processor*. A *master-slave* approach is usually taken: a single processor executes a master process and the other processes are started from within the master process.

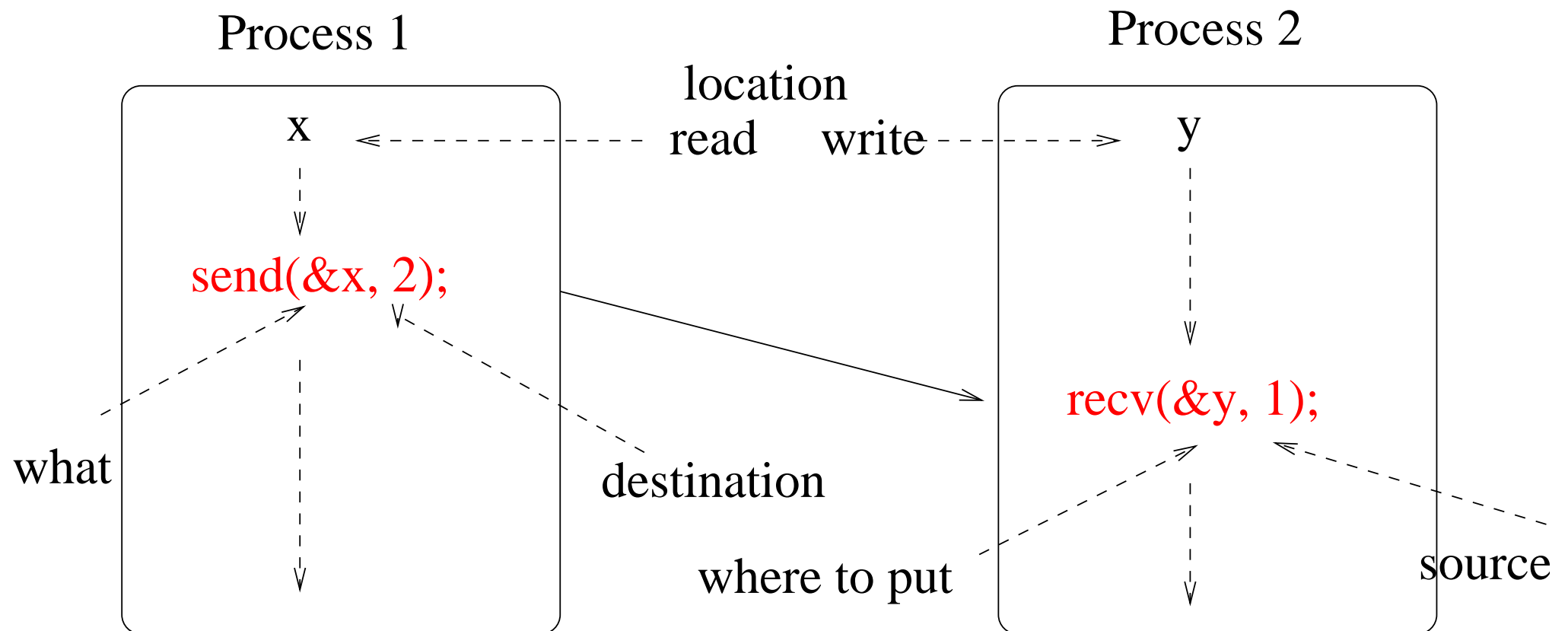
Basic features:

- usually *dynamic* process creation
- a basic model is PVM



Basic send and receive routines

Passing a message between processes using `send()` and `recv()` library calls





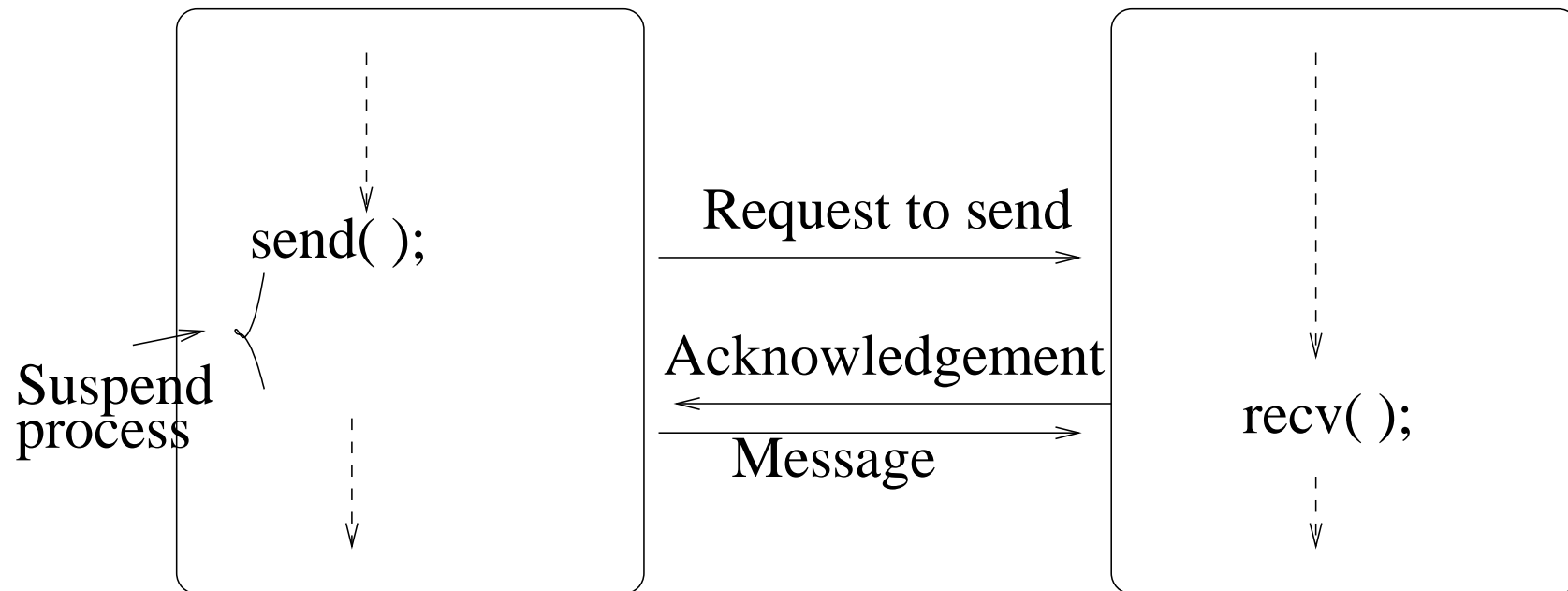
Synchronous message-passing

- *Synchronous message-passing routines return* when the message transfer has been completed.
- There is no need for message buffer storage.
 - The synchronous send routine could wait until the complete message can be accepted by the receiving process before sending the message.
 - The synchronous receive routine wait until the message it is expecting arrives.
- Synchronous routines perform two basic actions: They
 - *transfer data* and
 - *synchronize* processes

..Synchronous message-passing

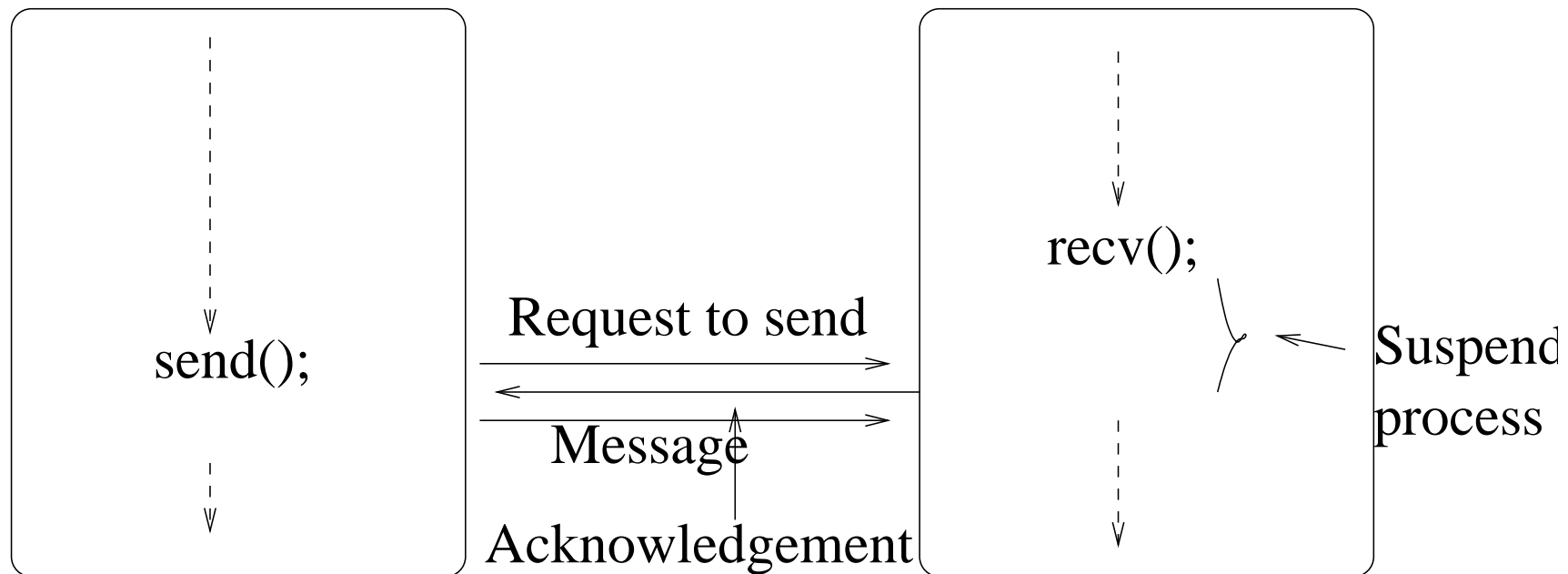
A three-way protocol is actually used here:

- Case 1: Process 1 arrives to `send()` before Process 2 arrives to `recv()`:



..Synchronous message-passing

- Case 2: Process 1 arrives to `send()` after Process 2 arrives to `recv()`:





Blocking and nonblocking message-passing

Blocking

- this term is used to describe routines that *do not return* until the transfer is *completed*.
- more precisely, the routines are *blocked from continuing* the process code
- generally speaking, the terms *synchronous* and *blocking* are synonymous

Non-blocking

- this term is used to describe routines that *return whether or not the message had been received*

Warning: These general terms were redefined in MPI, see below.



MPI definition of blocking and nonblocking

Blocking - return after their local actions are finished, though the message transfer may not have been completed (E.g., for `send()` it may return after the data are put in a buffer to be sent.)

Nonblocking - return immediately. In such a case it is assumed that the *data storage* to be used for the transfer *is not modified* by the subsequent statements before the transfer is completed and it is the programmer duty to ensure this.

Notice: This type of message passing is based on the use of message buffers between the source and destination processes. As the buffers are of finite length, it may happen that the `send()` routine is blocked because the available buffer space has been exhausted.



Message tag

A *message tag* is an extra information put in the message to differentiate between different messages being sent.

Example:

```
⋮  
send( &x, 2, 5 );  
⋮  
(process 1)
```

```
⋮  
recv( &x, 1, 5 );  
⋮  
(process 2)
```

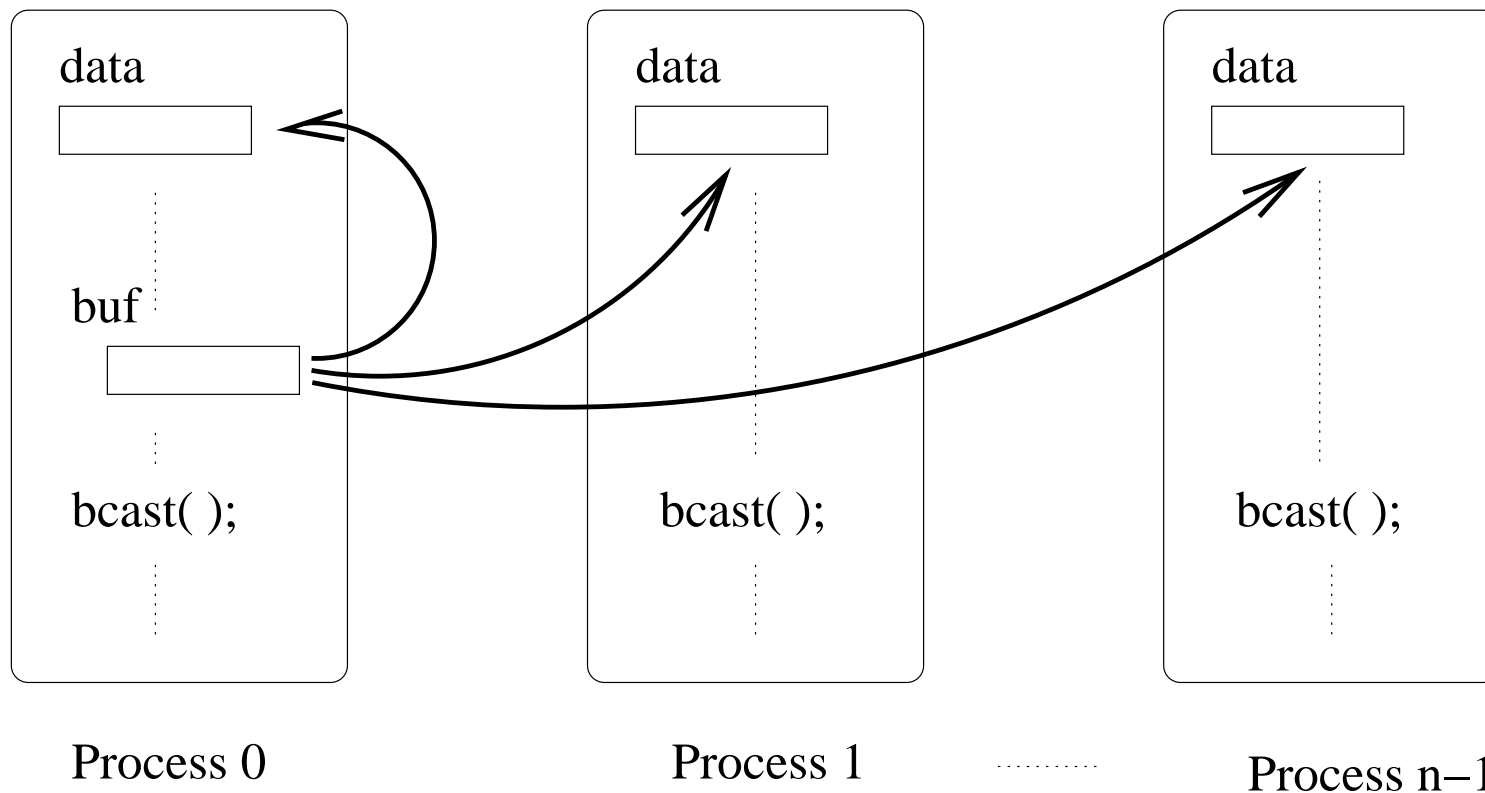
the tag 5 is used to match the send statement in process 1 to the receive statement in process 2.

Notice: If such a special type matching is not required, then a *wild card* message tag is used, so that `recv()` will match *any* `send()`

Broadcast

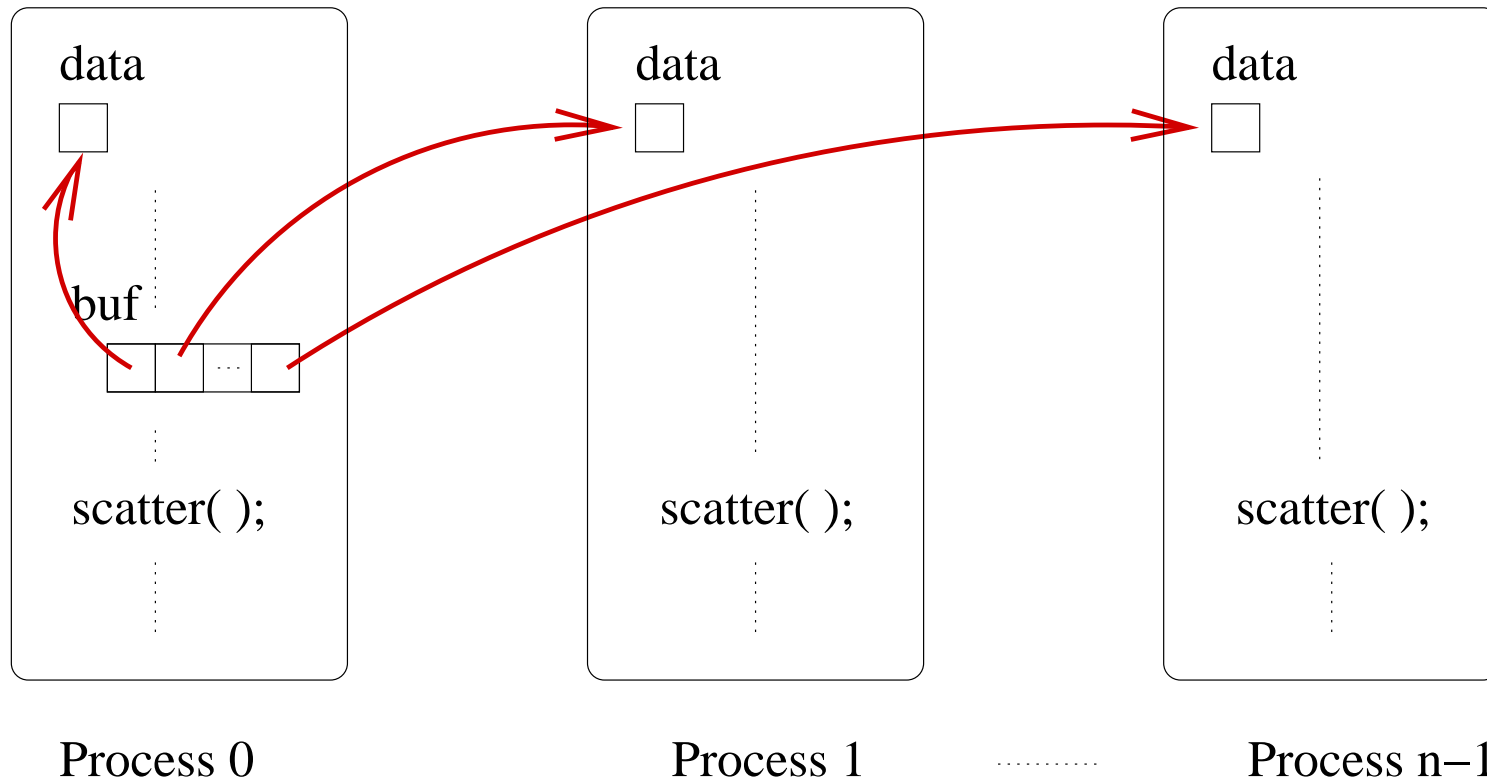
Broadcast is used to send the same message *to all* processes concerned with the problem.

Multicast is similar, but it is used to send a message *to a defined group* of processes.



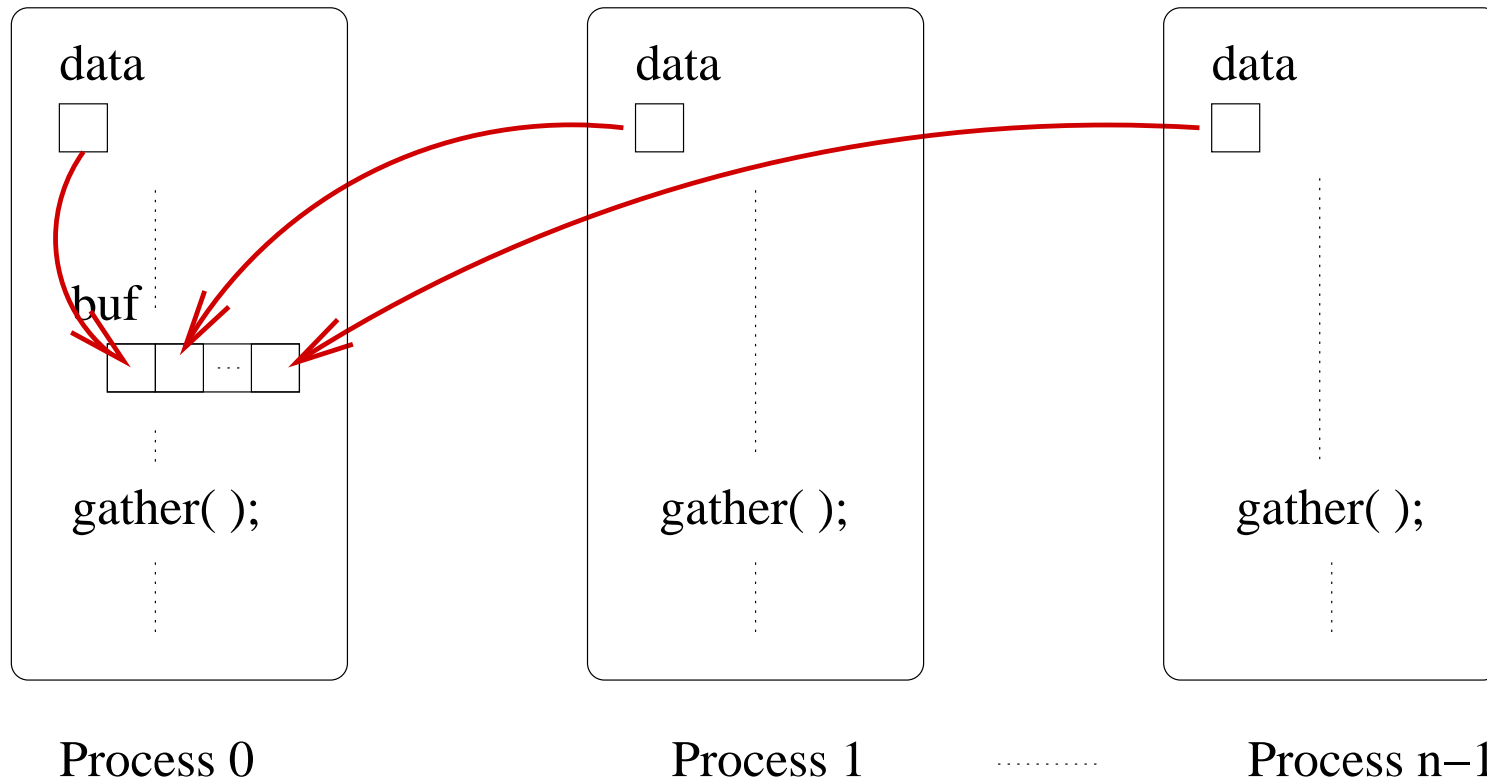
Scatter

Scatter is used to send *each element in an array* of data of the sending process to corresponding separate processes (datum from the i -th location goes to the i -th process).



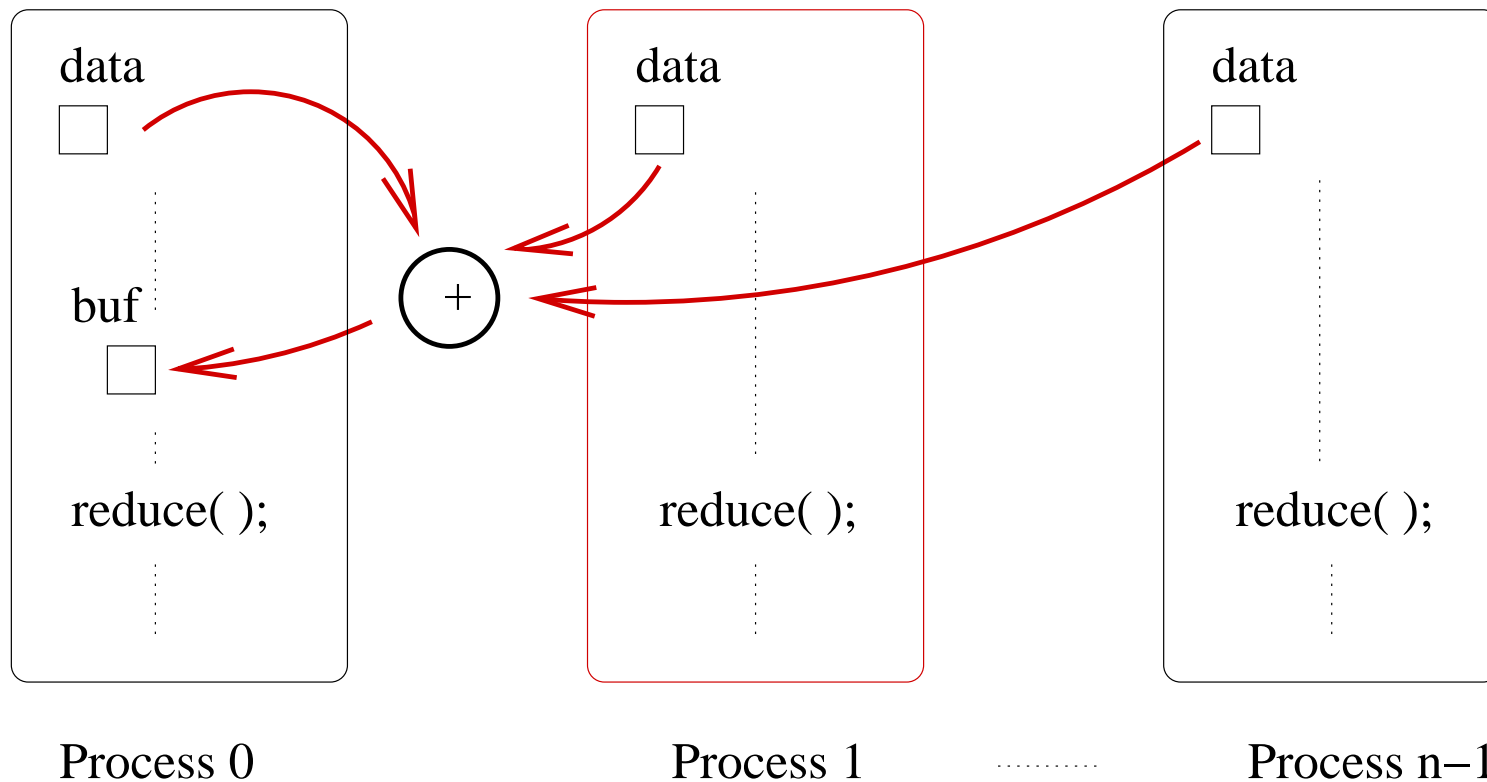
Gather

Gather is the opposite operation: the receiving process *collect in an array* the data sent by separate processes (datum from the i -th process goes to the i -th location).



Reduce

Reduce combines the `gather()` routine with an arithmetic or logical operation: the receiving process *collects* the data, *applies* the operation and *saves* it in its own memory.



PVM (Parallel Virtual Machine)

- a first wildly accepted attempt to use a workstation cluster as a multicomputer.
- it may be used to run programs on both homogeneous or heterogeneous multicomputers
- it has a collection of library routines to be used with C or FORTRAN programs
- free

MPI (Message Passing Interface)

- it is a standard developed by a group of academics and industrial people to increase the use and the portability of message passing
- several free implementation exists [we use the one from Chicago, `mpich`]



PVM

- The programmer decomposes the problem into *separate programs*; each program is written in C (or FORTRAN) and compiled to be run on a specific type of computers in the network
- The *set of computers* used for the problem must be defined prior the running of the programs. (A convenient way to do this is by using a *host-file* listing the names of the computers available. This host-file is then read by PVM.)
- The *routing* of messages between computers is done by *PVM demon processes* installed by PVM on the computers that form the virtual machine.



Basic PVM message-passing routines

General:

- all PVM *send* routines are *nonblocking* (or *asynchronous*), while PVM *receive* routines are either blocking (or *synchronous*) or *nonblocking*.
- both *messages tags* and *wild cards* may be used to match send and the corresponding receive occurrences.



Basic PVM message-passing routines

Basic send-receive routines

- `pvm_bsend()` and `pvm_brecv()` - it has two goals: to pack data and to send them
- `pvm_send()` and `pvm_recv()` - send and receive without packing; it has to be used with explicit packing statements:
 - clear the buffer (`pvm_init_send()`) and pack the data (with `pvm_pkint()`, `pvm_pkstr()`, ...) before sending and
 - unpack the data (using `pvm_upkint()`, `pvm_upkstr()`, ...) after receiving
- broadcast, scatter, gather, reduce may all be used; the PVM statements are:
`pvm_bcast()`, `pvm_scatter()`, `pvm_gather()`, `pvm_reduce()`



Example PVM program

We illustrate PVM programming with a simple *Sum* program: *the question is to add the numbers from a file using multiple processes.*

We use a master-slave approach:

- A master process creates the slave processes, reads data from the file and sends them to slaves (by multicast).
- Each slave identifies its portion of data, adds them and sends the result to the master [together with their identification number].
- The master receives the partial sums, adds them and prints the final result.



..Example PVM program

Master code

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <pvm3.h>
04  #define SLAVE "spsum"
05  #define PROC 10
06  #define NELEM 1000
07  main(){
08      int mytid,tids[PROC];
09      int n = NELEM, nproc = PROC;
10      int no,i,who,msgtype;
11      int data[NELEM],result[PROC],tot=0;
12      char fn[255];
13      FILE *fp;
```



..Example PVM program

```
14  /* Start slave tasks */
15      mytid = pvm_mytid(); /*enroll in PVM*/
16      no = pvm_spawn(SLAVE, (char**)0, 0, "", nproc, tids);
17      if (no < nproc){
18          printf("trouble spawning slaves \n");
19          for (i=0; i<no; i++) pvm_kill(tids[i]);
20          pvm_exit(); exit(1);
21      }
22  /* Open input file and initialize data */
23      strcpy(fn, getenv("HOME"));
24      strcat(fn, "/pvm3/src/rand_data.txt");
25      if ((fp = fopen(fn, "r")) == NULL){
26          printf("Can't open input file %s\n", fn);
27          exit(1);
28      }
29      for (i=0; i<n; i++) fscanf(fp, "%d", &data[i]);
```




..Example PVM program

```
30  /* Broadcast data to slaves */
31      pvm_initsend(PvmDataDefault);
32      msgtype = 0;
33      pvm_pkint(&nproc,1,1);
34      pvm_pkint(tids,nproc,1);
35      pvm_pkint(&n,1,1);
36      pvm_pkint(data,n,1);
37      pvm_mcast(tids,nproc,msgtag);  → out.0
38  /* Get results from Slaves */
39      msgtype = 5;
40      for (i=0; i<nproc; i++){
41          pvm_recv(-1,msgtype);  ← in.5
42          pvm_upkint(&who,1,1);
43          pvm_upkint(&result[who],1,1);
44          printf("%d from %d\n",result[who],who);
45      }
```



..Example PVM program

```
46  /* Compute global sum */
47      for (i=0; i<nproc; i++) tot += result[i];
48      printf ("The total is %d.\n\n",tot);
49  /* Program finished; exit PVM */
50      pvm_exit();
51      return(0);
52 }
```

..Example PVM program

Slave code

```
01  #include <stdio.h>
02  #include "pvm3.h"
03  #define PROC 10
04  #define NELEM 1000
05  main(){
06      int mytid,tids[PROC],n,me,i,msgtype;
07      int x,nproc,master,data[NELEM],sum;
08      mytid = pvm_mytid();
09  /* Receive data from master */
10      msgtype = 0;
11      pvm_recv(-1,msgtype); ← in.0
12      pvm_upkint(&nproc,1,1);
13      pvm_upkint(tids,nproc,1);
14      pvm_upkint(&n,1,1);
15      pvm_upkint(data,n,1);
16  /* Determine my tid */
17      for (i=0; i<nproc; i++)
18          if (mytid == tids[i])
19              {me = i; break;}
```



..Example PVM program

```
20  /* Add my portion of data */
21      x = n / nproc;
22      low = me * x;
23      high = low + x;
24      for (i=low; i<high; i++)
25          sum += data[i];
26  /* Send result to master */
27      pvm_initsend(PvnDataDeafult);
28      pvm_pkint(&me,1,1);
29      pvm_pkint(&sum,1,1);
30      msgtype = 5;
31      master = pvm_parent();
32      pvm_send(master,msgtype);  —→ out.5
33  /* Exit PVM */
34      pvm_exit();
35      return(0);
36  }
```



MPI

General: MPI is a *standard* with various implementations; one writes a *single program*, each process running its own copy;

Process creation and execution: generally it is not defined; it is specified at compiling time how many processes are using; only static process creation is supported (in MPI, version 1)

Communications: one defines the *scope* of the communication operation; the set of all involved processes may be accessed using the predefined variable `MPI_COMM_WORLD`; each process has a unique rank, a number from 0 to $n - 1$ (where n is the number of processes); other communication groups may be defined



..MPI

SPMD model: The shape of an MPI program is

```
main (int argc, char *argv[])
{
    MPI_Init(&argc,&argv);
    :
    /* find process rank */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    if (myrank == 0)
        /* master code */
    else
        /* slave code */
    :
    MPI_Finalize();
}
```



..MPI

Global and local variables: By default, any global declaration of variables will be *duplicated* in each process; the variables that are not to be duplicated need to be declared within the code executed by that process

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
if (myrank == 0){  
    int x,y;  
    :  
}  
elseif (myrank == 1){  
    int x,y;  
    :  
}
```

(x, y from process 0 are different from x, y in process 1)



..MPI

Point-to-point communication: message tags and wild cards may be used (MPI_ANY_TAG, or MPI_ANY_SOURCE)

Blocking routines: return when they are locally complete, i.e., when the location used for the message can be used again without affecting the message being sent; general format:

`MPI_Send(buf, count, datatype, dest, tag, comm)`

where: buf - address of send buffer, count - number of items to send, datatype - datatype of each item, dest - rank of destination process, tag - message tag; comm - communicator

and

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

where: buf - address of receive buffer, count - maximum number of items to receive, datatype - datatype of each item, src - rank of source process, tag - message tag, comm - communicator, status - status after operation

Example (blocking communication): To send an integer x from process 0 to process 1

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
if (myrank == 0){  
    int x;  
    MPI_Send(&x, 1, MPI_INT, 1, 73, MPI_COMM_WORLD);  
} elseif (myrank == 1){  
    int x;  
    MPI_Recv(&x, 1, MPI_INT, 0, 73, MPI_COMM_WORLD, status);  
}
```

Non-blocking communication: `MPI_Isend()` and `MPI_Irecv()` - return “*i*mediately”, even if the communication is not safe; to be used in combination with `MPI_Wait()` and `MPI_Test()` in order to ensure a complete communication.

Example (non-blocking communication): - same example

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
    int x;
    MPI_Isend(&x, 1, MPI_INT, 1, 73, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} elseif (myrank == 1){
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, 73, MPI_COMM_WORLD, status);
}
```

Send communication modes: Four basic modes are

1. *Standard mode send* - it is not assumed that the corresponding receive routine has started (buffer space is not defined here; if buffering is provided, send can complete before the corresponding receive was reached)
2. *Buffered mode* - send may start and return before a matching receive was reached (here it is necessary to specify buffer space)
3. *Synchronous mode* - send and receive have to complete together (however, they may start at any time)
4. *Ready mode* - send can only start if a matching receive was already reached (use it with care...)



..MPI

Collective communication: This applies to processes included in a communicator. The main operations are:

`MPI_Bcast()` - broadcast from root to all other processes

`MPI_Gather()` - gather values from processes in the group

`MPI_Scather()` - scatter parts of the buffer to processes

`MPI_Alltoall()` - send data from all processes to all processes

`MPI_Reduce()` - collect and combine values from processes

`MPI_Reduce_scatter()` - combine values and scatter results

Barrier: May be used to synchronize processes by stopping each process until all have reached the barrier call



Example of MPI program

We illustrate MPI programming style with the same simple question: *add the numbers from a file using multiple processes.*

A similar master-slave approach is used:

- A master process (process 0) detects the number of processes from communicator, reads data from the file and sends them to all processes (by broadcast).
- Each process (including the master) identifies its portion of data and adds them.
- The master collects the partial sums and adds them (using reduce statement) and prints the final result.



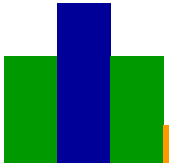
..Example (MPI program)

```
01  #include "mpi.h"
02  #includes <stdio.h>
03  #include <math.h>
04  #define MAXSIZE 1000
05  void main(int argc, char *argv){
06      int myid, numprocs;
07      int data[MAXSIZE], i, x, low, high, myresult, result;
08      char fn[255];
09      char *fp;
10      MPI_Init(&argc,&argv);
11      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
12      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13      if (myid == 0){
14          strcpy(fn,getenv("HOME"));
15          strcat(fn,"/MPI/rand_data.txt");
16          if((fp = fopen(fn,"r")) == NULL){
17              printf("Can't open the input file %s\n\n",fn);
18              exit(1);
19          }
}
```



..Example (MPI program)

```
20         for (i=0; i<MAXSIZE; i++) fscanf(fp,"%d",&data[i]);
21     }
22     /* Broadcast data */
23     MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);
24     /* Add my portion of data */
25     x = n / nproc;
26     low = myid * x;
27     high = low + x;
28     for (i=low; i<high; i++)
29         myresult += data[i];
30     printf("I got %d from %d\n", myresult,myid);
31     /* Compute global sum */
32     MPI_Reduce(&myresult,&result,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
33     if (myid == 0) printf("The sum is %d.\n",result);
34     MPI_Finalize();
35 }
```



SoC Cluster

SoC has a Linux cluster *Tembusu*:

- a 64-node cluster of Dell PCs; each node has two 1.4GHz Intel PIII CPUs with 1GB of memory
- the nodes are connected by Myrinet as well as Gigabit ethernet
- besides the standard Linux suite of tools, MPI (versions using Myrinet and Gigabit ethernet) and PVM are available on the cluster.

See

[https : //www.comp.nus.edu.sg/cf/tembusu/index.html](https://www.comp.nus.edu.sg/cf/tembusu/index.html)

for more.

Lesson 3a: Message-passing (II)

Lesson 3b: Embarrassingly parallel computations

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004

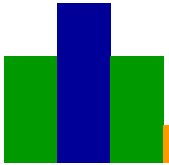
Lesson 3a: Message-passing (II)



Parallel program evaluation

Program evaluation:

- Both *theoretical* and *empirical* techniques may be used to determine the efficiency of (parallel) programs.
- The ultimate goal is to discriminate between various parallel processing techniques; a fine tune is also necessary to find the best *number of processes* and a good balance of the *computation time* and of the *communication time* of each process.
- An extra goal to find out if a parallel processing approach is actually better suited than a (usually simpler) sequential one.



Parallel execution time

Theoretical evaluation of parallel programs is based on:

- Parallel execution time, t_p , consists of the *computation time* t_{comp} and the *communication time* t_{comm} , namely

$$t_p = t_{comp} + t_{comm}$$

- Computation time, t_{comp} , is estimated similarly as in the case of sequential algorithms. (It is supposed here that all processes use identical computers.)
- Communication time, t_{comm} , consists of the *startup time* $t_{startup}$ (also called “message latency”) and the *time to send data* $n t_{data}$, where t_{data} is the time to transfer a data word, namely

$$t_{comm} = t_{startup} + n t_{data}$$



..Parallel execution time

- Theoretical analysis is intended to give a *starting point* to how an algorithm might perform in practice.
- Parallel computation time is evaluated in terms of *arithmetical and logical operations* - their actual time depends on the computer system (also assume all computers are identical)
- Communication time should use the same time units as computation time; all (unit) data types are supposed to require the same time to be delivered

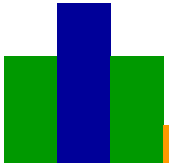
A typical example (IBM: SP-2; rough estimation):

- computation (1 arithmetical operation) - 1 unit
- time to transfer a data word - 55 units
- startup time - 8333 units



Latency hiding

- A general method to overcome the significant message communication time is to *overlap communication with subsequent computations*
- *Nonblocking send routines* are particularly useful to enable latency hiding
- A different technique is to *map multiple processes on a single processor*. (Due to the time-sharing facilities, the processor switches from one process to another when the first is stopped because of an incomplete message passing. *Threads*, a kind of “light processes” used in shared memory model, are particularly good for an efficient switching.)



Time complexity

Generalities:

- The starting point is to estimate the *number of computation steps*; here only *arithmetical* and *logical operations* are considered, ignoring other aspects such as computation tests, etc.
- The number of computation steps often depends on the *number of data items* handled by the algorithm.

We use various notations for the *order of magnitude* of a function.
(They are listed below.)



..Time complexity

- **The O -notation:** We say $f(x) = O(g(x))$ iff there exist positive constants c_2 and x_0 such that

$$0 \leq f(x) \leq c_2 g(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is “*at most as*” the growth of g)

- Example: if $f(x) = 4x^2 + 2x + 12$, then $f(x) = O(x^2)$ (reason: for $x \geq 3$, $0 < 4x^2 + 2x + 12 \leq 6x^2$).
Notice: x^{2003} is also good, i.e., $f(x) = O(x^{2003})$; hence, to be useful, always take the least growing function for g .



..Time complexity

- **The Θ -notation:** We say $f(x) = \Theta(g(x))$ iff there exist positive constants c_1, c_2 , and x_0 such that

$$0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is “*same as*” that of g . Clearly $f(x) = \Theta(g(x))$ implies $f(x) = O(g(x))$, but the converse is not true.)

- **The Ω -notation:** We say $f(x) = \Omega(g(x))$ iff there exist positive constants c_1 and x_0 such that

$$0 \leq c_1 g(x) \leq f(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is “*at least as*” that of g .)

Clearly, Θ is equivalent to O & Ω .



Time complexity of parallel algorithms

Time complexity analysis hides the lower terms, giving an estimation of the shape of time complexity function.

Example:

- Suppose we add n numbers on two computers using the following algorithm:
 1. Computer 1 sends $n/2$ numbers to computer 2.
 2. Both computers add $n/2$ numbers simultaneously.
 3. Computer 2 sends its partial sum back to computer 1.
 4. Computer 1 adds the partial sums to produce the final result.



..Time complexity of parallel algorithms

- Computation time (steps 2,4):

$$t_{comp} = n/2 + 1$$

- Communication time (steps 1,3):

$$\begin{aligned} t_{comm} &= (t_{startup} + n/2 t_{data}) + (t_{startup} + t_{data}) \\ &= 2t_{startup} + (n/2 + 1)t_{data} \end{aligned}$$

- Both computational and communication complexities are $O(n)$, hence the overall time complexity is $O(n)$.



Cost-optimal algorithms

A *cost-optimal* (or *processor-time optimal*) algorithm is one such that

$$(\text{parallel time complexity}) \times (\text{number of processors}) \\ = \text{sequential time complexity}$$

Example:

- Suppose the best known algorithm for problem P has time complexity $O(n \log n)$
- A parallel algorithm solving the same problem using n processes and having the time complexity $O(\log n)$ is cost-optimal, while a parallel algorithm which uses n^2 processes and has time complexity $O(1)$ is not cost-optimal.



A case study: Broadcasting

Broadcast on a hypercube network: Consider a three-dimensional hypercube. To broadcast from node 000 to every other node an efficient algorithm is:

- 1st step: 000 \rightarrow 001
- 2nd step: 000 \rightarrow 010
 001 \rightarrow 011
- 3rd step: 000 \rightarrow 100
 001 \rightarrow 101
 010 \rightarrow 110
 011 \rightarrow 111

The time complexity is $O(\log n)$ for an n -dimensional hypercube, which is optimal because the diameter is $\log n$.



..Broadcasting

Gather on a hypercube network: The *reverse algorithm* can be used to gather data from all nodes to a node, say 000 (in the 3-dim hypercube case):

- 1st step: $100 \rightarrow 000$
 $101 \rightarrow 001$
 $110 \rightarrow 010$
 $111 \rightarrow 011$
- 2nd step: $010 \rightarrow 000$
 $011 \rightarrow 001$
- 3rd step: $001 \rightarrow 000$

In such a case the messages become *longer* as the data are gathered, hence the time complexity is increased over $\log n$.

..Broadcasting

Broadcast on a mesh network:

- Nodes of top line: firstly send to left (if any), then down.
- Other nodes: send down.

Denoting the nodes in order from left-to-right and top-to-down, the broadcasting in a 4×4 mesh is:

step-1: 1 → 2

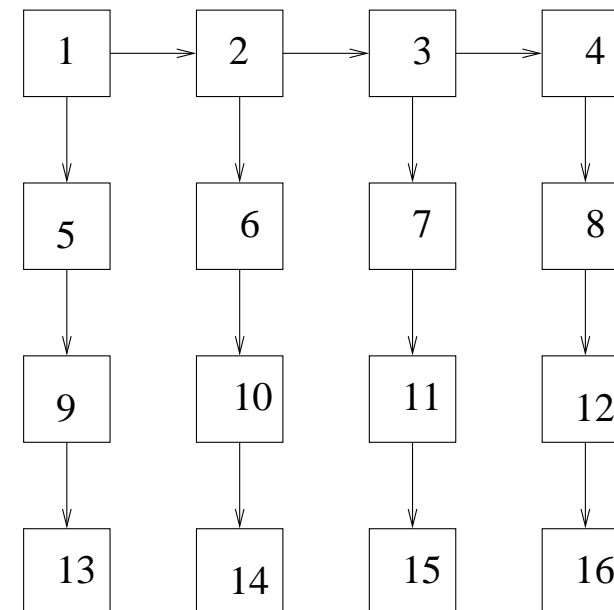
step-2: 1 → 5; 2 → 3

step-3: 5 → 9; 2 → 6; 3 → 4

step-4: 9 → 13; 6 → 10; 3 → 7; 4 → 8

step-5: 10 → 14; 7 → 11; 8 → 12

step-6: 11 → 15; 12 → 16



The algorithm is optimal as the number of steps is equal to the diameter of the mesh.



..Broadcasting

Broadcast on a workstation cluster:

- Broadcasting on a *single Ethernet connection* can be done using a single message that is read by all the destinations on the network *simultaneously*, hence time complexity is $O(1)$.



Evaluating programs empirically

Measuring execution time: To measure the execution time between point *L1* and *L2* in the code, one may use the following construction

```
L1:  time(&t1);                /* start timer */
    :
L2:  time(&t2);                /* stop timer */
elapsedTime = difftime(t2,t1); /* elapsedTime = t2-t1 */
printf(`Elapsed time = %5.2f seconds`,elapsedTime);
```

Notice: MPI provides the routine *MPI_Wtime* for returning time (in seconds). Each processor may have its own clock, hence be careful not to mix such timers.



..Evaluating programs empirically

Communication time by Ping-Pong methods: To empirically estimate the communication time from a process P_1 to a process P_2 one may use the following method: *Immediately* after receiving the message P_2 *send the message back* to P_1 .

P_1

```
L1:  time(&t1);  
send(&x, P2);  
recv(&x, P2)  
L2:  time(&t2);  
elapsedTime = 0.5 * difftime(t2, t1);  
printf(`Elapsed time = %5.2f seconds`, elapsedTime);
```

P_2

```
recv(&x, P1)  
send(&x, P1);
```



Debugging strategies

A useful *three-step approach* to debugging message-passing programs is:

1. If possible, run the program as a single process and debug as a normal *sequential program*.
2. Execute the program using *two to four multi-tasked processes* on a *single computer*. Now examine actions such as checking that messages are indeed being sent to correct places. It is very common to make mistakes with message tags and have messages sent to wrong places.
3. Execute the program using *two to four processes* but now across *several computers*. This step helps to find the impact of network delays on synchronization and timing constraints of your program.

Lesson 3b: Embarrassingly parallel computations



b) Embarrassingly parallel computations

Embarrassingly parallel computations:

- A (truly) *embarrassingly parallel computation* is a computation that can be divided into a number of completely *independent parts*.
- More common is a *nearly embarrassingly parallel computation* where there is a *loose communication* between processes: a “master” process distributes initial data and collects the results; the set of “slave” processes is (truly) embarrassingly parallel.



Example 1: Image processing

A few (low level) image operations

- *Shifting*: Objects are shifted by Δx in x -dimension and by Δy in y -dimension

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

- *Scaling*: Objects are scaled by factor S_x in x -dimension and by S_y in y -dimension

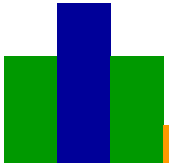
$$x' = x S_x$$

$$y' = y S_y$$

- *Rotation*: Objects are rotated through an angle θ about the origin of the coordinate system:

$$x' = x \cos \theta + y \sin \theta$$

$$y' = -x \sin \theta + y \cos \theta$$



..Image processing

Algorithm

- Divide the area into regions for individual processes (square or row regions may be used).
- Each process independently performs the transformation for the pixels of its own region.

The (pseudo)code for shifting transformation (using row partition) is given below: the image area in 480×640 ; each slave process is supposed to handle 10 lines.



