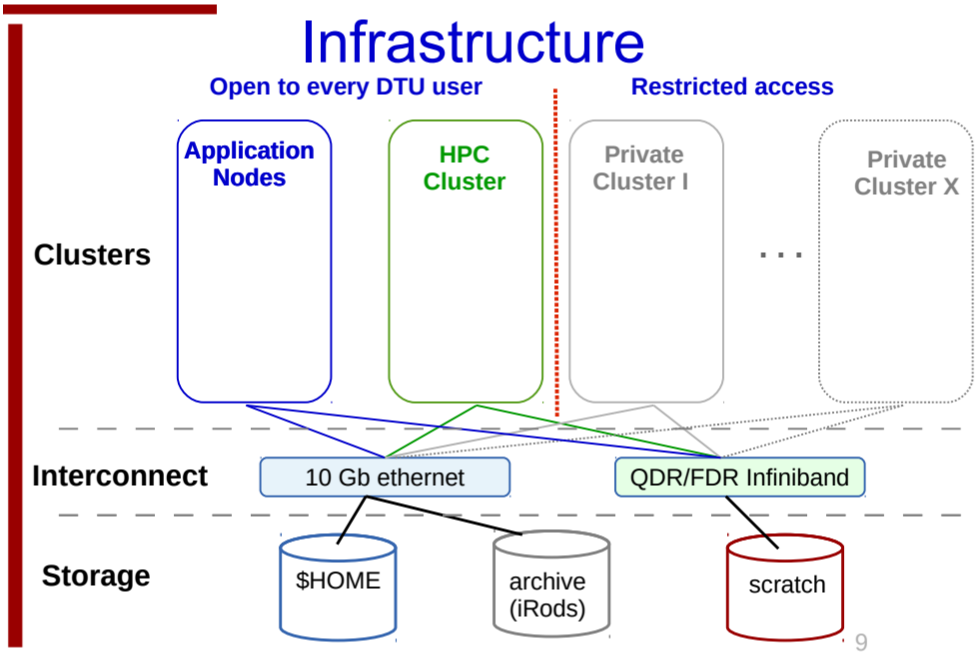
Pensumnoter - Foredragsnoter Alle

Distribuerede og parallelle systemer (Danmarks Tekniske Universitet)

Parallel Exam Notes



Infrastructure

Open to every DTU user

Restricted access

Clusters

Application Nodes

HPC Cluster

Private Cluster I

. . .

Private Cluster X

Interconnect

10 Gb ethernet

QDR/FDR Infiniband

Storage

$HOME

archive (iRods)

scratch

9

HW-arkitekturer (hardware arkitekturer)

Når der tales om HW arkitekturer, refereres der ofte til de følgene 4 bergreber:

Single-instruction, single-data (SISD) systems

- Executes a single instruction at a time and it can fetch or store one item of data at a time
- Vector processors, vector instructions
- GPU

Single-instruction, multiple-data (SIMD) systems

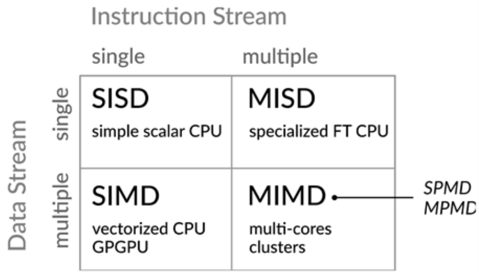
- SIMD systems are parallel systems
- Operates on multiple data streams by applying the same instruction to multiple data items

Multiple-instruction, single-data (MISD) systems

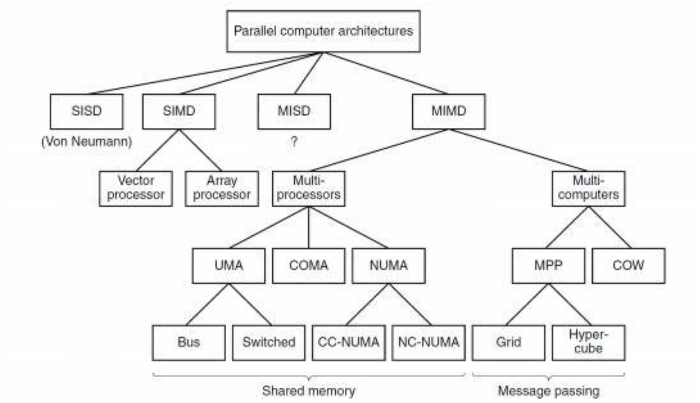
Multiple-instruction, multiple-data (MIMD) systems

- Simultaneous instruction streams operating on multiple data streams.
- MIMD systems typically consist of a collection of fully independent processing units or cores
- unlike SIMD systems, MIMD systems are usually asynchronous,

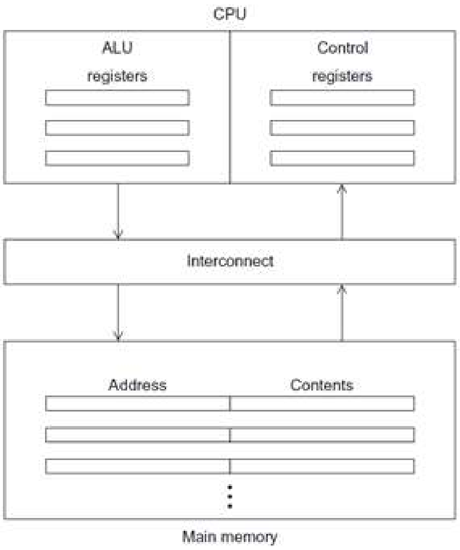
Flynn’s taxonomy



		Instruction Stream single	Instruction Stream multiple
Data Stream	single	SISD simple scalar CPU	MISD specialized FT CPU
	multiple	SIMD vectorized CPU GPGPU	MIMD multi-cores clusters



Von Neumann architecture (example på SISD)
Separationen af CPU og Memory kaldes ofte “the Von Neumann bottleneck”, sammenkoblingen bestemmer hastigheden på hvilken instruktion/data kan fås. Da en moderne CPU kan udregne instruktioner 100x hurtigere end de kan hente dem er dette et problem



Shared-memory system:

- a collection of autonomous processors is connected to a memory system via an interconnection network
- each processor can access each memory location
- processors usually communicate implicitly by accessing shared data structures
- The most widely available shared-memory systems use one or more multicore processors.
- Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores

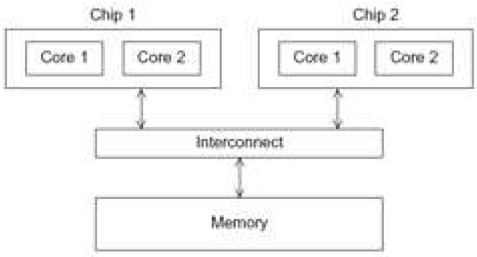


FIGURE 2.5
A UMA multicore system

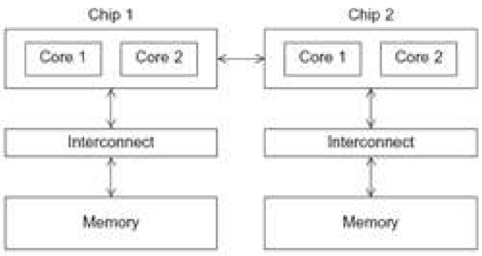


FIGURE 2.6
A NUMA multicore system

UMA - uniform memory access

- The time to access all the memory locations will be the same for all the cores,
- programmer doesn’t need to worry about different access times for different memory locations

NUMA - non-uniform memory access

- NUMA systems have the potential to use larger amounts of memory than UMA systems.
- A memory location to which a core is directly connected can be accessed more quickly than a memory location that must be accessed through another chip.

distributed-memory system

- each processor is paired with its own private memory
- Processors/memory communicates over an interconnection network
- processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor.
- The most widely available distributed-memory systems are called clusters.
- connected by a commodity interconnection network—for example, Ethernet (InfiniBand)

Forbindelsesnetværk

Interconnection - sammenkobling - forbindelse netværk

Shared-memory interconnect

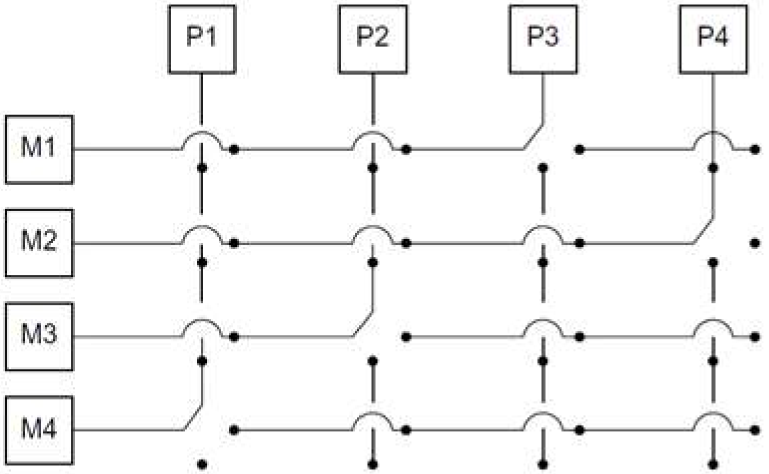
- Plays a big role in the performance for both shared and distributed memory
- a slow interconnect will seriously degrade the overall performance
- the two most widely used interconnects on shared-memory systems are buses and crossbars.