..Image processing

```
/* send starting row number to processes */ Master code
for (i=0,row=0; i<48; i++,row=row+10)
    send(row,Pi);

/* initialize tmp */
for (i=0; i<480; i++)
    for (j=0; j<640; j++)
        tmpMap[i][j] = 0;

/* accept new coords for each pixel */
for (i=0; i < (480*640); i++){
    recv(oldRow,oldCol,newRow,newCol,Pany);
    if (!((newRow<0) || (newRow>=480) || (newCol<0) || (newCol>=640)))
        tmpMap[newRow][newCol] = map[oldRow][oldCol];
}

/* update bitmap */
for (i=0; i<480; i++)
    for (j=0; j<640; j++)
        map[i][j] = tmpMap[i][j];
```




..Image processing

```
/* receive starting row number */
```

Slave code

```
recv(row, Pmaster) ;
```

```
/* for each pixel compute new coords and send to master */
```

```
for (oldRow=row; oldRow<(row+10);oldRow++)
```

```
    for (oldCol=0; oldCol<640;oldCol++){
```

```
        newRow = oldRow + deltaX
```

```
        newCol = oldCol + deltaY
```

```
send(oldRow,oldCol,newRow,newCol,Pmaster) ;
```

```
}
```



..Image processing

Analysis (the time to compute new coordinates is fixed and small):

- **Sequential:** $t_s = n^2 = O(n^2)$
- **Parallel ($p + 1$ processes):**
 - Communication: In general $t_{comm} = t_{startup} + m t_{data}$, hence for p processes
$$t_{comm} = p(t_{startup} + t_{data}) + n^2(t_{startup} + 4 t_{data})$$
$$= (n^2 + p)t_{startup} + (4 n^2 + p)t_{data} = O(p + n^2)$$
 - Computation: (2 additions; each process handle n^2 / p pixels)
$$t_{comp} = 2 \frac{n^2}{p} = O(n^2 / p)$$
 - Overall execution time: $t_p = t_{comm} + t_{comp}$

Conclusion: *Perform badly*, as the *communication part far exceeds the computation part* and the overall parallel time is significantly grater than the sequential time.

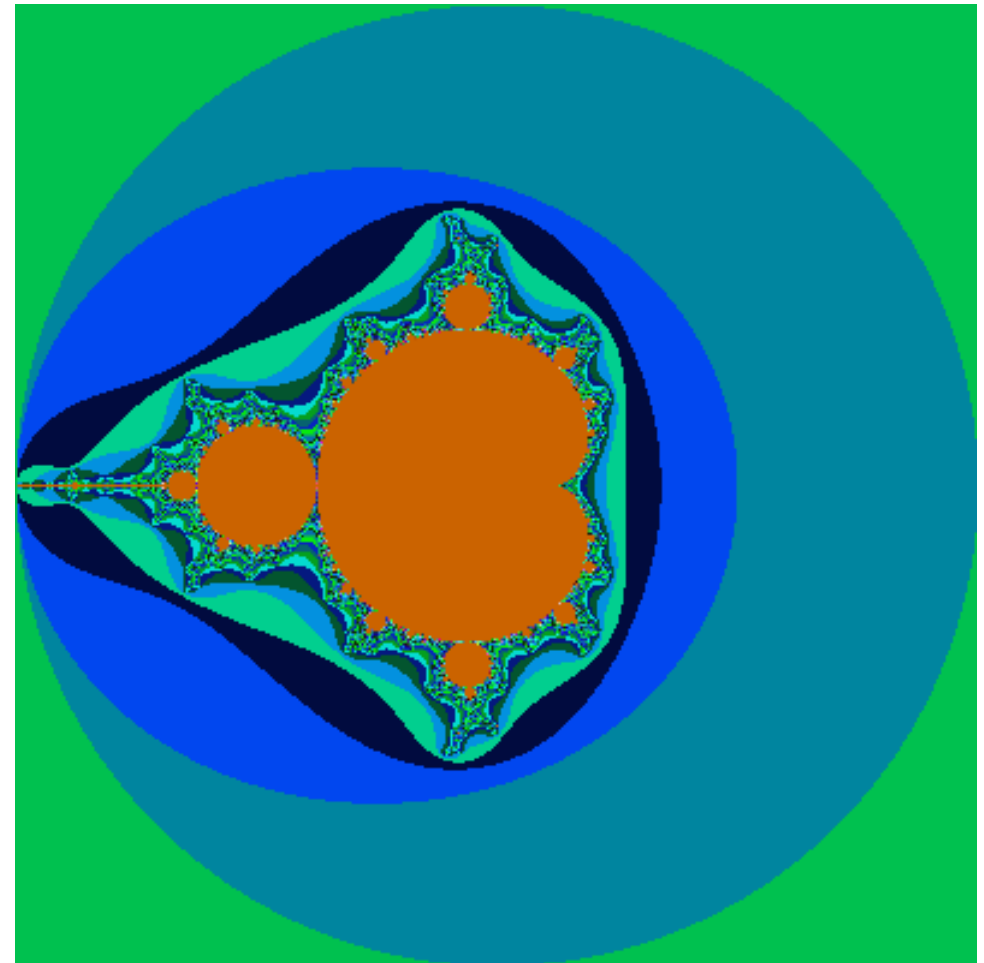
Example 2: Mandelbrot set

Mandelbrot set: Use the transformation

$$z \mapsto z^2 + c$$

on complex numbers. For given c iterate this starting with 0 till

- the module of the number is grater than 2 or
- the number of iterations reaches some arbitrary limit.



Notice: This is an example of an intensive computation for each pixel. The colors describe the numbers of steps necessary to get a module grater than 2.



..Mandelbrot set

A sequential program

```
structure complex{
    float real;
    float imag;
}

int calcPixel(complex c){
    int count, max;
    complex z;
    float tmp, lengthSq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0;
    do{
        tmp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag - c.imag;
        z.real = tmp;
        lengthSq = z.real * z.real + z.imag * z.imag;
        count++;
    } while (lengthSq < 4.0) && (count < max);
    return count;
}
```



..Mandelbrot set

Parallel version (Static task assignment):

Master code

```
for (i=0,row=0; i<48; i++,row=row+10)
    send(row, $P_i$ );
for (i=0; i<(480*640); i++){
    recv(c,color, $P_{any}$ );
    display(c,color);
}
```

Slave code (scaling factors are used to fit the display, i.e., $scaleReal = (realMax - realMin) / dispWidth$ and similarly for $scaleImg$)

```
recv(row, $P_{master}$ );
for (x=0; x<dispWidth, x++){
    for (y=row; y<(row+10), y++){
        c.real = minReal + ((float)x * scaleReal);
        c.imag = minImag + ((float)y * scaleImag);
        color = calcPixel(c);
        send(c,color, $P_{master}$ );
    }
}
```



..Mandelbrot set

Rough analysis (a more precise analysis is complicate as the number of iterations per pixel is difficult to estimate):

- **Sequential:** $t_s \leq \max \times n = O(n)$
- **Parallel ($p + 1$ processes):**
 - Comm-1: send row number to each slave
$$t_{comm1} = p(t_{startup} + t_{data})$$
 - Computation: slaves perform their computation in parallel
$$t_{comp} \leq \frac{\max \times n}{p}$$
 - Comm-2: results are passed to master using individual sends
$$t_{comm2} = \frac{n}{p}(t_{startup} + t_{data})$$
 - Overall execution time:
$$t_p \leq \frac{\max \times n}{p} + \left(\frac{n}{p} + p\right)(t_{startup} + t_{data})$$

Conclusion: When *max is large* the first factor is dominating and speedup may get closed to $p - 1$.



..Mandelbrot set

A few improvements may be used to increase the efficiency of the parallel program:

- The startup time is generally long, hence a better approach for slaves may be to wait till all the pixels in a row are computed and then to send the full row to the master
- The time to compute may differ from slave to slave, hence a better approach is to use a *dynamic task assignment*:
 - Allocate the rows one by one to slaves;
 - When a slave is ready and return the results a new task is send to him for processing.



..Mandelbrot set

Parallel version (Dynamic task assignment):

Master code

```
count = 0;
row = 0;
for (k=0; k < procNo; k++){
    send(row,  $P_k$ , dataTag);
    count++;
    row++;
}
do{
    recv(slave, r, color,  $P_{any}$ , resultTag);
    count--;
    if(row < dispHeight){
        send(row,  $P_{slave}$ , dataTag);
        count++;
        row++;
    } else
        send(row,  $P_{slave}$ , terminatorTag);
    rowRecv++;
    display(r, color);
} while(count > 0);
```




..Mandelbrot set

Slave code

```
recv(y, Pmaster, sourceTag);
while(sourceTag == dataTag){
    c.imag = imagMin + ((float) y * scaleImag);
    for (x=0; x < dispWidth; x++){
        c.real = realMin + ((float) x * scaleReal);
        color[x] = calcPixel(c);
    }
    send(i, y, color, Pmaster, resultTag);
    recv(y, Pmaster, sourceTag);
}
```

(Variable count is used to count the number of busy slaves.)

Notice: For this type of algorithms an empirical estimation of the overall execution time is for sure a better suited method.



Example 3: Monte-Carlo methods

Monte-Carlo methods are *approximative* methods based on random selections in computation.

Examples:

- *Computing π* : Take a circle of unit radius included into a 2×2 square. Then one gets

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi \times 1^2}{2 \times 2} = \frac{\pi}{4}$$

Then, choose randomly points within the square and score how many points happen to lie within the circle.

Notice: May be used for arbitrary integrals, but it's not too efficient.



..Monte-Carlo methods

- *Computing an Integral:* An integral

$$\int_a^b f(x)dx$$

is approximatively computed taking (uniformly distributed) randomly generated values between a and b , say, x_1, \dots, x_n and computing

$$\frac{b-a}{n} \sum_{i=1}^n f(x_i)$$



..Monte-Carlo methods

Pseudo-random number generation:

- Monte-Carlo methods are based on a procedure for generating pseudo-random numbers.
- A common method is to use the function

$$x_{i+1} = (ax_i + c) \bmod m$$

where a, c, m are well-chosen constants

- A good selection is with $m = 2^{31} - 1$ (prime number), $a = 16807$ and $c = 0$. (For any non-zero a , a sequence including all $2^{31} - 2$ different numbers is created before a repetition occurs.)
- A disadvantage is that these generators are relatively slow.



..Monte-Carlo methods

Parallel random number generation:

- One can see that

$$x_{i+1} = (ax_i + c) \bmod m$$

gives

$$x_{i+k} = (Ax_i + C) \bmod m$$

where

$$A = a^k \bmod m \text{ and}$$

$$C = c(a^{k-1} + \dots + a^1 + a^0) \bmod m$$

- Using such a recurrence relation one may do the generation in parallel:
 - the first k numbers are generated sequentially
 - then, one generates in parallel appropriate subsequences:
 $1, k+1, 2k+1, \dots$ and $2, k+2, 2k+2, \dots$ and so on.

Lesson 4: Partitioning and Divide-and-Conquer strategies

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004



Partitioning strategies

Partitioning strategy is used as a generic term for any procedure based on the division of a problem into parts.

Example (adding numbers): The goal is to add a sequence of n numbers x_0, \dots, x_{n-1} . The algorithm divides the sequence into p parts

$$x_0, \dots, x_{(n/p)-1}$$

$$x_{(n/p)}, \dots, x_{2(n/p)-1}$$

$$\vdots$$

$$x_{(p-1)(n/p)}, \dots, x_{p(n/p)-1} (= x_{n-1})$$

The numbers in each subsequence are added independently, then the resulting partial sums are added to get the final result.



..Partitioning strategies

1st implementation (using separate `send()` and `recv()`):

Master:

```
r = n/p
for (i=0, x=0; i<p; i++, x=x+r)
    send(numbers[x], r,  $P_i$ );
sum = 0;
for (i=0; i<p; i++){
    recv(partSum,  $P_{any}$ );
    sum = sum + partSum;
}
```

Slave:

```
recv(numbers, r,  $P_{master}$ );
partSum = 0;
for (i=0; i<r; i++)
    partSum = partSum + numbers[i];
send(partSum,  $P_{master}$ );
```




..Partitioning strategies

Analysis:

Sequential: Computation requires $n - 1$ additions.
Hence time complexity is $O(n)$.

Parallel (1st implementation):

—1st communication: $t_{comm1} = p(t_{startup} + (n/p)t_{data})$

—computation (in slaves): $t_{comp1} = n/p - 1$

—2nd communication (return partial sum):

$$t_{comm1} = p(t_{startup} + t_{data})$$

—final computation: $t_{comp1} = p - 1$

—overall:

$$t_p = 2pt_{startup} + (n + p)t_{data} + n/p + p - 2$$



..Partitioning strategies

Conclusion:

- The computation part is *decreasing* from $n - 1$ to $n/p + p - 2$.
- The communication part is *linear* on both the size of data and the number of processes.
- The overall parallel time complexity is *worse* than sequential time complexity.
- To be useful, the slaves should have heavier computations (say, $t(n)$ such that the $t(n) - t(n/p) > 2pt_{startup} + (n + p)t_{data}$)



..Partitioning strategies

2nd implementation (using broadcast/multicast):

Master:

```
r = n/p
bcast(numbers, r,  $P_{master}$ ,  $P_{group}$ ) ;
sum = 0 ;
for (i=0; i<p; i++){
    recv(partSum,  $P_{any}$ ) ;
    sum = sum + partSum ;
}
```

Slave:

```
bcast(numbers, r,  $P_{master}$ ,  $P_{group}$ ) ;
start = slaveNumber * r ;
end = start + r ;
partSum = 0 ;
for (i=start; i<end; i++)
    partSum = partSum + numbers[i] ;
send(partSum,  $P_{master}$ ) ;
```



..Partitioning strategies

3rd implementation (using scatter/reduce):

Master:

```
r = n/p  
scatter(numbers, r, Pmaster, Pgroup) ;  
reduceAdd(sum, Pmaster, Pgroup) ;
```

Slave:

```
scatter(numbers, r, Pmaster, Pgroup) ;  
start = slaveNumber * r ;  
end = start + r ;  
partSum = 0 ;  
for (i=start; i<end; i++)  
    partSum = partSum + numbers[i] ;  
reduceAdd(partSum, Pmaster, Pgroup) ;
```

Notice: The analysis of the 2nd or 3rd implementation is similar. (The real values depend on the network type and the particular implementations of broadcasting procedures.)



Divide-and-conquer

Divide-and-conquer is a particular partitioning strategy where *the subproblems are of the same form as the larger problem*.

Hence one can recursively apply the partition procedure to get smaller and smaller problems.

Add a list of numbers:

```
int add(list){
if (numberOfElements(list) =< 2) return theirSum;
else{
    divide(list,list1,list2);
    return(add(list1) + add(list2));
}
}
```

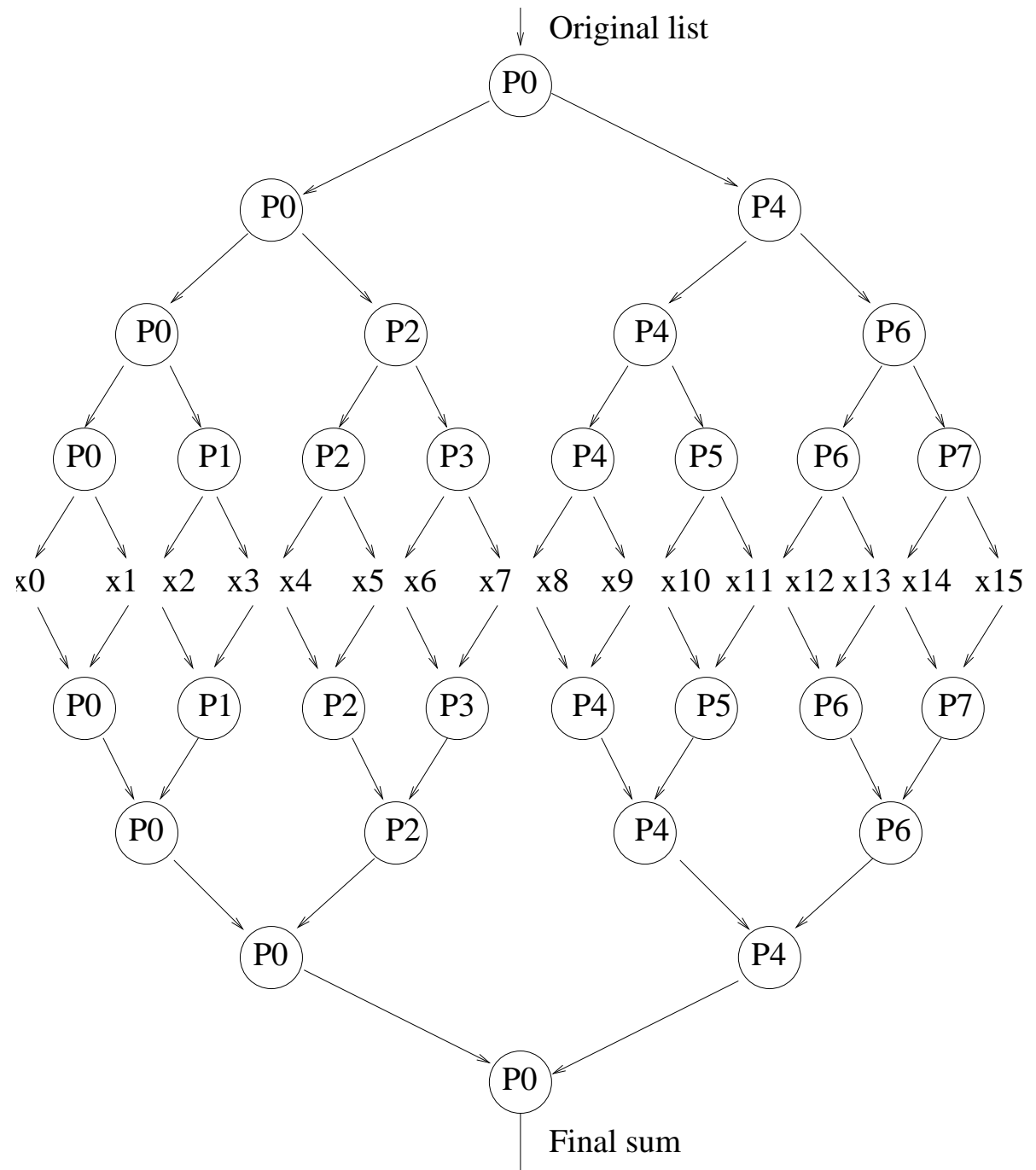
Notice: This is a general procedure which may be implemented either in a sequential or a parallel way.

..Divide-and-conquer

Parallel implementation:

(a) In a first stage, the list is recursively divided till each process receives a two-element list.

(b) In the next stage the partial results are collected using an opposite tree structure.



..Divide-and-conquer

Parallel code (Process P_i): Let

$$k(i) = \begin{cases} \bullet r & \text{if } i = 0 \text{ (and there are } 2^r \text{ processes)} \\ \bullet \text{ the greatest power of 2 which divides } i & \text{otherwise} \\ & [\text{i.e., } 2^{k(i)} \text{ divides } i, \text{ but } 2^{k(i)+1} \text{ does not divide } i] \end{cases}$$

```
if ( !(myRank == 0) ) recv(list, PmyRank-2k(i)) ;
for(i=k-1; i>=0; i--){
    divide(list, list, list2);
    send(list2, PmyRank+2i) ;
}
partSum = sumOf(list);
for(i=0; i<k; i++){
    recv(partSum2, PmyRank+2i) ;
    partSum = partSum + partSum2;
}
if ( !(myRank == 0) ) send(partSum, PmyRank-2k(i)) ;
```



..Divide-and-conquer

Examples:

Process P0

```
divide(list,list,list2);
send(list2,P4);
divide(list,list,list2);
send(list2,P2);
divide(list,list,list2);
send(list2,P1);
partSum = sumOf(List);
recv(partSum2,P1);
partSum = partSum + partSum2;
recv(partSum2,P2);
partSum = partSum + partSum2;
recv(partSum2,P4);
partSum = partSum + partSum2;
```

Process P4 ($k(4) = 2$):

```
recv(list,P0);
divide(list,list,list2);
send(list2,P6);
divide(list,list,list2);
send(list2,P5);
partSum = sumOf(List);
recv(partSum2,P5);
partSum = partSum + partSum2;
recv(partSum2,P6);
partSum = partSum + partSum2;
send(partSum,P0);
```




..Divide-and-conquer

Analysis: Assume $n = 2^k$ and $p = 2^r$. Then:

- division: $t_{comm1} = t_{startup}\log_2 p + (n/2 + n/4 + \dots + n/(2^r))t_{data}$
 $= t_{startup}\log_2 p + [n(p-1)/p]t_{data}$
- collecting results: $t_{comm2} = t_{startup}\log_2 p + t_{data}\log_2 p$
- computation: $t_{comp} = n/p - 1 + \log_2 p$
- overall:

$$t_p = 2t_{startup}\log_2 p + [\log_2 p + n(p-1)/p]t_{data} + n/p - 1 + \log_2 p$$



..Divide-and-conquer

Conclusion:

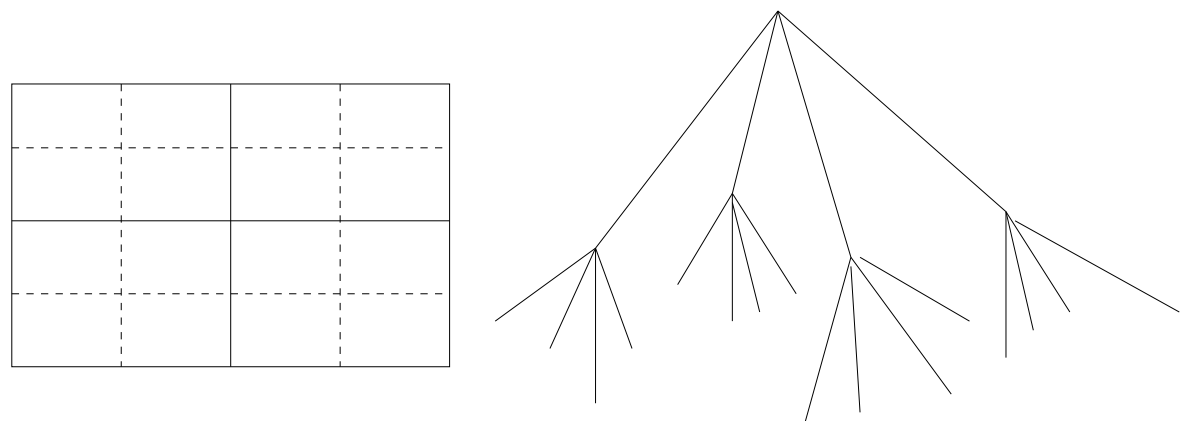
- The computation part is *decreasing*.
- The communication is still large: *linear* on the size of data, but now *logarithmic* on the number of processes.
- The algorithm becomes efficient when the computation part is large enough.

Notice that the work is not equally loaded on processes.

..Divide-and-conquer

M-ary divide and conquer: A similar approach may be used when the problem is divided into more than two parts at each stage [if possible]. Now, m – *ary* trees are to be used, where m is the branching degree used at each stage.

Example: Such a technique may be applied to image processing by dividing each dimension into two parts at each stage, hence 4-ary trees are to be used [trees in which each node has four children.]





Case studies: 1. Sorting

Sorting using bucket sort:

1. The range of the numbers, say the interval $[min, max)$, is divided into p equal regions:
 $[min, min + (max - min)/p)$ (Range 1)
 $[min + (max - min)/p, min + 2(max - min)/p)$ (Range 2)
 \vdots
 $[min + (p - 1)(max - min)/p, max)$ (Range p)
2. For each region a *bucket* is assigned to hold the numbers that fall within that region.
3. The numbers are placed into the appropriate buckets.
4. Finally, the numbers from each bucket are sorted using a sequential algorithm.

Notice: Works well if the numbers are uniformly distributed in $[min, max)$.



..Sorting

Sequential algorithm: Suppose there are n numbers uniformly distributed in $[min, max)$. Then each bucket has an average of n/p numbers. Hence

$$t_s = n + p(n/p)\log_2(n/p)$$

where:

- n — time to parse the numbers and place into buckets
- p — number of buckets
- $(n/p)\log_2(n/p)$ — sequential time to sort n/p numbers of each bucket



..Sorting

Parallel time (1st version): Sort the buckets in parallel. Hence

$$t1_p = n + (n/p)\log_2(n/p) + t_{startup} + nt_{data}$$

where, comparing with the former equation,

- p in the 2nd term disappears (work in parallel)
- $t_{startup} + nt_{data}$ is added (send data, via broadcast)



..Sorting

Parallel time (2nd version): A further parallelization is possible by *placing numbers into buckets in parallel*, as well. Suppose we have n numbers and p regions.

1. Separate the unsorted numbers a_0, \dots, a_{n-1} into groups of n/p elements: a_0 to $a_{n/p-1}$, $a_{n/p}$ to $a_{2n/p-1}$, \dots , one for each processor $P_i (i = 1, \dots, p)$.
2. Each processor P_i parses its group of n/p numbers and generates *small buckets* b_{i1}, \dots, b_{ip} (b_{ik} contains the numbers that fall into Range k).
3. The small buckets are sent to appropriate processors (P_i sends b_{ik} to P_k).
4. Each processor sorts its bucket of numbers.



..Sorting

Analysis: Assume we have n numbers, p processes and the numbers are uniformly distributed into buckets. Then:

1. send numbers to processes: $t_{comm1} = t_{startup} + nt_{data}$ (broadcast)
2. generate small buckets: $t_{comp1} = n/p$
3. send small buckets: $t_{comm2} = (p - 1)(t_{startup} + (n/p^2)t_{data})$
(if communications could overlap; otherwise multiply by p)
4. sort large buckets: $t_{comp} = (n/p)\log_2(n/p)$

Overall:

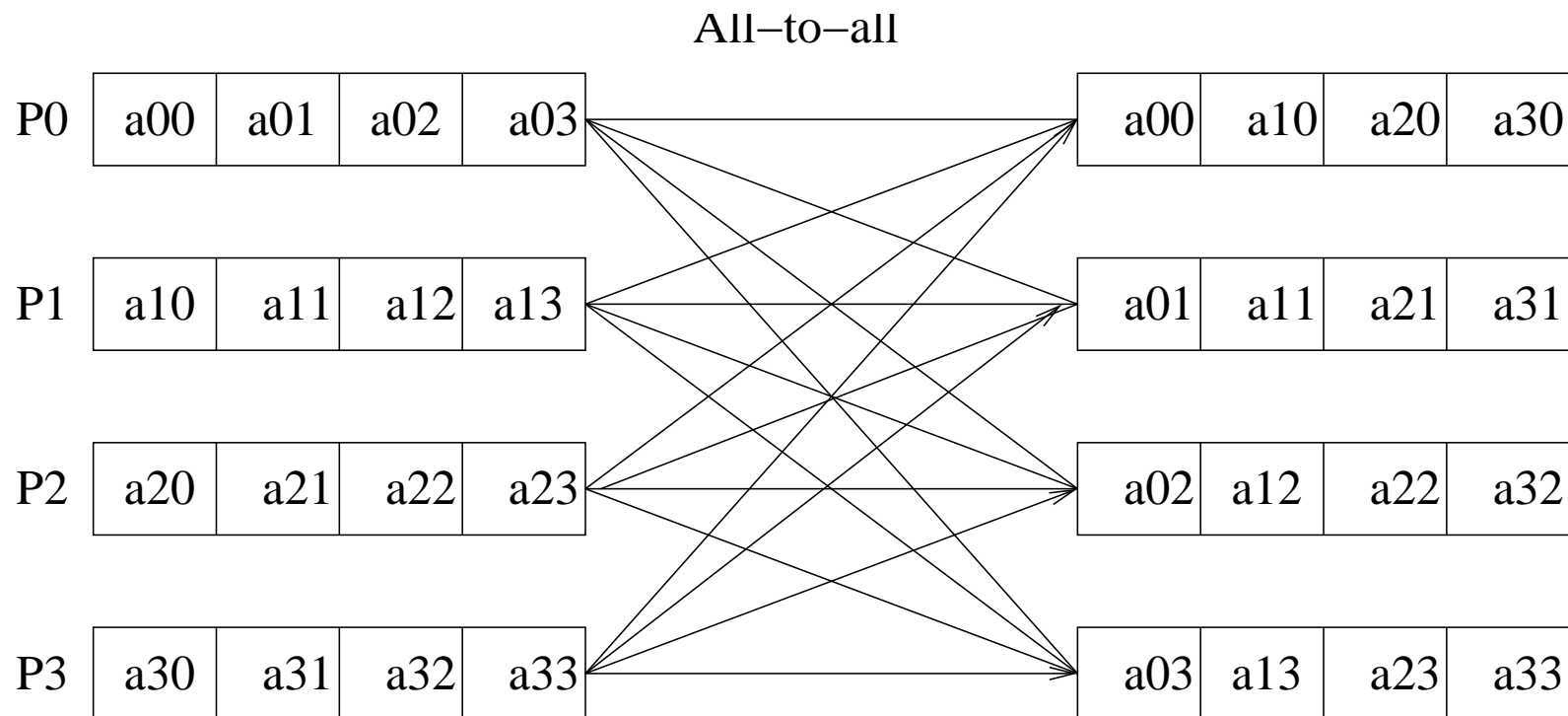
$$t_p = n/p + (n/p)\log_2(n/p) + pt_{startup} + (n + n/p^2)t_{data}$$

Conclusion: The computation time to generate buckets is decreasing from n to n/p , but the communication time is increasing.

..Sorting

All-to-all routines:

- A point where further improvement is possible is Step 3 where each process has to send specific data to all other processes.
- Such sort of communication may be made more efficient using standard routines, e.g., `MPI_Alltoall()`.





Case studies: 2. Numerical integration

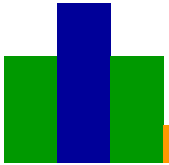
Numerical integration:

- A general divide-and-conquer technique divides the region continuously into parts.
- The procedure stops when some optimization function decides that the regions are sufficiently divided.

Example: For

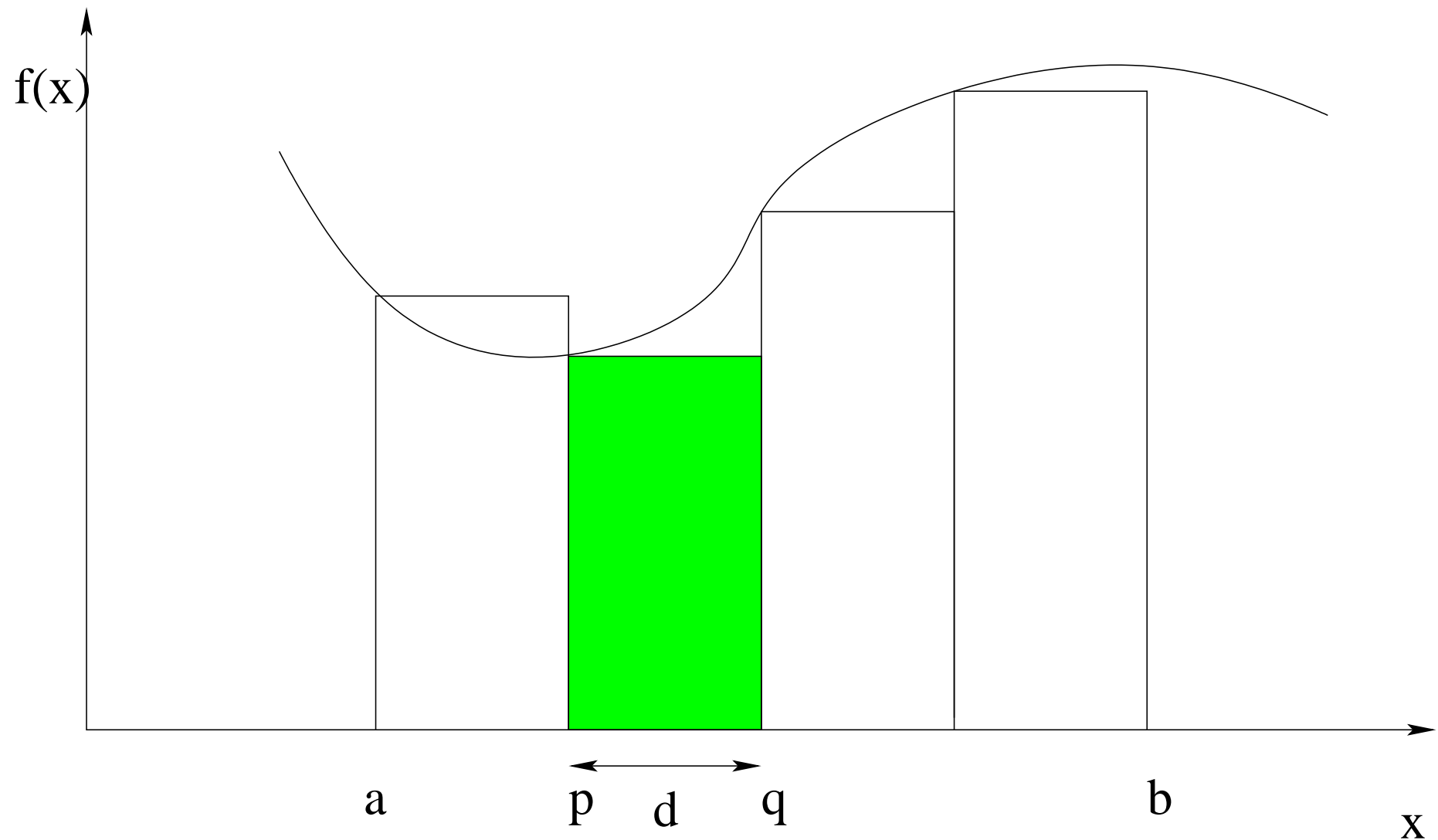
$$I = \int_a^b f(x)dx$$

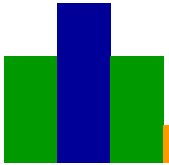
divide the range $[a, b]$ into separate parts and perform the computation in each part using a separate process.



..Integration

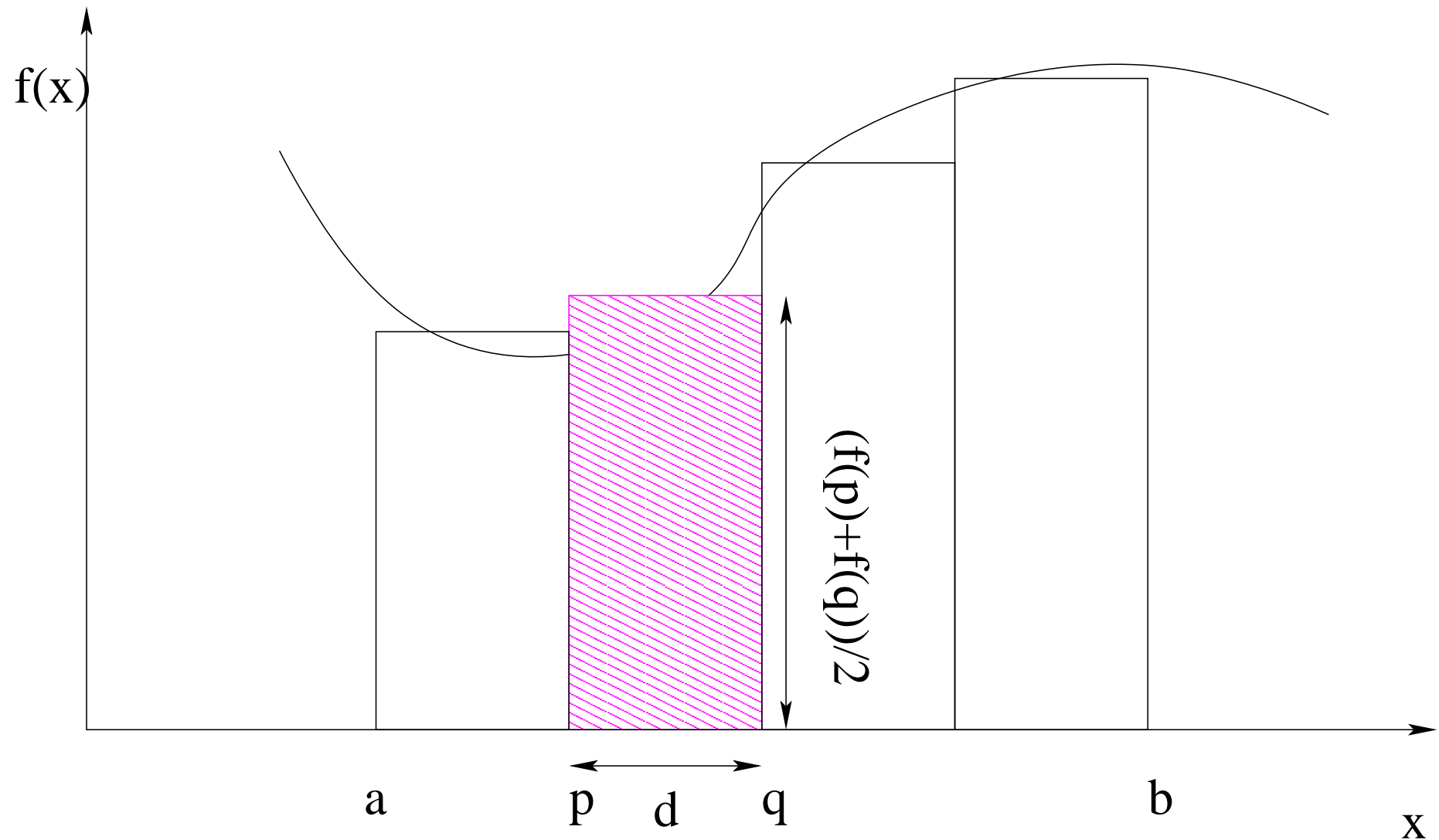
Approximation using rectangles:

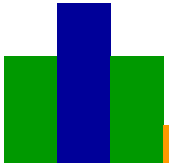




..Integration

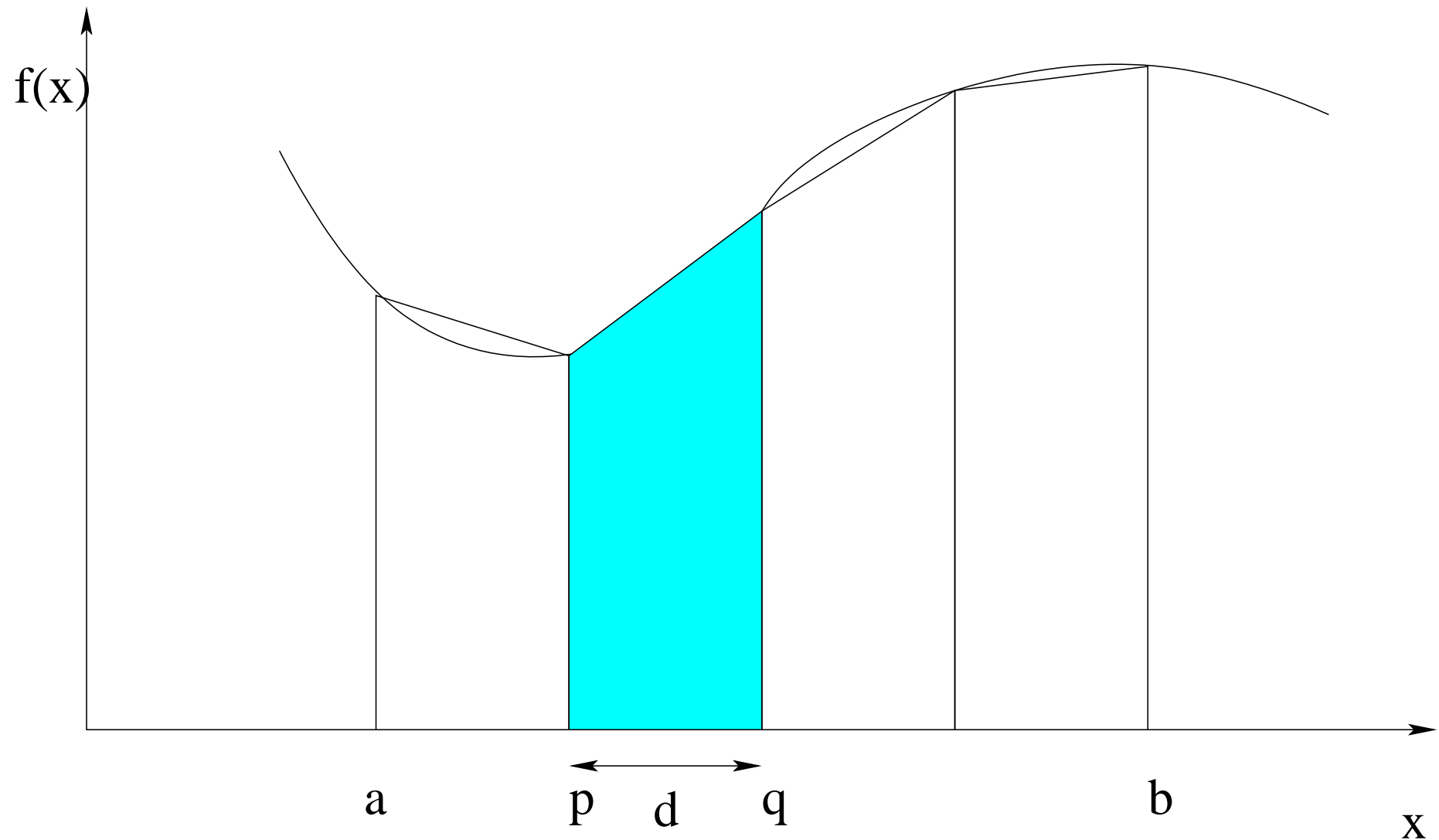
Better approximation using rectangles (average value):





..Integration

Using trapezoidal approximations:





..Integration

Sample pseudocode:

(Static assignment, SPMD program, 2nd rectangular method)

Process P_i :

```
if (i == master){
    printf("`Enter number of interval`");
    scanf("%d",&n);
}
bcast(&n,  $P_{master}$ ,  $P_{group}$ );
region = (b-a)/p;
start = a + region * i;
end = start + region;
area = 0.0;
for(x=start; x<end; x=x+d)
    area = area + f(x) + f(x+d);
area = 0.5 * area * d;
reduceAdd(&integral, &area,  $P_{group}$ );
```



..Integration

Final comments:

- This program is using a simple partitioning strategy. An approach using divide-and-conquer strategy may be used as well.
- The above program does not consider any termination condition. It may be (1) repeated for larger and larger n till a good approximation is obtained, or (2) the convergence criteria may be included into the program.
- Sometimes the function is not so smooth and a nonuniform division of the interval may fit better. E.g., using more processes closer to 0 gives a better approximation for $\int_{0.01}^1 (x + \sin(1/x)) dx$.



Case studies: 3. Gravitational N -body problem

Gravitational N -body problem:

- This is a (computationally) difficult, practical problem which was widely studied.
- Goal: find positions and movements of bodies in space subject to gravitational forces using Newtonian laws of physics.
- The gravitational force between two bodies of masses m_a and m_b is

$$F = \frac{Gm_a m_b}{r^2}$$

where G is the gravitational constant and r is the distance between the bodies.



.. N -body problem

- Subject to the (resulting) force F , a body will accelerate according to Newton's 2nd law

$$F = ma$$

where m is the mass of the body and a is the resulting acceleration.

- Let the time interval be Δt and v^{t+1} (resp. v^t) be the velocity at time $t + 1$ (resp. t). Using the approximation

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

one gets

$$v^{t+1} = v^t + \frac{F \Delta t}{m}$$



..N-body problem

- The position of the body is changed from x^t to x^{t+1} by

$$x^{t+1} = x^t + v\Delta t$$

- After all these computations a new position is obtained and the procedure is repeated:

position \mapsto force \mapsto acceleration \mapsto velocity \mapsto position...

- In a 3-dim space, the distance between bodies $a = (x_a, y_a, z_a)$ and $b = (x_b, y_b, z_b)$ is

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

and the general force gives actions along each direction

$$F_x = k(x_b - x_a)/r, \quad F_y = k(y_b - y_a)/r, \quad F_z = k(z_b - z_a)/r$$

where $k = \frac{Gm_a m_b}{r^2}$



.. N -body problem

Sequential (pseudo)code: (sketched, one iteration step)

```
for(t=0; t<tmax; t++)
    for(i=0; i<N; i++){
        F = Force(i); /* compute force on i-th body */
        v[i]new = v[i] + F * dt / m;
        x[i]new = x[i] + v[i]new * dt;
    }
for(i=0; i<N; i++){
    v[i] = v[i]new;
    x[i] = x[i]new;
}
```



.. N -body problem

Parallel approach: (sketched)

- The algorithm uses $O(N^2)$ operations for one iteration (each body acts on each other body), hence is not practical for usual N -body problems where N is large.
- The time complexity is reduced using the observation that a cluster of distant bodies can be approximated as a single body (containing the total mass of the bodies in the cluster)



.. N -body problem

Barnes-Hut algorithm:

- Start with a cube containing all bodies. The cube is divided into eight subcubes.
- If a subcube contains no particles, then the subcube is deleted from further considerations
- If a subcube contains more than one body, then it is recursively divided until every subcube contains one body.

*Notice: This way **octrees** are obtained (each node has up to 8 children).*



.. N -body problem

- The total mass and the center of mass is stored in each node of the tree.
- The force on each body can be obtained by traversal the tree from the root and stopping when a clustering approximation may be used, usually,

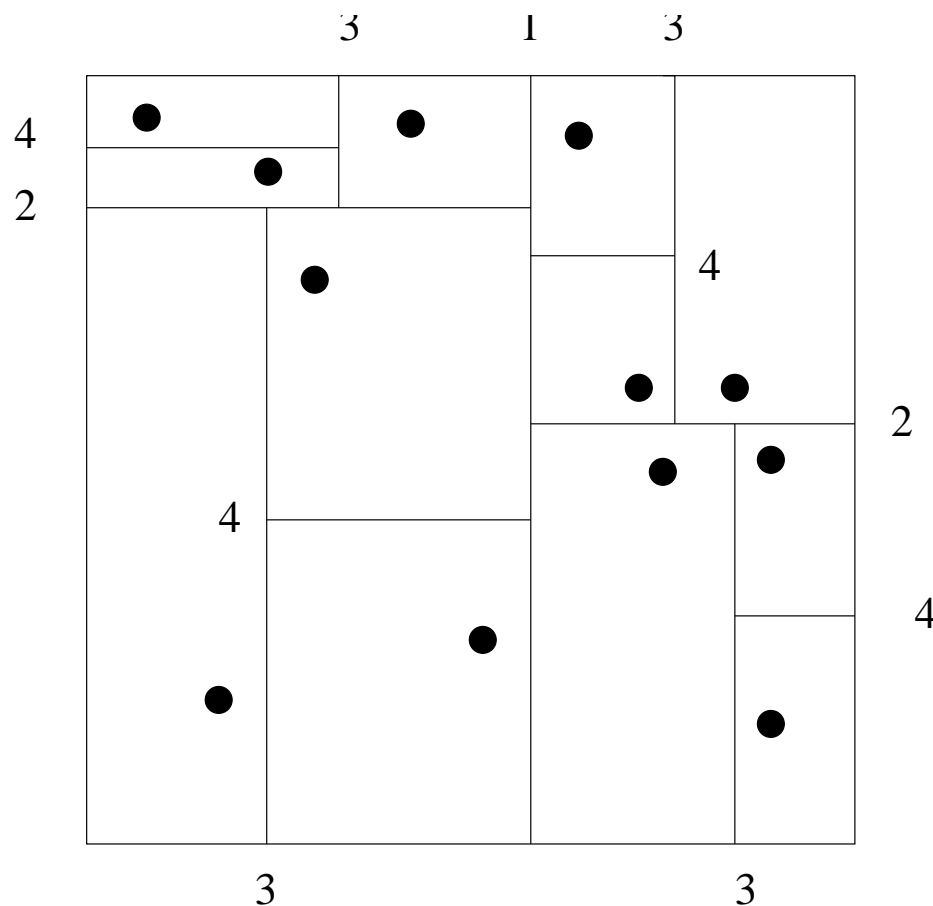
$$r \geq d/\theta$$

where r - distance to the mass center, d - cube dimension, and θ constant, typically 1.0.

Both, the construction of the tree and the computation of all forces require $O(n \log n)$ time, hence the algorithm is $O(n \log n)$ (rather than the direct approach requiring $O(N^2)$ time).

..N-body problem

The resulting tree may be not well balanced, hence a different approach will be to divide the space into subcubes having (closed to) an equal number of bodies. An illustration is below (in 2-dim case).



Lesson 5: Pipeline computations

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004

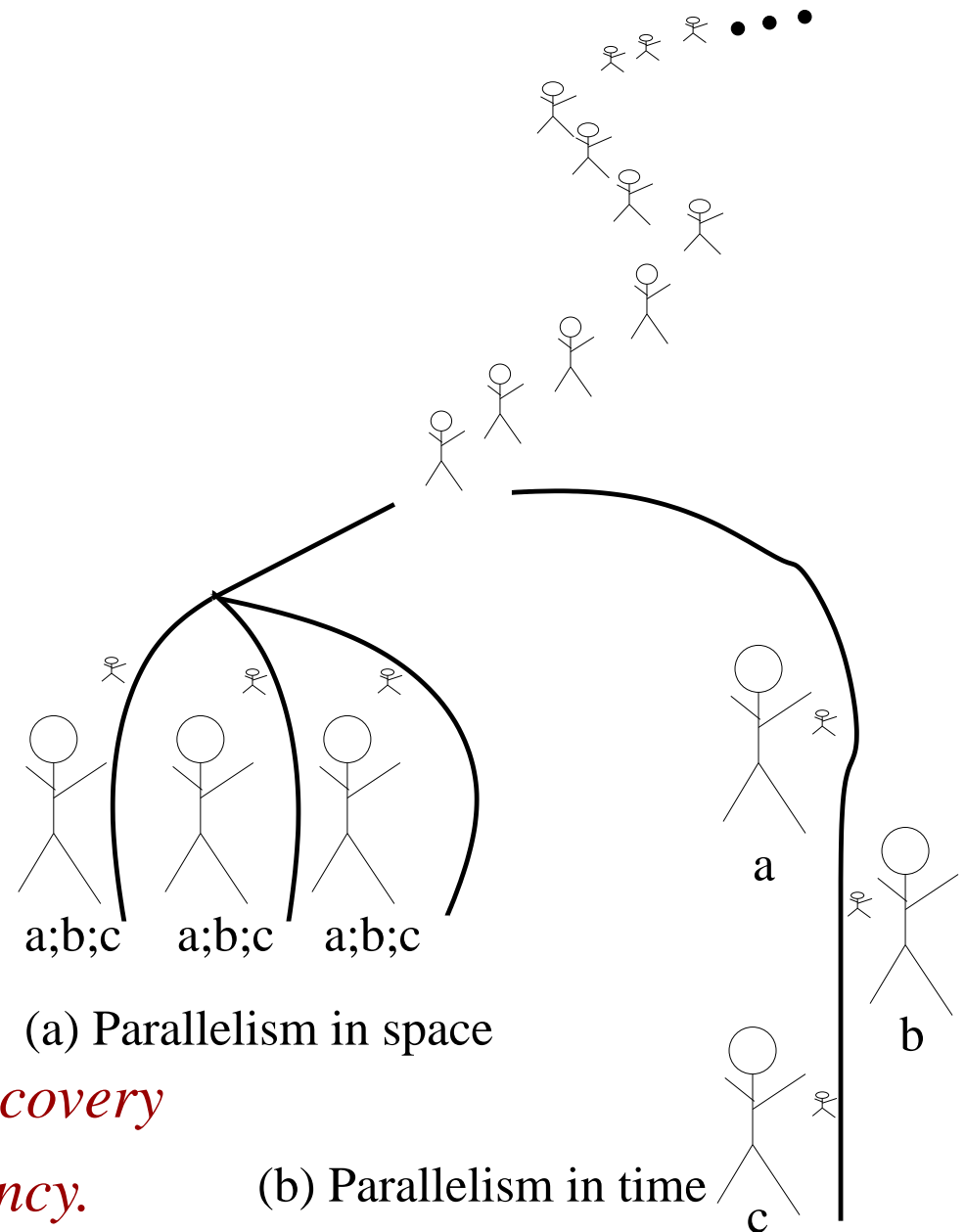
Parallelism in space and in time

One may identify two kinds of parallelism:

- in space* and
- in time*.

In the former case (a), the full sequence of operations $a;b;c;$ is done by a single process, while in the latter case (b) one has specialized processes for each action a , b , and c .

Notice: Actually, (b) reflects Ford's discovery that specialization often increases efficiency.





Applications

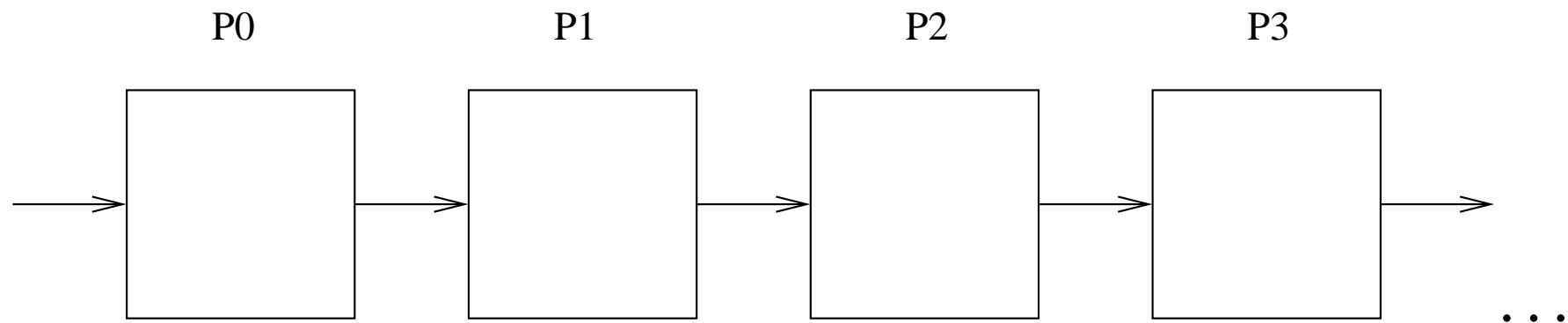
Applications:

- The “*pipeline*” approach was the main source of increasing *processor speed* in the last decade. (Currently, there is a shift to instruction-level parallelism in processor design.)
- It was also a key part in the design of *parallel data-flow architectures* aiming as an alternative to classical Von Neumann architecture.

Pipeline technique

In the *pipeline technique*,

- The problem is divided into *a series of tasks* that have to be completed *one after the other*. [Actually, this is the basis of sequential programming.]
- Then, *each task* will be executed by a *separate process* (or processor).



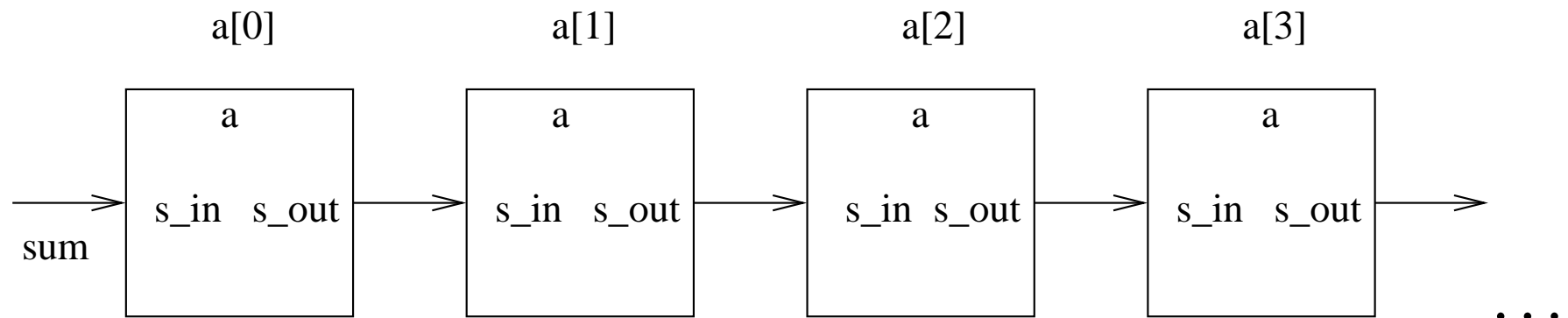
..Pipeline technique

Example 1 (adding numbers): Add all the elements of array a:

```
for (i=0; i<n; i++)  
    sum = sum + a[i];
```

The loop may be *unfolded* to yield

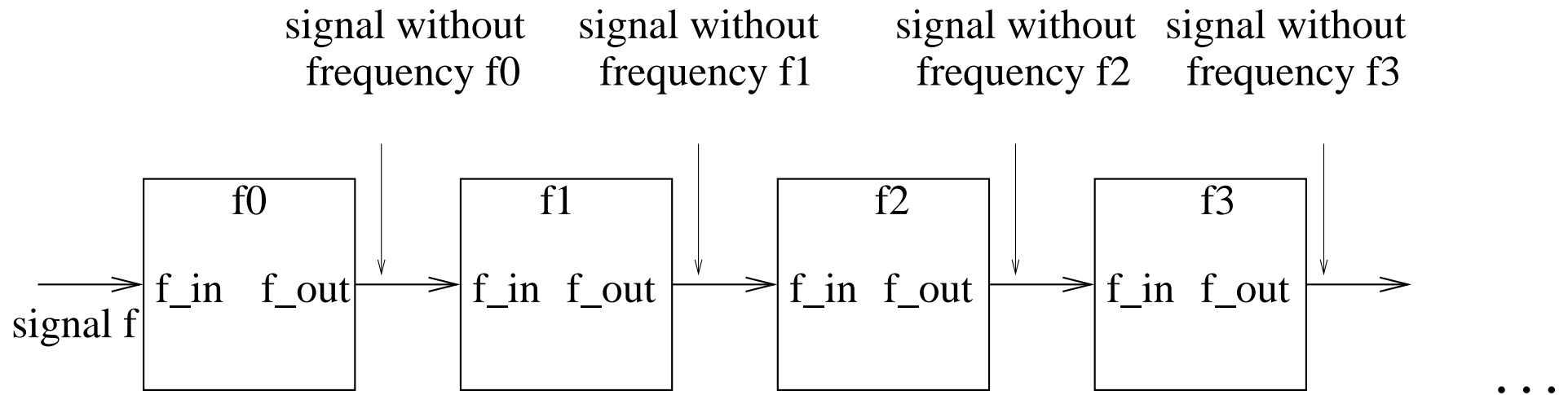
```
sum = sum + a[0];  
sum = sum + a[1];  
sum = sum + a[2];  
sum = sum + a[3];  
⋮
```



..Pipeline technique

Example 2 (frequency filter):

- *Frequency filter*: The objective here is to remove specific frequencies, say $f_0, f_1, f_2, f_3, \dots$ from a given digital signal $f(t)$.
- The pipeline is described in the figure below





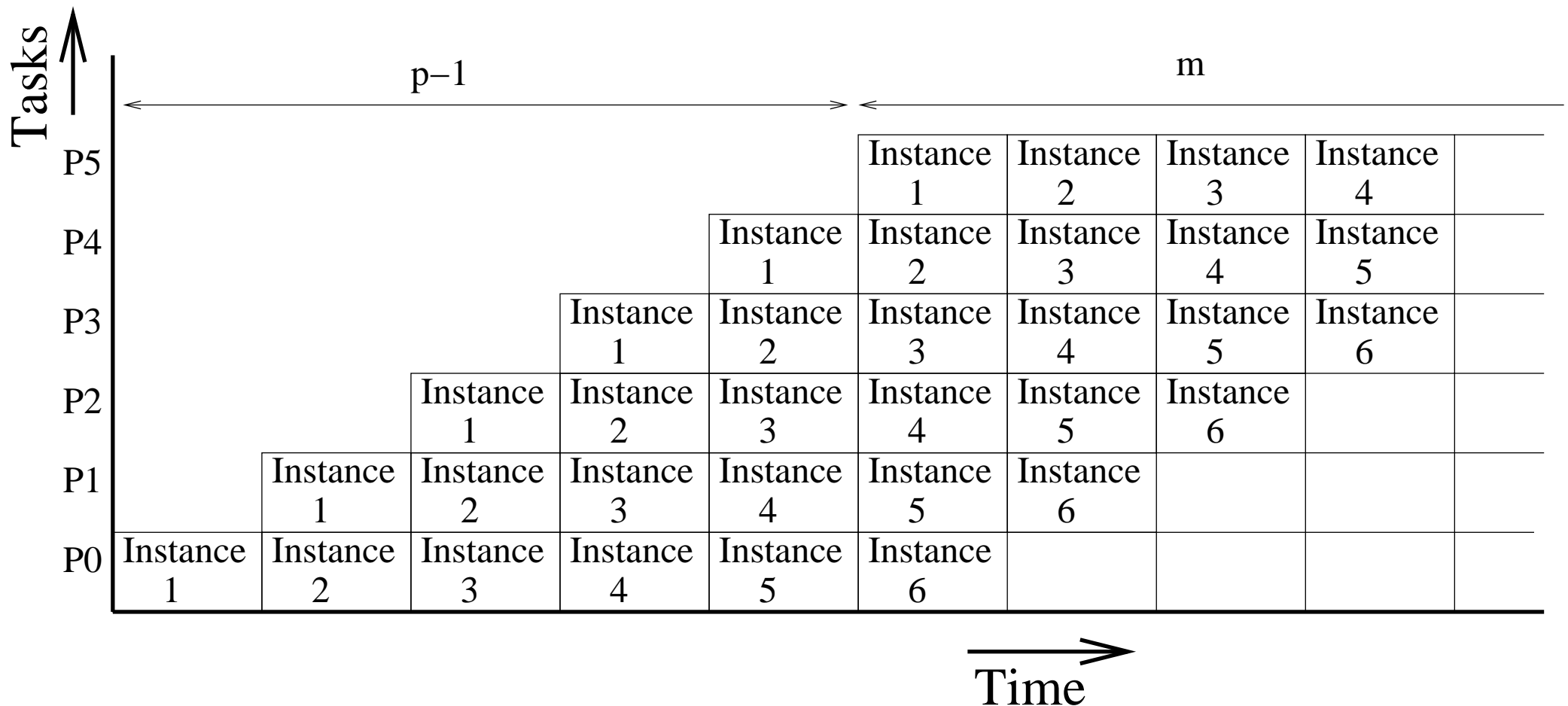
..Pipeline technique

Good for: Given that the problem can be divided into a series of sequential tasks, the pipeline approach can provide *increased speed* under the following three types of computations:

- **Type 1:** If *more than one instance* of the complete problem is to be executed
- **Type 2:** If a *series of data items* must be processed, each requiring multiple operations
- **Type 3:** If information to start a next process can be *passed forward before the current process has completed* all its internal operations.

Space-time diagrams

Type 1: Multiple instances of the same problem.





..Space-time diagrams

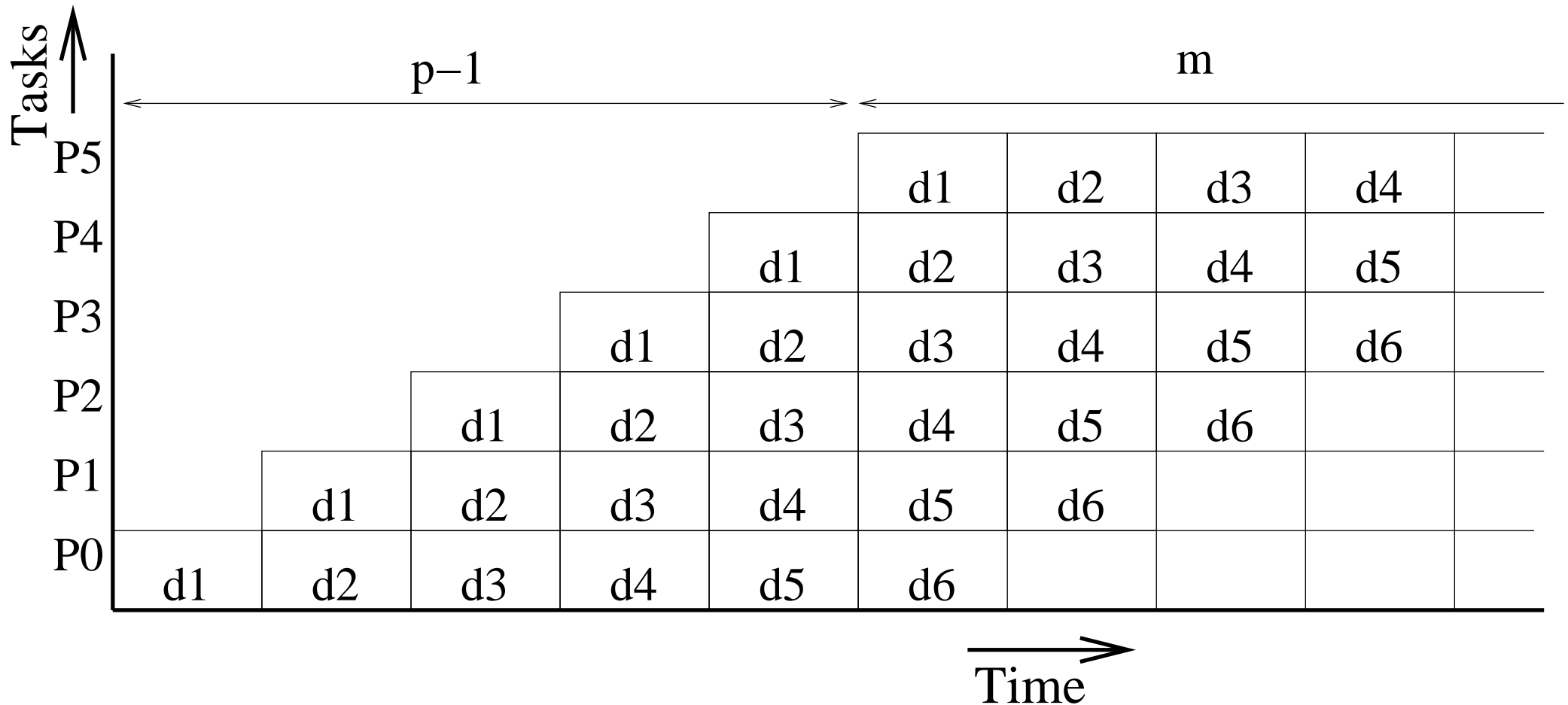
Type 1:

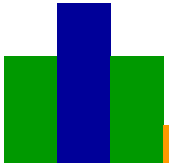
- mostly used in hardware design, particularly for processor design
- a staircase effect at the beginning; after that, one instance of the problem is completed at each pipeline cycle.
- $p - 1$ is the *pipeline latency*

..Space-time diagrams

Type 2: Pipeline structure

d6 d5 d4 d3 d2 d1 d0 \rightarrow P0 \rightarrow P1 \rightarrow P2 \rightarrow P3 \rightarrow P4 \rightarrow P5





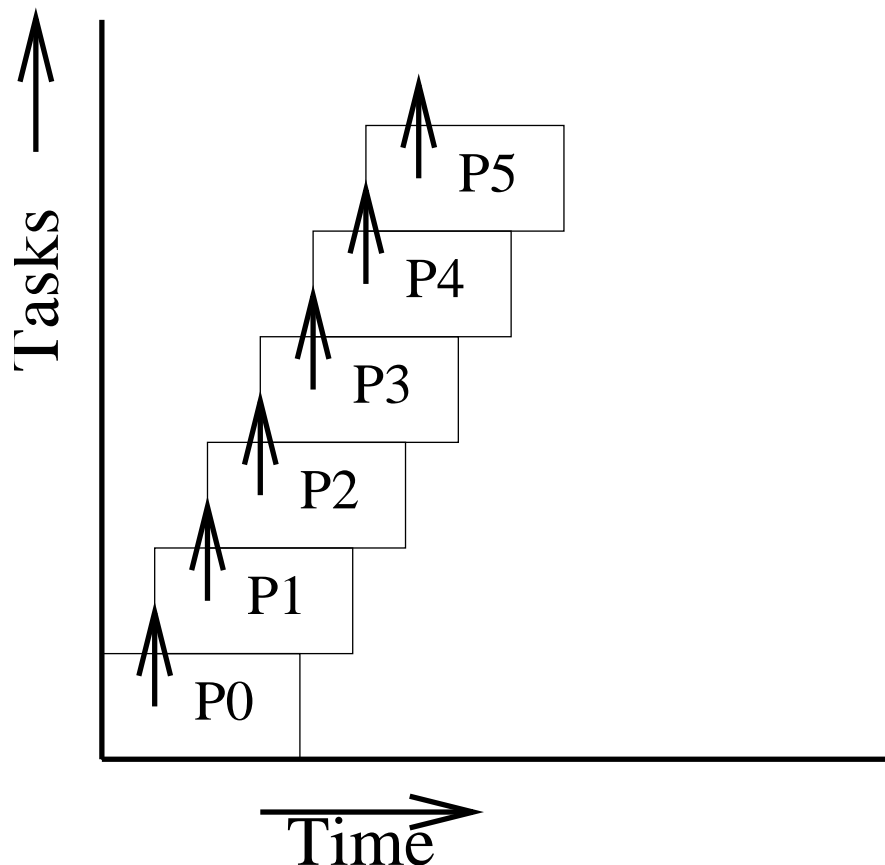
..Space-time diagrams

Type 2:

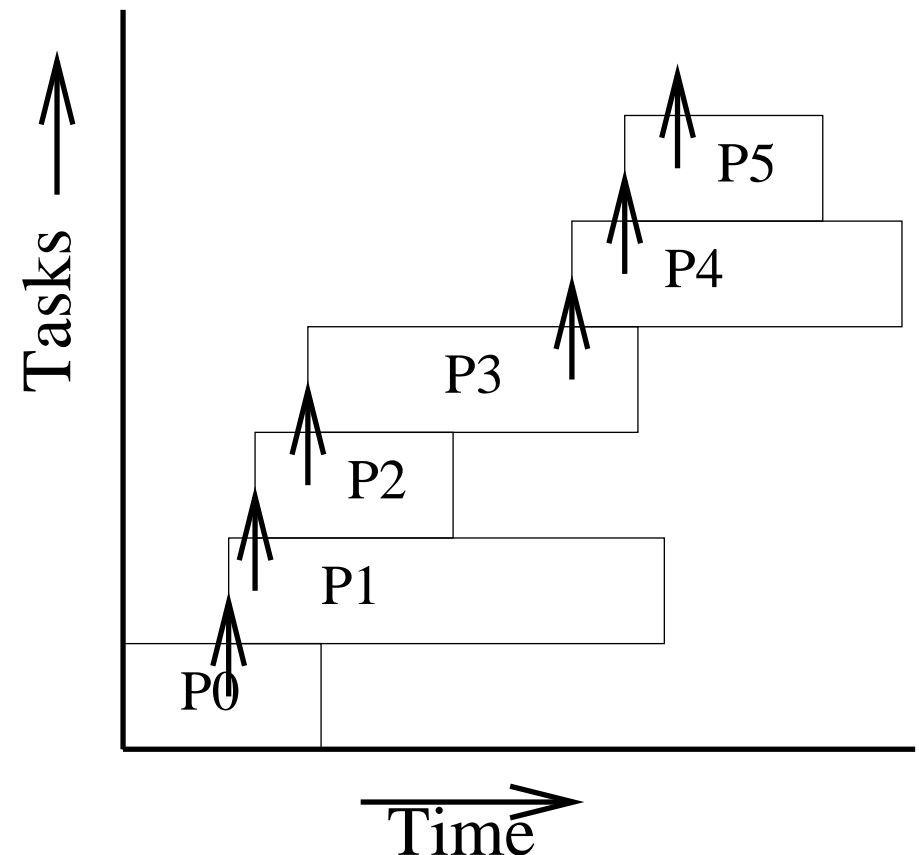
- a single instance of a problem, but with a series of data items to be processed
- examples: multiplications, sortings, etc.

..Space-time diagrams

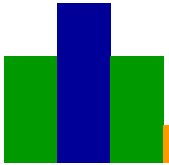
Type 3: Pipeline processing where relevant *information for starting a next stage is passed before the end of the current task.*



(a) Processes with the same execution time



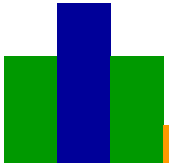
(b) Processes with possible different execution time



..Space-time diagrams

Type 3:

- a single instance of a problem, but now each process can pass on information to the next process before it has completed
- example: triangular systems of linear equations
- generally difficult to analyze

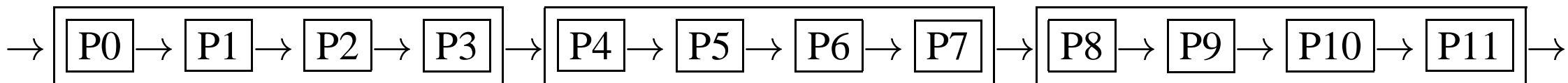


..Space-time diagrams

Stages vs. processes (processors):

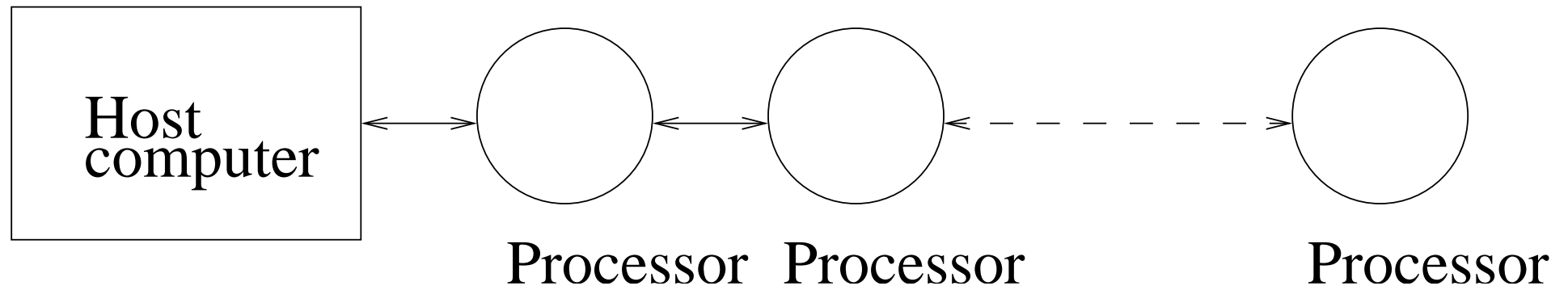
- What we have counted before was the number of *logical stages* in a pipeline solution of a problem.
- If their number is larger than the number of processors, then a *group of stages* can be assigned to each processor.

Example of partitioning:



A computing platform

A possible computing platform for pipeline applications is described below:

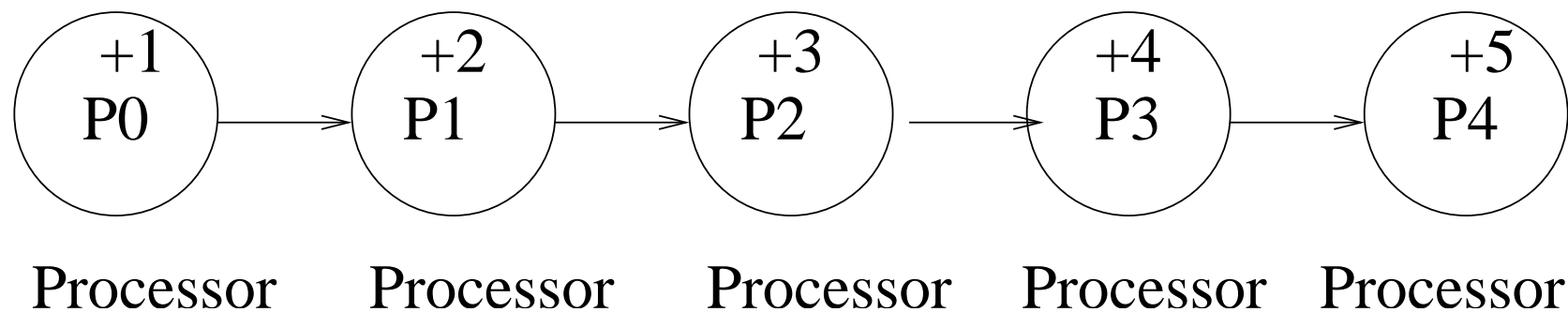


Data-flow architecture:

- Pipeline computation was one of the basic models used to develop *data-flow computation & machines*.
- Examples of programming languages based on this model are: *Lucid, Lustre, Signal*, etc.

Simple example

Adding numbers: A first, simple example is for adding some numbers (say, $\sum_1^k i$) using pipeline technique. The task is to add such numbers, each process holding a number. An illustration is below:





..example

Suppose we have n numbers/processes. The code is as follows:

- Code for process $P_i (0 < i < n)$:

```
recv( &sum,  $P_{i-1}$  ) ;
```

```
sum = sum + number ;
```

```
send( &sum,  $P_{i+1}$  ) ;
```

- Code for process P_0 :

```
sum = number ;
```

```
send( &sum,  $P_1$  ) ;
```

- Code for process P_n :

```
recv( &sum,  $P_{n-1}$  ) ;
```

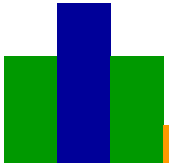
```
sum = sum + number ;
```




..example

One may write a SPMD program as follows

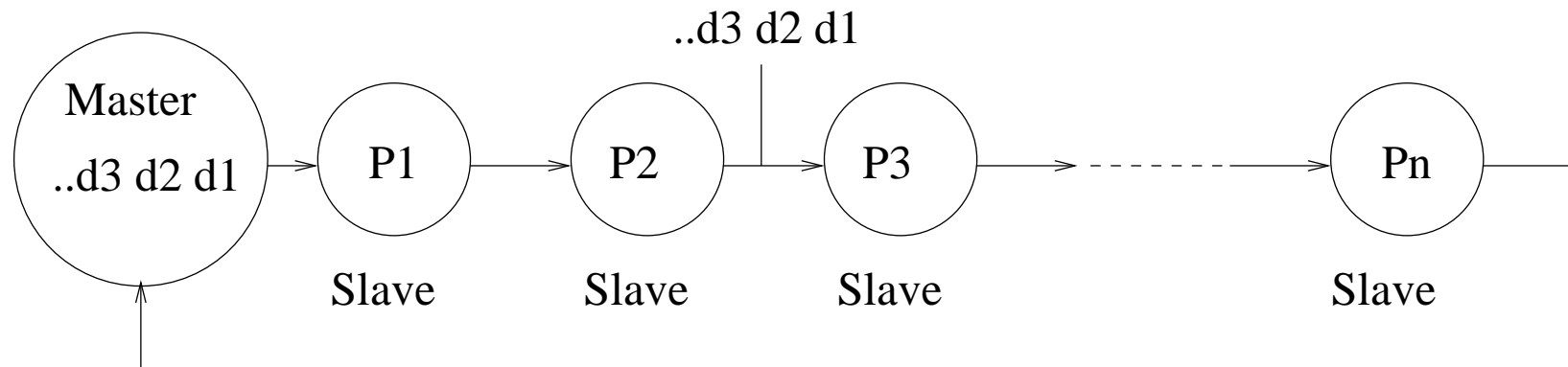
```
if (pid > 0) {  
    recv(&sum,  $P_{i-1}$ ) ;  
    sum = sum + number ;  
}  
if (pid < n-1) {  
    send(&sum,  $P_{i+1}$ ) ;  
}
```



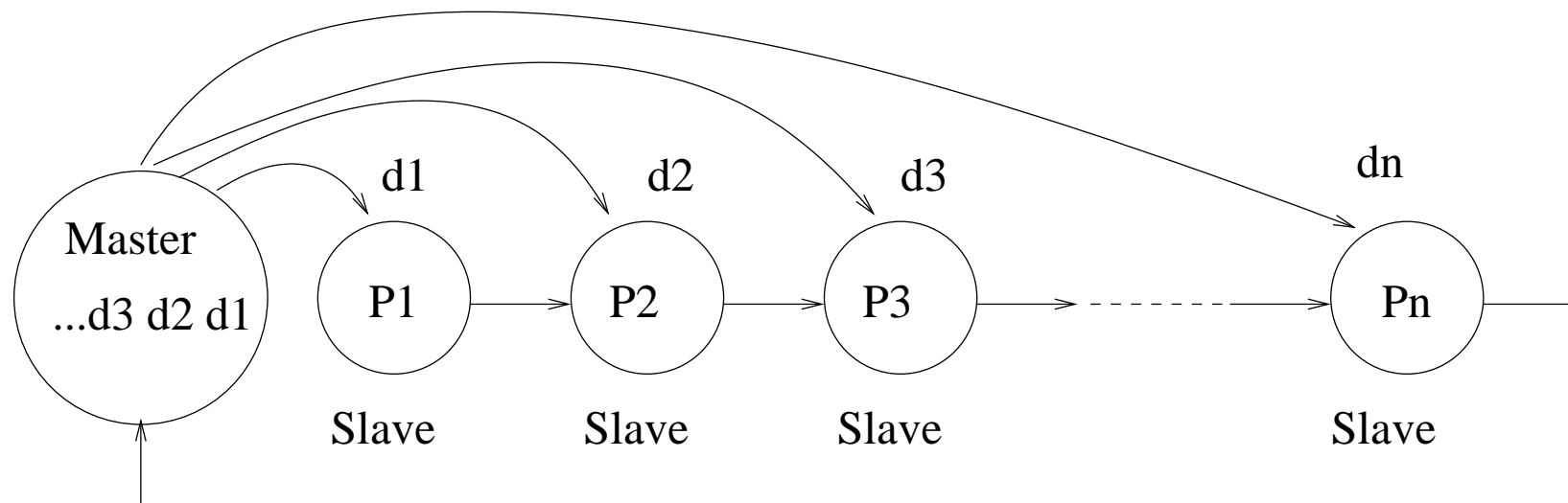
Concrete architectures

Concrete architectures:

- master process + ring configuration



- master process + direct access to slave processes





Case studies 1: Type 1, adding numbers

Analysis, adding numbers, Type 1: Suppose each process performs similar actions in each pipeline cycle. Then

- *(total time) = (time for one pipeline cycle) \times (number of cycles)*

If there are m instances and p pipeline stages, then the number of cycles is $m + p - 1$, hence

$$t_{total} = (t_{comp} + t_{comm}) \times (m + p - 1)$$

- *(average time) = (total time) / (number of instances solved)*

With the above notations,

$$t_a = \frac{t_{total}}{m}$$



..Type 1, adding numbers

Single instance of problem:

Suppose we have to add *one set ($m = 1$) of p data*. Then

$$t_{comp} = 1 \text{ (one addition)}$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

(left+right communications; only sum is to be send)

$$t_{total} = [2(t_{startup} + t_{data}) + 1]p.$$

Notice: Here (and in the following 2 slides) we suppose processes hold the numbers, hence only sum is communicated. If also numbers are to be sent, then one has to count the time to send these numbers. The result depends on the architecture - see Slide 16.



..Type 1, adding numbers

Multiple instances of problem: Suppose we have to add m ($m > 1$) *sets of p data*. Then

$$t_{comp} = 1 \text{ (one addition)}$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

(left+right communications; only sum is to be send)

$$t_{total} = [2(t_{startup} + t_{data}) + 1](m + p - 1)$$

hence the average time is

$$t_a = \frac{t_{total}}{m} = [2(t_{startup} + t_{data}) + 1]\left(1 + \frac{p-1}{m}\right)$$

When m is large this is approximatively one pipeline cycle, i.e., $2(t_{startup} + t_{data}) + 1$. In other words, after a warming-up period the pipeline solves one instance per pipeline cycle.



..Type 1, adding numbers

Data partitioning with multiple instances of problem: Suppose we have to add m ($m > 1$) sets of n data, using p processes, hence each process handle n/p data in a pipeline cycle. Then

$$t_{comp} = n/p \text{ (additions)}$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

(left+right communication; only sum is to be send)

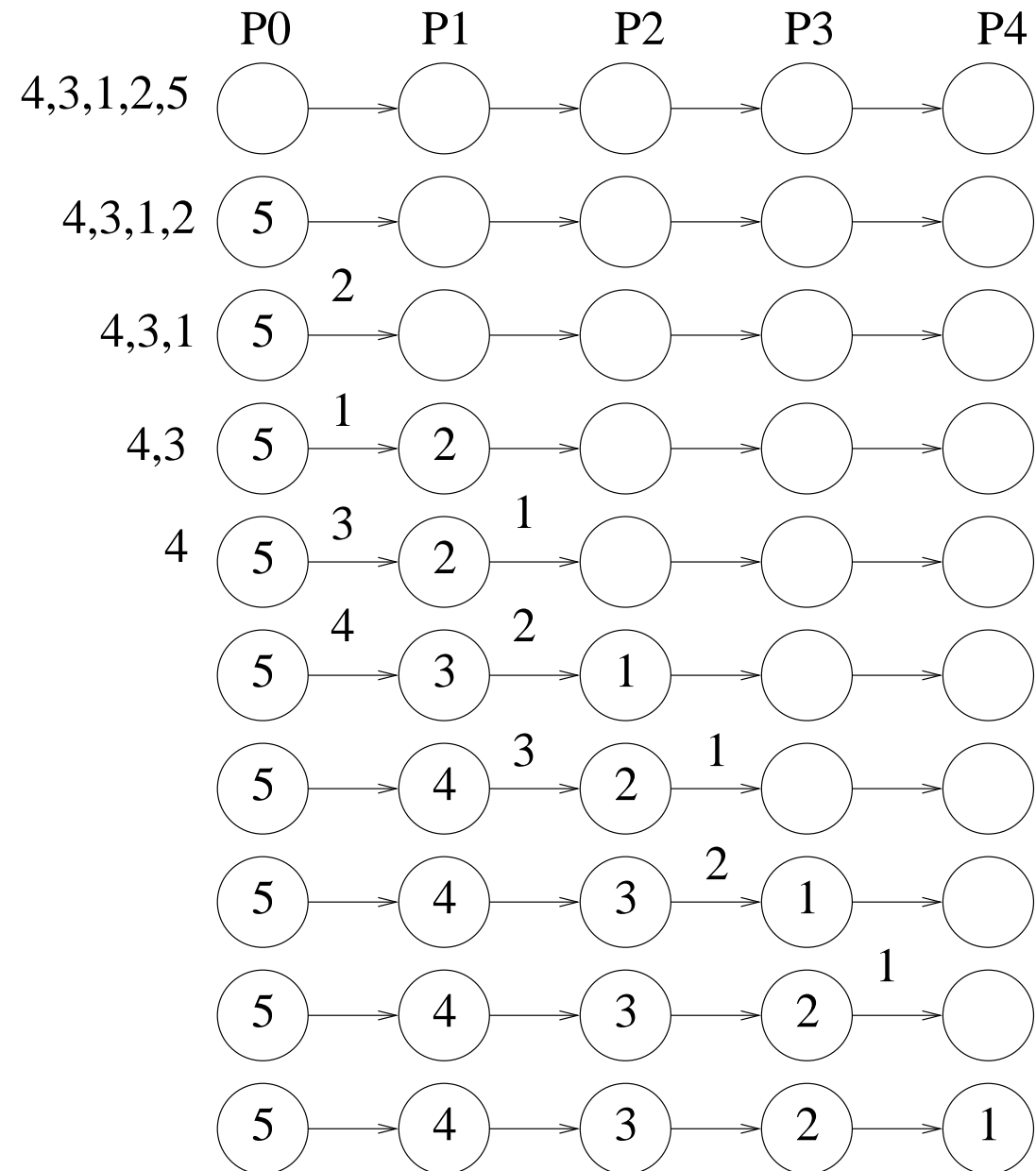
$$t_{total} = [2(t_{startup} + t_{data}) + n/p](m + p - 1)$$

Notice: By increasing the length of data partition n/p the impact of communication is diminished. Increasing the length too much will decrease the parallelism and the execution time may increase.

Case studies 2a: Type 2, sorting

Sorting numbers, Type 2:

A parallel version of the *insertion sort*. ((1) The basic step is to insert a new number in a previously ordered sublist. (2) An extra step requires to shift numbers, if necessary. (3) The final algorithm is obtained allowing such sequences of operations to interfere in a pipeline style.)





..Type 2, sorting

The basic algorithm for process P_i is:

```
recv( &number,  $P_{i-1}$  ) ;  
if ( number > x ) {  
    send( &x,  $P_{i+1}$  ) ;  
    x = number ;  
} else send( &number,  $P_{i+1}$  )
```

Notice: If the number of numbers (say, n) and the process number (say i) are known, then the above basic step is repeated $n - i + 1$ times.



..Type 2, sorting

The final code for process $P_i (i > 0)$ is (provided an extra requirement is to have all sorted numbers hold by master process P_0):

```
rightProcNo = n-i-1;
recv(&number,  $P_{i-1}$ );
for (j=0; j<rightProcNo; j++) {
    recv(&number,  $P_{i-1}$ );
    if (number > x) {
        send(&x,  $P_{i+1}$ );
        x = number;
    } else send(&number,  $P_{i+1}$ )
}
send(& number,  $P_{i-1}$ );
for (j=0; j<rightProcNo; j++) {
    recv(&number,  $P_{i+1}$ );
    send(&number,  $P_{i-1}$ )
}
```



..Type 2, sorting

Analysis:

- *Sequential*:

$$t_s = (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n - 1)}{2}$$

(poor sequential sorting algorithm; good only for very small n)

- *Parallel*: (for n numbers there are $2n - 1$ pipeline circles)

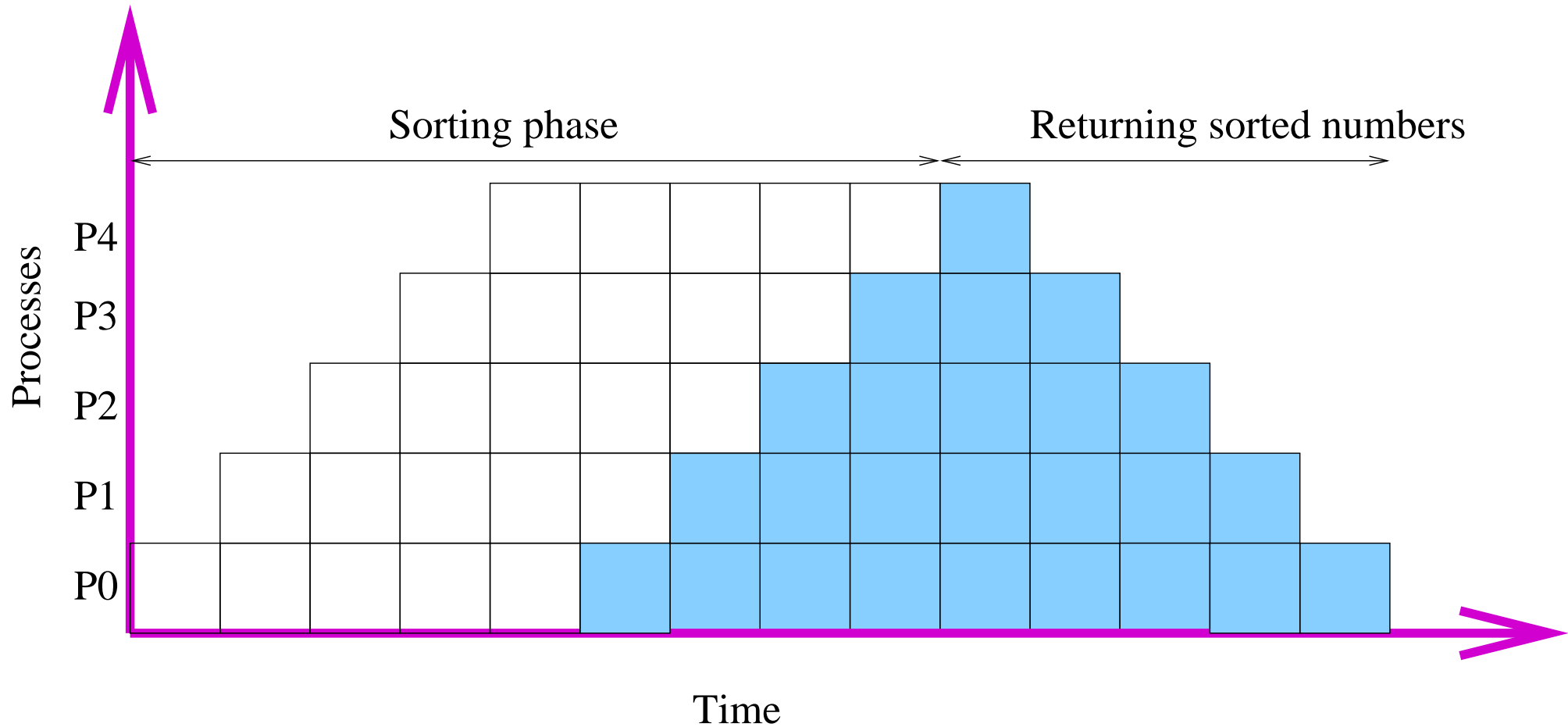
$t_{comp} \leq 2$ (a comparison in `if` and sometimes an update of `x`)

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (t_{comp} + t_{comm})(2n - 1) \leq 2(1 + t_{startup} + t_{data})(2n - 1)$$

..Type 2, sorting

Insertion sort with sorted numbers returned ($n = 5$)



("sorting + returning" require $3n - 1$ cycles)



Case studies 2b: Type 2, prime numbers

Generate prime numbers using the Sieve of Eratosthenes.

Suppose we want to find the prime numbers from 2 to 20.

We start with all numbers

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

1st number 2 is prime; strike out the number and all its multiples

~~2~~, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~

1st (non-marked) number 3 is prime; strike out the number and all its multiples

~~2~~, ~~3~~, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~

... and so on.

To find all prime numbers up to n requires to repeat the marking procedure starting with (prime) numbers up to \sqrt{n} . (Each composite number up to n has at least one factor less than \sqrt{n} .)



..Type 2, prime numbers

Sequential code:

```
for (i=2; i<n; i++)
    prime[i] = 1;
for (i=2; i<=sqrt(n); i++){
    if (prime[i] == 1){
        for (j=i+i; i<n; j=j+i)
            prime[j] = 0;
    }
}
```

The program use an array prime such that finally prime[i] is 1 if i is prime number, otherwise 0.



..Type 2, prime numbers

Sequential time:

- The number of striking out steps depend on the prime number: $\lfloor n/2 - 1 \rfloor$ for 2, $\lfloor n/3 - 1 \rfloor$ for 3, and so on.
- Hence:

$$t_s = \lfloor n/2 - 1 \rfloor + \lfloor n/3 - 1 \rfloor + \lfloor n/5 - 1 \rfloor + \dots + \lfloor n/p_k - 1 \rfloor$$

where p_k is the greatest prime number $\leq \sqrt{n}$.

- Roughly, the growing of t_s is less than $\sqrt{n}n$ which is $O(n^{1.5})$.



..Type 2, prime numbers

Parallel code:

- A partitioning procedure may be quite inefficient.
- A pipeline solution acts as follows:
 - each process *keeps the 1st* received number, say p , as a prime number and
 - *passes forward* those received numbers which are *not multiple of p* .
- This procedure is more efficient as it avoids to multiple strike out the composite numbers for all their prime factors.



..Type 2, prime numbers

The basic piece of (pseudo)code for P_i may be

```
recv(&x,  $P_{i-1}$ ) ;  
for (i=0; i<n; i++) {  
    recv(&number,  $P_{i-1}$ ) ;  
    if (number == terminator) break ;  
    if ((number % x) != 0) send (&number  $P_{i+1}$ ) ;  
}
```

We need a special “terminator” message, as the number of iteration steps is not a-priori known. The mod operator “%” is usually expensive and should be avoided.



Case studies 3: Type 3, systems of equations

Upper-triangular systems of linear equations:

These systems have the following shape:

$$\begin{array}{ccccccc} a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} & = & b_{n-1} \\ \vdots & & \\ a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & = & b_2 \\ a_{1,0}x_0 + a_{1,1}x_1 & = & b_1 \\ a_{0,0}x_0 & = & b_0 \end{array}$$

where a 's and b 's are constants and x 's are unknown to be found.



..Type 3, systems

They are easily solved by backwards substitution method:

- compute x_0 from the last equation

$$x_0 = \frac{b_0}{a_{0,0}}$$

- substitute the value of x_0 into the next equation to obtain x_1 , i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

- substitute the values of x_0, x_1 into the next equation to obtain x_2 , i.e.,

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

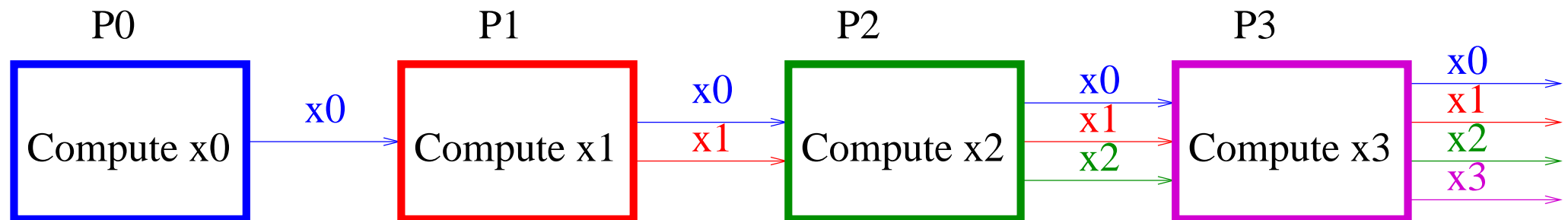
- ... and so on, till all unknowns are found.

..Type 3, systems

The general formula for solving x_i is:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

A parallel *type 3, pipeline solution* comes naturally here: once x_i is obtained, it may be passed to all other upper processes to use it.





..Type 3, systems

The pseudocode of P_i ($0 < i < n$) for a pipeline version is

```
sum = 0;
for (j=0; j<i; j++){
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
    sum = sum + a[i][j]*x[j];
}
x[i] = (b[i]-sum)/a[i][i];
send(&x[i], Pi+1);
```

(The code of P_0 consists of the last 2 statements, only. The code of P_{n-1} consists of the above statements, except for the last send.)



..Type 3, systems

The analysis is quite difficult, as (1) one cannot assume the computational effort at each pipeline stage is the same and (2) the pipeline stages overlap. A rough analysis is as follows:

- Process P_0 performs one computation (division) and one send.
- The i -th process ($0 < i < n - 1$) performs i sets of send / recv / multiply / addition, operations and finally one subtract / division / send set.
- Process P_{n-1} does similar operations as above, except for the last send.
- Finally, one has to consider the overlapping of the tasks/stages and to appropriately count the total execution time.