Bus

- a bus is a collection of parallel communication wires together with some hardware that controls access to the bus
- the communication wires are shared by the devices that are connected to it.
- Have the virtue of low cost and flexibility; multiple devices can be connected to a bus with little additional cost
- If a lot of processors is connected the performance is expected to decrease and we would expect the processors to wait on the main memory access
- Thus, as the size of shared-memory systems increases, buses are rapidly being replaced by *switched* interconnects

Switched

- switched interconnects use switches to control the routing of data among the connected devices
- There will only be a problem if two processors try to access the same memory module at the same time
- Crossbar is an example of this:



- Crossbars allow simultaneous communication among different devices
- The switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Distributed-memory interconnect
often divided into two groups: direct interconnects and indirect interconnects

Direct interconnect

- each switch is directly connected to a processor-memory pair
- the switches are connected to each other
- allows multiple simultaneous communications
- The ideal direct interconnect is a fully connected network in which each switch is directly connected to every other switch. ("theoretical best possible", in practical not used

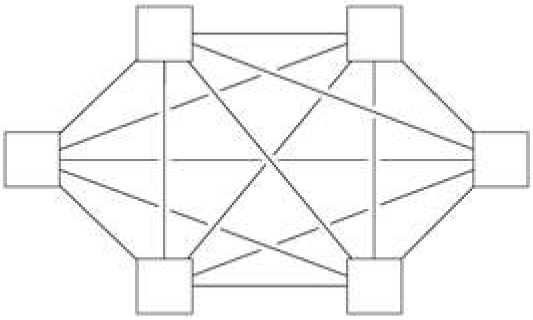


FIGURE 2.11

A fully connected network

- hypercube is a highly connected direct interconnect that has been used in actual systems.
- A one-dimensional hypercube is a fully-connected system with two processors.
- A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches
- a hypercube of dimension d has *p= 2^d* nodes, and a switch in a d-dimensional hypercube is directly connected to a processor and *d* switches
- A bisection of a hypercube is *p/2*

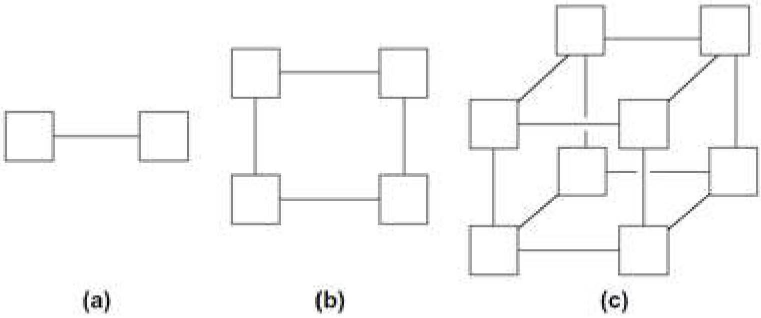


FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects

- the switches may not be directly connected to a processor
- often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network
- The crossbar and the omega network are relatively simple examples of indirect networks.
- **For crossbars:** as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.
- **For omega (see figure):** there are communications that cannot occur simultaneously a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7

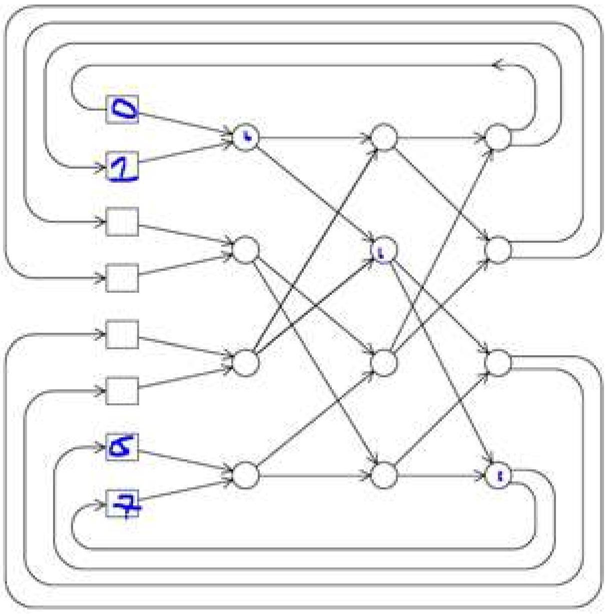


FIGURE 2.15

An omega network

- omega network is less expensive than the crossbar

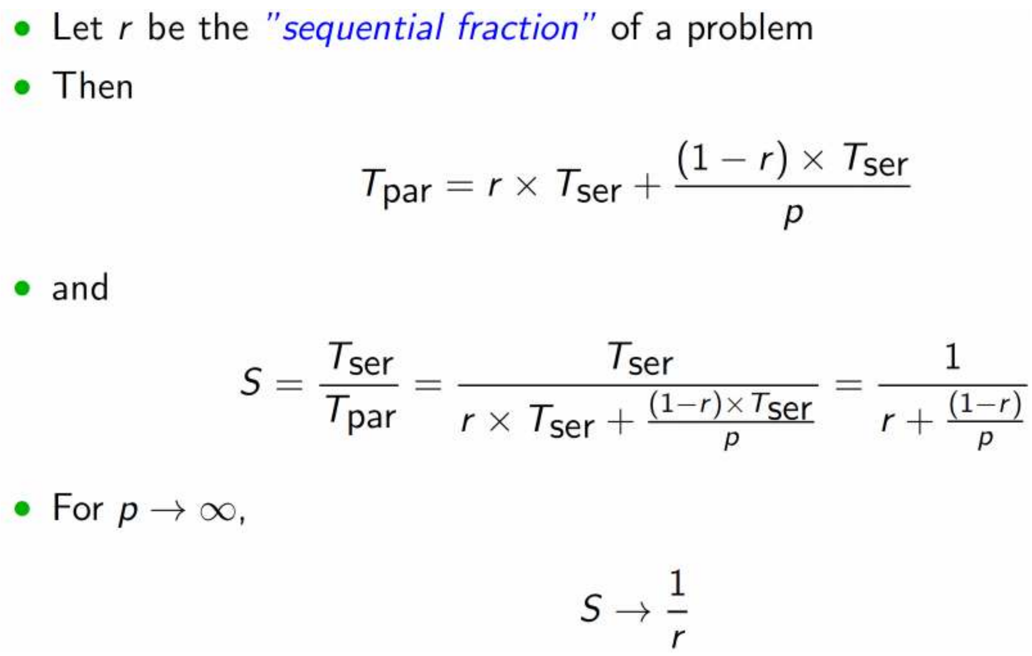
Message transmission time = l + n/b
l = latency i sekunder
n= number of bytes
b= bytes pr second

Amdahls lov

generally, if a fraction r of our serial program remains unparallelized, then Amdahl’s law says we can’t get a speedup better than 1=r
In our example, r = 1-0.9 = 1/10, so we couldn’t get a speedup better than 10. Therefore, if a fraction r of our serial program is “inherently serial,” that is, cannot possibly be parallelized then we can’t possibly get a speedup better than 1=r.

- Amdahls law doesn’t take into consideration the problem size.
- Often as the problem size increase, the “inherently serial” fraction of the program decreases in size

there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems



- Let r be the *"sequential fraction"* of a problem
- Then

$$T_{\text{par}} = r \times T_{\text{ser}} + \frac{(1-r) \times T_{\text{ser}}}{p}$$

- and

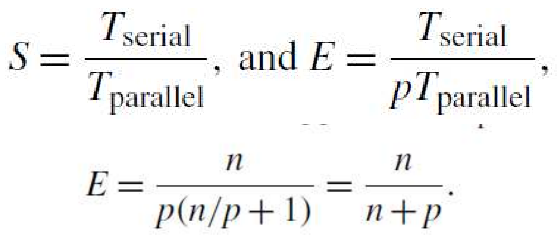
$$S = \frac{T_{\text{ser}}}{T_{\text{par}}} = \frac{T_{\text{ser}}}{r \times T_{\text{ser}} + \frac{(1-r)\times T_{\text{ser}}}{p}} = \frac{1}{r + \frac{(1-r)}{p}}$$

- For $p \to \infty$,

$$S \to \frac{1}{r}$$

Skalerbarhedsmål

Her referere til hvordan system effektivitet opfører sig når systemet skaléres. Et system kan skaleres når problem størrelsen stiger eller antallet af processor stiger.



$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \text{ and } E = \frac{T_{\text{serial}}}{pT_{\text{parallel}}},$$

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

E=efficiency

n=problem size
p=processors

Hvis vi skalere n med x og indsætter k ekstra processors til p får vi:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

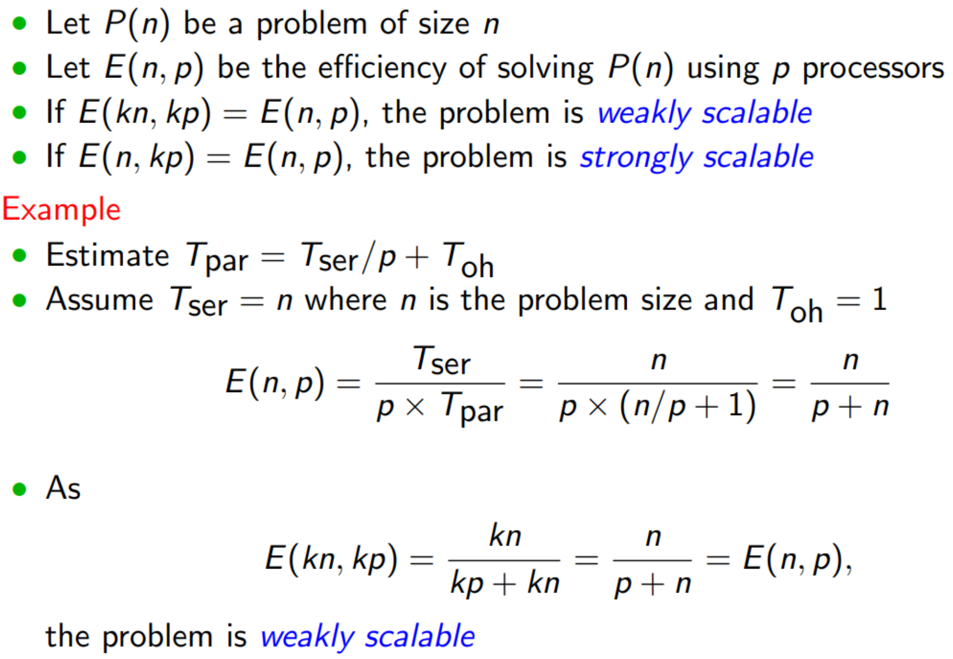
Hvis vi ønsker at se om systemet er skalerbart, er den nemmeste fremgangsmåde at se på hvordan systemet opføre sig når x = k, altså når problem størrelsen skaleres med samme faktor antallet af processorer:

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

Dette eksempel er (weakly scalable)

if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

- If when we increase the number of processes/threads, we can keep the efficiency fixed without increasing the problem size, the program is said to be strongly scalable
- If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be weakly scalable.