Lesson 6: Synchronous computations

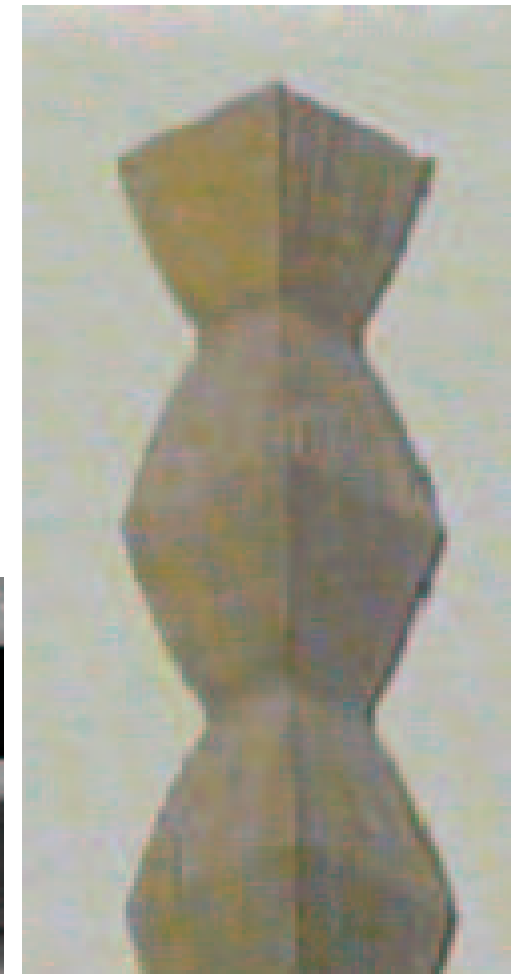
G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004

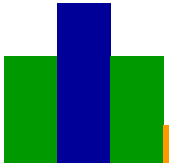
Synchronous computations

Synchronous computations:

- *all processes are repeatedly synchronized at regular points*
- a very important class of problems (e.g., 70% of cases in a Caltech set of parallel programming applications)



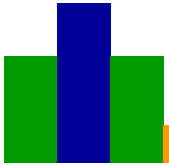
Brancusi: The endless column



Barrier

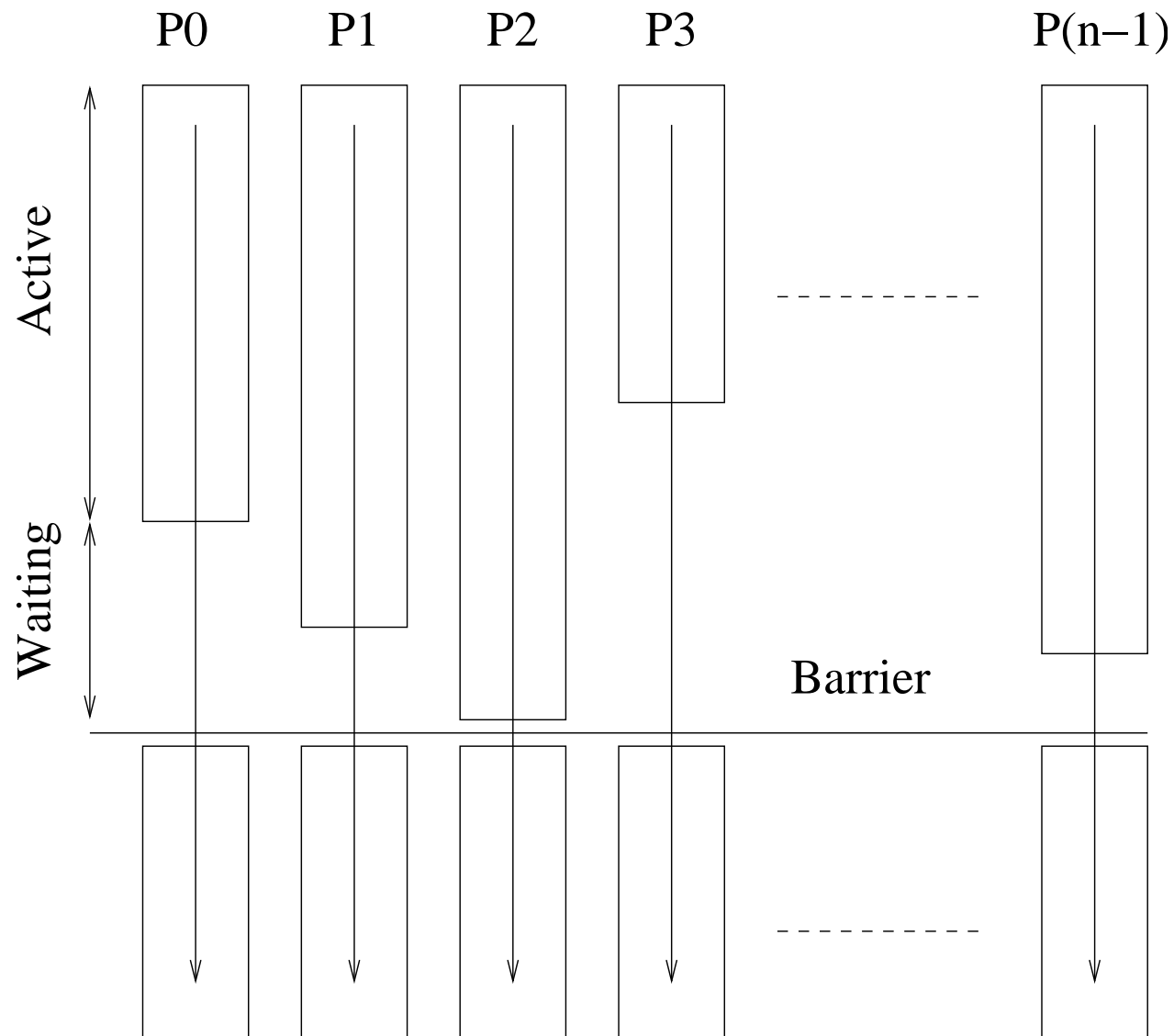
Barrier:

- a barrier is a *basic mechanism* for synchronizing processes
- the barrier statement has to be inserted in each process at the point *where it has to wait* for synchronization
- a processes can *continue* from the synchronization point *when all processes have reached the barrier* (or, in a partially synchronous case, a stated number of processes have reached the barrier).



..Barrier

Processes reaching the barrier at different times:





MPI/PVM barrier routines

In message passing systems, barriers are often provided as library routines

- MPI: `MPI_Barrier()`
 - it is a barrier with only one parameter, namely the communicator group name
 - it has to be called by each process in the group, blocking processes until all members of the group have reached the barrier
- PVM: `pvm_barrier`
 - it is similar
 - PVM has the unusual feature of specifying the number of processes that must reach the barrier in order to release the processes

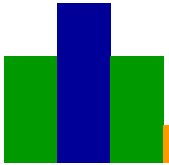


Implementation

Centralized counter implementations (or *linear barrier*): Such counter-based barriers often have two phases:

- a process enter an **arrival phase** and does not leave this phase until all processes have arrived in this phase
- then the process move to a **departure phase** and it is released

Good implementation of a barrier must take into account that a barrier might be used more than once in a process. So, it might be possible for a process to enter the barrier for a second time before previous processes have left the barrier for the first time. The two-phase above handles this scenario.



..Implementation

Example of implementation:

Master:

```
for (i=0; i<n; i++)  
    recv( $P_{any}$ ) ;  
for (i=0; i<n; i++)  
    send( $P_i$ ) ;
```

Slave:

```
send( $P_{master}$ ) ;  
recv( $P_{master}$ ) ;
```



Tree implementation

Tree implementation - it is *more efficient*; suppose there are eight processes P0,P1,..., P7:

—1st stage:

P1 sends message to P0 (when P1 has reached its barrier)

P3 sends message to P2 (when P3 has reached its barrier)

P5 sends message to P4 (when P5 has reached its barrier)

P7 sends message to P6 (when P7 has reached its barrier)

—2nd stage:

P2 sends message to P0 (P1,P2,P3 have reached their barrier)

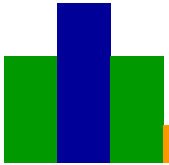
P6 sends message to P4 (P5,P6,P7 have reached their barrier)

—3rd stage:

P4 sends message to P0 (P1,P2,P3,P4,P5,P6,P7 have arrived)

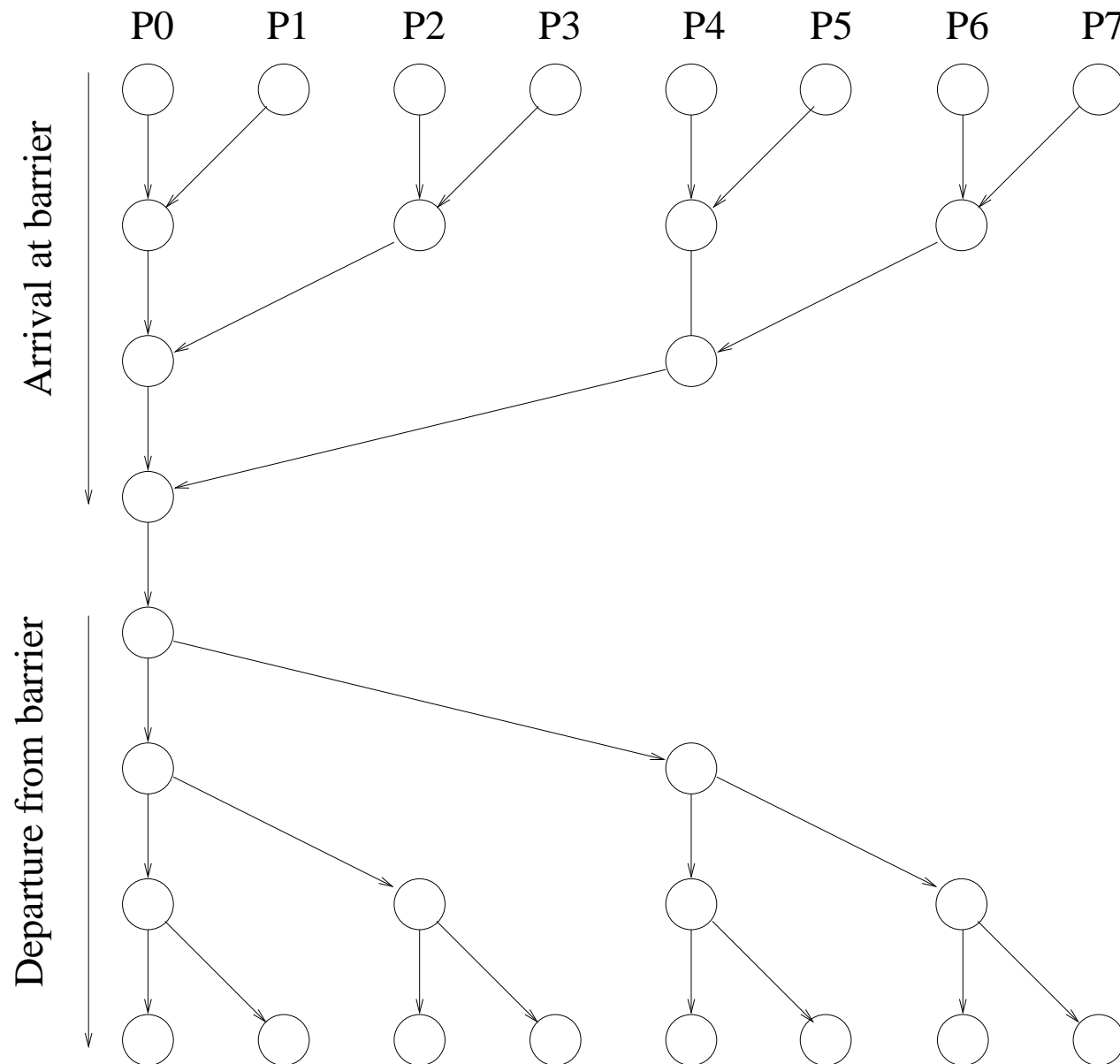
—Final stage:

P0 terminates the arrival phase (when P0 has reached its barrier and it has received message from P4)



..Tree implementation

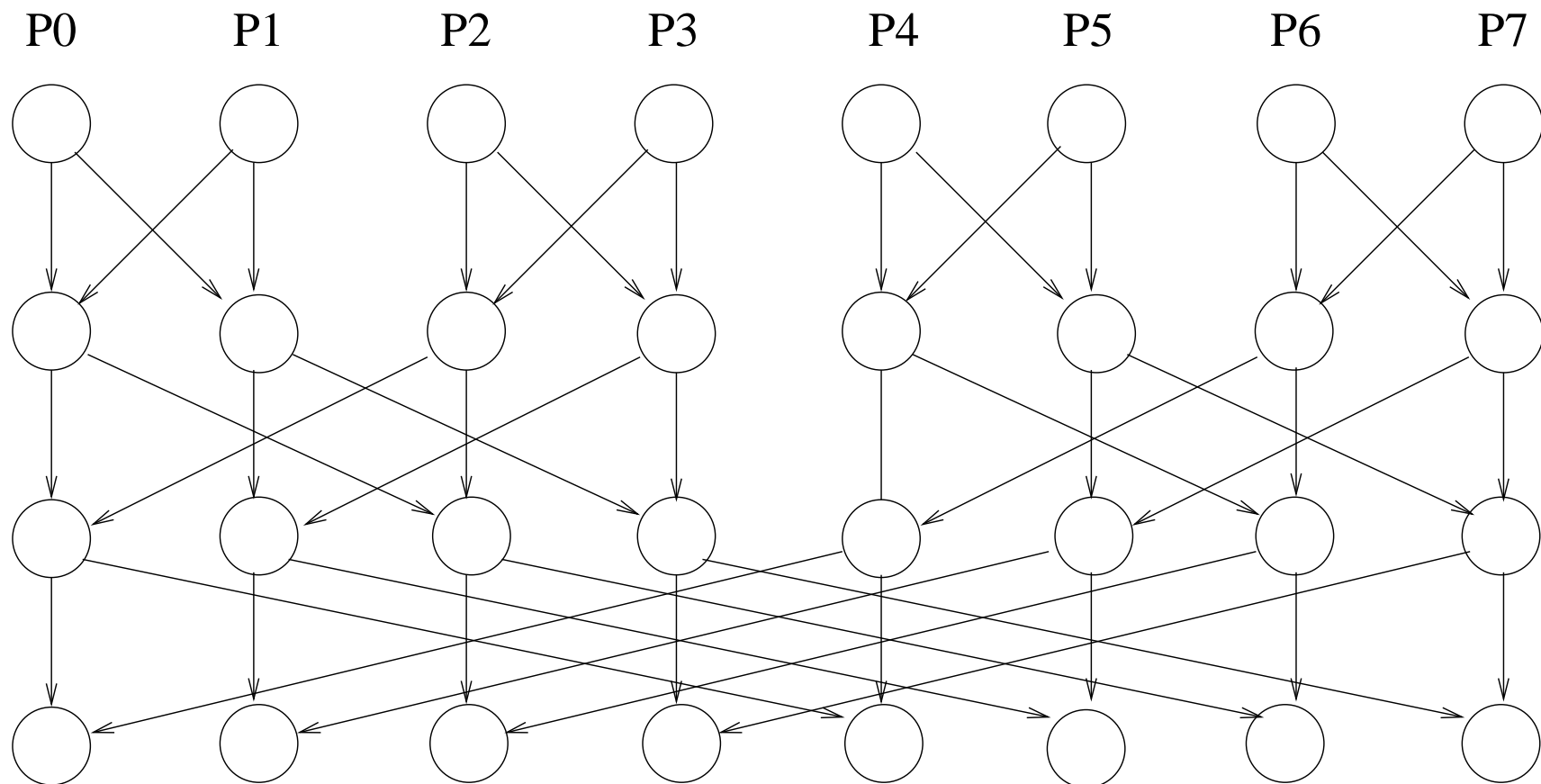
Tree barrier:



Butterfly barrier

Butterfly barrier:

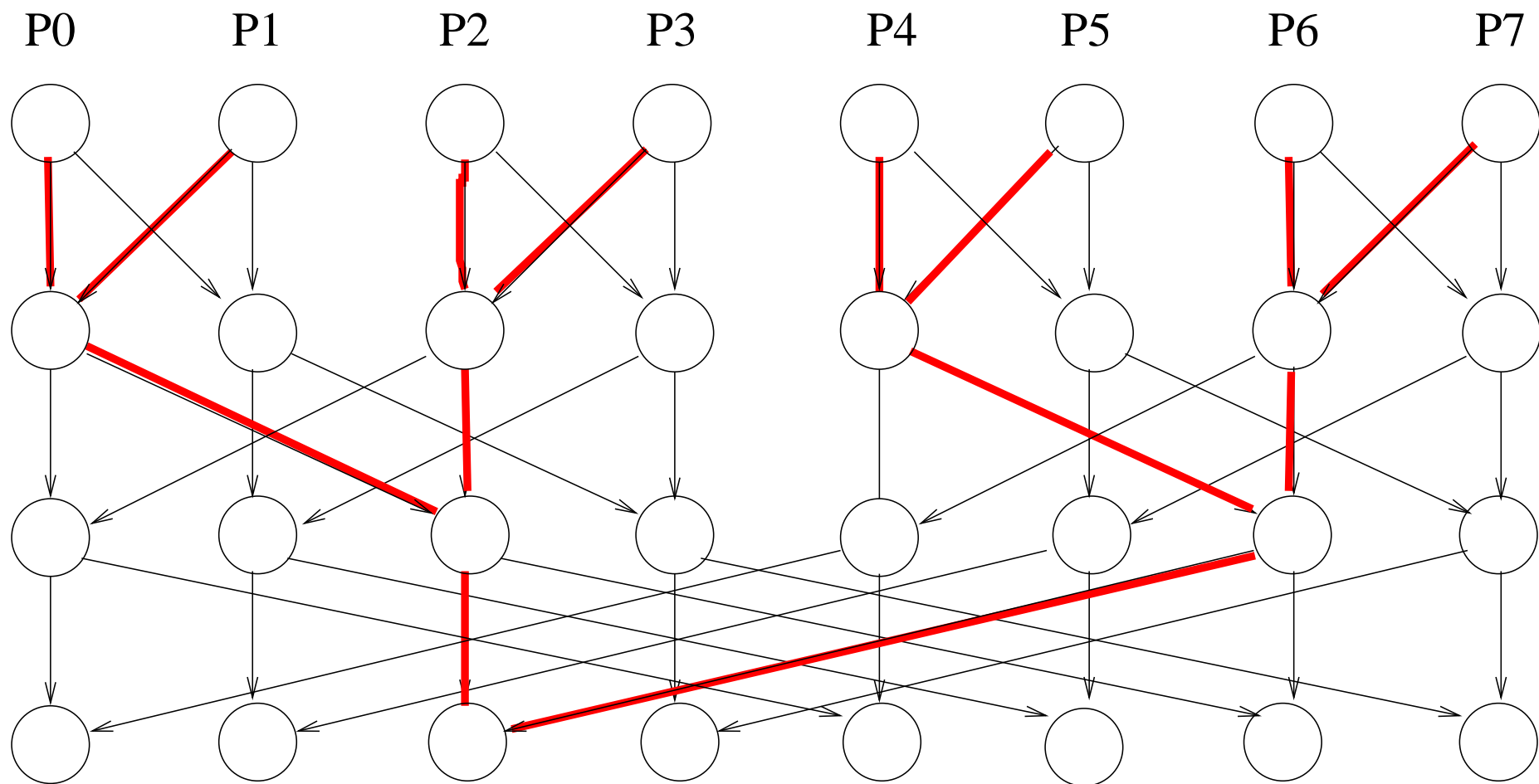
- 1st stage: $P0 \longleftrightarrow P1, P2 \longleftrightarrow P3, P4 \longleftrightarrow P5, P6 \longleftrightarrow P7$
- 2nd stage: $P0 \longleftrightarrow P2, P1 \longleftrightarrow P3, P4 \longleftrightarrow P6, P5 \longleftrightarrow P7$
- 3rd stage: $P0 \longleftrightarrow P4, P1 \longleftrightarrow P5, P2 \longleftrightarrow P6, P3 \longleftrightarrow P7$



..Butterfly barrier

Useful if *data are exchanged* at the barrier.

E.g., observe how P2 gathers data from all processes:



Gather data from all processes



Partial barriers

Local synchronization: Suppose a process P_i needs to be synchronized and to exchange data with processes P_{i-1} and P_{i+1} before continuing.

Process P_{i-1}	Process P_i	Process P_{i+1}
$\text{recv}(P_i) ;$	$\text{send}(P_{i-1}) ;$	$\text{recv}(P_i) ;$
$\text{send}(P_i) ;$	$\text{send}(P_{i+1}) ;$	$\text{send}(P_i) ;$
	$\text{recv}(P_{i-1}) ;$	
	$\text{recv}(P_{i+1}) ;$	

Notice that this is not a perfect three-process barrier because, for instance, process P_{i-1} will only be synchronized with P_i and continue as soon as P_i allows. Similarly for P_{i+1} and P_i .



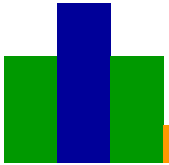
Deadlock

Deadlock: When a pair of processes send and receive from each other, deadlock may occur.

- Deadlock will occur if *both processes first perform the send, using synchronous routines* (or blocking routines without sufficient buffering).
- This is because neither will return; they will wait for matching receives that are never reached.

A solution: Arrange for *one* process *first to receive* and then to send and for the *other* process *first to send* and then to receive.

Example: Linear pipeline deadlock can be avoided by arranging so that *even*-numbered processes *first* perform their *sends* and the *odd*-numbered processes *first* perform their *receives*.



..Deadlock

A different solution is to use *combined deadlock-free blocking* routines, e.g., `sendrecv()` routines.

In MPI there are two versions:

- `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`
(the former uses 2 buffers, while the latter uses only one buffer)



Synchronized computations

Data parallel computations:

- *Some operation* is to be performed on *different data* elements *simultaneously* (in parallel)
- This is particularly convenient because:
 - it is *easy to write programs*
 - it easily *scales* to larger problem sizes
 - *many numeric and some non-numeric problems* can be casted into such a data parallel format

Example: Add the same constant to each element of an array:

```
for (i=0; i<n; i++)  
    a[i] = a[i] + k;
```

The statement $a[i] = a[i] + k$ may be executed simultaneously by multiple processors using different indices i



Forall

Forall construct: There are special “parallel” constructs in parallel programming languages to specify such a data parallel operation.

E.g.,

```
forall (i=0; i<n; i++) {  
    body;  
}
```

states that n instances of the body can be executed simultaneously, one for each value of i in the given range.

The term “forall” is unfortunate here, as there are not iterations, each instance performing one execution of the body, only. The previous example may be written as

```
forall (i=0; i<n; i++)  
    a[i] = a[i] + k;
```



Prefix sum problem

Prefix sum problem:

- Given a list of numbers x_0, \dots, x_{n-1} , compute *all the partial summations* (i.e., $x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots$).
- It may use *other associative operations* rather than addition.
- It was *wildly studied*, having practical applications in areas such as: *process allocation, data compaction, sorting, polynomial evaluation, etc.*



..Prefix sum problem

A sequential code may be:

```
for (i=0; i<n; i++){  
    sum[i] = 0;  
    for (j=0; j <= i; j++){  
        sum[i] = sum[i] + x[j]  
    }  
}
```

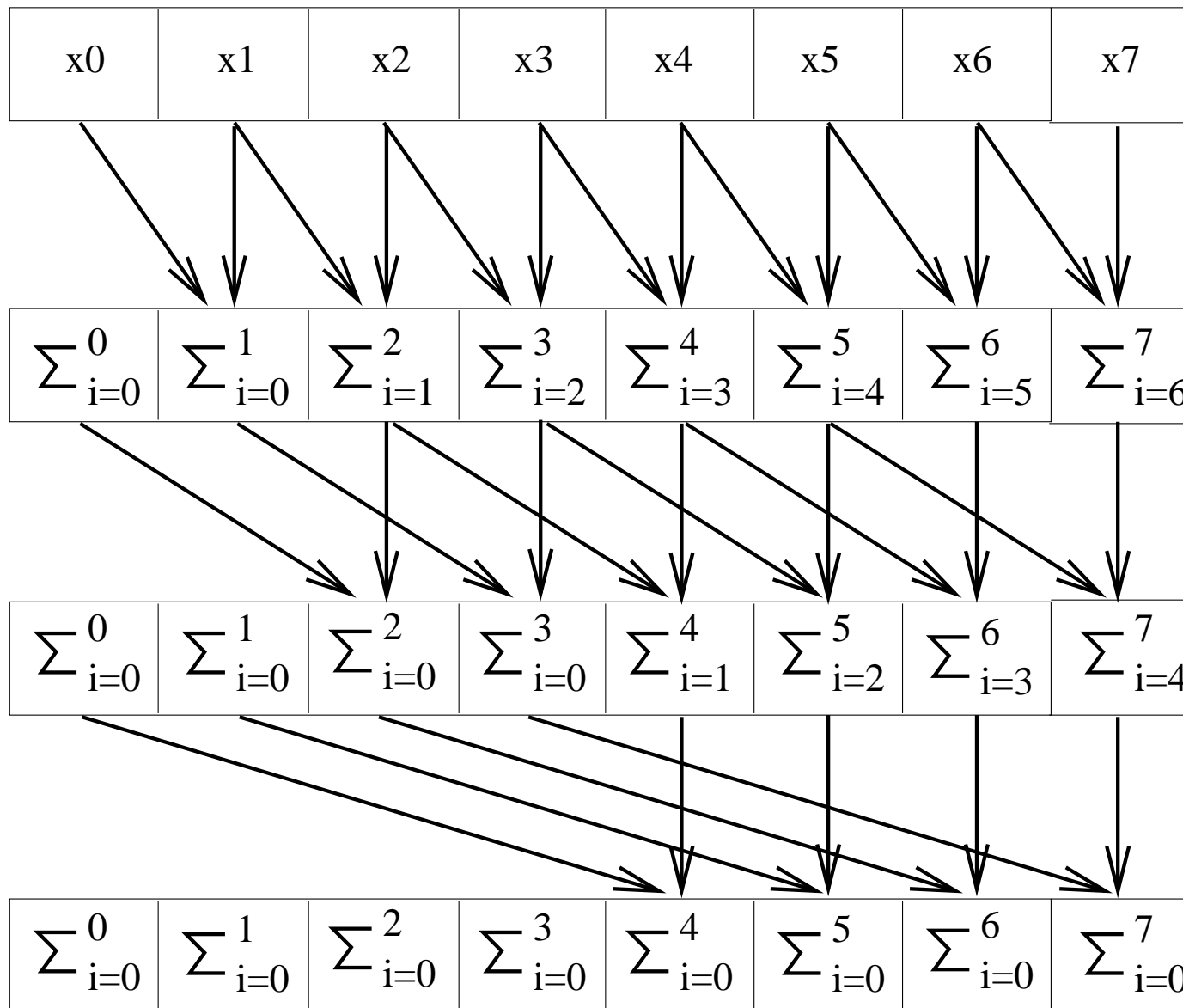
(this is an $O(n^2)$ algorithm)

Parallel code

```
for (j=0; j < log(n); j++){  
    forall (i=0; i<n; i++){  
        if (i >= 2j) x[i] = x[i] + x[i-2j]    }  
}
```

..Prefix sum problem

A concrete example, namely for 8 numbers:





Synchronous iteration

Synchronous iteration: this term is used to describe a situation where a problem is *solved by iteration* and

- *each iteration step* is composed of *several processes that start together* at the beginning of the iteration step and
- *next iteration step* cannot begin until *all processes have finished* the current iteration step.

Example:

```
for (j=0; j<n; j++) {  
    i = myrank;  
    body(i);  
    barrier(mygroup);  
}
```



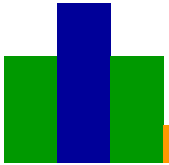
Case studies: 1. Systems of linear equations

Systems of linear equations:

A (general) systems of linear equations looks like follows:

$$\begin{array}{ccccccc} a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & a_{n-1,2}x_2 & + \dots + & a_{n-1,n-1}x_{n-1} & = & b_{n-1} \\ \vdots & & & & & & & & \\ a_{2,0}x_0 & + & a_{2,1}x_1 & + & a_{2,2}x_2 & + \dots + & a_{2,n-1}x_{n-1} & = & b_2 \\ a_{1,0}x_0 & + & a_{1,1}x_1 & + & a_{1,2}x_2 & + \dots + & a_{1,n-1}x_{n-1} & = & b_1 \\ a_{0,0}x_0 & + & a_{0,1}x_1 & + & a_{0,2}x_2 & + \dots + & a_{0,n-1}x_{n-1} & = & b_0 \end{array}$$

where a 's and b 's are constants and x 's are unknown to be found.



..Case studies: Systems

Iterative methods: One way to solve these systems of equations is by *iterations*.

One may rearrange the i -th equation

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 + \dots + a_{i,n-1}x_{n-1} = b_i$$

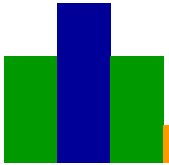
to get

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + \dots + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} + \dots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = \frac{1}{a_{i,i}}[b_i - \sum_{j \neq i} a_{i,j}x_j]$$

Such an equation specifies an unknown in terms of the other unknowns and can be used as an iteration formula (with the hope to get a convergent iteration process)



..Case studies: Systems

Jacobi iteration: in this iterative process (based on the above general schema) *all* values of x 's are *updated together*.

It can be proved that Jacobi method will *converge if the diagonal values of a 's have an absolute value greater than the sum of the absolute values of the others a 's on the row* (or, in other words, if the matrix of a 's is diagonally dominant), i.e.,

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}| \quad (\forall i)$$

(this is a sufficient, but not a necessary condition)



..Case studies: Systems

Termination (Jacobi iteration): A simple approach is

- to *compare the values computed at one iteration step* to the values obtained at the *previous iteration step* and
- to terminate the computation at the t -th iteration when all these differences are within a *given tolerance*, i.e.,

$$\forall i : |x_i^t - x_i^{t-1}| < \varepsilon \quad (\varepsilon - \text{error tolerance})$$

(x_k^t denotes the value of x_k at t -th iteration)

Other appropriate termination conditions are:

- $\sqrt{\sum_{i=0}^{n-1} (x_i^t - x_i^{t-1})^2} < \varepsilon$ (*Pacheco*) or
- $\forall i : |\sum_{j=0}^{n-1} a_{i,j} x_j^t - b_i| < \varepsilon$ (*Bertsekas & Tsitsiklis*)



..Case studies: Systems

Sequential code:

```
for (i=0; i<n; i++)
    x[i] = b[i];
for (iter = 0; iter < limit; iter++)
    for (i=0; i<n; i++){
        sum = 0;
        for (j=0; j<n; j++)
            if (i != j)
                sum = sum + a[i][j]*x[j];
        newx[i] = (b[i]-sum)/a[i][i]
    }
for (i=0; i<n; i++)
    x[i] = newx[i];
}
```



..Case studies: Systems

More efficient sequential code (avoiding the if statement):

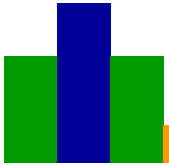
```
for (i=0; i<n; i++)
    x[i] = b[i];
for (iter = 0; iter < limit; iter++)
    for (i=0; i<n; i++){
        sum = -a[i][i]*x[i];
        for (j=0; j<n; j++)
            sum = sum + a[i][j]*x[j];
        newx[i] = (b[i]-sum)/a[i][i]
    }
    for (i=0; i<n; i++)
        x[i] = newx[i];
}
```



..Case studies: Systems

Parallel code: suppose we have a process P_i for each unknown x_i ; the code for process P_i may be:

```
x[i] = b[i];
for (iter = 0; iter < limit; iter++){
    sum = -a[i][i]*x[i];
    for (j=0; j<n; j++)
        sum = sum + a[i][j]*x[j];
    newx[i] = (b[i]-sum)/a[i][i]
    broadcast_receive(&newx[i]);
    global_barrier();
}
```

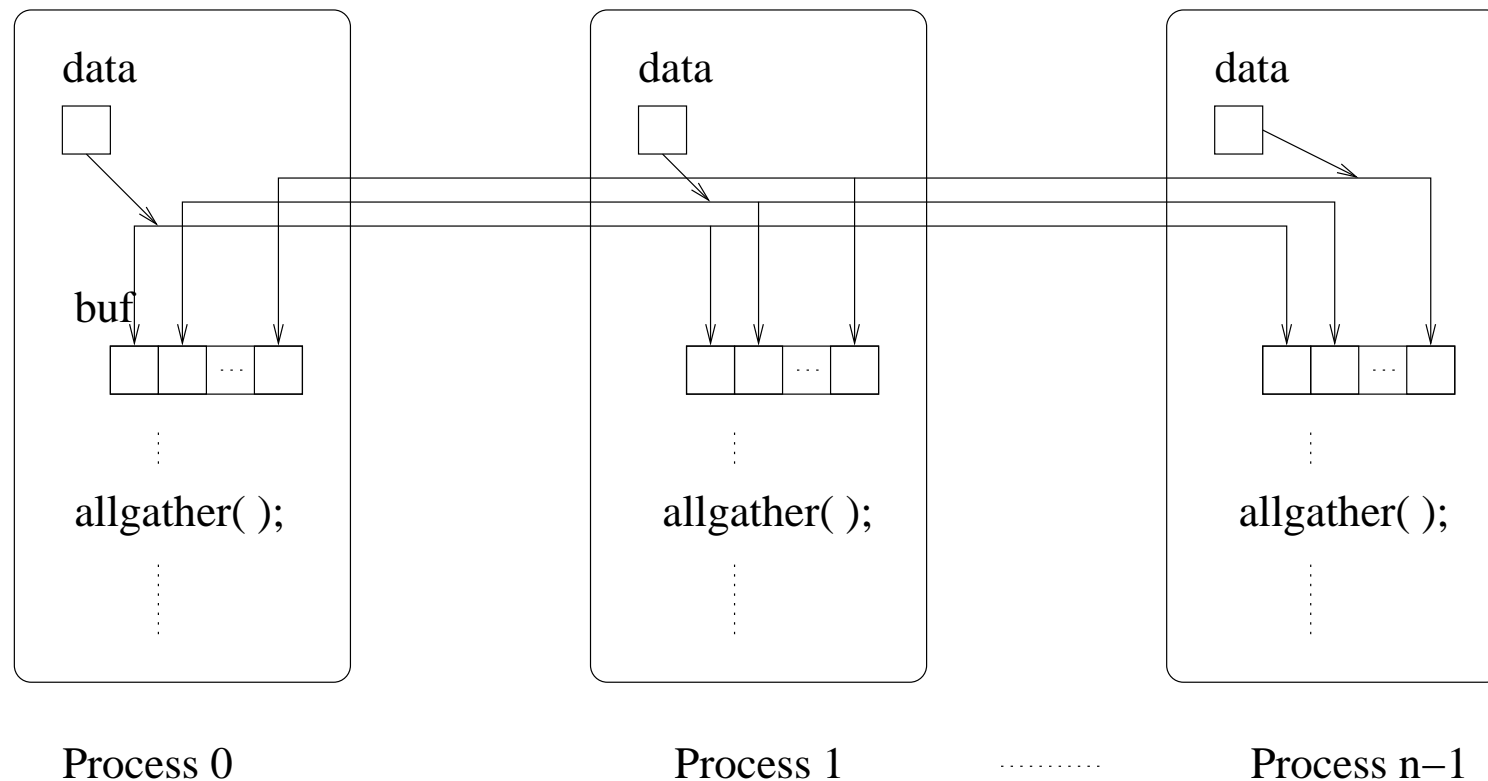
..Case studies: Systems

Notice:

- `broadcast_receive()` is used here (1) to *send* the newly computed value of `x[i]` from process P_i *to every* other process and (2) to *collect* data broadcasted *from any* other processes to process P_i
- an alternative simple solution is to use individual `send/recv` routines

..Case studies: Systems

All gather: Broadcast and gather values in *one* composite construction



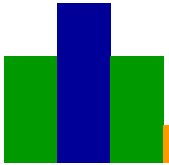
Notice: MPI_Allgather is also a global barrier, so you do not need to add global_barrier().



..Case studies: Systems

A version where the termination condition is included:

```
x[i] = b[i];  
iter = 0;  
do {  
    iter++;  
    sum = -a[i][i]*x[i];  
    for (j=0; j<n; j++)  
        sum = sum + a[i][j]*x[j];  
    newx[i] = (b[i]-sum)/a[i][i];  
    broadcast_receive(&newx[i]);  
} while (tolerance() && (iter < limit));
```



..Case studies: Systems

Partitioning:

- Usually the number of *processors* is *smaller* than the number of *unknowns*, hence each process can be responsible for computing a group of unknowns
- one may use
 - *block* partitions: allocate consecutive unknowns to a process
 - *cyclic* partition: P_0 handles $x_0, x_p, \dots, x_{((n/p)-1)p}$, etc.
(this is worse here as many such complex indices have to be computed)



..Case studies: Systems

Analysis: Suppose there are n equations and p processors; hence a processor handle n/p unknowns. Suppose there are τ iterations. Then:

- Computation time: $t_{comp} = n/p(2n + 4)\tau$
- Communication time (broadcast):
$$t_{comm} = p(t_{startup} + (n/p)t_{data})\tau = (pt_{startup} + nt_{data})\tau$$
- Overall: $t_p = (n/p(2n + 4) + pt_{startup} + nt_{data})\tau$

When $t_{startup} = 10000$ and $t_{data} = 50$, for various p and τ this has a minimum value around $p \approx 16$, $\tau \approx 0.4 \times 10^6$.



Case studies: 2. Heat distribution problem

Heat distribution problem: Suppose *a square metal sheet has known temperatures along each of its edges*. The goal is to *find temperature distribution of the metal square* (at equilibrium).

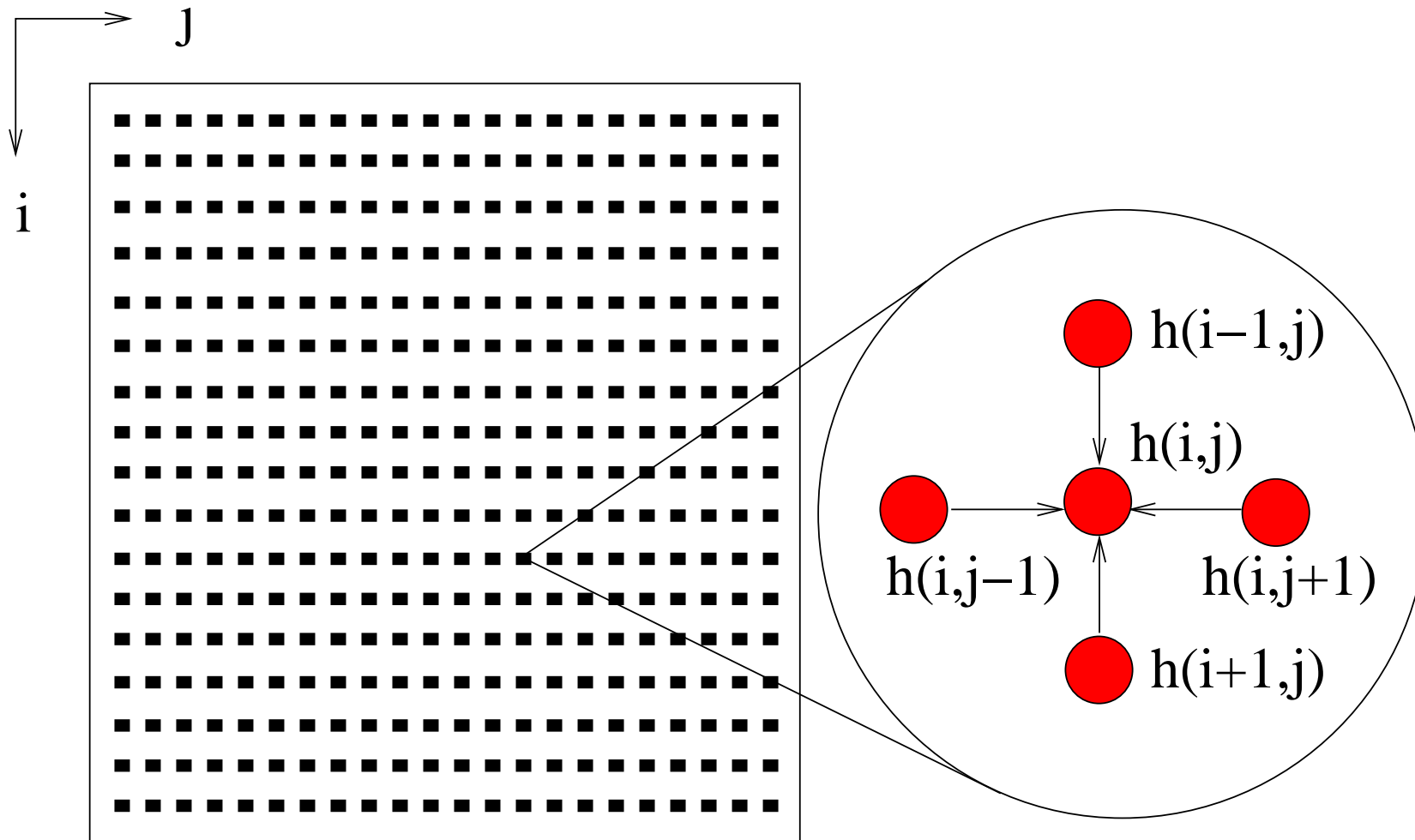
Solution:

- divide the area into a *fine mesh* of points $h_{i,j}$
- the evolution in time is obtained by using the following relation

$$h_{i,j}^{t+1} = \frac{h_{i-1,j}^t + h_{i+1,j}^t + h_{i,j-1}^t + h_{i,j+1}^t}{4}$$

- as in a usual iterative method, we *repeat* for a fixed number of iterations or until the difference between the values at two consecutive iteration steps is less than a very small prescribed amount in each point.

..Case studies: Heat distribution





..Case studies: Heat distribution

Connection with systems of linear equations: For convenience, the points are numbered from 1 to k^2 . Each point will have an associated equation

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4}$$

leading to a system of linear equations

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i+k} = 0$$

This is the *finite difference* system associated to *Laplace equation*.

Back: We come back to our double indices. The temperature on the boundary is known, i.e., we know $h_{0,r}, h_{n,r}, h_{r,0}, h_{r,n}$ for any $0 \leq r \leq n$. A sequential code (including a termination condition) is:



..Case studies: Heat distribution

```
do{
  for (i=1; i<n; i++)
    for (j=1; j<n; j++)
      g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]
                    +h[i][j-1]+h[i][j+1]);
  for (i=1; i<n; i++)
    for (j=1; j<n; j++)
      h[i][j] = g[i][j];
  cont = FALSE;
  for (i=1; i<n; i++)
    for (j=1; j<n; j++)
      if (!converged(i,j)){
        continue = TRUE;
        break;
      }
} while (continue == TRUE)
```



..Case studies: Heat distribution

Parallel code: (version with a fixed number of iterations and a process $P_{i,j}$ for each point:)

```
for (iter=0; iter<limit; iter++)
    h[i][j] = 0.25*(w+e+n+s);
    send(&h[i][j],  $P_{i-1,j}$ ); /* nonblocking sends */
    send(&h[i][j],  $P_{i+1,j}$ );
    send(&h[i][j],  $P_{i,j-1}$ );
    send(&h[i][j],  $P_{i,j+1}$ );
    recv(&w,  $P_{i-1,j}$ ); /* blocking receivers */
    recv(&e,  $P_{i+1,j}$ );
    recv(&n,  $P_{i,j-1}$ );
    recv(&s,  $P_{i,j+1}$ );
}
```

Notice: It is important to use nonblocking sends, otherwise the processes deadlock. On the other hand, some blocking routines (as our blocking receives) are also needed - here, a local synchronization technique is used.



..Case studies: Heat distribution

Some care is needed for *processes operating at the edges*:

- one may allocate processes for the edges which simply send the data; e.g., for $P_{0,j}$

```
for (iter=0; iter<limit; iter++)  
    send(&h[0][j],  $P_{1,j}$ )
```

- other possibility will be to remove send/rcv statements from the code of processes acting near the border, e.g.,



..Case studies: Heat distribution

```
⋮  
if (i != 1) send(&h[i][j],  $P_{i-1,j}$ ) ;  
if (i != n-1) send(&h[i][j],  $P_{i+1,j}$ ) ;  
if (j != 1) send(&h[i][j],  $P_{i,j-1}$ ) ;  
if (j != n-1) send(&h[i][j],  $P_{i,j+1}$ ) ;  
if (i != 1) recv(&w,  $P_{i-1,j}$ ) ;  
if (i != n-1) recv(&e,  $P_{i+1,j}$ ) ;  
if (j != 1) recv(&n,  $P_{i,j-1}$ ) ;  
if (j != n-1) recv(&s,  $P_{i,j+1}$ ) ;  
⋮
```



..Case studies: Heat distribution

Partitioning: As the number of points is usually large, one has to allocate more points to a process. Natural partitions are into *square blocks* or *strips*.

- Block partition: Communication time is
$$t_{commsq} = 8(t_{startup} + \sqrt{(n/p)}t_{data})$$
- Strip partition: Communication time is
$$t_{commcol} = 4(t_{startup} + \sqrt{n}t_{data}).$$



..Case studies: Heat distribution

Communication time: Strips are better for large startup time; for small startup time blocks are better. Indeed:

strips better than blocks

iff

$$t_{commcol} < t_{commsq}$$

iff

$$4(t_{startup} + \sqrt{n}t_{data}) < 8(t_{startup} + \sqrt{(n/p)}t_{data})$$

iff

$$t_{startup} > \sqrt{n}\left(1 - \frac{2}{\sqrt{p}}\right)t_{data}$$



..Case studies: Heat distribution

Implementation details:

- a good technique is to use an additional row of points (*ghost points*) at each edge of the area of a process that hold the values from the adjacent edge of a neighbor process
- code with ghost points (for row strips; $m = \sqrt{n}$ and $m1 = m/p$)

```
for (i=1; i<m1; i++)
    for (j=1; j<m; j++)
        g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]
                        +h[i][j-1]+h[i][j+1]);
for (i=1; i<m1; i++)
    for (j=1; j<m; j++)
        h[i][j] = g[i][j];
send(&g[1][1], &m, Pi-1);
send(&g[1][m], &m, Pi+1);
recv(&h[1][0], &m, Pi-1);
recv(&h[1][m+1], &m, Pi+1);
```



..Case studies: Heat distribution

Unsafe send/receive:

- If *all* processes *first send, then receive* the amount of buffering may be large. If there is *no storage available*, then locally blocking send() could become as a synchronous send() and *deadlock may occur*.
- A *solution* to this problem is to alternate send/rcv routines. E.g., in the case of row strips partitions one may use:



..Case studies: Heat distribution

```
if(myid % 2) == 0){
    send(&g[1][1], &m,  $P_{i-1}$ );
    recv(&h[1][0], &m,  $P_{i-1}$ );
    send(&g[1][m], &m,  $P_{i+1}$ );
    recv(&h[1][m+1], &m,  $P_{i+1}$ );
} else {
    recv(&h[1][0], &m,  $P_{i-1}$ );
    send(&g[1][1], &m,  $P_{i-1}$ );
    recv(&h[1][m+1], &m,  $P_{i+1}$ );
    send(&g[1][m], &m,  $P_{i+1}$ );
}
```



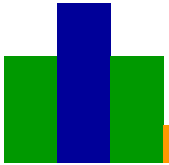
..Case studies: Heat distribution

Alternatives for safe communication:

- *combine send and receive* in the same routine
MPI_Sendrecv() (which is guaranteed not to deadlock)
- use MPI_Bsend() where the user *provides explicit buffering* storage
- use nonblocking routines like MPI_Isend() and MPI_Irecv()
 - the *routines return immediately* and one has to use separate routines to test if the communication was successful

Notice: MPI routines used for successful communication testing include:

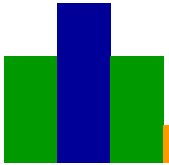
*MPI_Wait(), MPI_Waitall(), MPI_Waitany(), MPI_Test(),
MPI_Testall(), MPI_Testany().*



Cellular automata

Cellular automata represent an interesting “synchronous” formal model where:

- the problem space is divided into *cells*
- each cell can be in one of a finite number of *states*
- cells are affected by *their neighbors* according to certain *rules*
- all cells are affected simultaneously, leading to an *evolution* from one “generation” to another



..Cellular automata

Conway Game of life: Rules

- every organism with *two or three* neighbors *survives* for the next generation
- every organism with *four or more* neighbors *dies* from over-population
- every organism with *one neighbor or none* *dies* from isolation
- an empty cell adjacent to *exactly three* occupied neighbors will *give birth* to an organism

Many applications of cellular automata in: computer science, physics, biology, etc.

Lesson 7: Load balancing and termination detection

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004



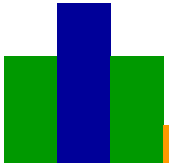
Topics

Load balancing:

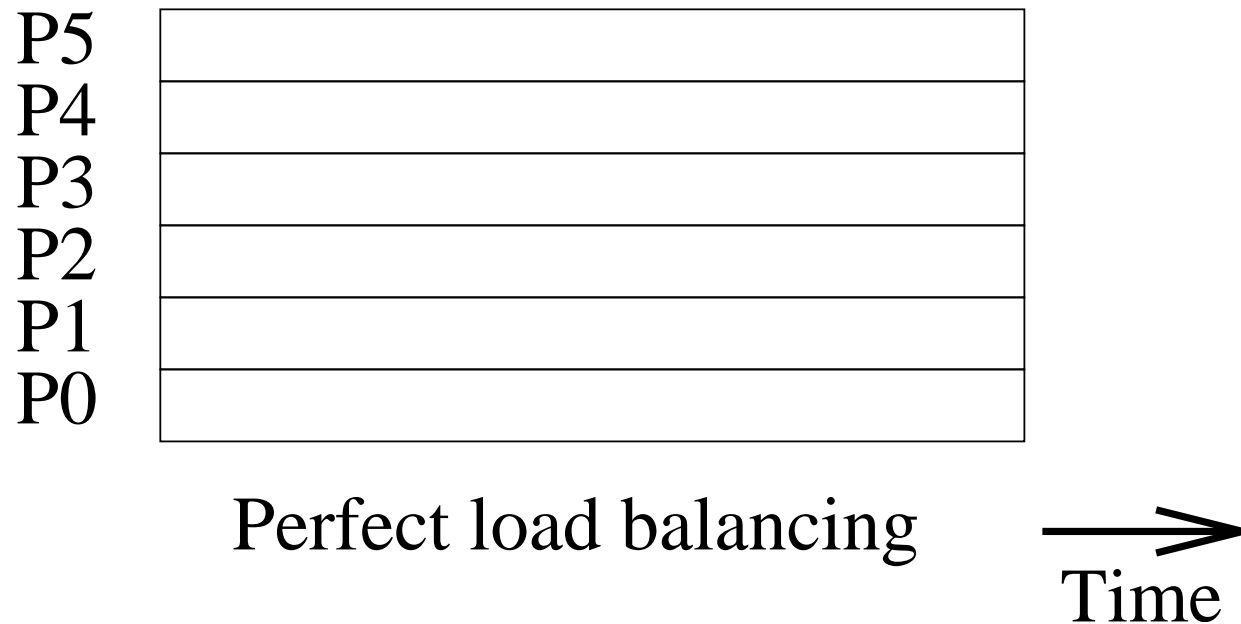
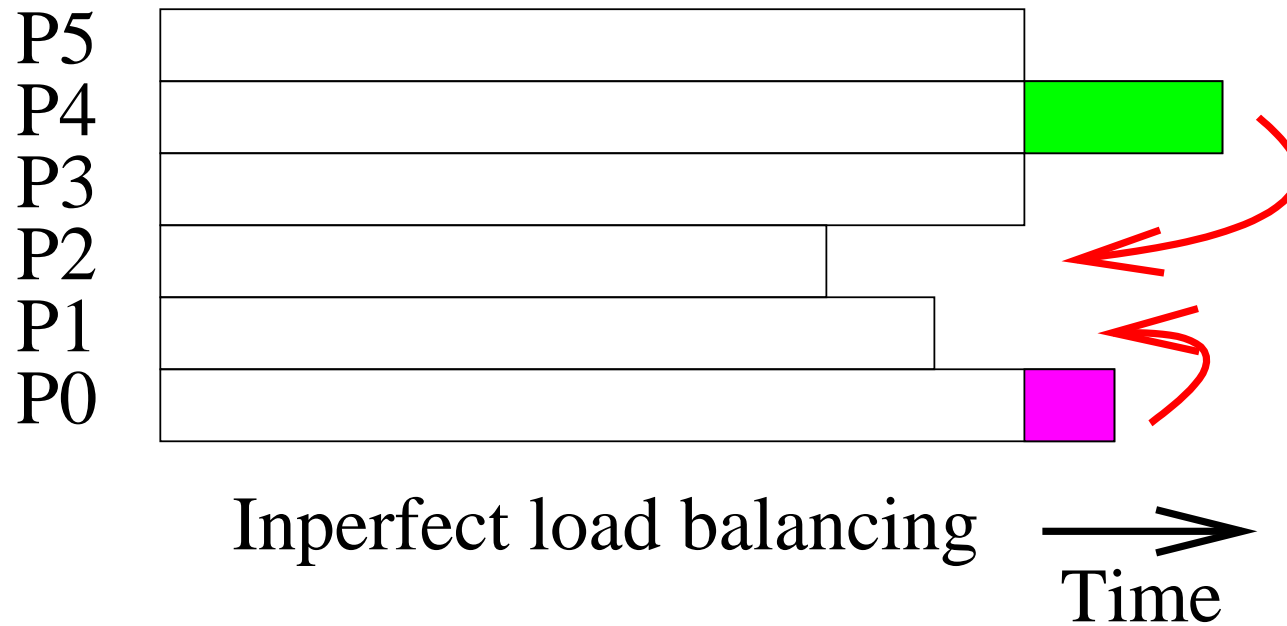
- a technique to make a *fair distribution of computation* across processors in order *to increase execution speed*

Termination detection:

- to detect when a computation has been completed
- it is usually more *difficult* when the computation is *distributed*



Load balancing





Static load balancing (SLB)

Static load balancing: in such a case the balancing is scheduled *before* the execution of any process.

A few static load balancing techniques are:

- *Round robin algorithm* - the tasks are passed to processes in a sequential order; when the last process has received a task the schedule continues with the first process (a new round)
- *Randomized algorithm*: the allocation of tasks to processes is random
- *Recursive bisection*: recursively divides the problem into sub-problem of equal computational effort
- *Simulated annealing* or *genetic algorithms*: mixture allocation procedure including optimization techniques



A theoretical result

A theoretical result:

- The problem of *mapping tasks to processes for arbitrary networks* is *NP-hard* (nondeterministically polynomial hard)

Hence

- by the common belief that $P \neq NP$, *no efficient polynomial time algorithm* may be found; one has to use various *heuristics*



Critics on static load balancing

Critics: - even when a good mathematical solution exists, static load balancing still have several flaws:

- it is very difficult to *estimate a-priori* [in an accurate way] the *execution time* of various parts of a program
- sometimes there are *communication delays* that vary in an uncontrollable way
- for some problems the *number of steps* to reach a solution is *not known* in advance



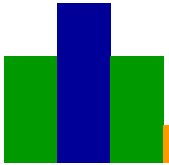
Dynamic load balancing (DLB)

Dynamic load balancing:

- the schedule of allocating tasks to processes is done *during the execution* of the processes

Features:

- the factors in the above critics are taken into account, improving the efficiency of the program
- there is an *additional overhead* during execution, but generally it is more effective than static load balancing
- *termination detection* is more *complicate* in dynamic load balancing



..DLB

(Features, cont.)

- computation is divided into *work* or *tasks* to be performed and *processes* perform these tasks
- processes are mapped onto *processors*; the ultimate goal is to keep processors busy;
- we will often map a single process onto each processor, so the terms *process* and *processor* are somehow interchangeably.



Type of DLB

Dynamic load balancing can be classified as *centralized* or *decentralized*.

Centralized load balancing:

- tasks are distributed to processes from a centralized location
- a typical *master-slave* structure exists

Decentralized load balancing:

- (worker) processes interact among themselves to solve the problem, finally reporting to a single process
- tasks are passed between arbitrary processes: a worker process may receive tasks from any other worker process and may send tasks to any other worker process



Centralized DLB

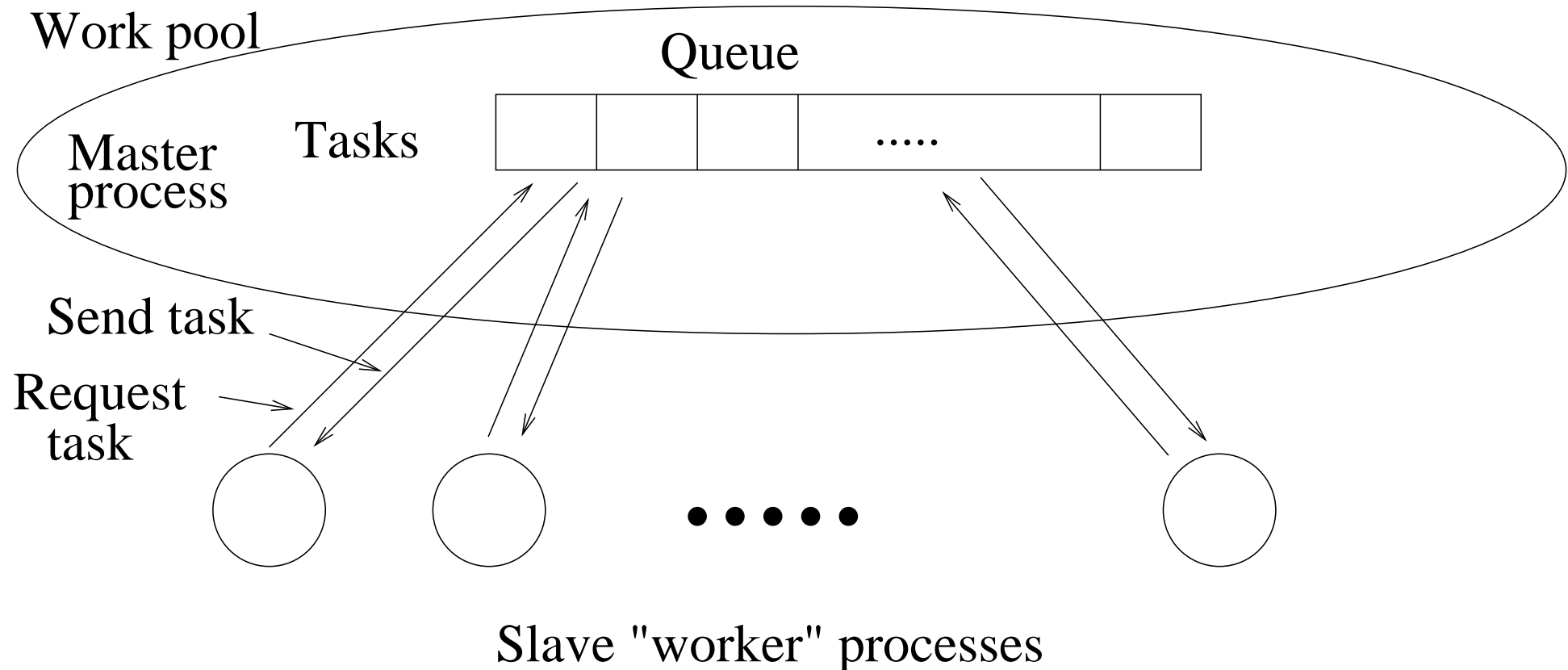
Good when: there is a *small number of slaves* and the problem consists of *computationally intensive tasks*

Basic features:

- a master process[or] holds the collection of tasks to be performed,
- tasks are sent to the slave processes
- when a slave process completes one task, it requests another task from the master process

One often uses the terms work pool, or replicated worker, or processor farm in this context. Technically, it is more efficient to start with the larger tasks first.

..centralized DLB





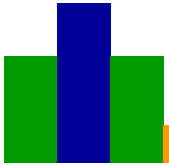
Termination in centralized DLB

Termination:

- stop the computation when the solution has been found
- when the tasks are taken from a task queue, computation terminates when
 - the task queue is empty *and*
 - every process has made a request for another task without any new tasks being generated

[It is not sufficient to check if the queue is empty if running processes are allowed to put tasks in the task queue.]

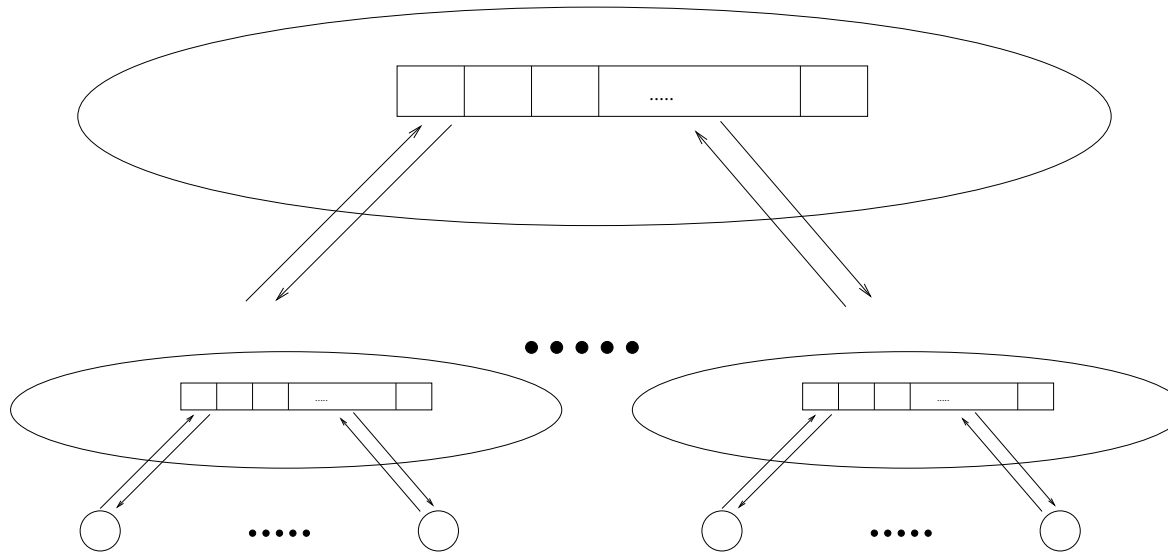
- in some applications a slave may detect the program termination by some local termination, for instance finding an item in a search algorithm



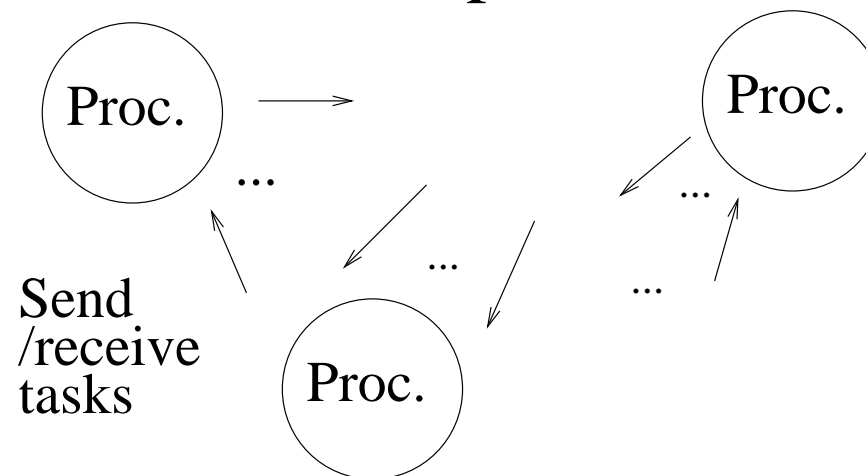
Decentralized DLB

Distributed work pool:

Tree structure



General (fully distributed work pool):





Task transfer mechanisms

Task transfer mechanisms:

- (1) *receiver-initiated method* or
- (2) *sender-initiated method*

Receiver-initiated method:

- *a process requests tasks* from other processes it selects; typically this is done when the process has few or no tasks to perform
- this method work *well* when there is a *high system load* [but, as we have seen, it can be expensive to determine process load]



..Task transfer mechanisms

Sender-initiated method:

- *a process sends tasks* to other processes it selects; typically this is done when the process has a heavy load and may pass some tasks to other processes that are willing to accept them
- this method work *well* when there is a *light system load*

Final comments:

- the above “pure” approaches may be mixed
- however, whatever method one may use, in very heavy system loads, load balancing can be difficult to achieve due to the lack of available processes.



Process selection in DLB

Process selection: Possible algorithms for selecting a process may be

- *Round robin algorithm:* process P_i requests tasks from process P_x , where x is given by a counter that is incremented modulo n , [if there are n processes], excluding $x = i$
- *Random polling algorithm:* process P_i requests tasks from process P_x , where x is a number that is randomly selected from the set $\{0, \dots, i-1, i+1, \dots, n-1\}$ [assuming there are n processes P_0, \dots, P_{n-1}]



DLB on a line structure

Procedure: (informal)

- the *master* process *feeds the queue with tasks* at one end; the tasks are shifted down the queue
- *when a worker* process $P_i (1 \leq i < n)$ *detects a task* and it is *idle*, it *takes the task* from the queue
- then the *tasks* to the left *shuffle* down the queue so that the space held by the task is filled; a *new task is inserted* into the left side end of the queue
- eventually, all processes will have a task and the queue is filled with new tasks
- for better results, high priority or larger tasks could be placed in the queue first



Code for DLB on a line structure

Shifting actions: - based on left and right communications with adjacent processes and handling of current task

Code (master):

```
for (i=0; i < noTasks; i++){  
    recv( $P_1$ , requestTag);  
    send(&task,  $P_1$ , taskTag);  
}  
recv( $P_1$ , requestTag);  
send(&empty,  $P_1$ , taskTag);
```



..Code for DLB on a line structure

Code $P_i(1 \leq i < n)$:

```
if (buffer == empty){
    send( $P_{i-1}$ , requestTag);
}
recv(&buffer,  $P_{i-1}$ , taskTag);
if ((buffer == full) && (!busy)){
    task = buffer;
    buffer = empty;
}
busy = TRUE;
nrecv( $P_{i+1}$ , requestTag, request)
if (request & (buffer == full)){
    send(&buffer,  $P_{i+1}$ );
}
buffer = empty;
if (busy) {
    do some work on task;
}
if task finished, set busy to false;
```

Notice: nrecv denotes a nonblocking receive routine.



Nonblocking routines

PVM:

- the nonblocking receive routine, `pvm_nrecv()`, returns a value that is zero if no message has been received
- a probe routine, `pvm_probe()`, may be used to check whether a message has been received without actual reading of this message [later, a normal `recv()` has to be used to accept and unpack the message]

PVM:

- the nonblocking receive routine, `MPI_Irecv()`, returns in a parameter a request “handle”, which is used later to complete the communication (using `MPI_Wait` and `MPI_Test`)
- actually, it posts a request for a message and return immediately



Distributed termination detection (TD)

Termination conditions: - the following conditions have to be fulfilled:

- the [application specific] *local termination* conditions are satisfied by all processes
- there are no messages in transit between processes

Notice: The second condition is necessary to avoid a situation where a message in transit may restart an already terminated process. It is not easy to check, as the communication time is not known in advance.



TD using acknowledgment messages

Termination detection using acknowledgment messages:

- each process P is either *inactive* (no task to handle) or *active*
- a process P' which sends a message to make an idle process P active becomes its *parent*
- whenever P *receives a task*, it immediately *sends an acknowledgment* message [except if the task is received from its parent]

(cont.)



..TD using acknowledgment messages

(cont.)

- it sends an *acknowledgment* message *to its parent* process when
 - its *local termination* condition holds [all its tasks have been completed]
 - it has *transmitted acknowledgment* messages for all *tasks* it has *received*
 - it has *received acknowledgment* messages for all *tasks* it has *sent out*

[hence, this process becomes inactive before its parent]

- the full *computation is finished* when the *first process* [which has started the computation] becomes *idle* (inactive)



Single-pass ring termination algorithm

Termination detection using ring termination algorithms:

Single-pass ring termination algorithm:

- When P_0 has terminated its computation, it generates a token that is passed to P_1 .
- When $P_i (1 \leq i < n)$ receives the token and has already terminated, it passes the token onward to P_{i+1} . Otherwise, it waits for its local termination condition and then passes the token onward. [P_{n-1} passes the token to P_0 .]
- When P_0 receives a token, it knows that all processes in the ring have terminated. Finally, a message can be sent to all processes informing them on global termination.

To insure the correctness of this algorithm, it is supposed that a process cannot be reactivated after reaching its termination condition.



Dual-pass ring termination algorithm

Dual-pass ring termination algorithm:

- can handle the case when *processes may be reactivated after their local termination*, but it requires a repeated passing of the token around the ring.
- technically, we use *colored tokens*: a token may be *black* or *white*
- subsequently, *processes* are also *colored*: a process is either *black* or *white*
- roughly speaking, a black color means global termination may have not occurred

The algorithm is as follows (starting with P_0):



..dual-pass ring termination

(*..dual-pass ring termination algorithm*):

- P_0 becomes white when it has terminated; it generates a white token that is passed to P_1
- The token is passed through the ring from one process P_i to the next when P_i has terminated.
- However, the color of the token may be changes: if a process P_i passes a task to a process P_j with $j < i$, then it becomes a *black process*; otherwise it is a *white process*.
- A black process will color a token black and pass on; a white process will pass on the token in its original color
- After P_i has passed on a token, it becomes a white process.
- When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated



Fixed energy distributed termination algorithm

Fixed energy distributed termination algorithm: - based on a *fixed quantity* within the system, called *energy*:

- the system starts with all the energy being held by one process, the master process
- master process passes out portions of the energy with the tasks to processes making requests for tasks
- if these processes receive requests for tasks, the energy is divided further and passed to these processes
- when a process becomes idle, it passes the energy it holds back to master before requesting a new task

(cont.)



..fixed energy distributed termination

(cont.)

- (it can return the energy to the process which has activated it, but then it has to wait until all the energy it handed out is back to him, i.e., we have a tree structure)
- when all energy is returned to the master and the master becomes idle, all the processes must be idle and the computation can terminate

Significant disadvantage:

- the process of dividing and adding energy may not have 100% precision [if the number of processes is large, the energy may become very small]
- hence, the termination may not always be correctly detected.



Case studies: Shortest path problem (SPP)

Shortest path problem: - the problem is to find the shortest distance between two points of a graph; more precisely,

SPP Given a set of interconnected nodes where the links between the nodes are marked with *weights*, find the path from one specific node to another specific node that has the smallest accumulated weights.

Terms: the interconnected nodes can be described by a *graph*; the nodes are also called *vertices* and the links *edges*; if the edges have directions [i.e., edges can only be traversed in one direction], then the graph is called *directed*.



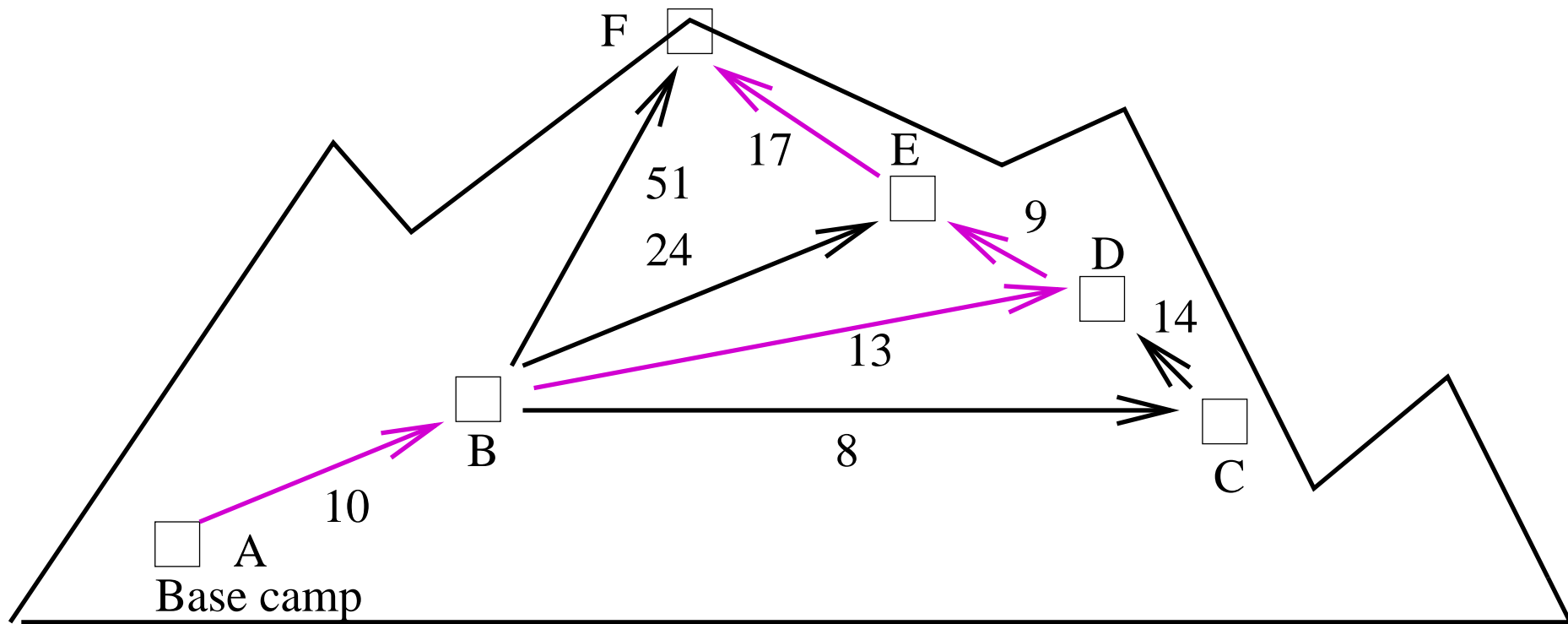
Particular interpretations of SPP

Particular interpretations of the problem:

- The *shortest distance* between two towns or other points on a map, where the weights represent distance;
- The *quickest route to travel*, where the weights represent time [different from “the shortest” if different modes of travel are available: plane, train, etc.];
- The *least expensive way to travel* by air, where the weights represent the cost of flights between cities [vertices];
- The *best way to climb* a mountain given a terrain map with contours
- The *best route through a computer network* for minimum message delay
- Etc.

Best way to climb a mountain

Example: The best way to climb a mountain:



- The weights of the graph indicate the amount of effort taking a route between two connected camp sites
- The effort in one direction may be different from the effort in the opposite direction, hence we have a *directed graph*



Graph representations

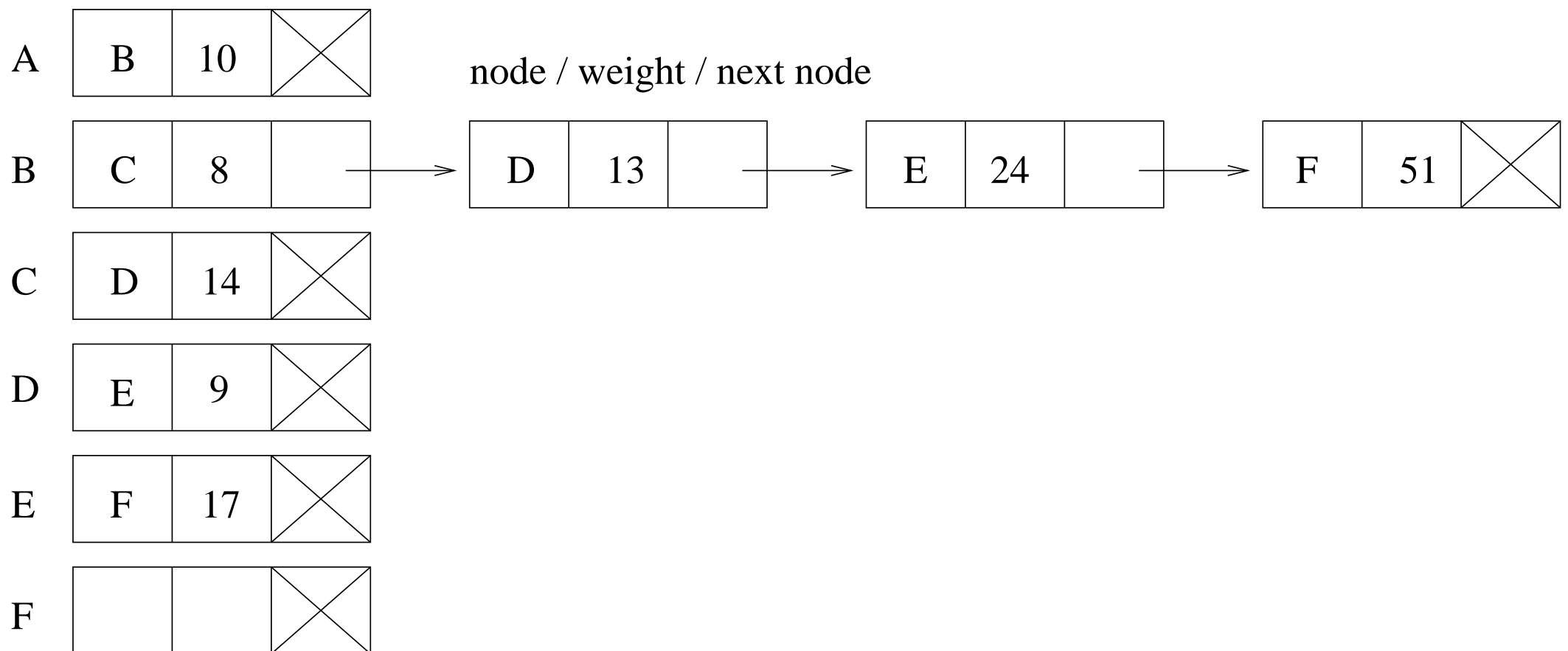
Graph representation - matrix: - that is, a two-dimensional array a , in which $a[i][j]$ holds the weight associated to the edge between vertex i and vertex j [good for *dense* graphs]

		<i>Destination</i>					
		<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
<i>Source</i>	<i>A</i>	∞	10	∞	∞	∞	∞
	<i>B</i>	∞	∞	8	13	24	51
	<i>C</i>	∞	∞	∞	14	∞	∞
	<i>D</i>	∞	∞	∞	∞	9	∞
	<i>E</i>	∞	∞	∞	∞	∞	17
	<i>F</i>	∞	∞	∞	∞	∞	∞

..graph representations

Graph representation - lists: to a vertex v we attach a list containing all directly connected vertices from v (by an edge) and the corresponding weights of those edges [good for *sparse* graphs]

null connection





Sequential algorithms for SPP

Searching a graph: Two well-known algorithms

- Moore's single-source shortest path algorithm [Moore 1957; any order for search]
- Dijkstra's single-source shortest path algorithm [Dijkstra 1959; nearest vertex first]

Here Moore algorithm is chosen because it is more amenable to parallel implementation, although it may do more work.

[Notice: The weights must be *positive* values; there are variations of the algorithms dealing with negative values.]



Moore algorithm for SPP

Moore algorithm: - based on $d_j := \min(d_j, d_i + w_{i,j})$

- Start with the source vertex;
- Suppose d_i is an [updated] current minimum distance from source vertex to a vertex i ; for an arbitrary vertex j , update the minimum distance d_j with $d_i + w_{i,j}$, if the latter is smaller
- Repeat the above for all vertices with updated distance, till there are no more updating.



..Moore algorithm for SPP

Moore algorithm, sequential code:

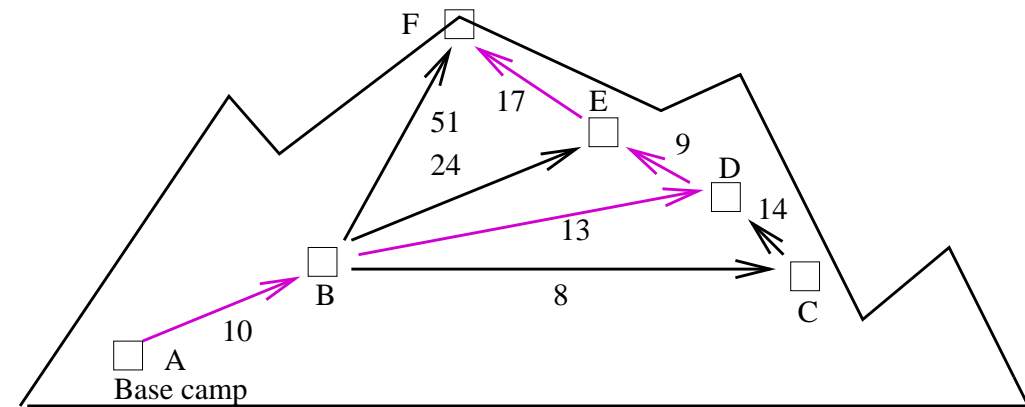
```
while(vertexQ != empty){
    i = getVetex(vertexQ);
    for (j=1; j<n; j++){
        if (w[i][j] != infinity){
            newDist = dist[i]+w[i][j];
            if(newDist < dist[j]){
                dist[j] = newDist;
                put(j,vertexQ);
            }
        }
    }
}
```


..Moore algorithm for SPP

Moore algorithm: example

Use a FIFO queue `vertexQ` holding the vertices with updated distances; `dist[i]` holds the current distance to vertex i . A running, using the above climbing example, is

vertexQ	dist to					
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B	0	10	∞	∞	∞	∞
E, D, C[, F]	0	10	18	23	34	61
D, C	0	10	18	23	34	51
C, E	0	10	18	23	32	50
E[, D]	0	10	18	23	32	49
[, F]						





Centralized parallel Moore algorithm

Centralized parallel Moore algorithm:

- vertices from `vertexQ` are used as tasks
- each slave takes vertices from the vertex queue and return new vertices
- the master holds an array with current distances; it updates the array when shorter distances are received
- as the graph structure is fixed, we assume the adjacency matrix is copied to each slave



Centralized parallel Moore algorithm

Master code

```
while ((vertexQ != empty) || (more messages to come)){
    recv(j, newDist,  $P_{any}$ , source =  $P_i$ , tag);
    if (tag == requestTag){                                /* request task */
        v = get(vertexQ);
        send(v,  $P_i$ );                                     /* send next vertex */
        send(&dist, &n,  $P_i$ );                             /* and dist array */
    } else {                                               /* got vertex and distance */
        if(newDist[j] < dist[j]){
            put(j, vertexQ);                               /* put vertex in queue */
            dist[j] = newDist;                             /* update dist */
        }
    }
}
for (j=1; j<n; j++){
    recv( $P_{any}$ , source =  $P_i$ );
    send( $P_i$ , terminationTag);
}
```



..Centralized parallel Moore algorithm

Slave code

```
send( $P_{master}$ )                                /* send request for task */
recv(&v,  $P_{master}$ , tag)                        /* get vertex/tag */
while (tag != termination tag){
    recv(&dist, &n,  $P_{master}$ ) ;                /* get distances */
    for (j=1; j<n; j++){
        if(w[v][j] != infinity){
            newDist = dist[v]+w[v][j]
            if (newDist < dist[j]){
                send(&j, &newDist,  $P_{master}$ ) ;
                /* send vertex and new distance */
            }
        }
    }
    send( $P_{master}$ )                                /* send request for task */
    recv(&v,  $P_{master}$ , tag)                    /* get vertex/tag */
}
```



Decentralized parallel Moore algorithm

Informal algorithm

- Process i handle vertex i ; it holds the [current] minimum distance `dist` from source to vertex i [initially this is ∞]; it also holds a list with its neighbors
- When process i receives a new, smaller value for the distance from the source to itself computed by a different process, then
 - it updates its current minimum distance `dist`
 - computes all new distances `dist + w[j]` to all vertices j it is connected to, and
 - send these values to the appropriate neighbors
- Normally, a process is idle; it is activated by receiving a distance sent by another process; initially only the source is active;
- The termination is checked using a distributed termination procedure.



Decentralized parallel Moore algorithm

Slave code

```
recv(newDist,  $P_{any}$ );  
if (newDist < dist){  
    dist = newDist;  
    for (j=1; j<n; j++){  
        if (w[j] != infinity){  
            d = dist + w[j];  
            send(&d,  $P_j$ );  
        }  
    }  
}
```

Notice: This is the basic step for the slave code; it has to be placed in a loop and integrated with a procedure for termination detection.

Lesson 8: Programming with shared memory

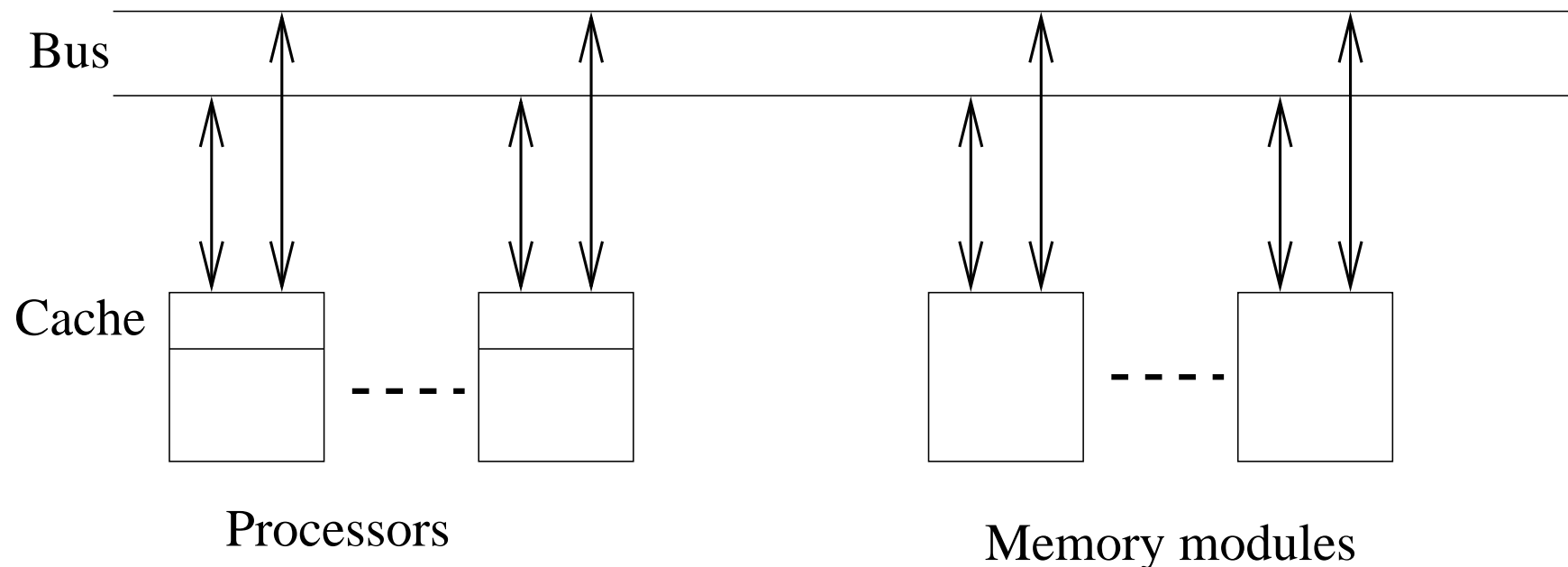
G Stefanescu — National University of Singapore

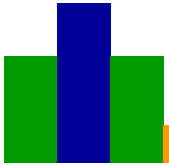
Parallel & Concurrent Programming, Spring, 2004

Shared memory model

Shared memory multiprocessor system:

- *any* memory location is *directly accessible* by any processor (not via message passing)
- there is a *single address space*
- example: a common architecture is the *single bus architecture* (good for a small number of processors, e.g. ≤ 8)





Parallel programming

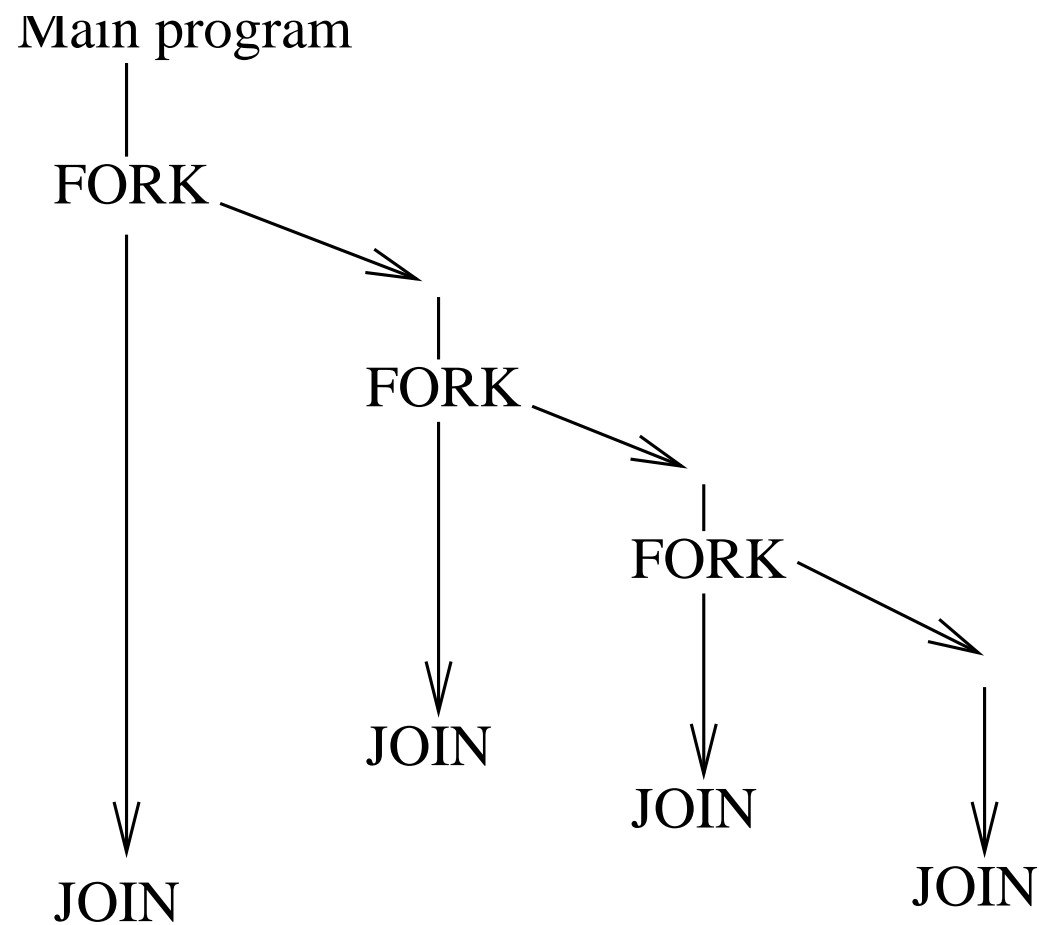
To our initial list of parallel programming alternatives

- use *new* programming languages
- *modify existing* sequential ones
- use *library* routines
- use sequential programs with *parallelizing compilers*

one may add

- *Unix processes* and
- *Threads* (Pthreads, Java, ...)

Fork-Join construct

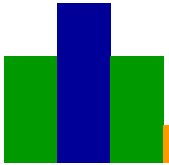




Unix (heavyweight) processes

A *fork* in Unix is obtained using system call `fork()`:

- the new process is an *exact copy* of the parent, except that it has its own process ID
- it has the same variables, whose initial values are the (current) values of the parent process
- the child process start execution from the point of the fork
- if successful,
 - `fork()` returns 0 to the child and child ID to the parent;
- if not,
 - return -1 to parent and no child process is created
- processes are joined using `wait()` and `exit()`
 - `wait(statusp)` - delay the caller
 - `exit(status)` - terminates a process



Fork in Unix: example

Simple example of using Unix fork:

```
pid = fork();  
if (pid == 0) {  
    code to be executed by slave  
} else {  
    code to be executed by parent  
}  
if (pid == 0) exit (0); else wait(0);
```



Threads

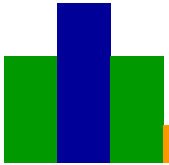
Threads vs. (heavyweight) processes:

- heavyweight *processes* are completely *separate* programs (with their own variables, stack, memory allocation)
- *threads share* the same memory space and global variables (but they have their own stack)

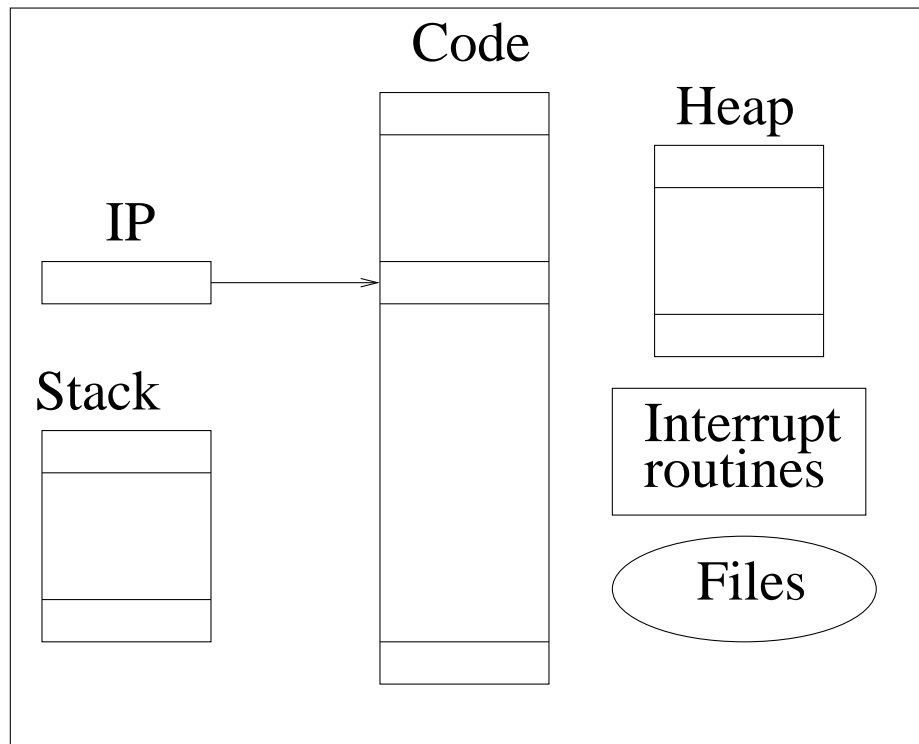
Threads' advantages:

- thread *creation* can take three orders of magnitude less time than process creation
- *communication* (via shared memory) and *synchronization* are also faster than in the case of processes

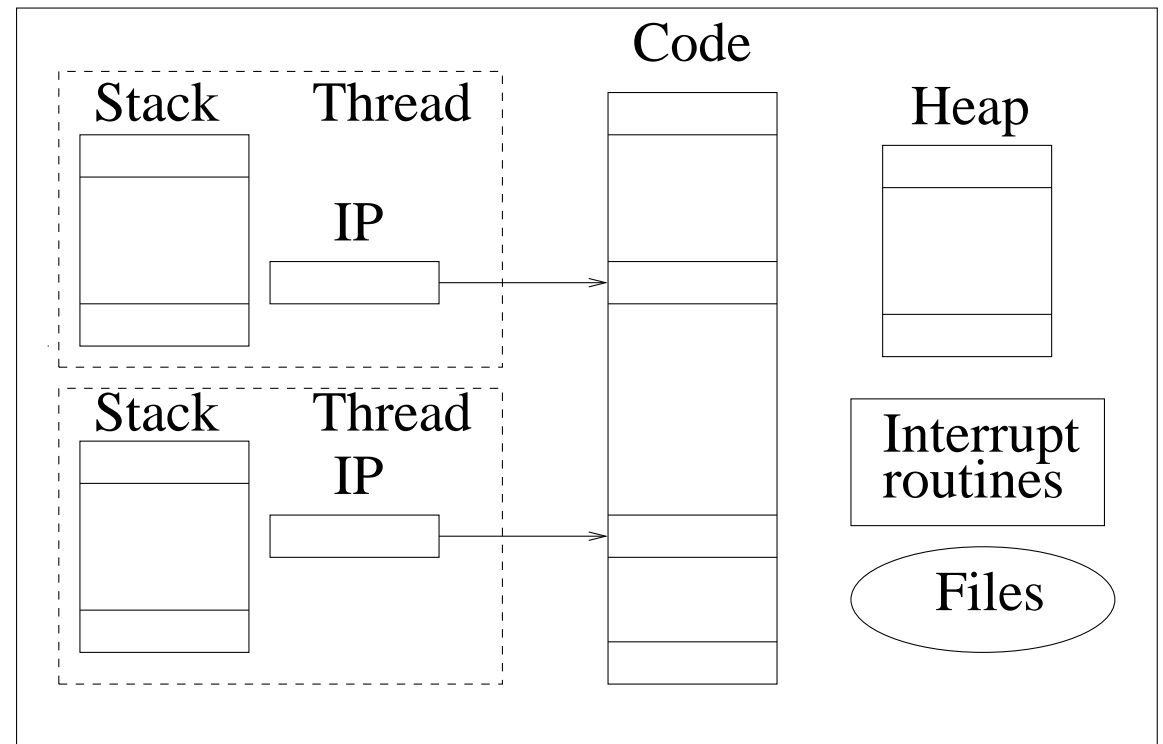
Sometimes there is no need to join a child thread with its parent; such threads are called *detached threads*.



Threads



(a) Process



(b) Threads



Pthreads

Pthreads is an IEEE standard

Main program

```
⋮  
pthread_create(&thread1,  
              NULL, proc1, &args);  
⋮  
pthread_join(thread1, *status);  
⋮
```

thread1

```
proc1(&arg) {  
    ⋮  
    return(*status);  
}
```



Thread barrier

A *barrier* may be defined in Pthreads as follows

```
for (i=0; i<n; i++)  
    pthread_create(&thread[i], NULL,  
                  (void *) slave, (void *) &arg);  
for (i=0; i<n; i++)  
    pthread_join(thread[i], NULL);
```

Notice that pthreads_join() waits for one specific thread to terminate. The current threads disappear after the barrier - if you want to reuse them after the barrier, you should recreate them.



Statement execution order

Execution order:

- on a single processor system the statements of parallel processes are *interleaved*
- on a multiprocessor system it is a chance they have real parallel processes, but usually the number of processes is larger than that of processors, so *some interleaving* appear here, as well

Interleaving: Given two sequences

abc and *ABCD*

any shuffle preserving the relative order of the letters from each string is possible, e.g.,

aABbCDc, AabcBCD, aAbBcCD, ...



..Execution order

Examples:

- *Printing*: when more processes print messages, their individual characters may be interleaved
- *Compiler optimization*: the order of some statements may be irrelevant; e.g.,

<code>a := b*3;</code>		<code>x := sin(y);</code>
<code>x := sin(y);</code>	and	<code>a := b*3;</code>

are equivalent; most of modern compilers/processors use such transformations to increase execution speed



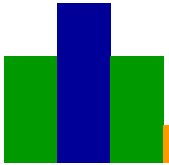
Thread-safe routines

Thread-safe routines are system calls or library routines that may be called from multiple threads simultaneously and always produce correct results

Examples

- standard I/O is print safe (no interleaving of characters)
- accessing shared/static data may be thread-safe, but some care is needed
- system routines that return time may be not thread safe

A general solution to thread-safeness is to *enclose* such routines into *critical section* (see forthcoming slides), but this is very inefficient



Accessing sharing data

Suppose x is a shared variable initially equal to 0. What is the result of running in parallel

Process 1:

$x = x+1$

Process 2:

$x = x+1$

?

The answer is: It depends...

- if the assignment statement is *atomic*, then the result is the expected one 2



..Accessing sharing data

- but if the assignment $x = x+1$ is *not atomic*, e.g., it is replaced by `read x; compute x+1; write x`

then the program becomes

Process 1:

`read x;`

`compute x+1;`

`write x;`

Process 2:

`read x;`

`compute x+1;`

`write x;`

with unexpected results, i.e.,

$r_1c_1w_1r_2c_2w_2 \rightarrow 2$ (ok), but

$r_1r_2c_1c_2w_1w_2 \rightarrow 1$!

where r_1, c_1, w_1 (resp. r_2, c_2, w_2) means read/compute/write in process 1 (resp. 2)



Critical section

We need a mechanism to ensure that only one process access a particular resource at any time; the section of code dealing with the access to such a resource is called *critical section*.