- Let $P(n)$ be a problem of size n
- Let $E(n,p)$ be the efficiency of solving $P(n)$ using p processors
- If $E(kn,kp) = E(n,p)$, the problem is *weakly scalable*
- If $E(n,kp) = E(n,p)$, the problem is *strongly scalable*

Example

- Estimate $T_{\text{par}} = T_{\text{ser}}/p + T_{\text{oh}}$
- Assume $T_{\text{ser}} = n$ where n is the problem size and $T_{\text{oh}} = 1$

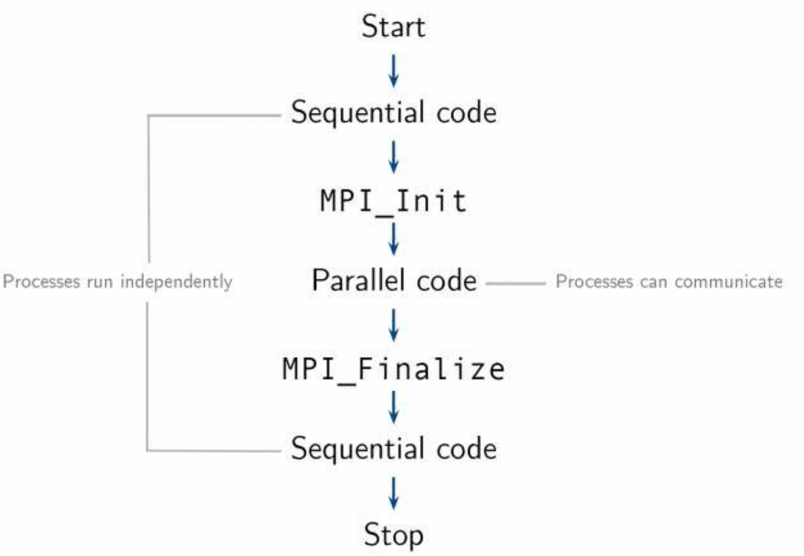
$$E(n,p) = \frac{T_{\text{ser}}}{p \times T_{\text{par}}} = \frac{n}{p \times (n/p+1)} = \frac{n}{p+n}$$

- As

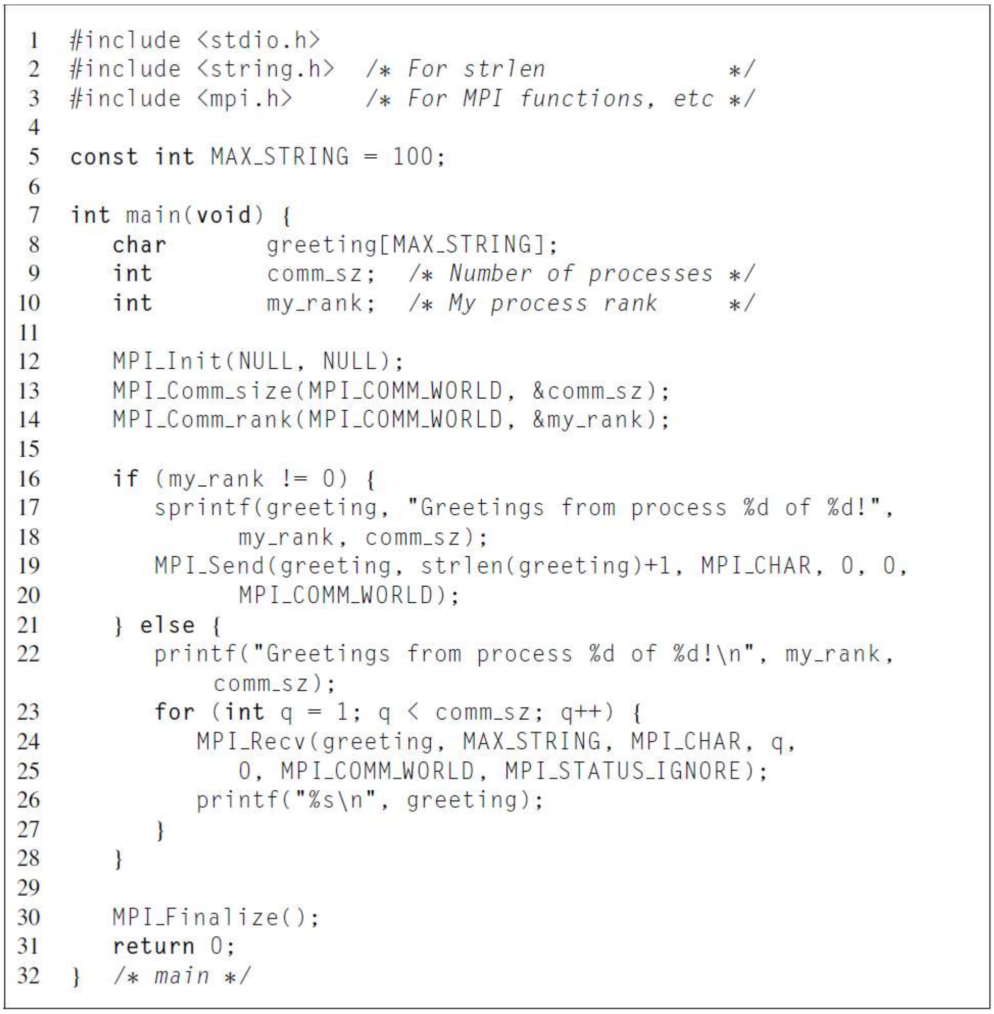
$$E(kn,kp) = \frac{kn}{kp+kn} = \frac{n}{p+n} = E(n,p),$$

the problem is *weakly scalable*

MPI-operationer

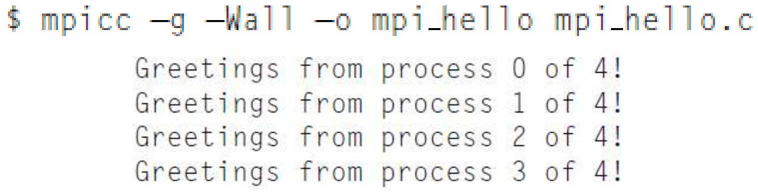


- Processes are grouped into a communicator
- Each process within a communicator has a unique rank
- Ranks are integers from $0 \ldots p-1$
- Initially all ranks belong to `MPI_COMM_WORLD`

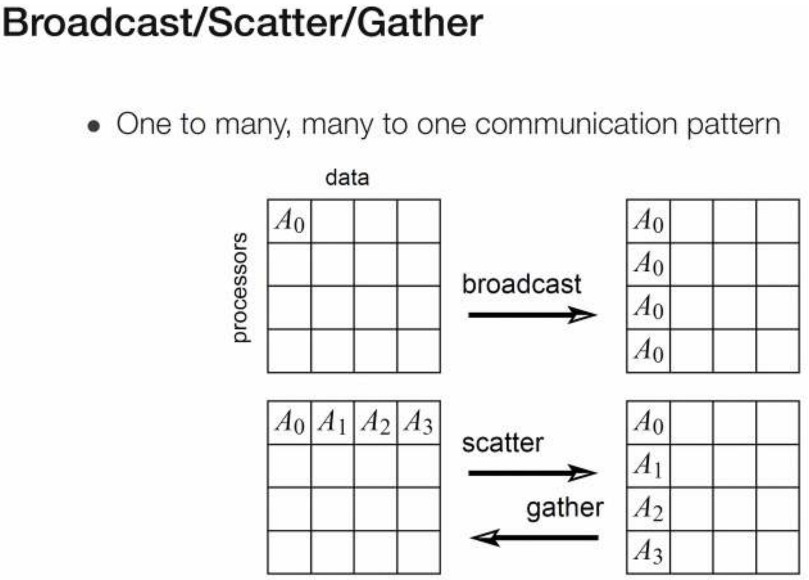


```
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
         comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Dette program udskriver



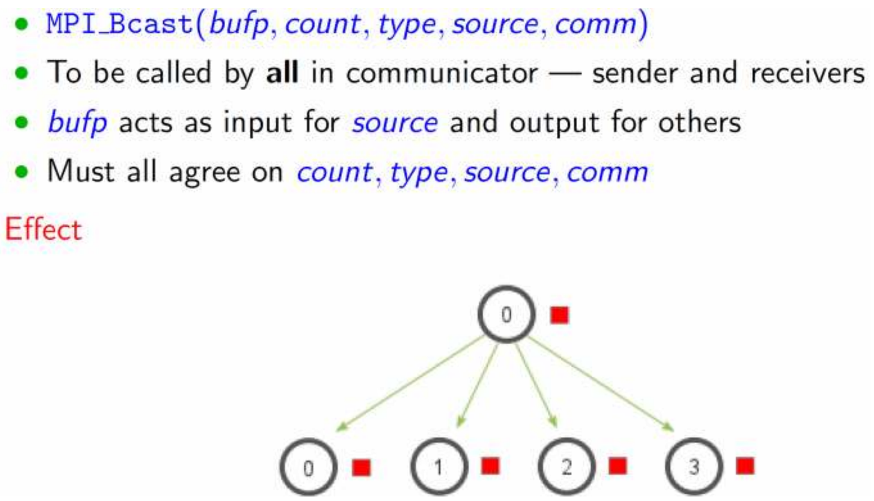
```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
          Greetings from process 0 of 4!
          Greetings from process 1 of 4!
          Greetings from process 2 of 4!
          Greetings from process 3 of 4!
```



Broadcast/Scatter/Gather

- One to many, many to one communication pattern

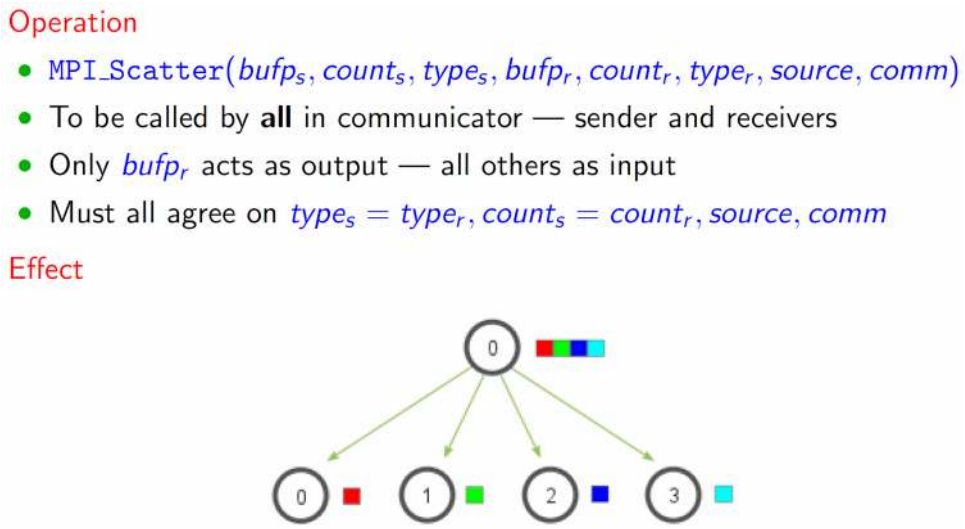
Broadcast



- MPI_Bcast(*bufp*, *count*, *type*, *source*, *comm*)
- To be called by **all** in communicator — sender and receivers
- *bufp* acts as input for *source* and output for others
- Must all agree on *count*, *type*, *source*, *comm*

Effect

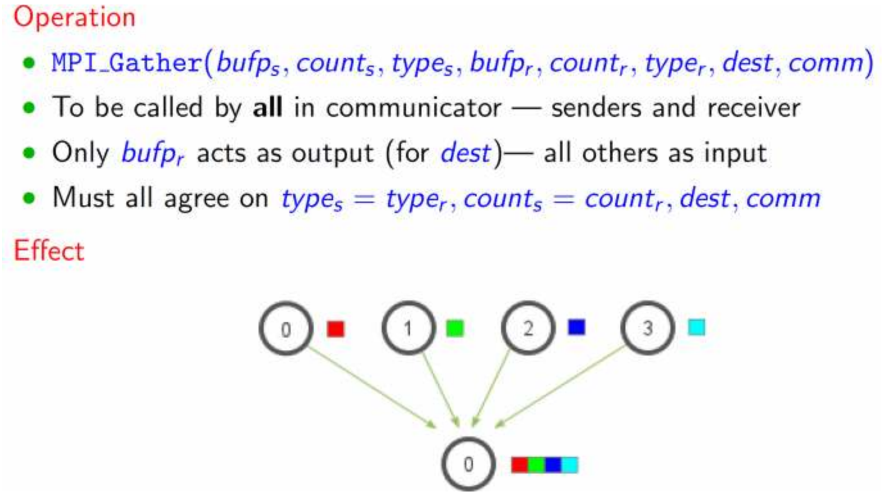
Scatter/Glatter



Operation

- `MPI_Scatter`$(bufp_s, count_s, type_s, bufp_r, count_r, type_r, source, comm)$
- To be called by **all** in communicator — sender and receivers
- Only $bufp_r$ acts as output — all others as input
- Must all agree on $type_s = type_r, count_s = count_r, source, comm$

Effect

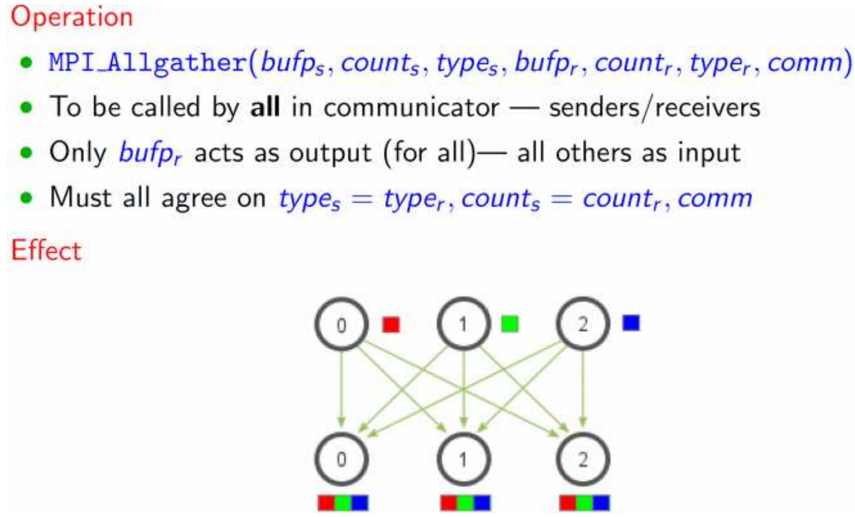


Operation

- `MPI_Gather`$(bufp_s, count_s, type_s, bufp_r, count_r, type_r, dest, comm)$
- To be called by **all** in communicator — senders and receiver
- Only $bufp_r$ acts as output (for $dest$)— all others as input
- Must all agree on $type_s = type_r, count_s = count_r, dest, comm$

Effect

All Gather

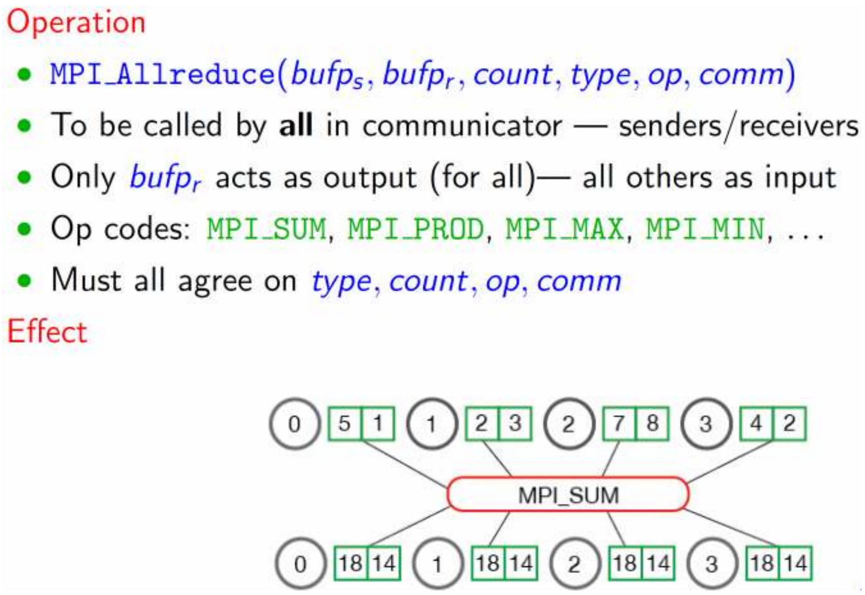


Operation

- `MPI_Allgather`$(bufp_s, count_s, type_s, bufp_r, count_r, type_r, comm)$
- To be called by **all** in communicator — senders/receivers
- Only $bufp_r$ acts as output (for all)— all others as input
- Must all agree on $type_s = type_r, count_s = count_r, comm$

Effect

All Reduce



Operation

- MPI_Allreduce($bufp_s$, $bufp_r$, *count*, *type*, *op*, *comm*)
- To be called by **all** in communicator — senders/receivers
- Only $bufp_r$ acts as output (for all)— all others as input
- Op codes: MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, . . .
- Must all agree on *type*, *count*, *op*, *comm*

Effect

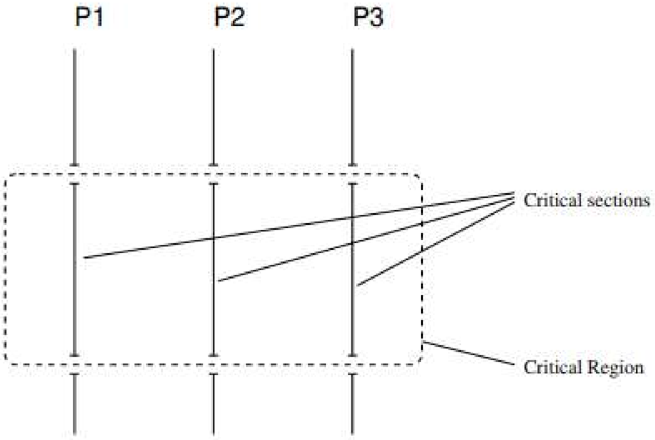
Oprette og afvente tråde (Pthreads)

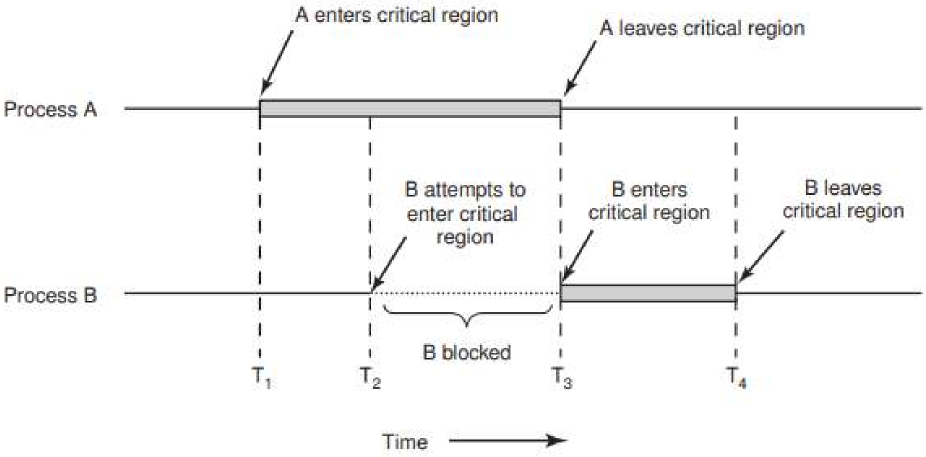
-Program 4.1: A Pthreads “hello, world” program
4.2 HELLO, WORLD
Program 4.1 shows a program in which the main function starts up several threads. Each thread prints a message and then quits.
4.2.1 Execution
The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library:1

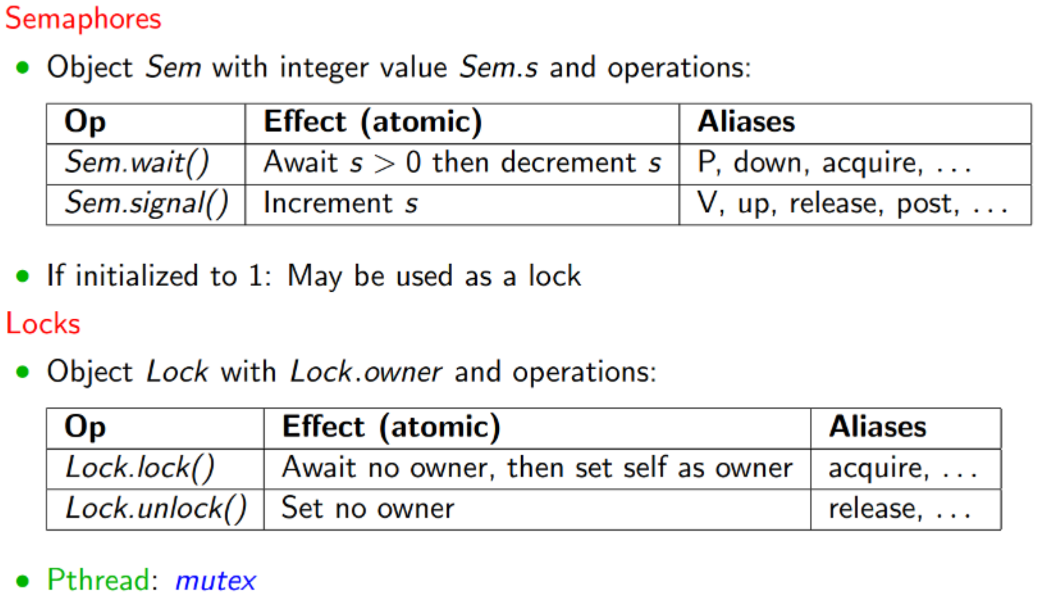
```
$ gcc −g −Wall −o pth_hello pth_hello.c −lpthread
```

The -lpthread tells the compiler that we want to link in the Pthreads library.
Note that it’s -lpthread, not -lpthreads. On some systems the compiler will automatically link in the library, and -lpthread won’t be needed.
To run the program, we just type
$ ./pth hello <number of threads>
We saw that in Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
In order to use Pthreads, we should include the pthread.h header file, and, when we compile our program, it may be necessary to link our program with the Pthread library by adding -lpthread to the command line. We saw that we can use the functions **pthread create** and **pthread join**, respectively, to start and stop a thread Function.

Semaforer og kritiske regioner(mutex)







Semaphores

- Object *Sem* with integer value *Sem.s* and operations:

Op	**Effect (atomic)**	**Aliases**
Sem.wait()	Await $s > 0$ then decrement s	P, down, acquire, . . .
Sem.signal()	Increment s	V, up, release, post, . . .

- If initialized to 1: May be used as a lock

Locks

- Object *Lock* with *Lock.owner* and operations:

Op	**Effect (atomic)**	**Aliases**
Lock.lock()	Await no owner, then set self as owner	acquire, . . .
Lock.unlock()	Set no owner	release, . . .

- Pthread: *mutex*

Three basic approaches to avoiding conflicting access to critical sections: busy-waiting, **mutexes, and semaphores**. Busy-waiting can be done with a flag variable and a while loop with an empty body. It can be very wasteful of CPU cycles. It can also be unreliable if compiler optimization is turned on, so mutexes and semaphores are generally preferable.

A **mutex** can be thought of as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

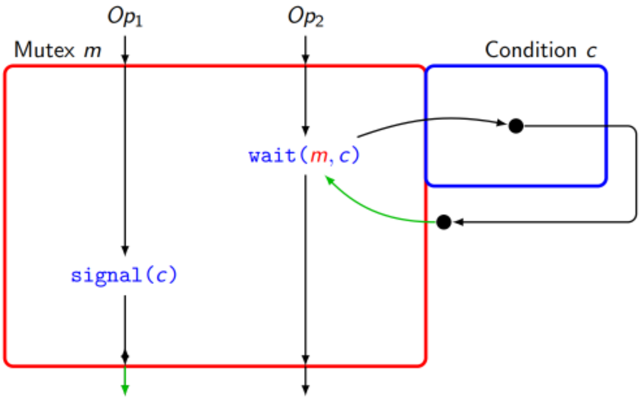
In Pthreads, a thread attempts to obtain a mutex with a call to ***pthread_mutex_lock***, and it relinquishes the mutex with a call to ***pthread_mutex_unlock***. When a thread attempts to obtain a mutex that is already in use, it blocks in the call to pthread mutex lock. This means that it remains idle in the call to pthread mutex lock until the system gives it the lock.