The mechanism has to obey a few conditions:

- a first process reaching a critical section for a particular resource *enters and executes* the critical section
- at the same time, it *prevents all other processes* from entering to access the some resource in their critical sections
- when the process has finished its critical section, *another* process is allowed to enter its critical section for the resource

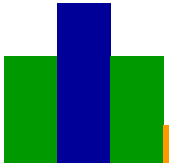
This mechanism is called *mutual exclusion*.



Locks

Locks provide the simplest mechanism for ensuring mutual exclusion of critical sections:

- a lock is a 0-1 variable that is
 - 1 to indicate that *a process* has *entered* the critical section and
 - 0 to indicate that *no process* is *in* the critical section
- it operates much like a door lock:
 - a process *coming to the door* of a critical resource and finding the *door open* may *enter* the critical section, *locking* the door behind to prevent other processes from entering;
 - when the process has *finished* the critical section, it *unlocks the door* and leaves



..Locks

Spin lock:

- a process trying to enter the critical section may *check continuously* the lock *doing nothing*; such a lock is called *spin lock* and the waiting mechanism *busy waiting*
- quite *inefficient* as no useful work is done

Atomic checks of the locks:

- it is important to have an atomic lock checking: e.g.,
 - no two processes check the door and enter simultaneously;
 - or
 - if a process find the door open no other process is able to check the door before the current process is able to close it



Pthreads locks

In Pthreads mutual exclusion is implemented using *mutex* variables; they are to be *declared* and initialized in the main program

```
pthread_mutex_t myMutex;  
:  
pthread_mutex_init(&myMutex, NULL);
```

and used as follows

```
pthread_mutex_lock(myMutex)  
    critical section  
pthread_mutex_unlock(myMutex);
```



Deadlock

Deadlock may appear when processes need to access resources hold by other processes and *no process can progress*.

Example:

P1	holds	R1	and needs	R2
P2	holds	R2	and needs	R3
⋮				
P(n-1)	holds	R(n-1)	and needs	Rn
Pn	holds	Rn	and needs	R1

A Pthread solution is `pthread_mutex_trylock()`: it *tests* the lock *without blocking* the thread. (It locks an unlock mutex variable and return 0 or return EBUSY if the variable is locked.)



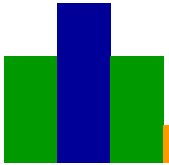
Semaphores

Dijkstra (1968) has introduced *semaphores*, as a way to control the access of critical sections. A semaphore is a positive integer s (initialized to 1) together with two operations

- $P(s)$ - waits until s is greater than 0 and then decrements s by one (wait to pass)
- $V(s)$ - increments s by one and release one of the waiting processes (release)

Each process has a sequence

$\dots \text{noncritical} \rightarrow P(s) \rightarrow \text{critical} \rightarrow V(s) \rightarrow \text{noncritical} \dots$



..Semaphores

- *binary semaphores* behave as locks, but P/V operations also provide a scheduling mechanism for process selection
- *general semaphores* (using arbitrary positive numbers) may be used to solve producer/consumer problems, keeping track on the number of free resources
- semaphores do *not* exist in *Pthreads*, but may be *found* in *Unix*
- semaphores are *powerful* enough to solve most of critical section problems, but they provide a very low-level mechanism which may be difficult to handle



Monitor

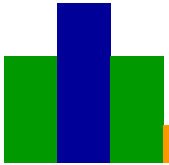
Hoare (1974) has introduced the notion of *monitor* which is

- a *higher-level* technique to control the access to critical sections, and
- encapsulates *data* and *access operations* in one structure

A monitor may be implemented using semaphores as follows

```
monitor_proc1{  
    P(monitor_semaphore);  
    monitor_body  
    V(monitor_semaphore);  
    return;  
}
```

Java has such “monitors”.



Condition variables

Problem:

- *entering* into a *critical* section may require *frequent checking* of certain (access) conditions
- *checking* them *continuously* may be very *inefficient*

A proposed solution is to use *condition variables* (used, e.g., in the case of monitors):

- such a condition variable is accompanied with three operations
 - Wait(condVar) - *wait* for the condition to occur
 - Signal(condVar) - *signal* that the condition has occurred
 - Status(condVar) - return the *number* of processes *waiting* for the condition to occur



..Condition variables

Example:

```
action() {  
    :  
    lock();  
    while(x != 0)  
        wait(s);  
    unlock(s);  
    doAction;  
    :  
}
```

```
counter() {  
    :  
    lock();  
    x--;  
    if (x == 0) signal(s);  
    unlock();  
    :  
}
```

`wait(s)` unlocks the lock and waits for signal `s`; the loop `while ...` is used as a double check for the condition; also notice that signals have no memory [if `counter` reaches its critical section first, the signal `s` is lost]



Pthread condition variables

Pthread has *condition* variables *associated* to *mutex* variables;
Signal and wait mechanisms have the following format:

—in action part

⋮

```
pthread_mutex_lock(&mutex1);
```

```
while (c <> 0)
```

```
    pthread_mutex_wait(cond1, mutex1);
```

```
pthread_mutex_unlock(&mutex1);
```

—in counter part

⋮

```
pthread_mutex_lock(&mutex1);
```

```
c--;
```

```
    if (c == 0) pthread_cond_signal(cond1);
```

```
pthread_mutex_unlock(&mutex1);
```




Language constructs for parallelism

A few *specific* language constructs for shared memory parallel programs are:

- *Shared data* may be specified as
`shared int;`
- *Different concurrent* statements may be introduced using the `par` statement
`par{S1;S2; ... ,Sn;}`
- *similar concurrent statements* may be introduced using the `forall` statement
`forall (i=0; i<n; i++){body}`
which generates n concurrent blocks `{body1; body2; ... ; bodyn;}`, one instance of the body for each i



Dependency analysis

Dependency analysis is an *automatic* technique used by *parallelizing compilers* to detect which processes/statements may be executed in *parallel*

Examples (1st - obvious; 2nd - easy, but not so obvious)

```
forall(i=0; i<5; i++)  
    a[i] = 0
```

```
forall(i=2; i<6; i++){  
    x = i - 2*i + i*i;  
    a[i] = a[x];  
}
```



Bernstein's conditions

Bernstein's conditions: A set of *sufficient* conditions for two processes to be executed simultaneously is the following *concurrent-read/exclusive-write* situation:

- define two sets of memory locations
 - I_i - the set of memory locations read by process P_i
 - O_i - the set of memory locations where process P_i writes
- two processes P_1, P_2 may be *executed simultaneously* if

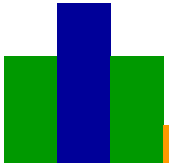
$$O_1 \cap O_2 = \emptyset \quad \& \quad I_1 \cap O_2 = \emptyset \quad \& \quad I_2 \cap O_1 = \emptyset$$

(in words: the writing locations are disjoint and no process reads from a location where the other process writes.)



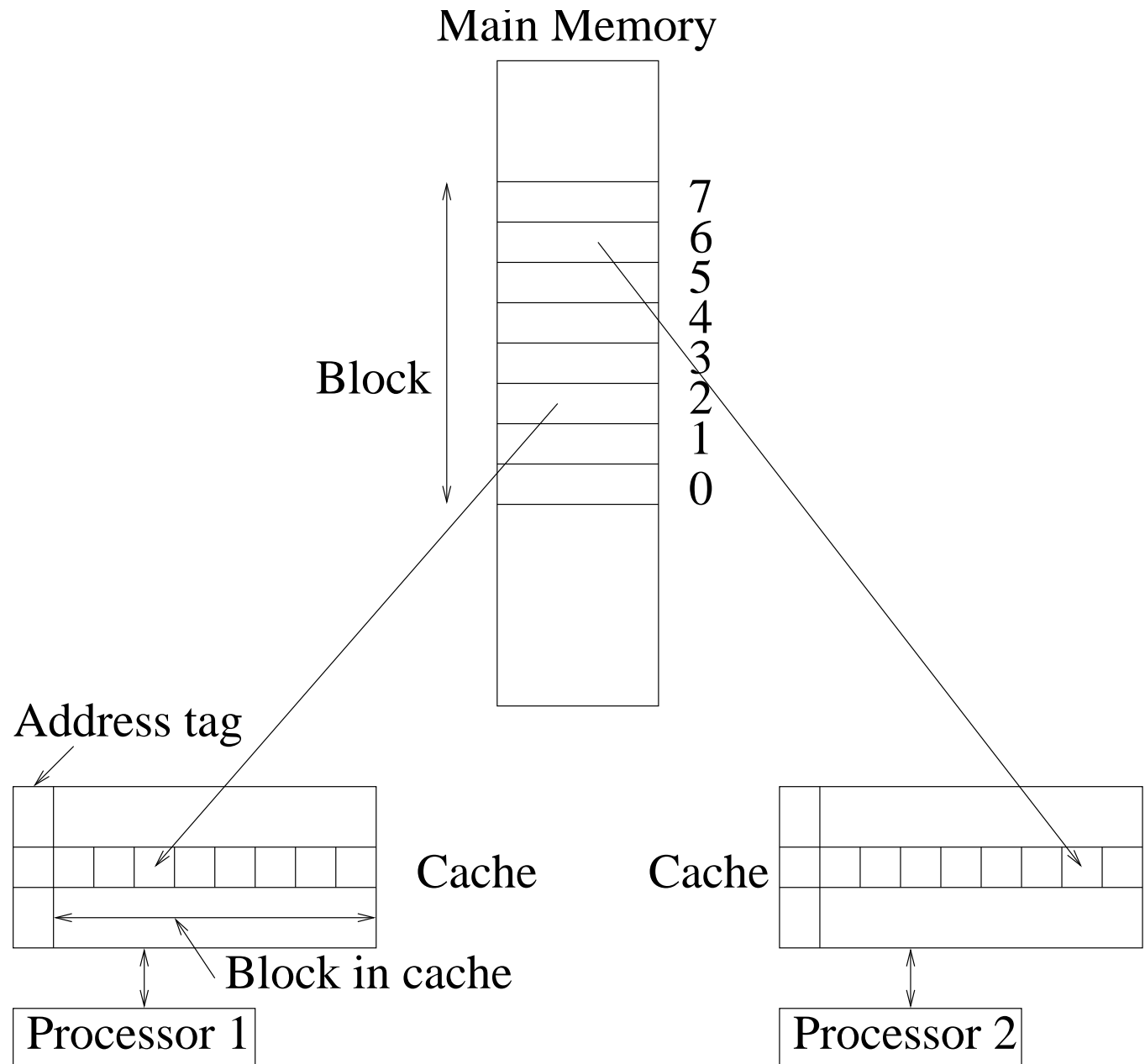
Sharing data in systems with caches

- *cache memory* is a high-speed memory closely attached to a processor to hold the recent used data and code
- *cache coherence protocols*: need to assure processes have coherent caches; two solutions:
 - *update policy* - copies in all caches are updated anytime when one copy is modified
 - *invalidate policy* - when a datum in one copy is modified, the same datum in the other caches is invalidated (resetting a bit); the new data is updated only if the processor needs it
- a *false sharing* may appear when different processors alter different data within the same block (solution - memory rearrangement)



Threads

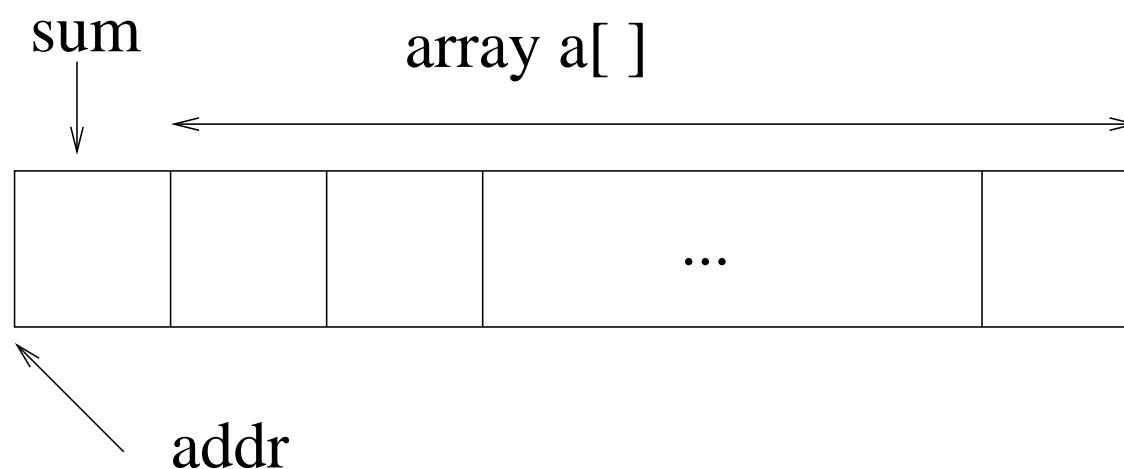
False sharing:



Examples (add 1000 numbers): 1 Unix

Adding numbers: The aim is to add 1000 numbers using two processes, namely P1 will add the numbers with an even index in array $a[]$, while P2 those with odd index in $a[]$.

The following memory structure is used





..Examples / Unix

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
    int shmid, s, pid;
    char *shm;
    int *a, *addr, *sum;
    int partial_sum;
    int i;
```



..Examples / Unix

```
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT));
if (s == -1){
    perror("semget");
    exit(1);
}
if (semctl(s,0,SETVAL, init_sem_value) < 0){
    perror("semctl");
    exit(1);
}

shmid = shmget(IPC_PRIVATE, (array_size*sizeof(int)+1),
               (IPC_CREAT|0600));
if (shmid == -1){
    perror("semget");
    exit(1);
}
```




..Examples / Unix

```
shm = shmat(shmid, NULL, 0);
```

```
if (shm == (char*)-1){  
    perror("shmat");  
    exit(1);  
}
```

```
addr = (int*)shm;  
sum = addr;  
addr++;  
a = addr;
```

```
*sum = 0;  
for (i = 0; i < array_size; i++)  
    *(a+i) = i+1;
```



..Examples / Unix

```
pid = fork();
if (pid == 0){
    partial_sum = 0;
    for (i = 0; i < array_size; i = i + 2)
        partial_sum += *(a + i);
} else {
    partial_sum = 0;
    for (i = 1; i < array_size; i = i + 2)
        partial_sum += *(a + i);
}
P(&s);
*sum += partial_sum;
V(&s);

printf("\nprocess pid = %d, partial sum = %d\n",
        pid, partial_sum);
if (pid == 0) exit(0); else wait (0);
printf("\nThe sum of 1 to %i is %d\n",
        array_size, *sum);
```



..Examples / Unix

```
if (semctl(s, 0, IPC_RMID, 1) == -1) {  
    perror("semctl");  
    exit(1);  
}  
  
if (shmctl(shmid, IPC_RMID, NULL) == -1) {  
    perror("shmctl");  
    exit(1);  
}  
  
exit(0);  
}
```



..Examples / Unix

```
void P(int *s)
{
    struct sembuf sembuffer, *sops;
    sops = &sembuffer;
    sops->sem_num = 0;
    sops->sem_op = -1;
    sops->sem_flg = 0;
    if (semop(*s, sops, 1) < 0) {
        perror("semop");
        exit(1);
    }
    return;
}
```



..Examples / Unix

```
void V(int *s)
{
    struct sembuf sembuffer, *sops;
    sops = &sembuffer;
    sops->sem_num = 0;
    sops->sem_op = 1;
    sops->sem_flg = 0;
    if (semop(*s, sops, 1) < 0) {
        perror("semop");
        exit(1);
    }
    return;
}
```

```
process pid = 0, partial sum = 250000
process pid = 2073, partial sum = 250500
```

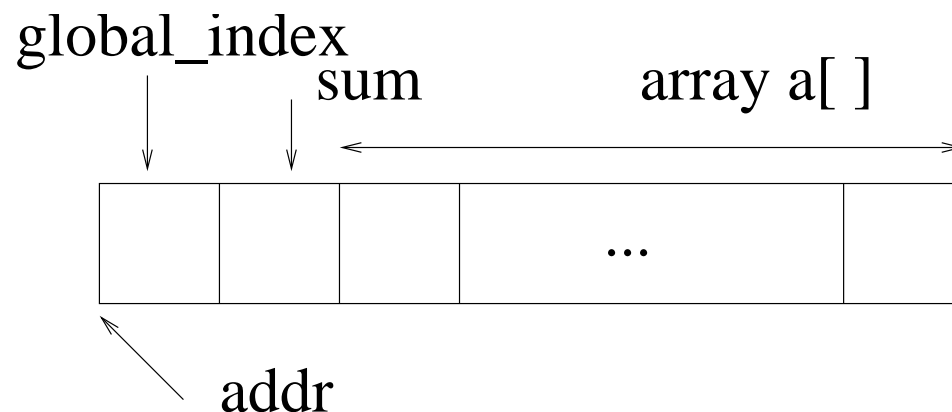
The sum of 1 to 1000 is 500500

Examples (add 1000 numbers): 2 Pthreads

Adding numbers: The aim is again to add 1000 numbers, now using ten threads:

- each thread reads in a `global_index` for scanning array `a[]` and adds the corresponding element to its local `partial_sum`
- then, each thread will add its `partial_sum` to a global `sum` variable
- both, the access/update of `global_index` and of `sum` variables are critical sections implemented using mutex variables

The following memory structure is used





..Examples / Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;
```



..Examples / Pthreads

```
void *slave(void *ignored)
{
    int local_index, partial_sum = 0;
    do {
        pthread_mutex_lock(&mutex1);
        local_index = global_index;
        global_index++;
        pthread_mutex_unlock(&mutex1);

        if (local_index < array_size)
            partial_sum += *(a + local_index);
    } while (local_index < array_size);

    pthread_mutex_lock(&mutex1);
    sum += partial_sum;
    pthread_mutex_unlock(&mutex1);
    return();
}
```




..Examples / Pthreads

```
main()  
{  
    int i;  
    pthread_t thread[10];  
    pthread_mutex_init(&mutex1, NULL);  
  
    for (i = 0; i < array_size; i++)  
        a[i] = i+1;  
  
    for (i = 0; i < no_threads; i++)  
        if (pthread_create(&thread[i], NULL, slave, NULL) != 0)  
            perror("Pthread_create fails");  
  
    for (i = 0; i < no_threads; i++)  
        if (pthread_join(thread[i], NULL) != 0)  
            perror("Pthread_join fails");  
  
    printf("The sum of 1 to %i is %d\n", array_size, sum);  
}
```



Examples (add 1000 numbers): 2 Java

Adding numbers: The aim is again to add 1000 numbers using ten threads, but now using Java programming.

The key mechanism to implement the access to the critical sections is by using *synchronized* methods.



..Examples / Java

```
public class Adder
{
    public int[] array;
    private int sum = 0;
    private int index = 0;
    private int number_of_threads = 10;
    private int threads_quit;

    public Adder()
    {
        threads_quit = 0;
        array = new int[1000];
        initializeArray();
        startThreads();
    }

    public synchronized int getNextIndex()
    {
        if (index < 1000) return (index++); else return (-1);
    }
}
```



..Examples / Java

```
public synchronized void addPartialSum(int partial_sum)
{
    sum = sum + partial_sum;
    if (++threads_quit == number_of_threads)
        System.out.println("The sum of the numbers is " + sum);
}
```

```
private void initializeArray()
{
    int i;
    for (i = 0; i < 1000; i++)
        array[i] = i;
}
```



..Examples / Java

```
public void startThreads()  
{  
    int i = 0;  
    for (i = 0; i < 10; i++)  
    {  
        AdderThread at = new AdderThread(this,i);  
        at.start();  
    }  
}
```

```
public static void main(String args[])  
{  
    Adder a = new Adder();  
}  
}
```



..Examples / Java

```
class AdderThread extends Thread
{
    int partial_sum = 0;
    Adder parent;
    int number;

    public AdderThread(Adder parent, int number)
    {
        this.parent = parent;
        this.number = number;
    }
}
```



..Examples / Java

```
public void run()
{
    int index = 0;

    while(index != -1) {
        partial_sum = partial_sum + parent.array[index];
        index = parent.getNextIndex();
    }

    System.out.println("Partial sum from thread "
        + number + " is " + partial_sum);
    parent.addPartialSum(partial_sum);
}
```



..Examples / Java

```
Partial sum from thread 8 is 0
Partial sum from thread 7 is 910
Partial sum from thread 1 is 165931
Partial sum from thread 2 is 51696
Partial sum from thread 9 is 10670
Partial sum from thread 6 is 54351
Partial sum from thread 0 is 98749
Partial sum from thread 3 is 62057
Partial sum from thread 5 is 45450
Partial sum from thread 4 is 9686
The sum of the numbers is 499500
```


Lesson 9: Sorting algorithms

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004



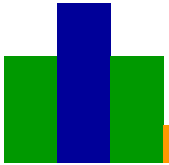
Parallel sorting algorithms

Potential speedup:

- best sequential sorting algorithms (for arbitrary sequences of numbers) have average time complexity $O(n \log n)$
- hence, the best speedup one can expect from using n processors is

$$\frac{O(n \log n)}{n} = O(\log n)$$

- there are such parallel algorithms, but the hidden constant is very large (Leighton'84)
- generally, a practical, useful $O(\log n)$ algorithm may be difficult to find



Rank sort

Rank sort:

- count the number of numbers that are smaller than a number a in the list
- this gives the position of a in the sorted list
- this procedure has to be repeated for all elements of the list; hence the time complexity is $n(n - 1) = O(n^2)$ (not so good sequential algorithm)



Rank sort, sequential code

Rank sort: sequential code

```
for (i=0; i<n; i++){  
    x=0;  
    for (j=0; j<n; j++){  
        if (a[i] > a[j]) x++;  
    }  
    b[x] = a[i];  
}
```

- work well if there are no repetitions of the numbers in the list (in the case of repetitions one has to change slightly the code)



Rank sort, parallel code

Rank sort: parallel code, using n processors

```
forall (i=0; i<n; i++) {  
    x=0;  
    for (j=0; j<n; j++)  
        if (a[i] > a[j]) x++;  
    b[x] = a[i];  
}
```

- n processors work in parallel to find the ranks of all numbers of the list
- parallel time complexity is $O(n)$, better than any sequential sorting algorithm



..Rank sort, parallel code

Rank sort: parallel code, using n^2 processors

- in the case n^2 processors may be used, the comparison of each $a[0], \dots, a[n-1]$ with $a[i]$ may be done in parallel, as well
- incrementing the counter is still sequential, hence the overall computation requires $1 + n$ steps;
- if a tree structure is used to increment the counter, then the overall computation time is $O(\log n)$ (but, as one expects, processor efficiency is very low)

These are just theoretical results: it is not efficient to use n or n^2 processors to sort n numbers.



Compare-and-exchange sorting algorithms

Compare-and-exchange:

- Compare-and-exchange is a basis for many sequential sorting algorithms
- sequential “compare-and-exchange”:

```
if (a > b) {  
    tmp = A;  
    A = B;  
    B = tmp;  
}
```



..Compare-and-exchange sorting

Asymmetric parallel “compare-and-exchange”:

- process P1 sends A to P2;
- process P2 compares A with its values B ; if B is larger than A it sends B to P1, otherwise it sends A back to P1
- code for process P1:

```
send(&A, P2);  
recv(&A, P2);
```
- code for process P2:

```
recv(&A, P1);  
if (A > B) {  
    send(&B, P1);  
    B = A;  
} else  
    send(&A, P1);
```




..Compare-and-exchange sorting

Symmetric parallel “compare-and-exchange”:

- each process sends its number to the other
- code for process P1:

```
send(&A, P2);  
recv(&B, P2);  
if (A > B) A = B;
```
- code for process P2:

```
recv(&A, P1);  
send(&B, P1);  
if (A > B) B = A;
```

(alternated send-recv are used here to avoid deadlock)

- *Duplicated computation:* the result of the comparison should be the same on each processor

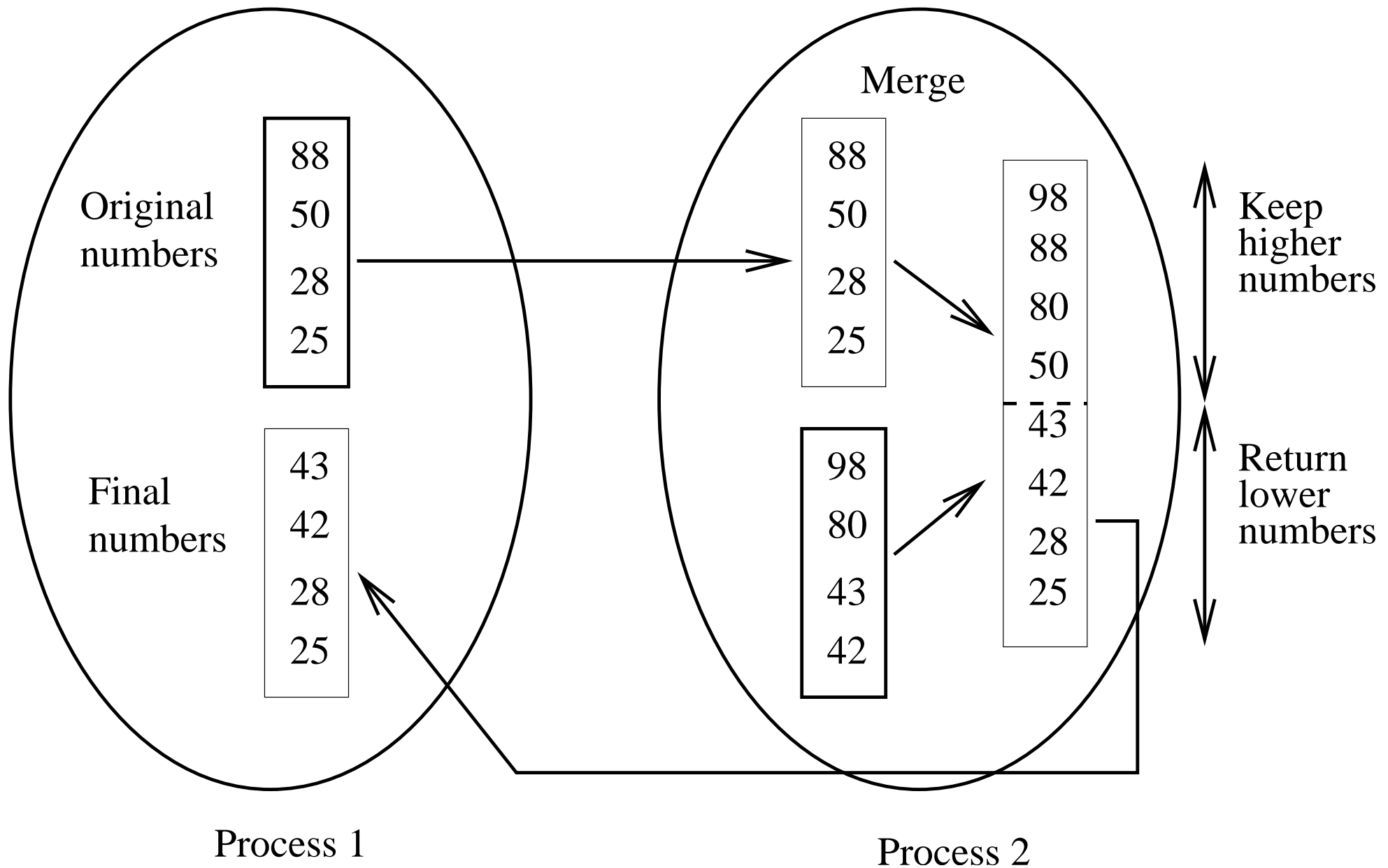


Data partitioning

Data partitioning:

- usually, the number n of numbers is much larger than the number p of processes
- in such cases, each process will handle a group of data, here a sorted sublist
- the algorithm is the same, but now each process
 - concatenate its list with that received from the other process,
 - then sort it, and
 - finally keep the corresponding (top or bottom) half of it

Merging two sublists





Bubble sort

Bubble sort:

- simple, but not too efficient, sequential algorithm

- sequential code;

```
for (i = n-1; i>0; i--)  
    for (j = 0; j<i; j++) {  
        k = j+1;  
        if (a[j] > a[k]) {  
            tmp = a[j];  
            a[j] = a[k];  
            a[k] = tmp;  
        }  
    }
```

- time complexity: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$



Parallel bubble sort

Parallel bubble sort: based on the idea that the bodies of the main loop may be overlapped

Odd-even transposition sort:

- Even phase (0,2,4,6,...)

$P_i \ (i=0, 2, \dots; \text{even}):$

`recv(&newA, P(i+1));`

`send(&A, P(i+1));`

`if(newA < A) A = newA;`

$P_i \ (i=1, 3, \dots; \text{odd}):$

`send(&A, P(i-1));`

`recv(&newA, P(i-1));`

`if(newA > A) A = newA;`

- Odd phase (1,3,5,...)

$P_i \ (i=2, 4, \dots; \text{even}):$

`recv(&newA, P(i-1));`

`send(&A, P(i-1));`

`if(newA > A) A = newA;`

$P_i \ (i=1, 3, \dots; \text{odd}):$

`send(&A, P(i+1));`

`recv(&newA, P(i+1));`

`if(newA < A) A = newA;`

parallel bubble sort

Example (sorting 8 numbers):

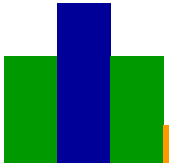
Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	\longleftrightarrow 2	7	\longleftrightarrow 8	5	\longleftrightarrow 1	3	\longleftrightarrow 6
1	2	4	\longleftrightarrow 7	8	\longleftrightarrow 1	5	\longleftrightarrow 3	6
2	2	\longleftrightarrow 4	7	\longleftrightarrow 1	8	\longleftrightarrow 3	5	\longleftrightarrow 6
3	2	4	\longleftrightarrow 1	7	\longleftrightarrow 3	8	\longleftrightarrow 5	6
4	2	\longleftrightarrow 1	4	\longleftrightarrow 3	7	\longleftrightarrow 5	8	\longleftrightarrow 6
5	1	2	\longleftrightarrow 3	4	\longleftrightarrow 5	7	\longleftrightarrow 6	8
6	1	\longleftrightarrow 2	3	\longleftrightarrow 4	5	\longleftrightarrow 6	7	\longleftrightarrow 8
7	1	2	\longleftrightarrow 3	4	\longleftrightarrow 5	6	\longleftrightarrow 7	8



Two dimensional sorting

Shearsort:

- the goal is to sort a two-dimensional array/mesh in *snakelike-style*, i.e., 1st line increasing, 2nd line decreasing, 3rd line increasing, and so on
- in *odd phases* rows are sorted in snakelike-style (alternated directions)
- in *even phases* columns are sorted increasingly from top to bottom (all columns are sorted in the same direction)
- result: after $\log n + 1$ phases the mesh is snakelike-style sorted



..Sheresort

Shearsort / Example:

4	14	8	2	2	4	8	14	1	4	7	3
10	3	13	16	16	13	10	3	2	5	8	6
7	15	1	5	1	5	7	15	12	11	9	14
12	6	11	9	12	11	9	6	16	13	10	15

Original numbers

Phase 1 - row sort

Phase 2 - col sort

1	3	4	7	1	3	4	2	1	2	3	4
8	6	5	2	8	6	5	7	8	7	6	5
9	11	12	14	9	11	12	10	9	10	11	12
16	15	13	10	16	15	13	14	16	15	14	13

Phase 3 - row sort

Phase 4 - col sort

Final phase - row sort

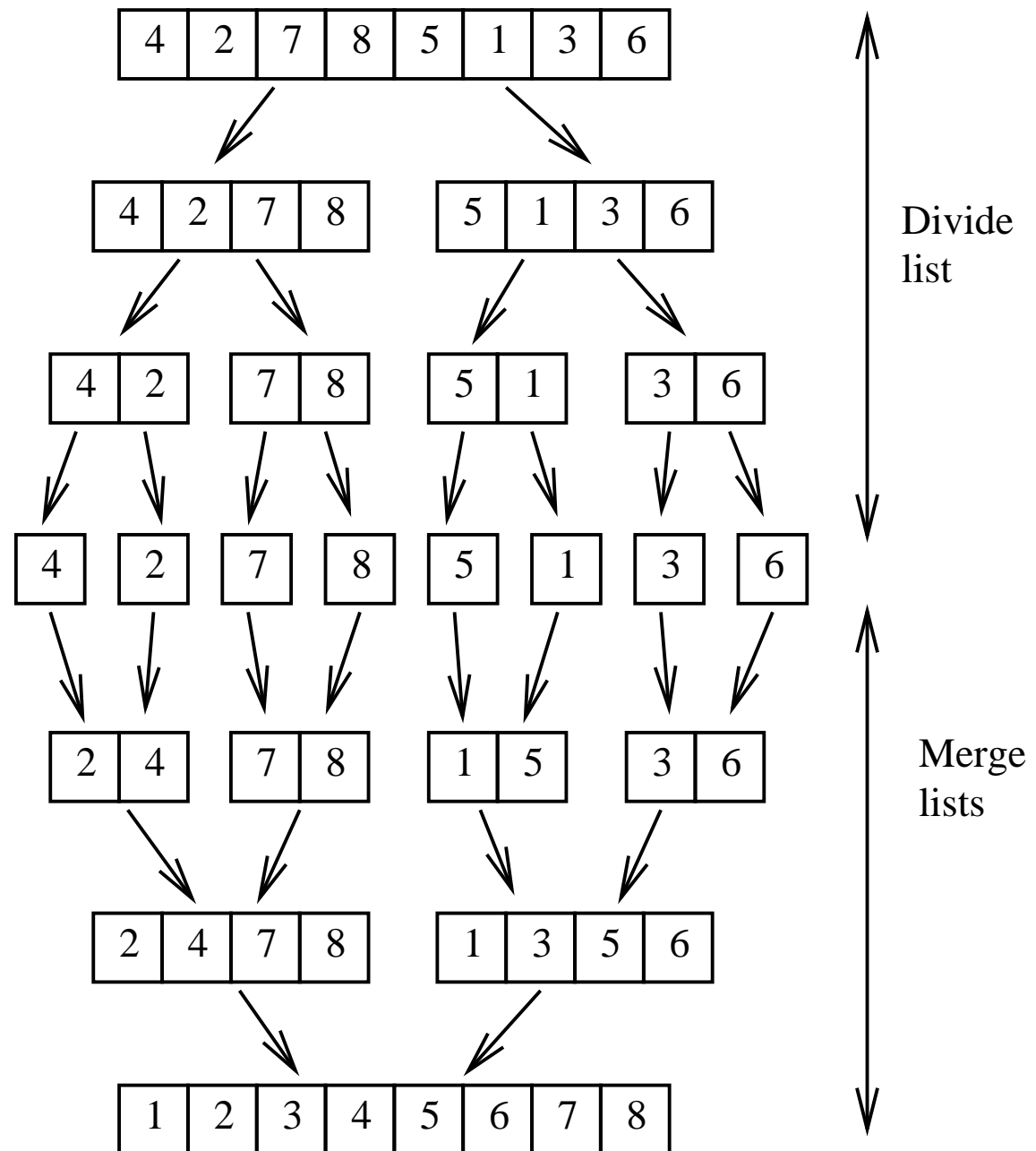
Using transpositions, one may arrange to use only line sorting (“transpose, then line sorting, then transpose” may replace column sorting).

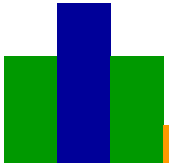
Mergesort

Mergesort:

is an optimal
 $O(n \log n)$ se-
quential sort-
ing algorithm;

an example:





..Mergesort

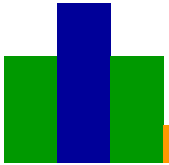
Mergesort / Analysis:

Sequential: $O(n \log n)$

Parallel: ($t_s = t_{startup}; t_d = t_{data}$)

- communication: splitting data using $\log n$ steps
 $(t_s + (n/2)t_d) + (t_s + (n/4)t_d) + (t_s + (n/8)t_d) + \dots$
and merging phase ($\log n$ steps, again)
 $\dots + (t_s + (n/8)t_d) + (t_s + (n/4)t_d) + (t_s + (n/2)t_d)$
total communication time: $t_{comm} \approx 2(\log n)t_{startup} + 2nt_{data}$
- computation (merging lists):
 $t_{comp} = \sum_{i=1}^{\log n} (2^i - 1) = O(n)$

The overall time complexity (using n processors) is $O(n)$.



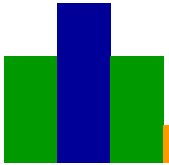
Quicksort

Quicksort:

- another optimal sequential sorting algorithm (sequential time complexity is $O(n \log n)$)
- select a number r , called *pivot*, and split the list into two sub-lists: one with all the elements at most equal to r , the other holding all the elements greater than r
- the procedure is recursively applied till one element lists are obtained (which are sorted)

- example:

4	2	7	8	5	1	3	6 [←]
3	→2	7	8	5	1	4	6
3	2	4	8	5	1 [←]	7	6
3	2	1	→8	5	4	7	6
3	2	1	4	5 [←]	8	7	6

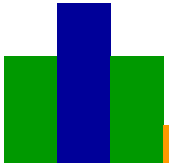


..Quicksort

Quicksort:

- sequential code: `list[]` holds the numbers; `pivot` is the index of the pivot

```
quicksort(list, start, end){  
    if (start < end) {  
        partition(list, start, end, pivot);  
        quicksort(list, start, pivot-1);  
        quicksort(list, pivot+1, end);  
    }  
}
```



..Quicksort

Parallelizing quicksort:

- the recursive shape of the algorithm suggests to apply a divide-and-conquer parallelization method
- the main problem of the approach is that the tree distribution (induced by the lengths of the sublists) heavily depends on pivot selection; in the worst case, the tree may consist of a single path
- analysis: provided an equal distribution of numbers within sublists is assured, one gets
 - computation: $t_{comp} = n + (n/2) + (n/4) + \dots \approx 2n$
 - communication: $(t_s + (n/2)t_d) + (t_s + (n/4)t_d) + (t_s + (n/8)t_d) + \dots \approx (\log n)t_s + nt_d$



Quicksort on a hypercube

Quicksort on a hypercube I: a root, say 00..0, is supposed to *hold all numbers*

- quicksort fits quite well for a hypercube implementation
- each processor receives a part of the list, divides it using a locally selected pivot, and submits the sublists to other nodes
- the selection of nodes where the sublists are to be submitted is similar to that used in the hypercube broadcast procedure (see next slide)



..Quicksort on a hypercube

Quicksort on a hypercube: example, using the standard binary coding of nodes

1st step:

000→100: 000 passes to 100 the numbers greater than p1 and keeps the others

2nd step:

000→010: 000 passes to 010 the numbers greater than p2 and keeps the others

100→110: 100 passes to 110 the numbers greater than p3 and keeps the others

3rd step:

000→001: 000 passes to 001 the numbers greater than p4 and keeps the others

010→011: 010 passes to 011 the numbers greater than p5 and keeps the others

100→101: 100 passes to 101 the numbers greater than p6 and keeps the others

110→111: 110 passes to 111 the numbers greater than p7 and keeps the others

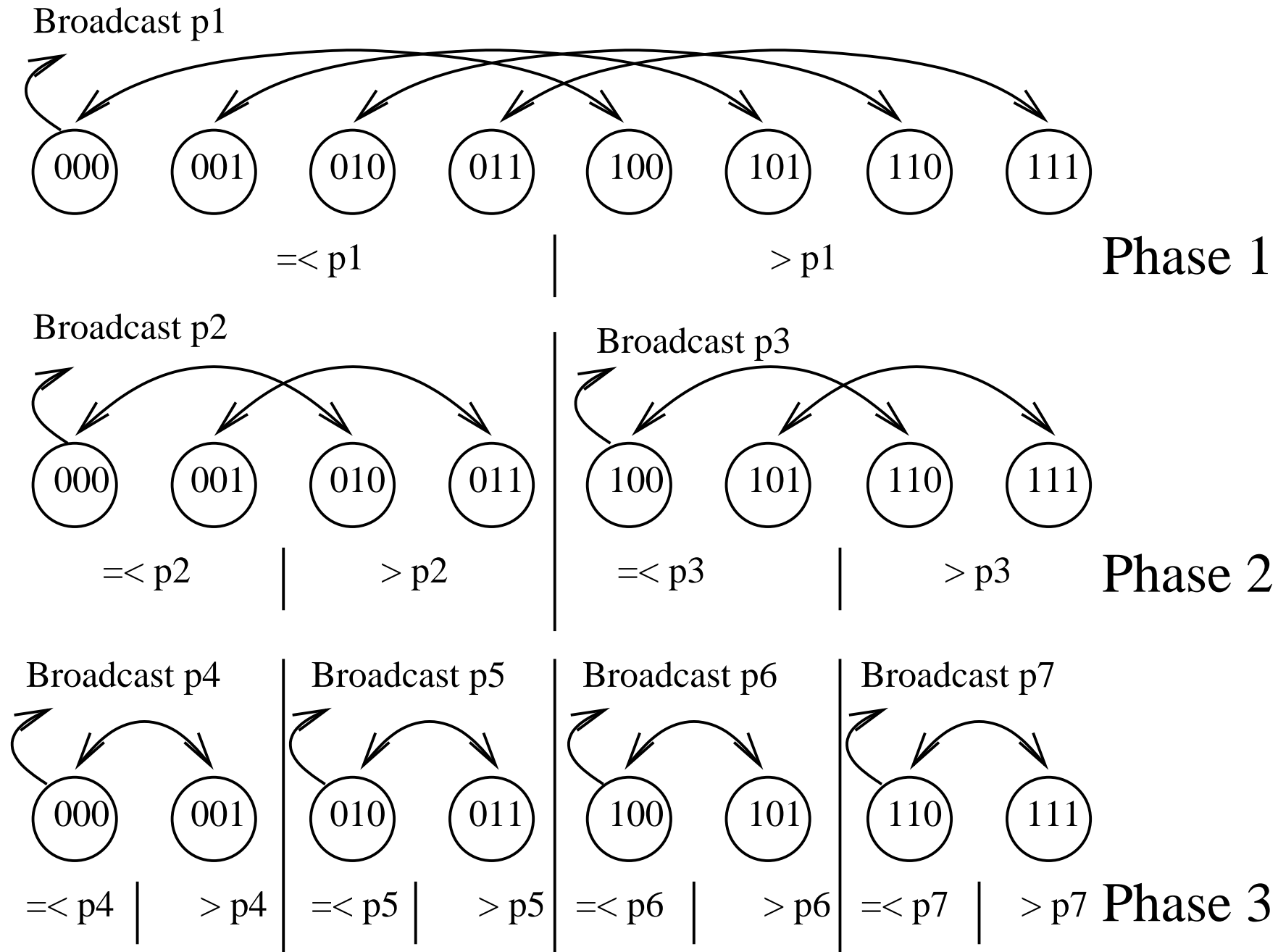


..Quicksort on a hypercube

Quicksort on a hypercube II: the list of numbers is *distributed* across all processors

- phase 1: —one processor, say $00..0$, selects a pivot p_1 and broadcast it to all nodes in the hypercube
—pairs of complementary nodes P, Q (namely, $P = 0i_2..i_k$ and $Q = 1i_2..i_k$) exchange numbers such that finally P holds all common numbers less than or equal to p_1 and Q holds those common numbers greater than p_1
- phase 2,3,...: repeat the above procedure for all the sub-hypercubes of smaller dimensions $k-1, k-2, ..$ (in each phase, each sub-hypercube selects its own pivot)
- finally, each node has a small list to sort; one gets the final sorted list concatenating these small sorted lists

..Quicksort on a hypercube





..Quicksort on a hypercube

Pivot selection:

- pivot selection is important for having equally loaded nodes
- in the sequential algorithm, usually the first number is selected;
- another possibility may be to take a sample, compute the mean value, and select the median as pivot;
- the latter is slightly more time consuming when the numbers are distributed across processes: one needs extra communication to select a sample
- the problems are simplified if each node keeps its small list *sorted*



Hyperquicksort

Hyperquicksort: quicksort on a hypercube with *distributed and sorted* lists across all processors

- each processor sorts its list sequentially
- the phases of the algorithm are as before (general quicksort on a hypercube), but
 - after the exchange of data, each node will merge its half list with the half list received to keep its numbers sorted

Analysis (hyperquicksort): suppose there are d dimensions (hence, $p = 2^d$ processors) and each processor initially holds n/p numbers

- computations: initial sorting $(n/p)\log(n/p)$ comparisons



Hyperquicksort

..Analysis (hyperquicksort):

- computation: pivot selection $O(1)$ (sorted lists, just take a middle list element)
- communication (pivot broadcast):
 - broadcast one pivot in a k hypercube $k(t_s + t_d)$
 - total $(d + \dots + 1)(t_s + t_d) = \frac{d(d-1)}{2}(t_s + t_d)$
- computation (split data using the pivot): for x numbers in a sorted list this requires $\log x$
- communication (exchange $x/2$ numbers): $2(t_s + (x/2)t_d)$
- computation (merge sorted sublists): this requires $x/2$ comparisons, if the longest list has $x/2$ numbers



Hyperquicksort

..Analysis (hyperquicksort):

- total time complexity: add the above items
- it is quite tedious to count the sum of the last 3 items (the lengths of the sublists is not known in advance; the processes are synchronized, hence only the longest time counts as the phase time)
- in the ideal case when all list splits are into equal sublists, just add the last 3 items for x equal to $(n/p), (n/p)/2, (n/p)/4, \dots$



Odd-even mergesort

Odd-even mergesort:

- the basic step is to merge two sorted lists a_1, \dots, a_n and b_1, \dots, b_n into one sorted list using the following parallel method
 - merge the elements of odd indices of each list
 - merge the elements of even indices of each list (to be done in parallel with the previous action)
 - finally, compare-and-exchange (in parallel) positions $(2, n+1), (3, n+2), \dots, (n, 2n-1)$.
- the lists should be of equal lengths
- moreover, usually n is a power of 2 in order to apply recursively the basic step



..Odd-even mergesort

Odd-even mergesort Example: - basic step

2, 4, **5**, 8 **1**, 3, **6**, 7

1, **2**, **5**, **6** **3**, **4**, **7**, 8

1, 2, 3, 4, 5, 6, 7, 8



Bitonic mergesort

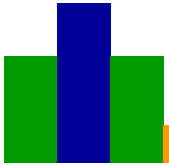
Bitonic sequences: a *bitonic sequence* of numbers is

- a sequence consisting of two subsequences (of consecutive numbers), one increasing and one decreasing; e.g.,

3, 5, 8, **19**, 17, 14, 12, 11 (*)

- or a sequence which may be brought to such a form (*) by a circular shifting of the elements of the sequence; e.g.,

12, **11**, 3, 5, 8, 19, 17, 14



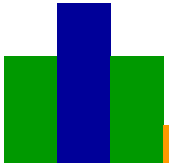
..Bitonic mergesort

Bitonic sequences have the following useful mathematical property:

- if one performs compare-and-exchange of elements $(1, 1 + n/2), (2, 2 + n/2), \dots$ of a bitonic sequence a_1, \dots, a_n , then
 - two bitonic subsequences $b_1, \dots, b_{n/2}$ and $b_{(n/2)+1}, \dots, b_n$ are obtained and
 - all numbers in one subsequence are less than all numbers in the other

Example: 12, 11, 3, 5, **8, 19, 17, 14**

8, 11, 3, 5, 12, 19, 17, 14



..Bitonic mergesort

Bitonic mergesort: the above observations lead to the following sorting method

1. Sorting a bitonic sequence (SBS): by a repeated application of the procedure explained in the above example one gets a sorted list

2. Creating bitonic sequences: here we can apply the same procedure (SBS)

- start with two-element lists (they are bitonic)
- sort adjacent lists (using SBS), but in opposite directions
- concatenate two adjacent lists to get a longer bitonic sequence;
- repeat the procedure till the full list becomes a bitonic sequence



..Bitonic mergesort

Bitonic mergesort / Example:

8 3 4 7 9 2 1 5

1. given list; mark starting 2-elem bitonic sublists

[8 3] [4 7] [9 2] [1 5]

2. alternate (increasing - decreasing) sorting of 2-elem bitonic lists

[3 8] [7 4] [2 9] [5 1]

3. mark obtained 4-elem bitonic lists

[3 8 7 4] [2 9 5 1]

4. alternate (increasing - decreasing) sorting of 4-elem bitonic lists

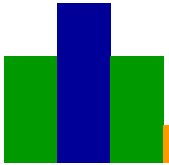
[3 4 7 8] [9 5 2 1]

5. mark obtained 8-elem bitonic list

[3 4 7 8 9 5 2 1]

6. sort (increasing) the 8-elem bitonic list

1 2 3 4 5 7 8 9



..Bitonic mergesort

..Bitonic mergesort / Example:

- Sorting of bitonic sequences (of steps 2,4,6) is explained here by expanding into microsteps the last case (step 6):

[3 4 7 8 9 5 2 1]

6.1. given 8-elem bitonic list; split into two 4-elem bitonic sublists

[3 4 2 1] [9 5 7 8]

6.2. split each 4-elem bitonic sublists into two 2-elem bitonic sublists

[2 1] [3 4] [7 5] [9 8]

6.3. sort (increasing) all 2-elem bitonic sublists

1 2 3 4 5 7 8 9



..Bitonic mergesort

Analysis (bitonic mergesort):

- with $n = 2^k$, there are k phases, each involving $1, 2, \dots, k$ steps, respectively;
- the total number of steps is

$$\sum_{i=1}^k i = \frac{k(k+1)}{2} = O(k^2) = O(\log^2 n)$$

Lesson 10: Numerical algorithms

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004



Matrices

Matrices:

- an *$m \times n$ matrix* A is a two-dimensional array with m rows and n columns (usually the entries are numbers); e.g.,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\ \dots & \dots & \dots & \dots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,n-1} \end{bmatrix}$$

- we briefly denote a matrix by $A = (a_{i,j})_{0 \leq i < m, 0 \leq j < n}$ (when no confusion arises, the specification of the index range is omitted)



..Matrices

Operations on matrices:

- *addition*: given two matrices $A = (a_{i,j})$ and $B = (b_{i,j})$ of the same size $m \times n$, their sum $A + B$ is a matrix of the same size $C = (c_{i,j})$ defined by element-wise addition

$$c_{i,j} = a_{i,j} + b_{i,j}, \quad \forall 0 \leq i < m, 0 \leq j < n$$

- *multiplication*: let $A = (a_{i,j})$ be an $m \times n$ matrix and $B = (b_{i,j})$ be an $n \times p$ matrix; their product $A \cdot B$ is the $m \times p$ matrix $C = (c_{i,j})$ defined by “row-column multiplication”

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j}, \quad \forall 0 \leq i < m, 0 \leq j < p$$



..Matrices

(Operations on matrices, cont.)