A **semaphore** is an unsigned int together with two operations: ***sem_wait*** and **sem post**. If the semaphore is positive, a call to sem wait simply decrements the semaphore, but if the semaphore is zero, the calling thread blocks until the semaphore is positive, at which point the semaphore is decremented and the thread returns from the call. The ***sem_post*** operation increments the semaphore, so a semaphore can be used as a mutex with ***sem_wait*** corresponding to ***pthread_mutex_lock*** and ***sem_post*** corresponding to ***pthread_mutex_unlock***. However, semaphores are more powerful than mutexes since they can be initialized to any nonnegative value. Furthermore, since there is no "ownership" of a semaphore, any thread can "unlock" a locked semaphore. We saw that semaphores can be easily used to implement producer-consumer synchronization. In producer-consumer synchronization, a "consumer" thread waits for some condition or data created by a "producer" thread before pro-ceeding. Semaphores are not part of Pthreads. In order to use them, we need to include the semaphore.h header file.

Monitor begrebet (mutex + condition)

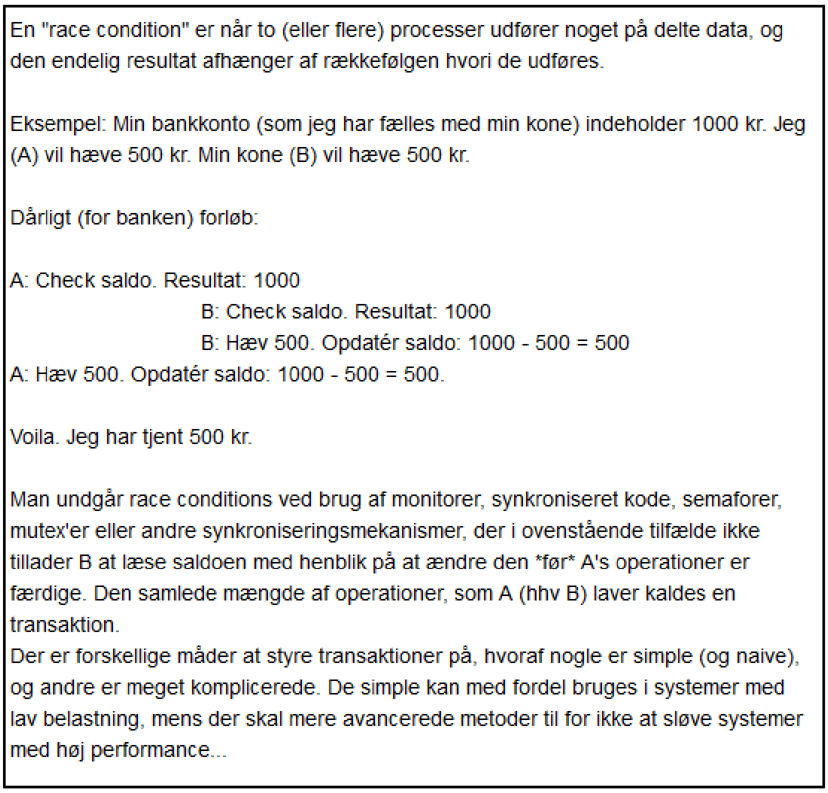
Idea: *Monitor* = *object* + *critical region*

- Java: **synchronized** + `wait()/notify()`
- Pthreads: `Mutex` + `Condition` variables



When multiple threads are executing, the order in which the statements are executed by the different threads is usually **nondeterministic(ikke samme efter hverkørsel)**. When nondeterminism results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. One of our most important tasks in writing shared-memory programs is identifying and correcting race conditions. A **..** is a block of code that updates a shared resource that can only be updated by one thread at a time, so the execution of code in a critical section should, effectively, be executed as serial code. Thus, we should try to design our programs so that they use them as infrequently as possible, and the critical sections we do use should be as short as possible.

Race condition:



En "race condition" er når to (eller flere) processer udfører noget på delte data, og den endelig resultat afhænger af rækkefølgen hvori de udføres.

Eksempel: Min bankkonto (som jeg har fælles med min kone) indeholder 1000 kr. Jeg (A) vil hæve 500 kr. Min kone (B) vil hæve 500 kr.

Dårligt (for banken) forløb:

A: Check saldo. Resultat: 1000
                    B: Check saldo. Resultat: 1000
                    B: Hæv 500. Opdatér saldo: 1000 - 500 = 500
A: Hæv 500. Opdatér saldo: 1000 - 500 = 500.

Voila. Jeg har tjent 500 kr.

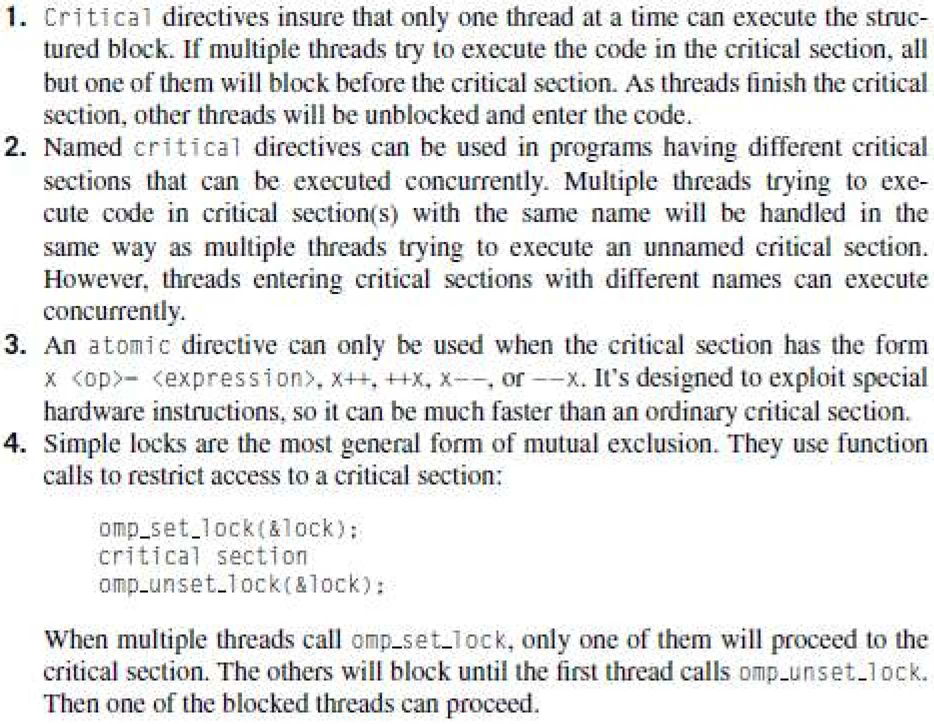
Man undgår race conditions ved brug af monitorer, synkroniseret kode, semaforer, mutex'er eller andre synkroniseringsmekanismer, der i ovenstående tilfælde ikke tillader B at læse saldoen med henblik på at ændre den *før* A's operationer er færdige. Den samlede mængde af operationer, som A (hhv B) laver kaldes en transaktion.
Der er forskellige måder at styre transaktioner på, hvoraf nogle er simple (og naive), og andre er meget komplicerede. De simple kan med fordel bruges i systemer med lav belastning, mens der skal mere avancerede metoder til for ikke at sløve systemer med høj performance...

Læse/skrive-lås begrebet

A read-write lock is used when it's safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

OpenMP direktiver

- An atomic directive: Specifies that a memory location that will be updated atomically and is very similar to the critical directive. The major difference is that, while the critical directive applies to anything that you would like to do one thread at a time, the atomic directive only applies to memory read/write operations.*(#pragma omp atomic)
- A for directive (parallel for directive):Clauses the work done in a for loop inside a parallel region to be divided among threads.
- A barrier directive:Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.



1. `Critical` directives insure that only one thread at a time can execute the structured block. If multiple threads try to execute the code in the critical section, all but one of them will block before the critical section. As threads finish the critical section, other threads will be unblocked and enter the code.
2. Named `critical` directives can be used in programs having different critical sections that can be executed concurrently. Multiple threads trying to execute code in critical section(s) with the same name will be handled in the same way as multiple threads trying to execute an unnamed critical section. However, threads entering critical sections with different names can execute concurrently.
3. An `atomic` directive can only be used when the critical section has the form `x <op>= <expression>`, `x++`, `++x`, `x--`, or `--x`. It's designed to exploit special hardware instructions, so it can be much faster than an ordinary critical section.
4. Simple locks are the most general form of mutual exclusion. They use function calls to restrict access to a critical section:

```
omp_set_lock(&lock);
critical section
omp_unset_lock(&lock);
```

   When multiple threads call `omp_set_lock`, only one of them will proceed to the critical section. The others will block until the first thread calls `omp_unset_lock`. Then one of the blocked threads can proceed.

Trådpulje

Tråd puljer, er når et vist antal tråde bliver skabt ved start, som derefter venter en opgave, der skal tildeles. Når en ny opgave ankommer, det vågner, fuldfører opgaven og går tilbage til at vente. Derved undgår de relativt dyre skabelse og destruktion tråd funktioner til hver opgave udføres, og tager tråden styring af anvendelsen udviklerens hånd og overlader det til et bibliotek eller det operativsystem, der er bedre egnet til at optimere tråd ledelse.

Begreber fra artikel

LAN (Local-area network)
Allow thousands of machines within a building or campus to be connected in such a way that small amounts of information can be transferred in a few microseconds or so. Larger amounts of data can be moved between machines at rates of billions of bits per second (bps)

WAN (wide-area network)
allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps, and sometimes even faster

Group membership

- Open group

    Any node is allowed to join the distributed system, effectively meaning that it can send messages to any other node in the system

- Close group

    Only the members of that group can communicate with each other and a separate mechanism is needed to let a node join or leave the group

Admission control can be difficult. First, a mechanism is needed to authenticate a node, and if not properly designed managing authentication can easily create a scalability bottleneck. Second, each node must, in principle, check if it is indeed communicating with another group member and not, for example, with an intruder aiming to create havoc. Finally, considering that a member can easily communicate with nonmembers, if confidentiality is an issue in the communication within the distributed system, we may be facing trust issues.

Overlay network and peer-to-peer (P2P)
Message passing is then done through TCP/IP or UDP channels, but higher-level facilities may be available as well.

- Structered overlay

    In this case, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring.

- Unstructered overlay

    In these overlays, each node has a number of references to randomly selected other nodes

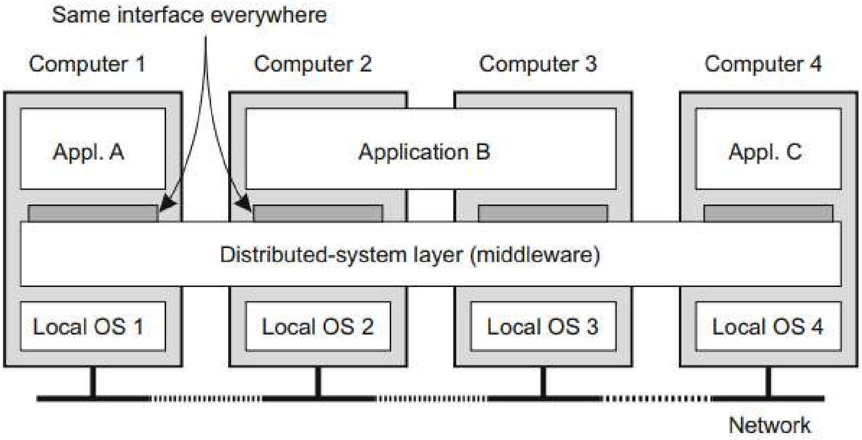
In any case, an overlay network should in principle always be connected, meaning that between any two nodes there is always a communication path allowing those nodes to route messages from one to the other. A well-known class of overlays is formed by **peer-to-peer (P2P)** networks. **Peer-to-peer** networks are often associated with distribution of media files such as audio and video. In other cases, the technology is used for distributing large amounts of data, as in the case of software updates, backup services, and data synchronization across multiple servers

Distribution transparency
Important design goal of distributed system. Offering a single coherent view is often challenging enough.

For example, it requires that an end user would not be able to tell exactly on which computer a process is currently executing, or even perhaps that part of a task has been spawned off to another process executing somewhere else. Likewise, where data is stored should be of no concern, and neither should it matter that the system may be replicating data to enhance performance.

Middelware
Seperate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network.

The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment



Atomic transaction.
In this case, the application developer need only specify the remote services involved, and by following a standardized protocol, the **middleware** makes sure that every service is invoked, or none at all.

Groupware
Software for collaborative editing, teleconferencing, and so on, as is illustrated by multinational software-development companies that have outsourced much of their code production to Asia

Transparency

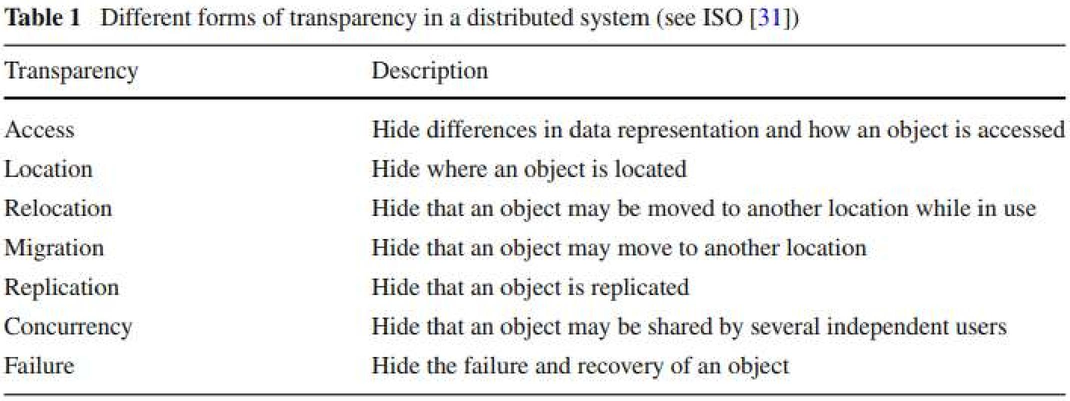


Table 1 Different forms of transparency in a distributed system (see ISO [31])

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Copy-before-use principle
According to this principle, data can be accessed only after they have been transferred to the machine

of the process wanting that data. Moreover, modifying a data item should not be done. Instead, it can only be updated to a new version

IDL (Interface definition language)
Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are given in an informal way by means of natural language. If properly specified, an interface definition allows an arbitrary process that needs a certain interface, to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate components that operate in exactly the same way.

Open distributed system
Essentially a system that offers components that can easily be used by, or integrated into other systems. At the same time, an open distributed system itself will often consist of components that originate from elsewhere.

Interoperability
characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard

Portability
characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

Exemption
When the cache fills up, which data is to be removed so that newly fetched pages can be stored?

Sharing
Does each browser make use of a private cache, or is a cache to be shared among browsers of different users?

Refreshing
When does a browser check if cached data is still up-to-date? Caches are most effective when a browser can return pages without having to contact the original Web site. However, this bears the risk of returning stale data. Note also that refresh rates are highly dependent on which data is actually cached: whereas timetables for trains hardly change, this is not the case for Web pages showing current highway-traffic conditions, or worse yet, stock prices

Size scalability
A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance

- The computational capacity, limited by the CPUs
- The storage capacity, including the transfer rate between CPUs and disks
- The network between the user and the centralized service

Geographical scalability
A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed. A wide-area system, we need to take into account that interprocess communication may be hundreds of milliseconds, three orders of magnitude slower. Building applications using synchronous communication in wide-area systems requires a great deal of care (and not just a little patience), notably with a rich interaction pattern between client and server

Administrative scalability
An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security

Scaling up
Simply improving their capacity (e.g., by increasing memory, upgrading CPUs, or replacing network modules).

Scaling out
Expanding the distributed system by essentially deploying more machines, there are basically only three techniques we can apply:

- Hiding communication latencies

    Latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote-service requests as much as possible.

- Distribution of work (Partitioning and distribution)

    Another important scaling technique. It involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system

- Replication

    Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before. For instance: caching but consequently, caching and replication leads to consistency problems

Partitioning and distribution
Another important scaling technique. It involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system

Assumptions not to make in distributed systems

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

DSM system (Distributed shared-memory multicomputers)

In essence, a DSM system allows a processor to address a memory location at another computer as if it were local memory. This can be achieved using existing techniques available to the operating system, for example, by mapping all main-memory pages of the various processors into a single virtual address space. Whenever a processor A addresses a page located at another processor B, a page fault occurs at A allowing the operating system at A to fetch the content of the referenced page at B in the same way that it would normally fetch it locally from disk. At the same time, processor B would be informed that the page is currently not accessible.

This elegant idea of mimicking shared-memory systems using multicomputers eventually had to be abandoned for the simple reason that performance could never meet the expectations of programmers.

RMI (Remote method invocations)

An RMI is essentially the same as an RPC, except that it operates on objects instead of functions.

Pervasive systems

Mobile and embedded computing devices are generally referred to as pervasive systems. They are naturally also distributed systems. What makes them unique in comparison to the computing and information systems, is that the separation between users and system components is much more blurred. There is often no single dedicated interface, such as a screen/keyboard combination. Instead, a pervasive system is often equipped with many sensors that pick up various aspects of a user's behavior.

Many devices in pervasive systems are characterized by being small, battery powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices.

Model for distribueret beregning og kunne benytte begreberne global tilstand og snit (cut)

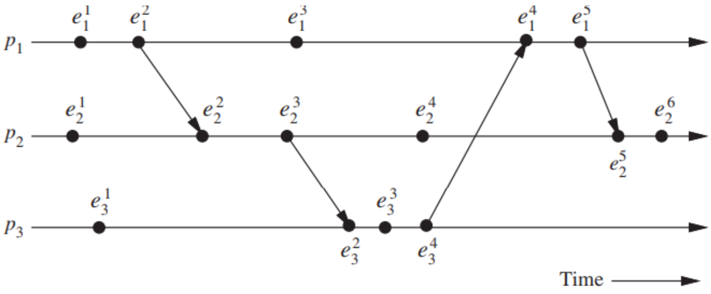
The global state

The global state of a distributed computation is composed of the states of the processes and the communication channels. It is a collection of the local states of its components. There is no shared memory.

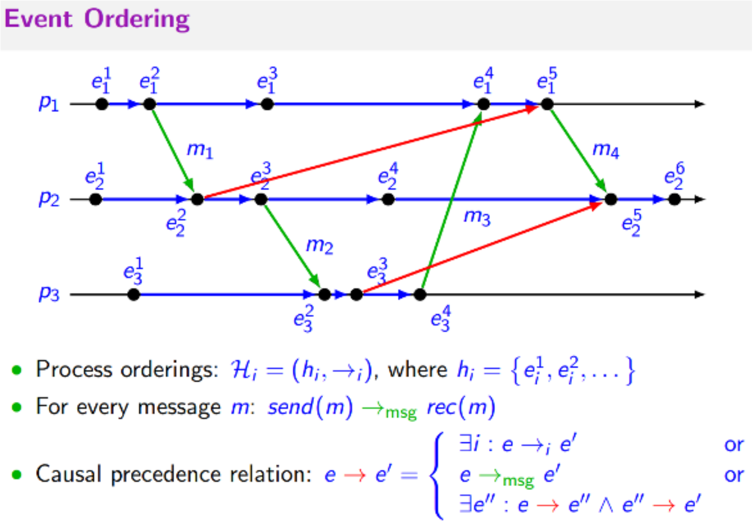
A global state of the distributed system is called a *checkpoint.*

Recording the **global state** of a distributed system is an important paradigm when one is interested in analyzing, monitoring, testing, or verifying properties of distributed applications, systems, and algorithms

The execution of a process consists of a sequential execution of its actions. The actions are atomic and the actions of a process are modeled as three types of events, namely, **internal events, message send events, and message receive events**



In this figure, for process p1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.



Event Ordering

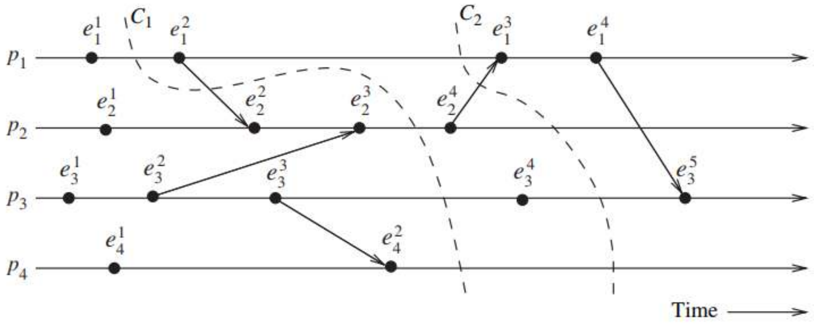
- Process orderings: $\mathcal{H}_i = (h_i, \rightarrow_i)$, where $h_i = \{e_i^1, e_i^2, \dots\}$
- For every message m: $send(m) \rightarrow_{msg} rec(m)$
- Causal precedence relation: $e \rightarrow e' = \begin{cases} \exists i : e \rightarrow_i e' & \text{or} \\ e \rightarrow_{msg} e' & \text{or} \\ \exists e'' : e \rightarrow e'' \wedge e'' \rightarrow e' \end{cases}$

Cut

In the space–time diagram of a distributed computation, a zigzag line joining one arbitrary point on each process line is termed a cut in the computation. Such a line slices the space–time diagram, and thus the set of events in the distributed computation, into a **PAST** and a **FUTURE**. The **PAST** contains all the events to the left of the cut and the **FUTURE** contains all the events to the right of the cut. For a cut C, let **PAST(C)** and **FUTURE(C)** denote the set of events in the **PAST** and **FUTURE** of C, respectively. Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space–time diagram

A **consistent global state** corresponds to a **cut** in which every message received in the PAST of the cut was sent in the PAST of that cut. Such a cut is known as a **consistent cut.**

A **cut is inconsistent** if a message crosses the cut from the FUTURE to the PAST.



For example, the space–time diagram of the figure shows two cuts, C1 and C2. C1 is an inconsistent cut, whereas C2 is a consistent cut

FIFO and non-FIFO

In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel. In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

Casual ordering

Casual ordering model is based on Lamport's "happens before" relation. For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency. Without causal ordering, each update must be checked to ensure that database consistency is not being violated. Causal ordering eliminates the need for such checks.

If A -> B and B -> C then A -> C

Process communications

There are two basic models of process communications synchronous and asynchronous.

- The synchronous communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process. The sender process resumes execution only after it learns that the receiver process has accepted the message. Thus, the sender and the receiver processes must synchronize to exchange a message
- Asynchronous communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message. After having sent a message, the sender process does not wait for the message to be delivered to the receiver process. The message is buffered by the system and is delivered to the receiver process when it is ready to accept the message. A buffer overflow may occur if a process sends a large number of messages in a

burst to another process

Neither of the communication models is superior to the other. Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver. However, an implementation of asynchronous communication requires more complex buffer management due to higher degree of parallelism and non-determinism. It is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications. Algorithms are likely to be much larger. Synchronous communication is simpler to handle. However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks

Logisk tid og kunne tildele konsistente tider til en distribueret beregning

System clocks
A system of logical clocks consists of a time domain **T** and a logical clock **C**. Elements of **T** form a partially ordered set over a relation <. This relation is usually called the happened before or causal precedence. Intuitively, this relation is analogous to the earlier than relation provided by the physical time. The logical clock **C** is a function that maps an event **e** in a distributed system to an element in the time domain **T**, denoted as **C(e)** and called the timestamp of **e**

Rules all logical clock system implements

- **R1** This rule governs how the **local logical clock** is updated by a process when it executes an event (send, receive, or internal).