- *matrix-vector multiplication*: given an $n \times n$ matrix $A = (a_{i,j})$ and a (column) vector $b = (b_i)_{0 \leq i < n}$ (i.e., an $n \times 1$ matrix)

$$c = A \cdot b$$

(the matrix multiplication of A and b) is a column vector again;

- a system of linear equations

$$a_{0,0}x_0 + \dots + a_{0,n-1}x_{n-1} = b_0$$

$$\vdots$$

$$a_{n-1,0}x_0 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

may be written in a simple form

$$A \cdot x = b$$

for an appropriate matrix A and corresponding vectors x, b



Implementing matrix multiplication

Sequential code:

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

Sequential time complexity: $O(n^3)$ ($2n^3$ arithmetical operations)



Parallel matrix multiplication

Computational parallel time complexity:

- with n processors: $O(n^2)$
- with n^2 processors: $O(n)$
(one processor for each pair (i, j))
- with n^3 processors (using a divide-and-conquer parallel procedure for the inner loop): $O(\log n)$

Notice: no communication time is included in these estimations; also, using n^3 processors to multiply $n \times n$ matrices is not a realistic assumption; usually, each processor handle blocks of sub-matrices



Partitioning into sub-matrices

Suppose a matrix is divided into s^2 sub-matrices, each having the same size $(n/s) \times (n/s)$.

Code for “block matrix multiplication”: denote by $A_{p,q}$ the (p,q) sub-matrix of this partition; similarly for B and C ; then

```
for (p=0; p<s; p++)  
  for (q=0; q<s; q++) {  
     $C_{p,q} = 0;$   
    for (r=0; r<s; r++)  
       $C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};$   
  }
```

The line of code $C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}$ denotes operations involving multiplication and addition of sub-matrices.

..Partitioning into sub-matrices

Example:

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array} \right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array} \right]$$

$$= \left[\begin{array}{cc|cc} \left[\begin{array}{cc} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{array} \right] \times \left[\begin{array}{cc} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{array} \right] & & \left[\begin{array}{cc} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{array} \right] \times \left[\begin{array}{cc} b_{0,2} & b_{0,3} \\ b_{1,2} & b_{1,3} \end{array} \right] \\ + \left[\begin{array}{cc} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{array} \right] \times \left[\begin{array}{cc} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{array} \right] & & + \left[\begin{array}{cc} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{array} \right] \times \left[\begin{array}{cc} b_{2,2} & b_{2,3} \\ b_{3,2} & b_{3,3} \end{array} \right] \\ \hline \left[\begin{array}{cc} a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{array} \right] \times \left[\begin{array}{cc} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{array} \right] & & \left[\begin{array}{cc} a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{array} \right] \times \left[\begin{array}{cc} b_{0,2} & b_{0,3} \\ b_{1,2} & b_{1,3} \end{array} \right] \\ + \left[\begin{array}{cc} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{array} \right] \times \left[\begin{array}{cc} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{array} \right] & & + \left[\begin{array}{cc} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{array} \right] \times \left[\begin{array}{cc} b_{2,2} & b_{2,3} \\ b_{3,2} & b_{3,3} \end{array} \right] \end{array} \right]$$

= ...



Direct implementation

Direct implementation, using n^2 processors:

- one process $P_{i,j}$ is used for each $C_{i,j}$
- process $P_{i,j}$ needs the i -th row of A and the j -th column of B
- data are separately sent to processes $P_{i,j}$
- alternatively, one may broadcast one row i to all processes $P_{i,j} (0 \leq j < n)$; similarly for columns



Analysis

Analysis (direct implementation, using n^2 processors):

Communication

- with separate messages to each of the n^2 slave processes (including row, column, and result)

$$t_{comm} = n^2(t_s + 2nt_d) + n^2(t_s + t_d) = n^2(2t_s + (2n + 1)t_d)$$

- with a unique broadcast of the full matrices A and B

$$t_{comm} = (t_s + 2n^2t_d) + n^2(t_s + t_d)$$

(now, returning the results becomes dominant)

Computation

- $t_{comp} = 2n$ (n additions and n multiplications)



Performance improvement

Performance improvement:

- using a tree construction n numbers can be added in $\log n$ steps using n processors
- computational time complexity decreases to $O(\log n)$ using n^3 processors.

Recursive algorithm

- we have seen that block matrix multiplication is based on multiplication (and addition) of smaller matrices
- the same procedure may be used recursively to multiply these smaller matrices



Recursive algorithm

```
matMult(A, B, s) {  
    if ( s == 1)  
        C = A * B  
    else {  
        s = s/2;  
        P0 = matMult (App, Bpp, s);  
        P1 = matMult (Apq, Bqp, s);  
        P2 = matMult (App, Bpq, s);  
        P3 = matMult (Apq, Bqq, s);  
        P4 = matMult (Aqp, Bpp, s);  
        P5 = matMult (Aqq, Bqp, s);  
        P6 = matMult (Aqp, Bpq, s);  
        P7 = matMult (Aqq, Bqq, s);  
        Cpp = P0 + P1;  
        Cpq = P2 + P3;  
        Cqp = P4 + P5;  
        Cqq = P6 + P7;  
    }  
}
```

Notice: s denotes the size of the blocks (number of rows). The number of processes is 8^k , so typically one takes 8 processes.



Mash implementations / Cannon's algorithm

Cannon's algorithm: use a *torus* (mash with wraparound connections) and shift the elements/blocks of A left and those of B up.

1. initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ ($0 \leq i, j < n$)
2. shift i -th row i places left and j -th column j places upward (the effect is that $P_{i,j}$ holds $a_{i,(j+i \bmod n)}$ and $b_{(j+i \bmod n),j}$ which are used for computing $c_{i,j}$)
3. each processor $P_{i,j}$ multiplies its elements
4. the i -th row of A is shifted one place left and the j -th column of B is shifted one place upward
5. each processor $P_{i,j}$ multiplies its elements and adds the result to the accumulating sum
6. the last 2 steps are repeated till the final result is obtained.

..Cannon's algorithm

Cannon's algorithm, example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}; B = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

After initial shift,

$$A \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 4 \\ 9 & 7 & 8 \end{bmatrix}; B \rightarrow \begin{bmatrix} a & e & i \\ d & h & c \\ g & b & f \end{bmatrix}; \text{ so } C \rightarrow \begin{bmatrix} 1a & 2e & 3i \\ 5d & 6h & 4c \\ 9g & 7b & 8f \end{bmatrix}$$

2nd shift,

$$A \rightarrow \begin{bmatrix} 2 & 3 & 1 \\ 6 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}; B \rightarrow \begin{bmatrix} d & h & c \\ g & b & f \\ a & e & i \end{bmatrix}; \text{ so } C \rightarrow \begin{bmatrix} 1a + 2d & 2e + 3h & 3i + 1c \\ 5d + 6g & 6h + 4b & 4c + 5f \\ 9g + 7a & 7b + 8e & 8f + 9i \end{bmatrix}$$

3rd shift,

$$A \rightarrow \begin{bmatrix} 3 & 1 & 2 \\ 4 & 5 & 6 \\ 8 & 9 & 7 \end{bmatrix}; B \rightarrow \begin{bmatrix} g & b & f \\ a & e & i \\ d & h & c \end{bmatrix}; \text{ so } C \rightarrow \begin{bmatrix} 1a + 2d + 3g & 2e + 3h + 1b & 3i + 1c + 2f \\ 5d + 6g + 4a & 6h + 4b + 5e & 4c + 5f + 6i \\ 9g + 7a + 8d & 7b + 8e + 9h & 8f + 9i + 7c \end{bmatrix}$$



Analysis / Cannon's algorithm

Analysis / Cannon's algorithm: suppose s^2 blocks of size $m \times m$ are used (hence $n = ms$)

Communication

- initial alignment requires maximum $s - 1$ shifts (communication operations)
- after that, there will be $s - 1$ more shift operations
- each shift operation involves $m \times m$ matrices, hence

$$t_{comm} = 2(s - 1)(t_{startup} + m^2 t_{data})$$

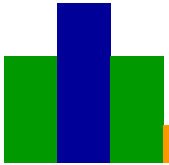
- the communication time complexity is $O(sm^2)$ or $O(mn)$

Computation

- each sub-matrix multiplication (done on a processor) requires m^3 multiplications and m^3 additions; the final summation is lighter: m^2 additions
- as there are $s - 1$ shifts and an initial multiplication step

$$t_{comp} = 2s m^3$$

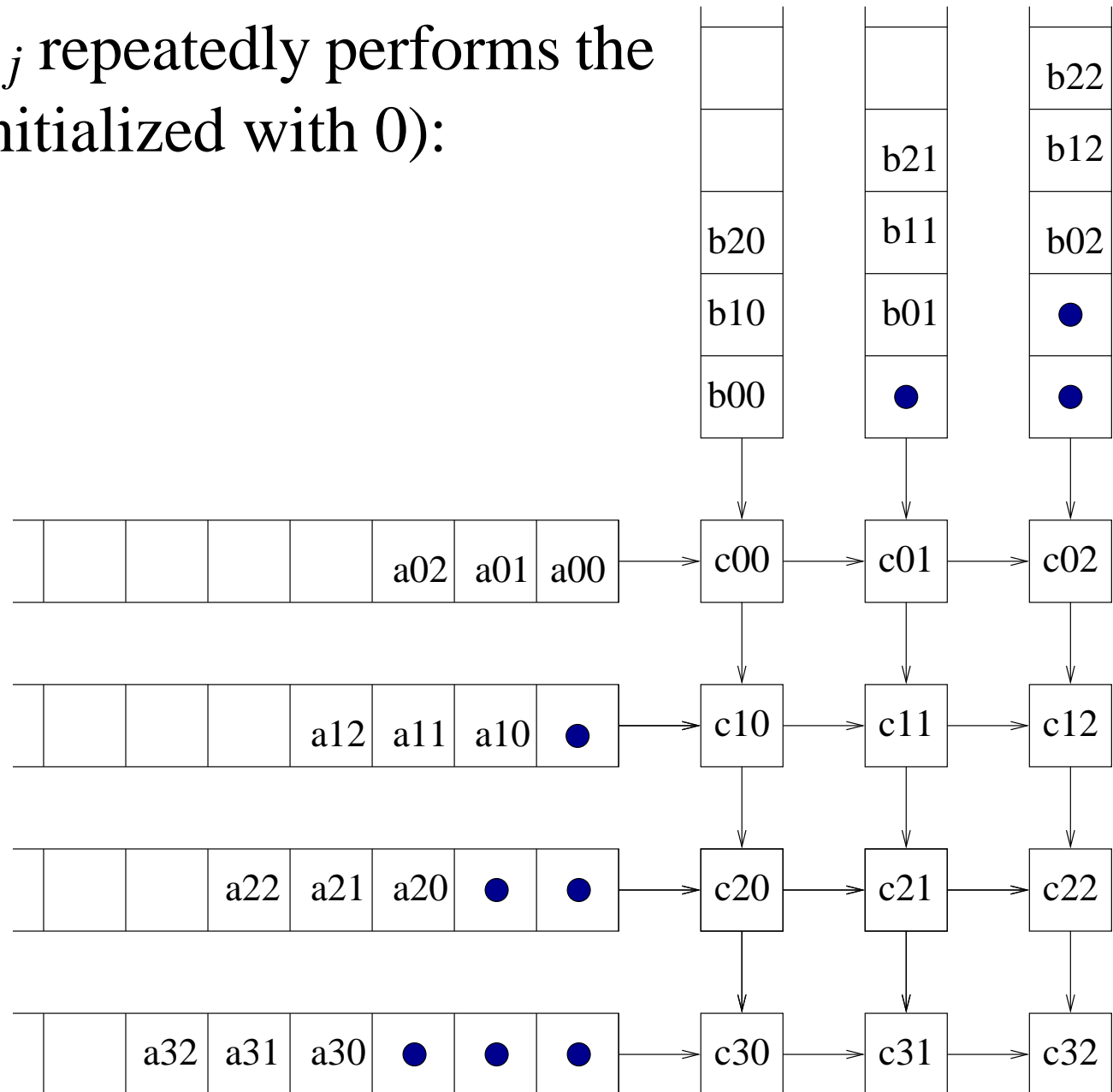
- hence the computational time complexity is $O(m^2n)$



Systolic array

Code: each processor $P_{i,j}$ repeatedly performs the following actions (c is initialized with 0):

```
recv(&a,  $P_{i,j-1}$ ) ;  
recv(&b,  $P_{i-1,j}$ ) ;  
c = c + a * b ;  
send(&a,  $P_{i,j+1}$ ) ;  
send(&b,  $P_{i+1,j}$ ) ;
```





Systems of linear equations

Systems of linear equations:

- a general system of linear equations has the following form

$$\begin{array}{ccccccc} a_{0,0}x_0 + & a_{0,1}x_1 + & a_{0,2}x_2 + \dots + & a_{0,n-1}x_{n-1} = & b_0 \\ a_{1,0}x_0 + & a_{1,1}x_1 + & a_{1,2}x_2 + \dots + & a_{1,n-1}x_{n-1} = & b_1 \\ & \vdots & & & \\ a_{n-1,0}x_0 + & a_{n-1,1}x_1 + & a_{n-1,2}x_2 + \dots + & a_{n-1,n-1}x_{n-1} = & b_{n-1} \end{array}$$

- the matrix form is

$$Ax = b$$

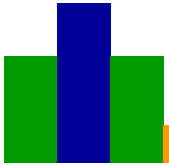
- the goal is to find the unknowns of vector x , given values for matrix A and vector b .



Gaussian elimination

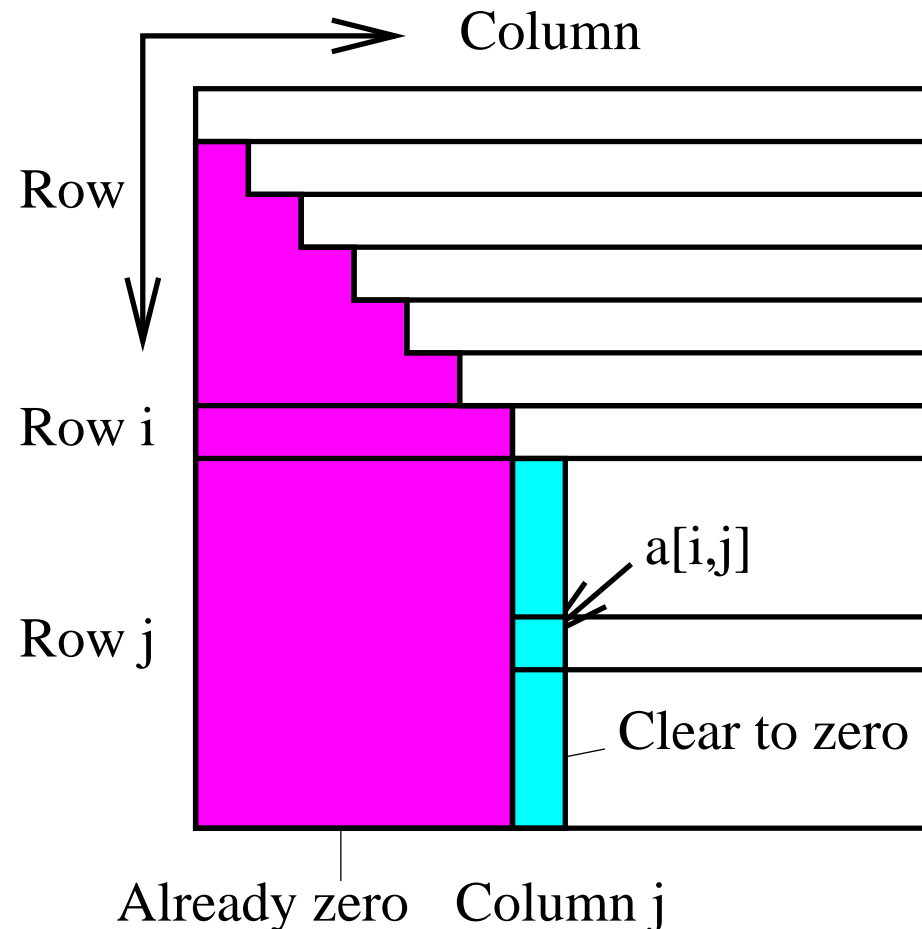
Gaussian elimination:

- convert the general system of linear equations into a *triangular system* of equations, then use *back-substitution* to solve it
- key property: the solution is not changed if any row is replaced by that row added with another row multiplied by a constant
- this leads to the following procedure:
 - start from the first row and work toward the bottom row
 - at the i -th row, each row j below it is replaced by $row\ j + (row\ i)(-a_{j,i}/a_{i,i})$; i.e., $a_{j,k} := a_{j,k} + a_{i,k} \left(\frac{-a_{j,i}}{a_{i,i}} \right)$, for all $k \geq i$
 - this way all elements of column i , below the i -th row, become zero; indeed, $a_{j,i} + a_{i,i} \left(\frac{-a_{j,i}}{a_{i,i}} \right) = 0$



..Gaussian elimination

Gaussian elimination:





Partial pivoting

Partial pivoting

- if $a_{i,i}$ is zero or close to zero, the value $-a_{j,i}/a_{i,i}$ can not be accurately computed
- a *partial pivoting* technique may be used to avoid such a situation:
 - swap the i -th row with the row below it that has the *largest* element (absolute value) in column i from all row below the i -th row

(reordering of equations does not affect the system - notice that each row i includes the corresponding free term b_i)



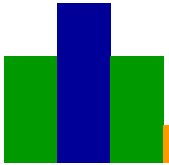
Sequential code

Sequential code

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++) {
        m = a[j][i]/a[i][i];
        for (k=i; k<n; k++)
            a[j][k] = a[j][k] - a[i][k] * m;
        b[j] = b[j] - b[i] * m;
    }
```

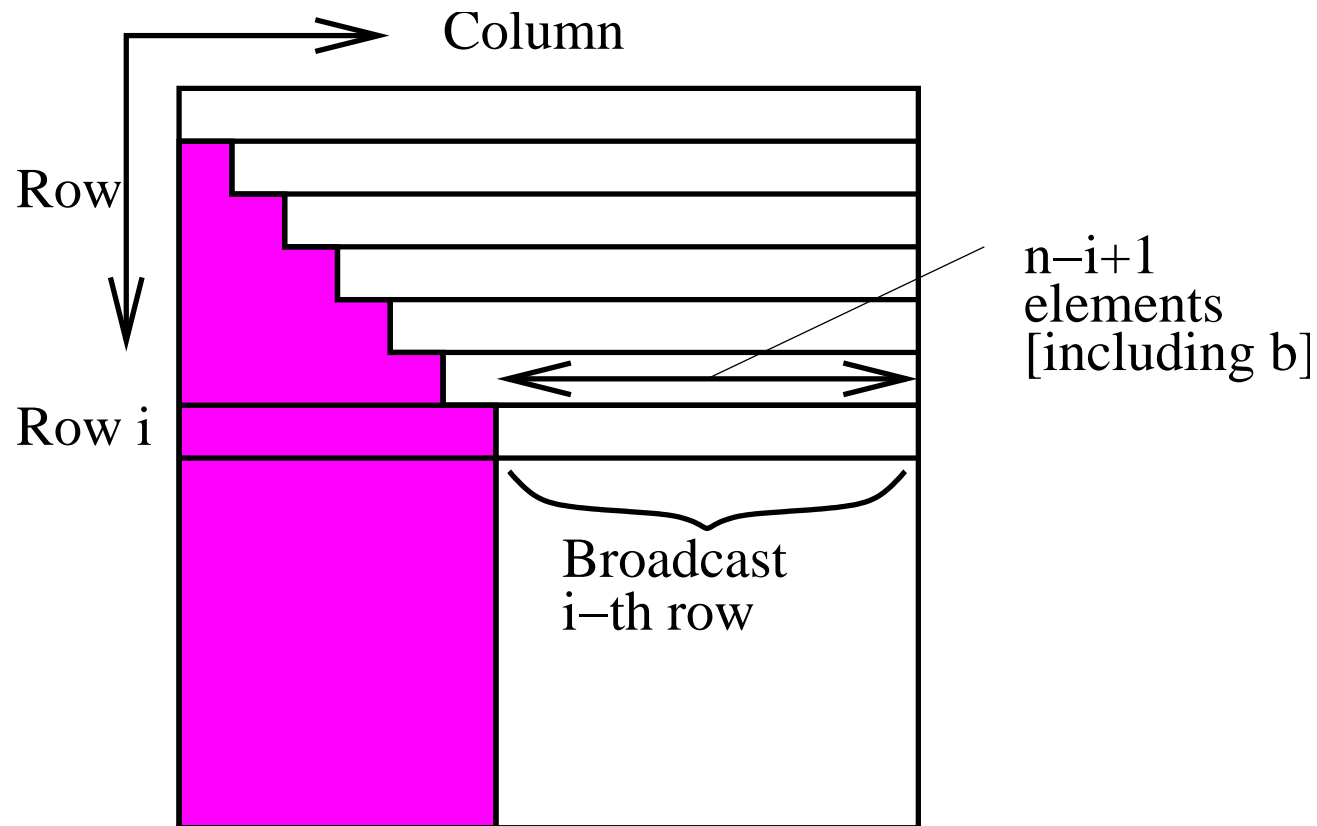
The time complexity is $O(n^3)$

(No pivoting method is considered here.)



..Gaussian elimination

Gaussian elimination (parallel):





Analysis, parallel algorithm

Analysis, parallel algorithm:

Communication

- using n processes (one for each row)
- using broadcast to pass row i (namely, $a[i][i..n-1]$) and b_i to processes $i+1, \dots, n-1$; we have to pass $n-i+1$ data

- hence,

$$\begin{aligned} t_{comm} &= \sum_{i=0}^{n-2} (t_{s-bcast} + (n-i+1)t_{d-bcast}) \\ &= (n-1)t_{s-bcast} + (3+4+\dots+(n+1))t_{d-bcast} \\ &= (n-1)t_{s-bcast} + ((n+1)(n+2)/2 - 3)t_{d-bcast} \end{aligned}$$

- the communication time complexity is $O(n^2)$



..Analysis, parallel algorithm

Computation:

- after broadcast, each process (below i) computes the multiplier and then update $n - i + 1$ elements - each updating requires 2 operations

- hence,

$$\begin{aligned} t_{comp} &= \sum_{i=1}^{n-1} (2(n - i + 1) + 1) \\ &= n - 1 + 2(n(n + 1)/2 - 1) = O(n^2) \end{aligned}$$

Notice: For this algorithm processes may be mapped onto a pipeline configuration where different stages of computation may overlap.



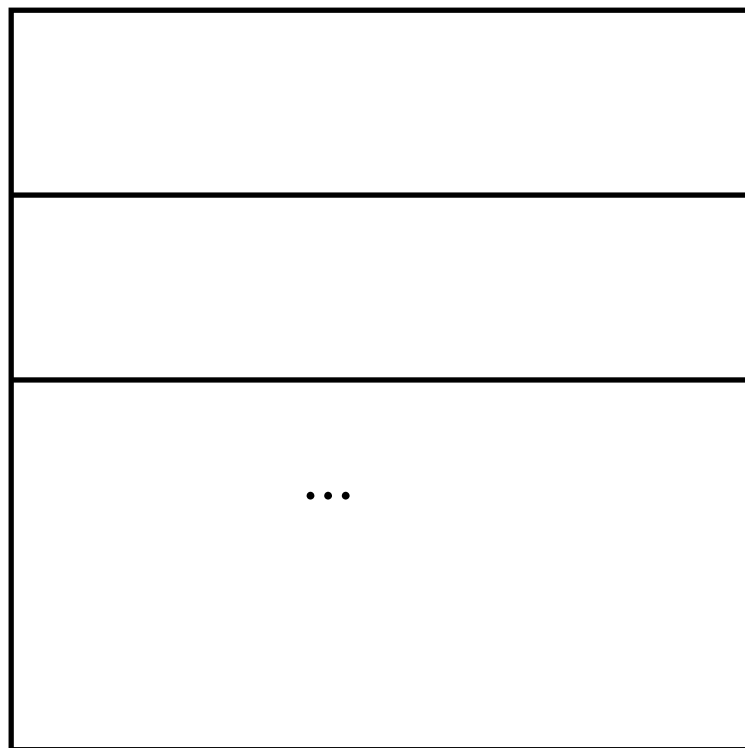
Partitioning

Partitioning:

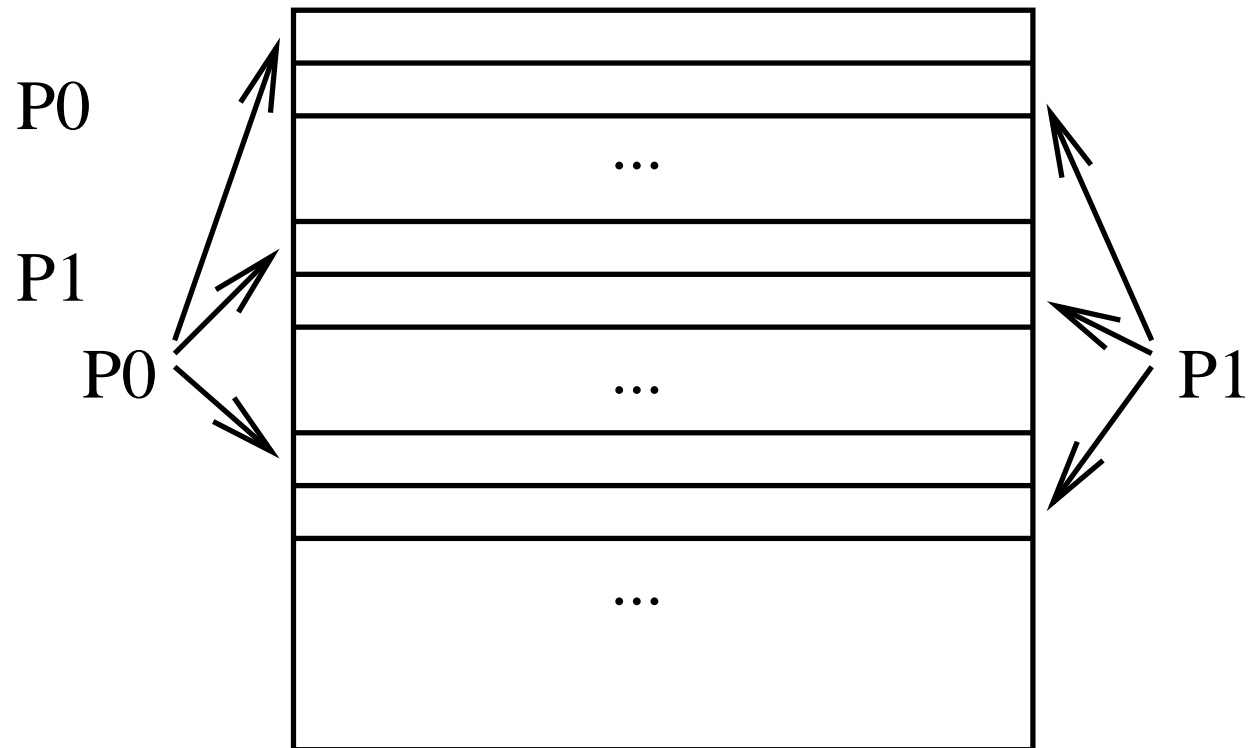
- *strip partitions* may be used when less processors are available
- ... but the work is not well balanced (the processors handling the top rows do less work - they do not participate in computation after their last row is processed)
- *cyclic-strip partitions* provide more balanced work:
if there are p processes, in this strategy
 - P_0 handles rows $0, n/p, 2n/p, \dots$,
 - P_1 rows $1, n/p + 1, 2n/p + 1, \dots$,
 - etc.

..higher order difference methods

Cyclic vs. strip partitions:



Strip partitions



Cyclic partitions



Iterative methods

Jacobi iteration:

- by rearranging the equations of a system of linear equations we get

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right)$$

- this relation may be used to get a approximating iterative method

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right)$$

where the superscript indicates iteration:

— x_i^k is x_i at the k -th iteration;

—it is computed using the x_j 's ($j \neq i$) from the previous iteration



..Iterative methods

Rough comparison (direct vs. iterative method):

- the direct method mainly requires $O(n^2)$ computation steps, when n processors are to be used
- in the iterative algorithm with n processors, one iteration is relatively light: $n + 1$ arithmetical operations are used;
- ... but the overall computation time complexity depends on the *number of iterations* needed to get the result with the required accuracy
- iterative methods are particularly useful for *sparse linear equations* (i.e., when there are many variables, but each equation constrains only a few of them)



Laplace's equation

Laplace's equation:

- if f is a function with two arguments x and y , the aim is to solve

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

- usually, this is to be solved within a finite area of a two-dimensional vector space, knowing the values at the border
- *finite difference* method is generally used for a computer solution: the space is *discretized* and Laplace's equation is transformed into a system of linear equations



..Laplace's equation

(Laplace's equation, cont.)

- if the distance between neighboring points in both x and y directions is small, say Δ , then

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2} (f(x + \Delta, y) - 2f(x, y) + f(x - \Delta, y))$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2} (f(x, y + \Delta) - 2f(x, y) + f(x, y - \Delta))$$

- use these approximations into Laplace's equation; one gets

$$\frac{1}{\Delta^2} (f(x + \Delta, y) + f(x - \Delta, y) + f(x, y + \Delta) + f(x, y - \Delta) - 4f(x, y)) = 0$$

- this leads to the following iterative formula

$$f^k(x, y) = (1/4) (f^{k-1}(x + \Delta, y) + f^{k-1}(x - \Delta, y) + f^{k-1}(x, y + \Delta) + f^{k-1}(x, y - \Delta))$$

for computing the values $f^k(x, y)$ at the k -th iteration using the values $f^{k-1}(\dots)$ of the previous iteration



..Laplace's equation

The result is just a system of linear equations. Indeed,

- for a square area with $n \times n$ points and the natural order (left-to-right, top to bottom), the equations are

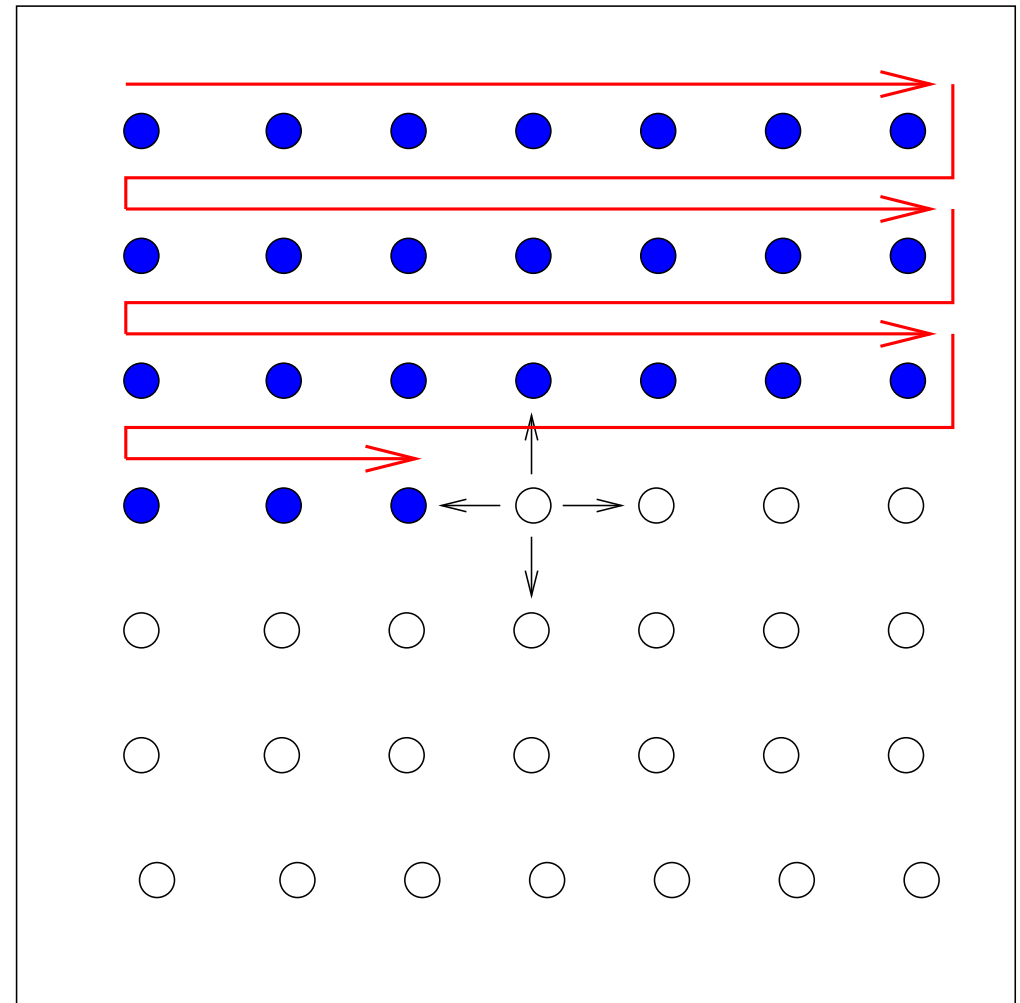
$$x_{i-n} + x_{i-1} - 4x_i + x_{i+1} + x_{i+n} = 0$$

- with usual pair-notation for indexes, the equations may be written as

$$x_{i-1,j} + x_{i,j-1} - 4x_{i,j} + x_{i,j+1} + x_{i+1,j} = 0$$

- near the border some variables are replaced by the corresponding known values (constants); these are collected in the constant b_i 's terms of the system
- a system of sparse equations is obtained

Faster convergence methods



Gauss-Seidel relaxation:

Notice: This order is for a sequential approach; for parallel methods other orders may be used.



..Gauss-Seidel relaxation

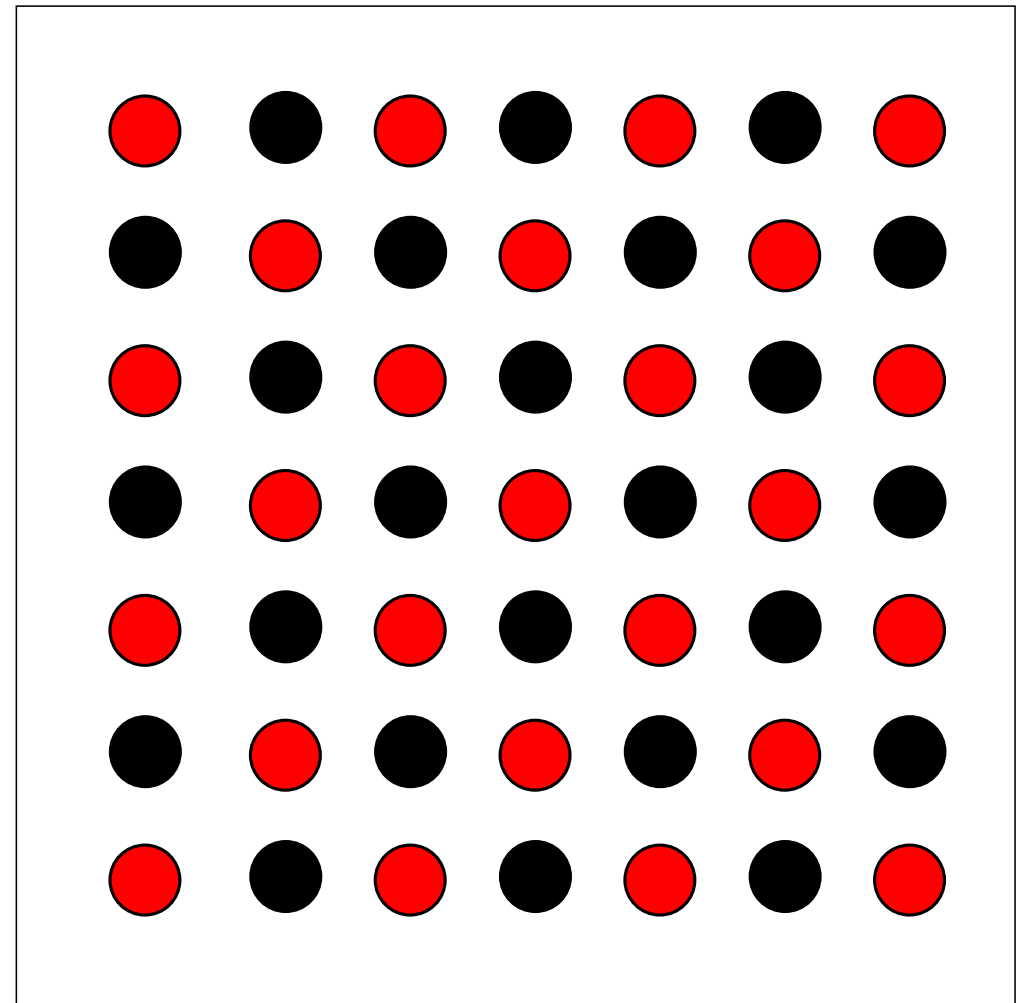
- this is a particular iterative method for system of linear equations
- mix old and newly computed values, according to the following scheme

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=0}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n-1} a_{i,j} x_j^{k-1} \right)$$

- for Laplace's equation, this scheme produces the following result

$$f^k(x, y) = (1/4) (f^k(x - \Delta, y) + f^k(x, y - \Delta) + f^{k-1}(x + \Delta, y) + f^{k-1}(x, y + \Delta))$$

Relaxation / Red-black ordering



Relaxation / Red-black ordering:

Notice: This order is for a parallel approach.



Red-black parallel code

Red-black parallel code:

```
forall (i=1; i<n; i++)
    forall (j=1; j<n; j++)
        if ((i+j)%2 == 0)
            f[i][j] = 0.25 * (f[i-1][j] + f[i][j-1]
                               + f[i+1][j] + f[i][j+1]);

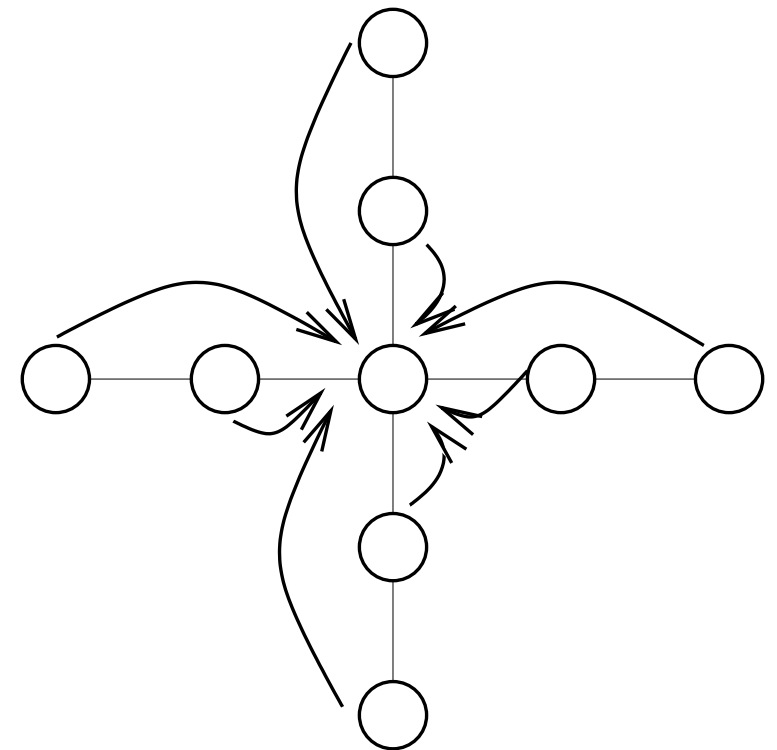
forall (i=1; i<n; i++)
    forall (j=1; j<n; j++)
        if ((i+j)%2 != 0)
            f[i][j] = 0.25 * (f[i-1][j] + f[i][j-1]
                               + f[i+1][j] + f[i][j+1]);
```

Higher order difference methods

In this case more distant points have to be used in the computation; an example is:

$$f^k(x, y) = (1/60)[16f^{k-1}(x - \Delta, y) + 16f^{k-1}(x, y - \Delta) + 16f^{k-1}(x + \Delta, y) + 16f^{k-1}(x, y + \Delta) - f^{k-1}(x - 2\Delta, y) - f^{k-1}(x, y - 2\Delta) - f^{k-1}(x + 2\Delta, y) - f^{k-1}(x, y + 2\Delta)]$$

—in this scheme eight nearby points (two points in each direction) are used to update a point





Over-relaxation

Over-relaxation:

- an improved convergence is obtained by including the values of the current point, i.e., adding a factor $(1 - \omega)x_i$ to Jacobi or Gauss-Seidel formulas; ω is called *over-relaxation parameter*
- *Jacobi over-relaxation formula* is

$$x_i^k = \frac{\omega}{a_{i,i}} \left(b_i - \sum_{j \neq i} x_j^{k-1} \right) + (1 - \omega)x_i^{k-1}$$

where $0 < \omega < 1$.



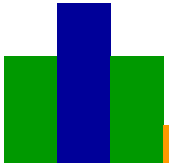
..Over-relaxation

(Over-relaxation, cont.)

- *Gauss-Seidel over-relaxation formula* is

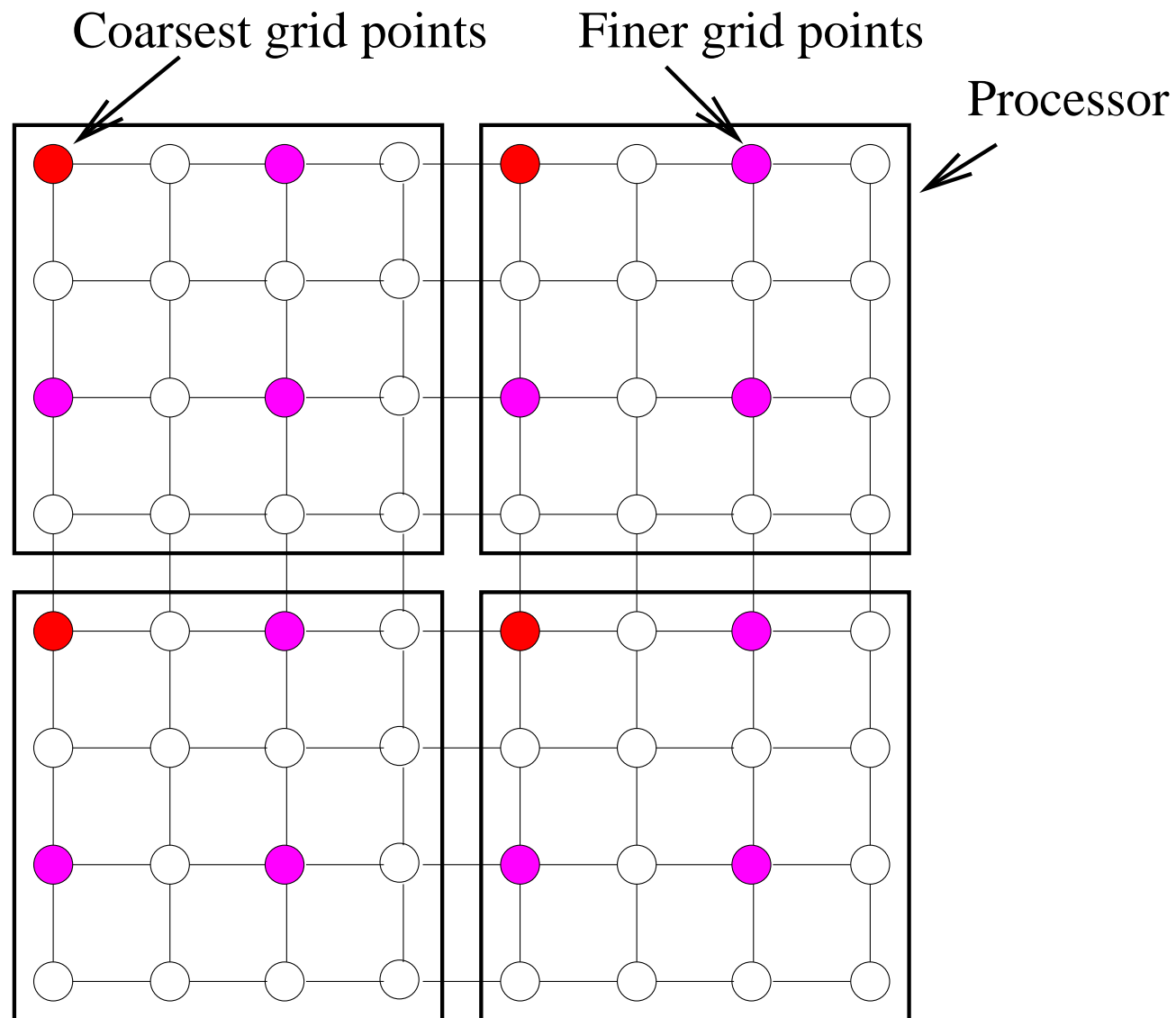
$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=0}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n-1} a_{i,j} x_j^{k-1} \right) + (1 - \omega) x_i^{k-1}$$

where $0 < \omega \leq 2$ (when $\omega = 1$, Gauss-Seidel method is obtained)



Multi-grid method

Multi-grid processor allocation:





..Multi-grid method

Multi-grid method:

- first, a *coarse grid* of points is used
 - with these points, the iteration process will start to *converge quickly*
 - .. but only a rough approximation for the starting problem is obtained
- at some stage
 - the *number of points* is *increasing*, including extra point between the points of the coarse grid;
 - the initial values of these extra points are obtained by *interpolation*;
 - the computation continues in this finer grid



..Multi-grid method

(Multi-grid method, cont.)

- the grid may be made finer and finer, or computation may alternate between fine and coarse grids
- the coarse grid takes into account distant effect more quickly and provide a good starting point for the next finer grid

Lesson 11: Image processing

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004

Generalities

Generalities:

- an *image* is represented as a two-dimensional array of *pixels* (picture elements)
- each pixel is either
 - a value in the *gray-scale* (usually, a number between 0 and 255)or it represents
 - a *color* (in rgb coding, i.e., the intensity of each primary red/green/blue color is specified; usually they are in the range 0-255)

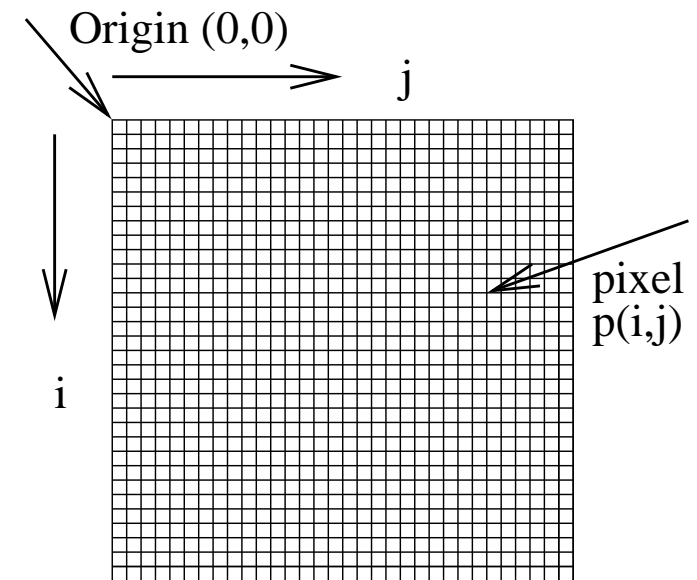




Image processing is computationally intensive

Estimation (the need for parallel processing):

- suppose an image has 1024×1024 8-bit pixels (1Mb)
- if each pixel requires one operation, then 10^6 operations are needed for one frame;
- with computers performing 10^8 operations/sec this would take 10^{-2} s (= 10ms)
- in real-time applications usually the rate is 60-80 frames/sec.; hence each frame need to be processed in 12-16ms
- however, many image processing operations are complex, requiring much more than 1 operation per pixel; hence parallel processing may be useful here



Image processing

Image processing methods: these methods use and modify the pixels; most of them are very suitable for parallel processing; examples:

- basic low-level image processing, including noise cleaning or noise reduction, contrast stretching, smoothing, etc.
- *edge detection*
- *matching* an image against a given template
- *Hough transformation* (identify the pixels associated to straight lines or curves)
- *(fast) Fourier transform* - passing from an image to a frequency domain (and back)



Point processing

Point processing: methods acting on individual pixels (they are embarrassingly parallel)

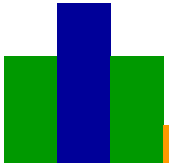
- *Thresholding*: the pixels below a threshold are reduced to 0:

$$if(x < threshold) x = 0; else x = 1$$

- *Contrast stretching*: the range of gray level values is extended to make details more visible

$$x = (x - x_l) \left(\frac{x_H - x_L}{x_h - x_l} \right) + x_L$$

a pixel of value x within the range $[x_l, x_h]$ is stretched to the range $[x_L, x_H]$ (this is often used in medical images, either for soft tissue portions, or for dense bone-like structures)



Histogram

Histogram

- an histogram shows the number of pixels in the image for each gray level (say, between 0 and 255)
- sequential code:

```
for(i=0; i<height_max; x++)  
    for(j=0; j<width_max; y++)  
        hist[p[i][j]] = hist[p[i][j]] + 1;
```

(hist[k] holds the number of pixels of gray level k)

- a parallel version may be easily developed (parallel addition of a set of numbers)



Smoothing, sharpening, noise reduction

Definitions:

- *smoothing* - suppress large fluctuations in intensity over the image area (can be achieved by reducing the high-frequency content)
- *sharpening* - accentuate the transitions, enhancing the detail (can be achieved in two ways: reduce low-frequency content or accentuate changes through differentiation)
- *noise reduction* - suppress a noise signal present in the image (not easy to distinguish noise from useful signal; a different method may be to capture the image more times and take the average on each pixel)



..Smoothing, sharpening, noise reduction

- the algorithms often require *local operations*, e.g., accessing all the pixels around the pixel to be updated

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

- Example: *mean*

$$x'_4 = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

it is used as a smoothing technique

- a sequential code for mean requires 9 operations for each pixel; hence sequential time complexity is $O(n)$



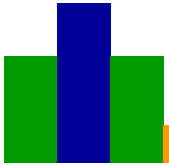
Parallel mean computation

Horizontal directions:

- 1 each processor (i, j) receives the value $x_{i,j-1}$ from *left* and adds it to an accumulating sum (the original value $x_{i,j}$ is retained)
- 2 then it receives the original value $x_{i,j+1}$ from *right* and adds it to the accumulating sum;

Hence processor (i, j) holds $x_{i,j} + x_{i,j-1} + x_{i,j+1}$; replace original values by these sums and repeat 1,2 for the vertical directions

- 3 processor (i, j) receives $x_{i-1,j} + x_{i-1,j-1} + x_{i-1,j+1}$ from above and adds it (retaining its previous sum $x_{i,j} + x_{i,j-1} + x_{i,j+1}$)
- 4 then it receives the sum $x_{i+1,j} + x_{i+1,j-1} + x_{i+1,j+1}$ from below and adds it getting the final sum $x_{i-1,j} + x_{i-1,j-1} + x_{i-1,j+1} + x_{i,j} + x_{i,j-1} + x_{i,j+1} + x_{i+1,j} + x_{i+1,j-1} + x_{i+1,j+1}$



Median

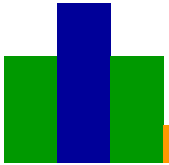
- mean value method tends to blur edges or other sharp details
- an alternative method is to use the *median*:
 - order the values of the neighborhood pixels and
 - choose the center value (provided an odd number of cells are compared)
- for the described 3×3 structure x_0, \dots, x_8 , order the values as $v_0 \leq v_1 \leq \dots \leq v_8$ and choose v_4
- one may use bubble sort, but stop when the center value was found (after 5 steps); this requires $8 + 7 + \dots + 4 = 30$ comparisons, hence $30n$ operations (for n pixels)



Parallel code / for median

- One may use a mash sorting algorithm, e.g., shear-sort;
- For greater speed an approximation method may be used:
 - use compare-and-exchange to sort any row:
 - Stage 1: $x_{i,j-1} \longleftrightarrow x_{i,j}$
 - Stage 2: $x_{i,j} \longleftrightarrow x_{i,j+1}$
 - Stage 3: $x_{i,j-1} \longleftrightarrow x_{i,j}$
 - repeat for columns:
 - Stage 1: $x_{i-1,j} \longleftrightarrow x_{i,j}$
 - Stage 2: $x_{i,j} \longleftrightarrow x_{i+1,j}$
 - Stage 3: $x_{i-1,j} \longleftrightarrow x_{i,j}$
 - the value $x_{i,j}$ after these 6 steps may not be the median, but it is a good approximation

This is the basic code for cell (i, j) ; it interferes with the code for other cells.

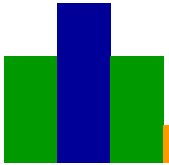


Weighted masks

- mean method gives equal weights to all neighborhood pixels
- generally, a *weighted mask* may be used
- for our standard 3×3 structure x_0, \dots, x_8 and weights w_0, \dots, w_8 , the new center pixel value is

$$x'_4 = \frac{w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8}{k}$$

- k is used to maintain a correct gray-scale balance; usually k is $\sum_{i=0}^8 w_i$
- this operation may be seen as the “cross-correlation” of vectors x and w
- masks of other size may also be used; e.g., 5×5 , 9×9 , etc.



..Weighted masks

Examples (of masks):

- | | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

 and $k = 9$ — may be used to compute mean

- | | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 8 | 1 |
| 1 | 1 | 1 |

 and $k = 16$ — a noise reduction mask

- | | | |
|----|----|----|
| -1 | -1 | -1 |
| -1 | 8 | -1 |
| -1 | -1 | -1 |

 and $k = 9$ — a sharpening filter mask



Edge detection

Edge detection:

- object identification often requires to find *edges*
- an “edge” is a *significant change* of the gray level intensity

Suppose a one variable function $f(x)$ is considered (e.g., corresponding to a row);

- if f is differentiable, then:
 - its derivative $\partial f / \partial x$ has a *spike* when f has a significant change
 - the polarity of this spike gives the sense of the changing: positive (resp. negative) \Rightarrow increasing (resp. decreasing)
- if f is double differentiable, then its second derivative has a *zero* in the interval where f has such a significant change



..Edge detection

Suppose a (two-dimensional) image is considered; then the change in gray level have a

- *gradient magnitude* (number)

$$\nabla f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

- ... and a *gradient direction* (the angle with respect to the y-axis)

$$\phi(x, y) = \tan^{-1} \left(\left(\frac{\partial f}{\partial y}\right) / \left(\frac{\partial f}{\partial x}\right) \right)$$

These formulas may be used to identify the edges in an image (the 1st one is approximated as $\nabla f = \left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|$ to simplify the computation)



Edge detection masks

- as usual, for discrete functions derivatives are approximated by

differences: for a 3×3 group,

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

we have:

$$\frac{\partial f}{\partial x} \approx x_5 - x_3 \quad \text{and} \quad \frac{\partial f}{\partial y} \approx x_7 - x_1$$

hence

$$\nabla f = |x_5 - x_3| + |x_7 - x_1|$$

- to compute this,
 - two masks may be used: one for $x_5 - x_3$, one for $x_7 - x_1$;
 - then add the resulting absolute values(the computation for each mask may be made in parallel)



Prewitt operator

Prewitt operator uses more points to approximate the gradient:

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

Then,

$$\nabla f = |x_2 - x_0 + x_5 - x_3 + x_8 - x_6| + |x_6 - x_0 + x_7 - x_1 + x_8 - x_2|$$

which, as above, requires two masks (one for each module)

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1



Sobel operator

Sobel operator is a popular edge detection method; it uses a different approximation method for the gradient:

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + 2(x_5 - x_3) + (x_8 - x_6)$$

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + 2(x_7 - x_1) + (x_8 - x_2)$$

which, as above, requires two masks (one for each module)

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Usually, the operators based of the 1st-order derivatives enhance noise; but Sobel operator has also a smoothing action.



Laplace operator

Laplace operator uses the 2nd-order derivatives

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

approximated to

$$\nabla^2 f = 4x_4 - (x_1 + x_3 + x_5 + x_7)$$

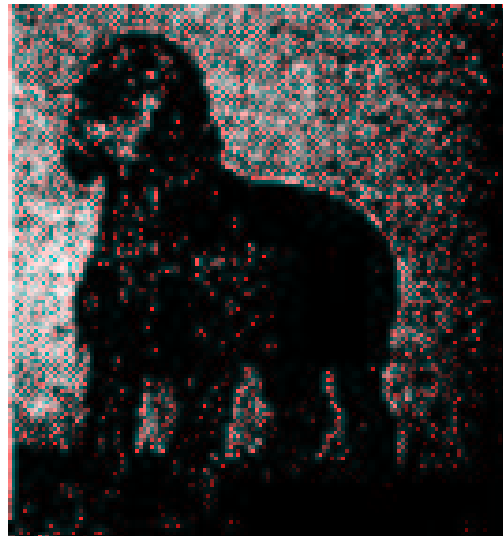
and computed using a single mask

0	-1	0
-1	4	-1
0	-1	0

Notice: We have studied this operator in other lectures, e.g., for heat distribution problem.

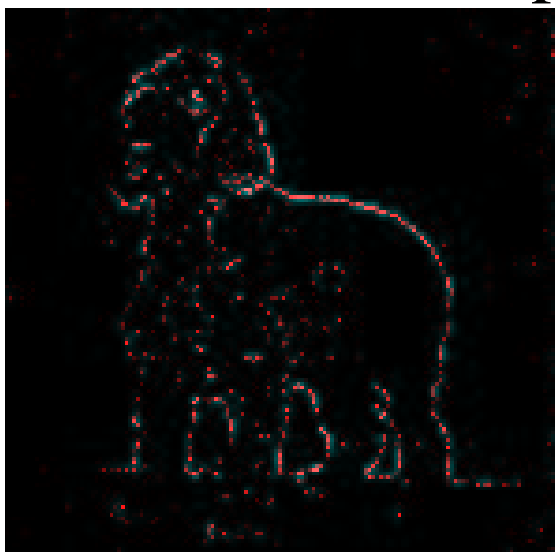
Edge detection

An original image

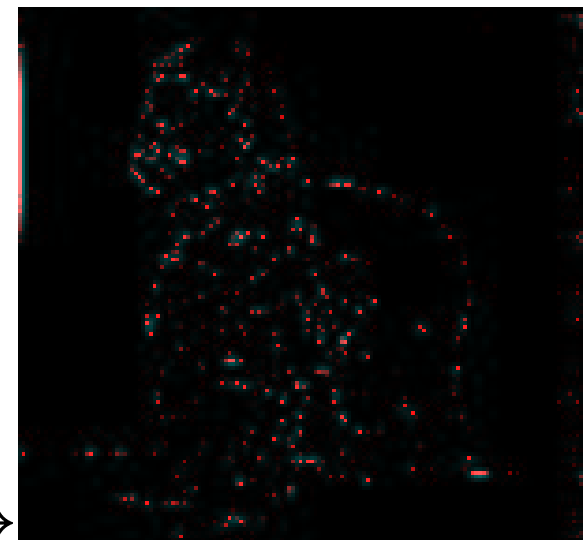


The effect of Sobel and Laplace operators

Sobel →



Laplace →



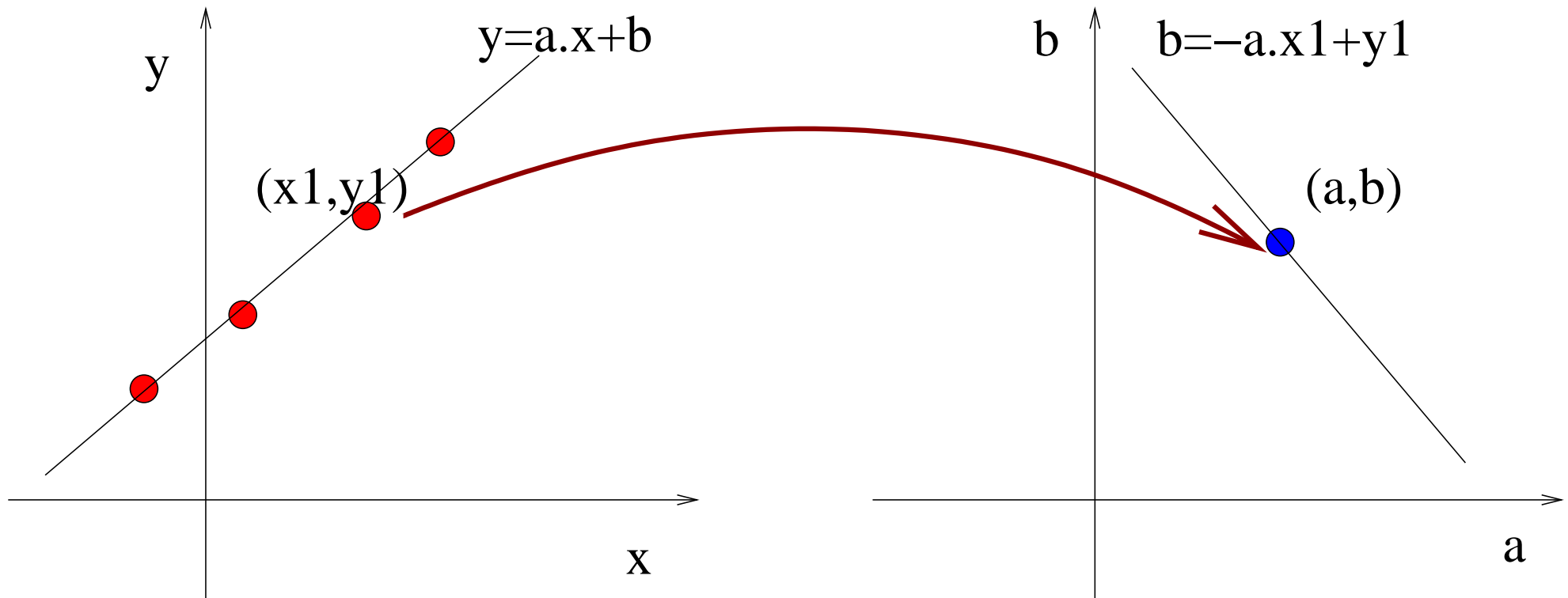


Hugh transform

Hugh transform:

- a method to “*find the parameters of equations of lines that most likely fit sets of pixels in an image*”
- may be used to fill in the gap between the points obtained in edge-detection result
- a line is described by the equation $y = ax + b$
- a direct search of the line including the largest number of pixels from a set of n pixels is expensive: $O(n^3)$ (for any 2 points check how many other points are on their line)
- rearrange the equation as $b = -xa + y$; then all points (x_i, y_i) on the line have the same associated (a, b) pair
- finally, count the number of points mapped to an (a, b) pair

..Hugh transform





Find “most likely” lines

1st version: Cartesian coordinates

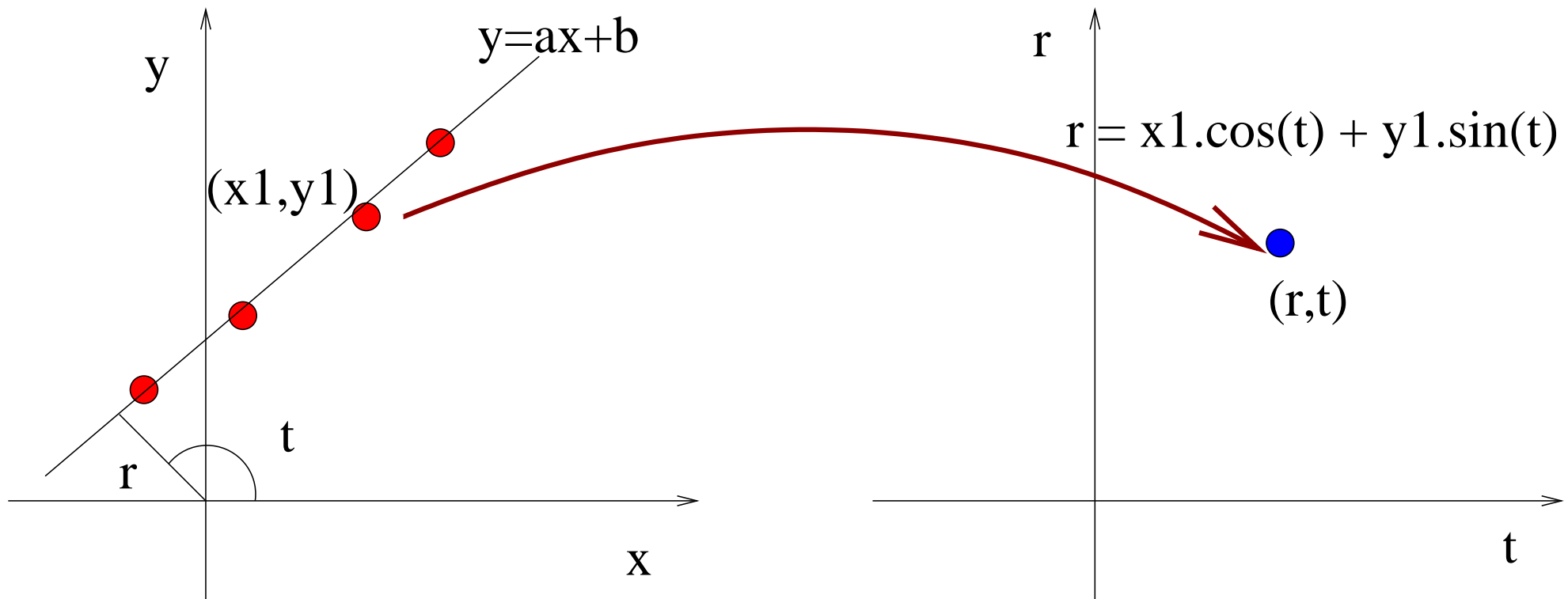
- a single original point (x_1, y_1) is mapped into *all* (a, b) -points of the line

$$b = -x_1 \cdot a + y_1$$

- a discrete grid for pairs (a, b) is used; the computation is then rounded to the nearest possible (a, b) coordinate
- this is to be repeated for all given points (x_i, y_i)
- each pair (a, b) uses an accumulator to count the number of points mapped into it
- finally, the point (a, b) with the maximum value is chosen

Disadvantage: can not handle vertical lines.

..Hugh transform





..Find “most likely” lines

2nd version: Polar coordinates

- a single original point (x_1, y_1) is mapped into *all* (r, θ) -points satisfying

$$r = x_1 \cos\theta + y_1 \sin\theta$$

- the rest of the procedure is as in the previous case:
 - a grid of points (r, θ) is selected,
 - the number of pixels mapped into each (r, θ) -point is counted, and
 - the (r, θ) -point with the maximum value is selected
- it works well for all directions



Implementation

Implementation (using the standard image representation, i.e., origin = top/left corner):

- the parameter space is divided into small rectangular regions
- each region has an associated accumulator
- accumulators for the regions were a pixel maps into are incremented
- if k intervals are chosen for θ , then the complexity is $O(kn)$
- the complexity can be significantly reduced by limiting the range of lines for individual pixels using some criteria



..Implementation

Sequential code: (only one θ is used, based on the gradient function)

```
for (x=0; x<xmax; x++)
  for(y=0; y<ymax; y++) {
    sobel(&x,&y,dx,dx) /* find x,y gradients */
    magnitude = grad_max(dx,dy);
    if (magnitude > threshold) {
      theta = grad_dir(dx,dy) /* use atan() fn */
      theta = theta_quantize(theta);
      r = x * cos(theta) + y * sin(theta);
      r = r_quantize(r);
      acc[r][theta]++;
      append(r, theta, x, y); /* gather points */
    }
  }
}
```



..Implementation

- from the resulting matrix `acc[][]` the points `(r, theta)` realizing a (local) maximum are chosen (hence an optimization algorithm has to be used/implemented)

Parallel code:

- there is a lot of room for parallelization in the above sequential algorithm, e.g.:
 - the accumulators may be computed in parallel
 - they use the same image, hence a shared memory model may be selected
 - only read actions on the image are performed, hence no critical sections are necessary



Transformation into frequency domain

Fourier transform:

- very useful transformation, used in many areas of science and engineering
- in image processing it was successfully applied for *image enhancement*, *restoration*, and *compression*
- we start with the one-dimensional case:
 - a periodic function $x(t)$ (of time) can be decomposed into a series of sinusoidal waveforms of various frequencies and amplitudes;
 - for each frequency f one gets an associated value $X(f)$
- the resulting series is called *Fourier series*; the transform is called *Fourier transform*



Fourier series

Fourier series:

- a *Fourier series* is a summation of sine and cosine terms

$$x(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} \left(a_j \cos \left(2\pi \frac{j}{T} t \right) + b_j \sin \left(2\pi \frac{j}{T} t \right) \right)$$

- T is the period of $x(t)$; $1/T = f$ is the frequency
- a more convenient representation (using complex numbers) is

$$x(t) = \sum_{j=-\infty}^{\infty} X_j e^{2\pi i \frac{j}{T} t}$$

X_j is called the j -th Fourier coefficient; $i = \sqrt{-1}$



Fourier transform

Fourier transform (for continuous functions):

- (direct) *Fourier transform*: given a continuous function of time $x(t)$, the function on frequency $X(f)$, called the *spectrum* of $x(t)$, is defined by

$$X(f) = \sum_{-\infty}^{\infty} x(t) e^{-2\pi i f t} dt$$

- *inverse Fourier transform*: given a continuous function of frequency $X(f)$, the function on time $x(t)$ is defined by

$$x(t) = \sum_{-\infty}^{\infty} X(f) e^{2\pi i f t} df$$

- key result: direct and inverse Fourier transforms are mutually converse one to the other



Discrete Fourier transform

Fourier transform (for discrete functions): similar, but integrals are replaced by finite sums

- *discrete Fourier transform* (DFT):

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{-2\pi i \left(\frac{jk}{N} \right)}$$

- *inverse Fourier transform*:

$$x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j e^{2\pi i \left(\frac{jk}{N} \right)}$$

- $0 \leq k < N$; N (real) numbers x_0, \dots, x_{N-1} produce N (complex) numbers X_0, \dots, X_{N-1}

The factor $1/N$ will be mostly omitted in the sequel; however, it has to be finally inserted to get a proper result



..Discrete Fourier transform

Example (16 points x_0, \dots, x_{15}):

$$X_0 = \frac{1}{16} (x_0 e^{-2\pi i (0 \frac{0}{16})} + x_1 e^{-2\pi i (1 \frac{0}{16})} + \dots + x_{15} e^{-2\pi i (15 \frac{0}{16})})$$

$$X_1 = \frac{1}{16} (x_0 e^{-2\pi i (0 \frac{1}{16})} + x_1 e^{-2\pi i (1 \frac{1}{16})} + \dots + x_{15} e^{-2\pi i (15 \frac{1}{16})})$$

$$X_2 = \frac{1}{16} (x_0 e^{-2\pi i (0 \frac{2}{16})} + x_1 e^{-2\pi i (1 \frac{2}{16})} + \dots + x_{15} e^{-2\pi i (15 \frac{2}{16})})$$

$$X_3 = \frac{1}{16} (x_0 e^{-2\pi i (0 \frac{3}{16})} + x_1 e^{-2\pi i (1 \frac{3}{16})} + \dots + x_{15} e^{-2\pi i (15 \frac{3}{16})})$$

\vdots

$$X_{15} = \frac{1}{16} (x_0 e^{-2\pi i (0 \frac{15}{16})} + x_1 e^{-2\pi i (1 \frac{15}{16})} + \dots + x_{15} e^{-2\pi i (15 \frac{15}{16})})$$



..Discrete Fourier transform

In a matrix form, this transformation may be written as follows (using $w = e^{-2\pi i(\frac{1}{16})}$):

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{15} \end{pmatrix} = \frac{1}{16} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & w^3 & \dots & w^{15} \\ 1 & w^2 & w^4 & w^6 & \dots & w^{2 \cdot 15} \\ 1 & w^3 & w^6 & w^9 & \dots & w^{3 \cdot 15} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{15} & w^{15 \cdot 2} & w^{15 \cdot 3} & \dots & w^{15 \cdot 15} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{15} \end{pmatrix}$$



Fourier transform in image processing

Two-dimensional Fourier transform (the factor $1/(NM)$ is omitted)

- a 2-dim DFT is

$$X_{lm} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left(\frac{jl}{N} + \frac{km}{M} \right)}$$

where j (resp. k) is the row (resp. column) coordinate

- the formula may be rewritten as

$$X_{lm} = \sum_{j=0}^{N-1} \left[\sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left(\frac{km}{M} \right)} \right] e^{-2\pi i \left(\frac{jl}{N} \right)}$$

showing that a 2-dim DFT may be obtained in two phases:

- an (inner) 1-dim DFT operating on row, followed by
- a 1-dim DFT operating on columns



..Fourier transform in image processing

Application of DFT in image processing:

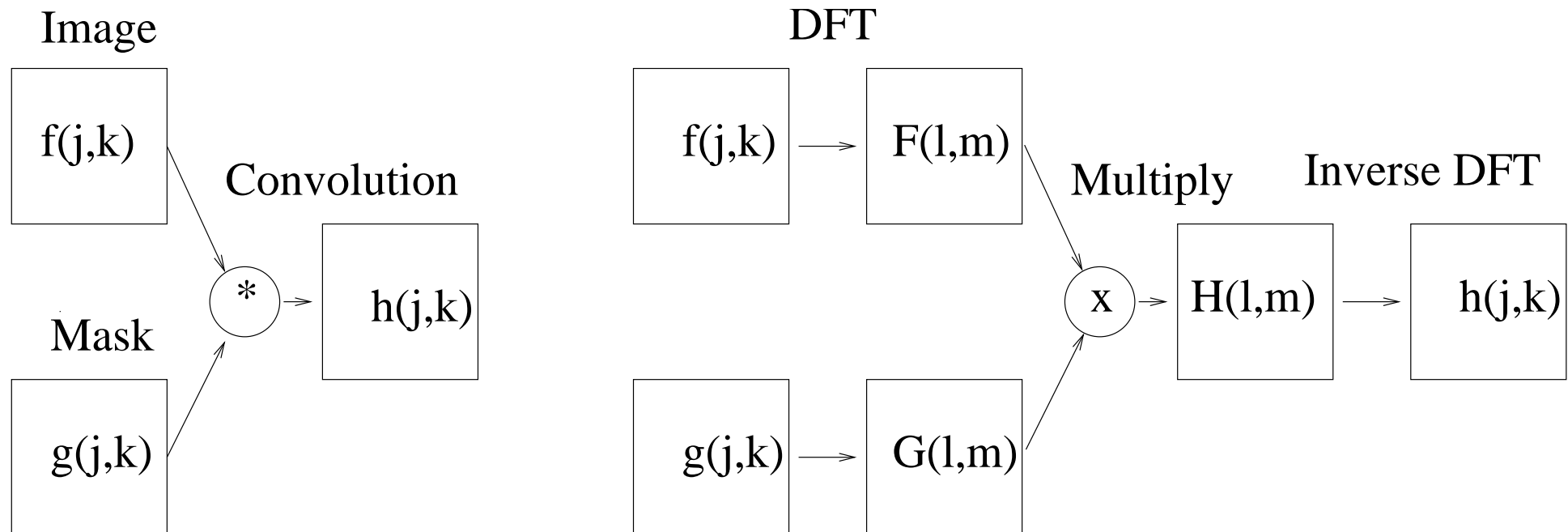
Frequency filtering is used for both smoothing and edge detection
—earlier we have used weighted masks for that purpose;
—but this is just a particular case of *convolution*

$$h(i, j) = g(j, k) * f(j, k)$$

where $g(j, k)$ describes the mask and $f(j, k)$ the image
—the convolution of functions corresponds to the product of their
Fourier transform (“ \times ” denotes element-wise multiplication)

$$H(l, m) = G(l, m) \times F(l, m)$$

..Fourier transform in image processing





Sequential code for DFT

Sequential code: denote $w = e^{-2\pi i/N}$; then $X_k = \sum_{j=0}^{N-1} x_j (w^k)^j$

```
for (k=0; k<n; k++) {  
    X[k] = 0;  
    a = 1;  
    for (j=0; j<N; j++) {  
        X[k] = X[k] + a * x[j];  
        a = a * wk;  
    }  
}
```



Parallel DFT

Parallel implementation (direct approach):

- use a master/slave approach (one slave for computing each $X[k]$); with N processes this gives an $O(N)$ algorithm

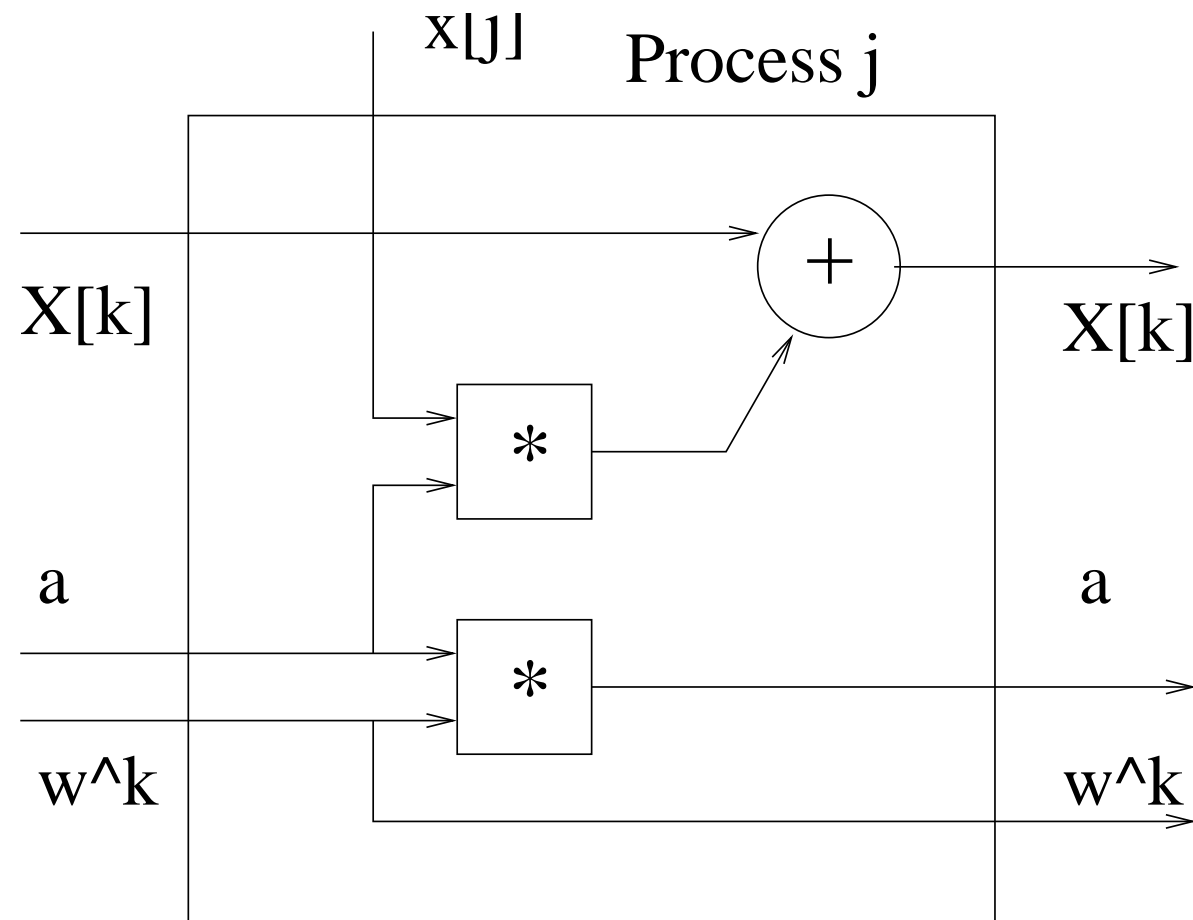
Pipeline implementation: Unfolding the inner loop we get

```
X[k] = 0;  
a = 1;  
X[k] = X[k] + a * x[0];  
a = a * wk  
X[k] = X[k] + a * x[1];  
a = a * wk  
⋮
```

..Parallel DFT

Pipeline implementation:

- one stage of the pipeline implementation is





Fast Fourier transform (FFT)

Fast Fourier transform: it's a method to reduce (sequential) time complexity from $O(N^2)$ to $O(N \log(N))$

Based on a recursive procedure:

- start with

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j w^{jk}$$

- divide the summation into two parts

$$X_k = \frac{1}{N} \left[\sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{(2j+1)k} \right]$$



..FFT

- rewrite the sum as $X_k = (1/2)[X_{even} + w^k X_{odd}]$, namely

$$X_k = \frac{1}{2} \left[\frac{1}{N/2} \sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + w^k \frac{1}{N/2} \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{2jk} \right]$$

- notice that $w^{k+(N/2)} = e^{(-2\pi i/N)(k+(N/2))} = -e^{(-2\pi i/N)k} = -w^k$
hence

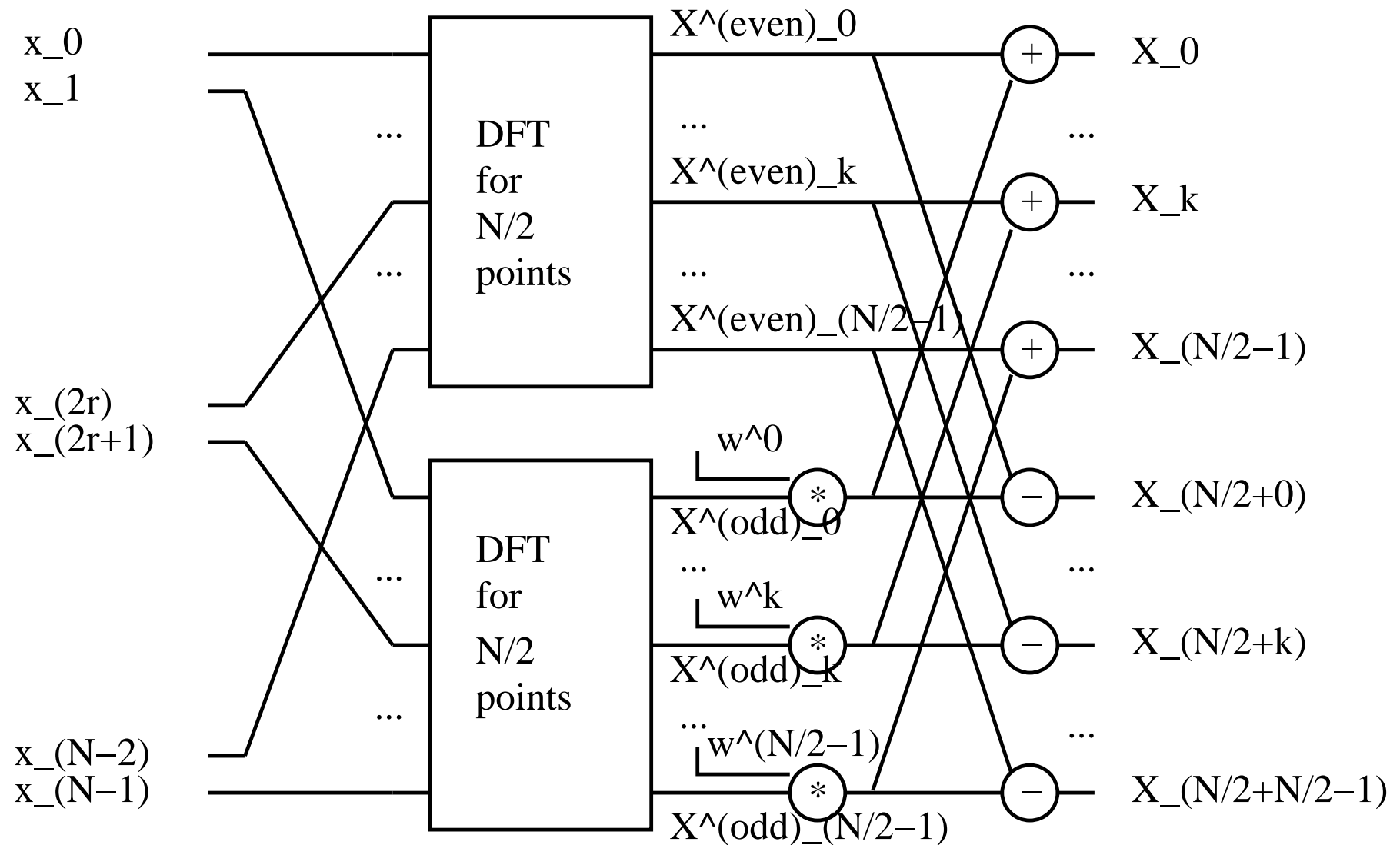
$$X_{k+(N/2)} = (1/2)[X_{even} - w^k X_{odd}]$$

showing that X_k and $X_{k+(N/2)}$ could be computed using two transforms involving $N/2$ points

- the procedure can now be recursively applied
- a sequential code leads to an $O(N \log(N))$ algorithm

..FFT

The recursive structure of *parallel FFT* is:





..FFT

Comments:

- notice that the terms w^k used in the figure are dependent on the number of points N , say w_N^k
- hence in the inner boxes of $N/2$ points different values have to be used $w_{N/2}^k$ for $k = 0, \dots, N/2 - 1$
- ... but we can normalize them: $w_{N/2}^k = w_N^{2k}$



..Parallel FFT

Analysis:

- Computation: with p processes and N points, each process will compute N/p points; each points requires 2 operations; with $\log(N)$ steps this gives $t_{comp} = 2(N/p) \log(N) = O(N \log(N))$
- Communication: if $p = N$, communication occurs at each step and one data is exchanges between pair of processors (a finer analysis may be give, depending on the network structure of the processors)

Lesson 12: Searching and optimization

G Stefanescu — National University of Singapore

Parallel & Concurrent Programming, Spring, 2004

Combinatorial search and optimization techniques:

- find a solution for a problem with many potential solutions — often exhaustive search is infeasible
- examples: traveling salesperson problem, 0/1 knapsack problem, n -queens problem, 15-tiles puzzle, etc.
- many “serious” applications in commerce, banking, industry; e.g., financial forecasting, airline fleet/crew assignment, VLSI chip layout, etc.
- basic techniques: *branch-and-bound search, dynamic programming, hill climbing, simulated annealing, genetic algorithms*
- parallel methods may be used for all these basic techniques



Branch-and-bound search

Branch-and-bound search:

- the state space of this problem is a *(static/dynamic) tree*:
 - a divide-and-conquer approach is used
 - at each level one choice between a finite number of choices C_0, \dots, C_{n-1} is selected
 - the leaves represents possible solutions to the problem
- the space tree may be explored in various ways:
 - depth-first* - first downward, then left-to-right
 - breadth-first* - first left-to-right, then downward
 - best-first* - select the branch most likely leading to the best solution
 - selective* (*pruning* some parts of the tree using *bounding* or *cut-off* functions)



..Branch-and-bound search

A few details regarding a sequential implementation:

- a node
 - is *live* if it was already reached but not all its children have been explored
 - is *E-node* if its children are currently explored
 - is *dead* if all its children have been explored
- in branch-and-bound search all children of an E-node becomes live nodes
- the live nodes are usually stored in a queue (this is a *priority queue* for best-first search strategy)

Backtracking is usually considered a distinct approach: when a search cannot proceed downward the search continues with a next alternative on the upper level; an appropriate data structure for this method is a stack (lifo queue).



Parallel branch-and-bound search

Parallel branch-and-bound

- a partitioning strategy may be used: separate processes may independently search different parts of the space tree
- even if each process uses a “pure” (e.g., depth-first search) on its own part, the resulting *parallel search* may be a sort of mixture as the order of parallel processing is difficult to predict
- the method is by far *less efficient* as one initially may *expect*:
 - the current “best solution” may be known by all processes in order to act properly on their part, for instance to prune certain irrelevant parts; hence, a lot of communication is to be used, decreasing the efficiency of the parallel approach
 - good load balancing is difficult to achieve



..Parallel branch-and-bound

- various strategies have been developed
- in a *shared-memory* model one may use a shared queue, but the serialization of the queue access limits the speedup to

$$S(n) \leq \frac{t_{queue} + t_{comp}}{t_{queue}}$$

where: t_{queue} is the mean time to access the queue and t_{comp} the mean time of process computation (this comes from Amdahl's law)

- for best-first strategies priority queues have to be used; they may be efficiently implemented using a *heap* data structure; furthermore, concurrent versions to handle the heap have been suggested (Rao and Kumar '88)



..Parallel branch-and-bound

Anomalies: with n processes one normally expect to reach the solution n times faster; but:

- *acceleration anomaly* - parallel speedup may be greater than n or “super-linear”:
 - in the parallel version a good solution may be found very early cutting down the time more than expected (n times)
- *deceleration anomaly* - parallel speedup is less than n , but more than 1:
 - this appears when the position of a feasible solution in the tree cannot be reached n times faster than with 1 process
- *detrimental anomaly* - parallel speedup less than 1:
 - total time with n processes is worse than using a single process



Genetic algorithms: generalities

Natural evolution:

- the basic information of living beings is contained into their *chromosomes*
- natural evolution works at this chromosome level:
 - when individuals reproduce, portions of parents' genetic informations are combined to generate the offspring' chromosomes
 - the combination is based on a *crossing over* mechanism
 - in addition, sometimes (random) *mutations* may appear
 - the environment selects the “most fit” individuals
 - most mutations *degrade* the individuals, but sometimes *favorables changes* may appear



..Genetic algorithms: generalities

Genetic algorithms: a *genetic algorithm* describes a computational method to solve a problem using ideas abstracted from this biological process of evolution:

- the algorithm starts with an *initial population* of solutions (individuals); *next generations* are iteratively created
- the individuals are evaluated using a set of *fitness* criteria
- a *subset* of population is selected, tending to favor the “most fit” individuals; this subset is used to produce a new generation of *offspring* (the “crossing over” mechanism is used here)
- finally, a small number of individuals of a new generation are subject to random *mutations*



Sequential genetic algorithms

Sequential genetic algorithms (sketch):

```
generation_no = 0;
initialize Population(generation_no);
evaluate Population(generation_no);
termination_condition = False
while (!termination_condition) {
    generation_no++;
    select Parents(generation_no) from
        Population(generation_no-1);
    apply crossing_over to Parents(generation_no) to get
        Offspring(generation_no);
    apply mutation to Offspring(generation_no) to get
        Population(generation_no);
    evaluate Population(generation_no);
}
```



..Sequential genetic algorithms

An example: a numerical computation problem

find the maximum of a function

Initial population:

- suppose the goal is to find the maximum of

$$f(x, y, z) = -x^2 + 10^6x - y^2 - 4 \cdot 10^4y - z^2$$

for integer x, y, z between -10^6 and 10^6 (actually, maximum value is for $x = 5 \cdot 10^5, y = -2 \cdot 10^4, z = 0$)

- initial population: $(2 \cdot 10^6 + 1)^3$ individuals (all combinations of values for x, y, z in the given range)



..Sequential genetic algorithms

Data representation:

- we need “strings” (as in the case of chromosomes)
- there are $2 \cdot 10^6 + 1$ values in the given range, hence we may use 21 bits to represent them (sign-plus-magnitude representation:
 $\overline{36}_{10} \rightarrow 0000000000000000100100;$
 $-\overline{7}_{10} \rightarrow 10000000000000000000111;$
 $-\overline{1024}_{10} \rightarrow 1000000000100000000000;)$
- concatenating all these bits, we get the representation of a potential solution by a string with 63 bits
 $(36, -7, -1024) \rightarrow$
0000000000000000010010010000000000000000111100000000010000000000



..Sequential genetic algorithms

Fitness function:

- for each (x, y, z) computes $f(x, y, z)$ (larger value, better fit)

Constraints:

- not all 63 bit strings represent valid combinations: the values outside $[-10^6, 10^6]$ should be deleted (or “surgically repaired”)

Number of individuals:

- small number \Rightarrow slow convergence;
large number \Rightarrow heavy computation
- usually 20 to 1000 pseudo-random generated strings



..Sequential genetic algorithms

Parents' election:

- more fit individuals have greater chance to be selected:
 - best fit individuals may be selected to produce offspring,
 - but choosing only the “best fit” individuals may lead to a fast convergence to a local optimum, totally missing the global one
- *tournament selection* - is a solution to avoid the stick into a local optimum:
 - a set of k individuals are randomly selected to enter into a tournament; the most fit individual is selected as a parent for the next generation;
 - the tournament is repeated n times (if n parents have to be selected)



..Sequential genetic algorithms

Offspring production:

- *single-point crossing over:*
 - given two parents $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$, randomly choose a cut point - say, this is between p -th and $p + 1$ -th bits
 - create 2 children:
child 1 = $a_1 \dots a_p b_{p+1} \dots b_n$ and child 2 = $b_1 \dots b_p a_{p+1} \dots a_n$
- other alternatives may be used (multi-point crossing over, uniform crossing over, gene sharing, etc.)

Mutation:

- randomly select and modify one or more bits of an individual (sometimes this may have huge effect - e.g., in our example, the 1st bit represents the sign; the last bit is less important)
- mutation should be kept at a small rate (for convergence)



..Sequential genetic algorithms

Variations:

- carry over a few well fit individuals to the next generation
- randomly create a few new individuals
- let the population size to vary from a generation to the next

Termination:

- either apply the basic step to generate an a-priori given number of generations
- or use other criteria (like the difference between the value of the optimum at a generation and at the next one; or the degree of similarity of the individuals within a generation, etc.)



Parallel genetic algorithms (PGA)

Parallel genetic algorithms: two quite different possible use of parallel processing are

1. *isolated subpopulations*: each processor operates independently on an isolated subpopulation; their communication is through *migration* of individuals
2. *common population*: in this approach there is a common population; each processor does a portion of the selection-crossover-mutation work



..(PGA) Isolated subpopulations

Isolated subpopulations:

- each processor handles an isolated subpopulation of individuals (it does the evaluation of fitness, selection, crossing over, mutation)
- periodically, say after k generations, the processors exchange certain individuals - a so called “migration” process takes place



..(PGA) Isolated subpopulations

Migration operator:

- migration requires a few activities: *selecting the emigrants; sending the emigrants; receiving the immigrants; integrating the immigrants*
- a selection criterion may be to send the best individuals to a particular processor; the convergence is then faster, but there is a risk of sticking into a local optimum
- sending/receiving the emigrants/immigrants may be easily done in a message-passing model
- migration introduces a communication overhead, hence both the frequency and the volume of communication should be carefully considered



..(PGA) Isolated subpopulations

Migration models:

- *the island model:* individuals are allowed to be sent to any other subpopulation
- *the stepping stone model:* individuals are allowed to be sent only to the neighboring subpopulations
- the island model allows more freedom, but also more communication overhead is introduced



..(PGA) Isolated subpopulations

Parallel genetic algorithms (sketch of slave code):

```
generation_no = 0;
initialize Population(generation_no);
evaluate Population(generation_no);
termination_condition = False
while (!termination_condition) {
    generation_no++;
    select Parents(generation_no) from
        Population(generation_no-1);
    apply crossing_over to Parents(generation_no) to get
        Offspring(generation_no);
    apply mutation to Offspring(generation_no) to get
        Population(generation_no);
    apply migration to Population(generation_no);
    evaluate Population(generation_no);
}
```



..(PGA) Parallelizing a common population

Parallelizing a common population:

- in this approach general techniques to parallelize the sequential algorithm are used
- for instance, if processors have access to the required information, then parallel selection may be used (e.g., each processor does separate tournaments),
- also crossover and mutation for different individuals may be done by different processors
- as with many other parallel programs, the approach is useful when the computational effort is large enough to hide the time spent on communication



Successive refinement

Successive refinement:

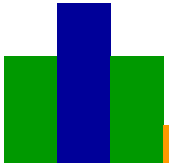
- this is another iterative approach on searching algorithms based on a series of successive refinements of the solution space
- start with a coarse “cube” of solution space; e.g., for the given example (slide 12.11) test every ten-thousandth point
- the best k points are retained and new finer, but smaller, cubes centered in these points are evaluated
- finer and finer grids are introduced and evaluated till a good solution is obtained
- this method may be easily parallelized



Hill climbing

Hill climbing:

- hill climbing is another common approach to searching and optimization
- based on the strategy of improving the result at any step (keep moving toward the pick - in the “hill climbing” metaphor)
- the problem with this approach is that one may reach a local pick and never go down to a little valley to climb up a higher pick on the other side
- a solution is to use more starting points - then the chance of reaching a global optimum is increasing



..Hill climbing

Parallel versions may be easy to implement:

- the starting points may be chosen in a random way (Monte-Carlo method)
- each processor may independently handle different starting points
- depending on the problem, either a static or a dynamic work pool may be used to allocate points to processes



..Hill climbing

A banking application:

- Problem:
 - how to distribute “lock-boxes” across a country to optimize the receiving of payments for a given company?
- the optimization function is in terms of *float_days* (a “float” is the delay between customer’s payment mailing date and the firm’s collection date, multiplied by the amount of payment)
- —the firm has to pay for the lock-boxes, hence not too many are to be used;
 - their optimal distribution depends on the mailing time, but also on the distribution of the money the firm is expecting to collect from various places



..Hill climbing

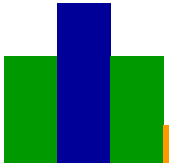
Sequential code (sketch):

```
set Float_days to 1000;
for (i=1; i < MaxLockBoxes; i++) {
    for (all possible placements of i lockboxes) {
        compute CurrentFloat_days;
        if (Float_days > CurrentFloat_days) {
            Float_days = CurrentFloat_days;
            save i and lockbox placement pattern;
        }
    }
}
```

Computational effort: to select k lock-box places among n places

requires to check $\frac{n!}{k!(n-k)!}$ combinations;

[for $(n, k) = (200, 6)$, approx. $8 \cdot 10^{10}$ combinations]



..Hill climbing

Parallel version:

- use hill climbing to avoid searching into such a huge space of possible placements
- incrementally change a lock-box location to another one that produces the greatest reduction of float_days
- eventually a location realizing a local minimum of float_days is reached
- one may randomly generate a number of initial starting locations to increase the chance of reaching a global minimum
- parallelization techniques (described when hill climbing method was introduced) may be used