- **R2** This rule governs how a process updates its **global logical clock** to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Scalar time
Attempt to totally order events in a distributed system. Time domain in this representation is the set of non-negative integers. The logical local clock of a process pi and its local view of the global time are squashed into one integer variable **Ci.**

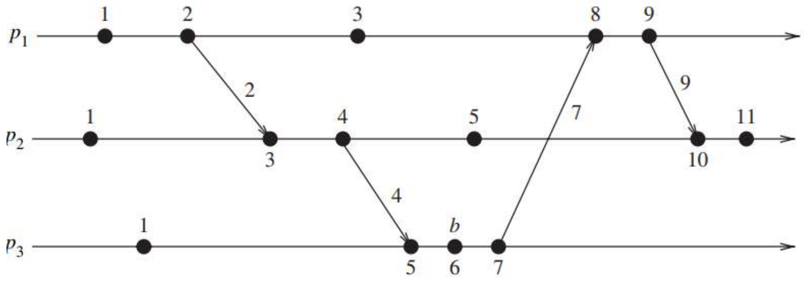
R1 and **R2** to update clock with scalar time:

- **R1** Before executing an event (send, receive, or internal), process pi executes the following:
  Ci = Ci +d d > 0
  In general, every time R1 is executed, d can have a different value, and this value may be application-dependent. However, typically d is kept at 1 because this is able to identify the time of each event uniquely at a process, while keeping the rate of increase of d to its lowest level.
- **R2** Each message piggybacks the clock value of its sender at sending time. When a process pi receives a message with timestamp Cmsg, it executes the following actions:
  1. Ci = max(Ci, C_msg);
  2. execute R1;
  3. deliver the message

Scalar clocks can be used to totally order events in a distributed system. The main problem in totally ordering events is that two or more events at different processes may have an identical timestamp. To solve this we need a tie-breaking mechanism: The lower the process identifier in the ranking, the higher the priority.

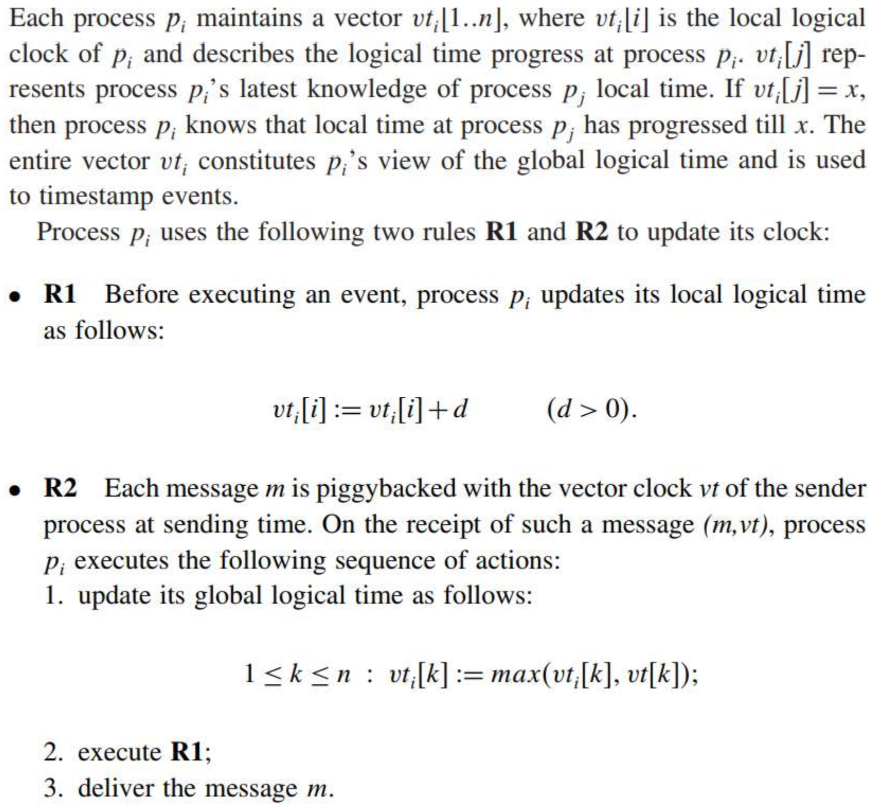
The system of scalar clocks is **not strongly consistent.** The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.

Figure 3.1 shows the evolution of scalar time with $d=1$.



Kendskab til begrebet vektor-tid

In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.



Each process p_i maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i. $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time. If $vt_i[j] = x$, then process p_i knows that local time at process p_j has progressed till x. The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

Process p_i uses the following two rules **R1** and **R2** to update its clock:

- **R1** Before executing an event, process p_i updates its local logical time as follows:

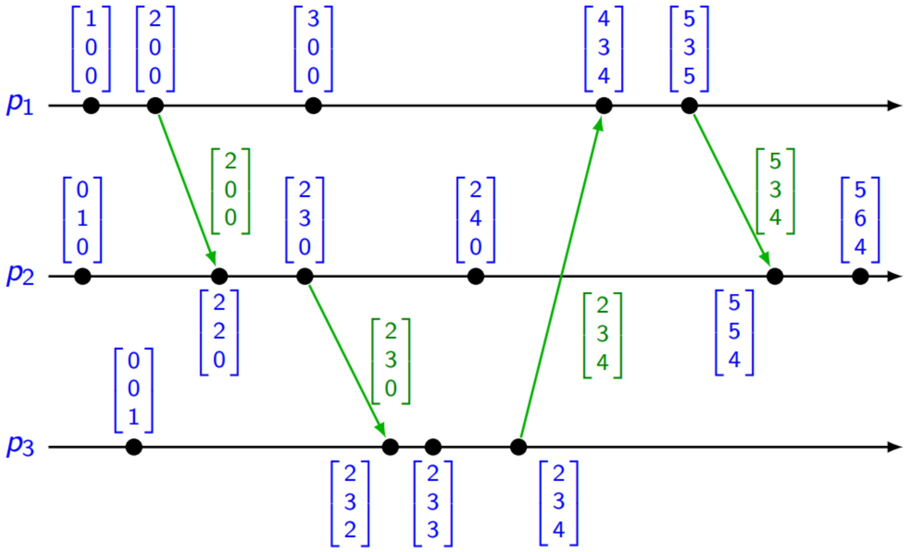
$$vt_i[i] := vt_i[i] + d \qquad (d > 0).$$

- **R2** Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message (m,vt), process p_i executes the following sequence of actions:
  1. update its global logical time as follows:

$$1 \le k \le n \ : \ vt_i[k] := max(vt_i[k], vt[k]);$$

  2. execute **R1**;
  3. deliver the message m.

The timestamp associated with an event is the value of the vector clock of its process when the event is executed. Initially, a vector clock is [0, ,0, 0..., 0].



The system of vector clocks is **strongly consistent.**

Begrebet globalt øjebliksbillede (snapshot).

The distributed nature of the local clocks and local memory makes it difficult to record the global state of the system efficiently

Reasons for recording globalstate:

- Detection of stable properties such as deadlocks and termination.
- Failure recovery
- Debugging
  - Industrial process control
  - Setting distributed breakpoints
  - Protocol specification and verification
  - Discarding obsolete information

Snapshot recording method has been used in the distributed debugging facility and records the global state.

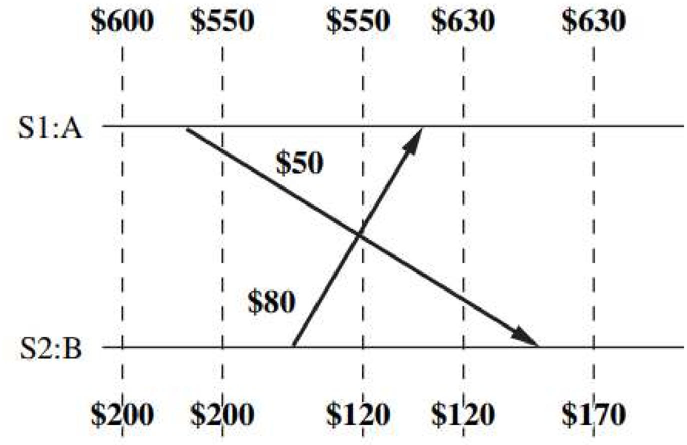
Snapshot recording algorithm:

- Various communication models such as FIFO communication channels, non-FIFO communication channels, and causal delivery of messages. They are called snapshots algorithms and records the global state.

Example of a problem:

- Time t0: Initially, Account A = $600, Account B = $200, C12 = $0, C21 = $0.
- Time t1: Site S1 initiates a transfer of $50 from Account A to Account B.
  Account A is decremented by $50 to $550 and a request for $50 credit to Account B is sent on Channel C12 to site S2. Account A = $550, Account B = $200, C12 = $50, C21 = $0
- Time t2: Site S2 initiates a transfer of $80 from Account B to Account A.
  Account B is decremented by $80 to $120 and a request for $80 credit to Account A is sent on Channel C21 to site S1.
  Account A = $550, Account B = $120, C12 = $50, C21 = $80.
  - Time t3: Site S1 receives the message for a $80 credit to Account A and updates Account A. Account A = $630, Account B = $120, C12 = $50, C21 = $0.
  - Time t4: Site S2 receives the message for a $50 credit to Account B and updates Account B. Account A = $630, Account B = $170, C12 = $0, C21 = $0.

Suppose the local state of Account A is recorded at time t0 to show $600 and the local state of Account B and channels C12 and C21 are recorded at time t2 to show $120, $50, and $80, respectively. Then the recorded global state shows $850 in the system. An extra $50 appears in the system. The reason for the inconsistency is that Account A's state was recorded before the $50 transfer to Account B using channel C12 was initiated, whereas channel C12's state was recorded after the $50 transfer was initiated



	t_0	t_1	t_2	t_3	t_4
C_{12}	$0	$50	$50	$50	$0
C_{21}	$0	$0	$80	$0	$0

Chandy-Lamports algoritme

This was the first snapshot algorithm to record a global state. There are at least three variations of this.

L1: How to distinguish between the messages to be recorded in the snapshot (either in a channel state or a process state) from those not to be recorded?

- The answer to this comes from conditions **C1** and **C2** as follows: Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**). Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

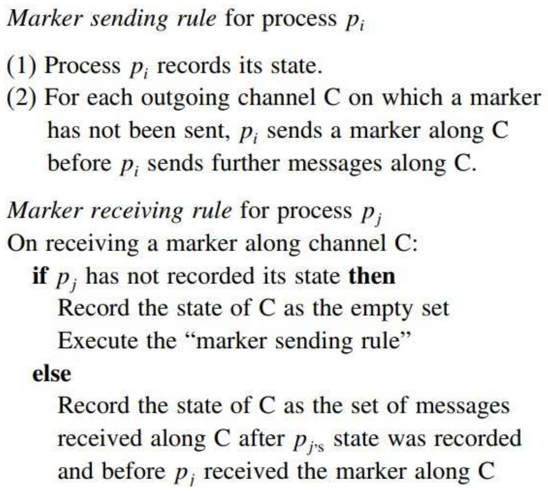
L2: How to determine the instant when a process takes its snapshot?

- The answer to this comes from condition **C2** as follows: A process pj must record its snapshot before processing a message mij that was sent by process pi after recording its snapshot.

The Chandy-Lamport algorithm uses a control message, called a marker. After a site has recorded its snapshot, it sends a marker along all of its outgoing channels before sending out any more messages. Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot (i.e., channel state or process state) from those not to be recorded in the snapshot. This addresses issue **L1**.

The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition **C2**. Since all messages that follow a marker on channel Cij have been sent by process pi after pi has taken its snapshot, process pj must record its snapshot no later than when it receives a marker on channel Cij. In general, a process must record its snapshot no later than when it receives a marker on any of its incoming channels. This addresses issue **L2**

Algorithm



```
Marker sending rule for process $p_i$

(1) Process $p_i$ records its state.
(2) For each outgoing channel C on which a marker
     has not been sent, $p_i$ sends a marker along C
     before $p_i$ sends further messages along C.

Marker receiving rule for process $p_j$
On receiving a marker along channel C:
   if $p_j$ has not recorded its state then
      Record the state of C as the empty set
      Execute the "marker sending rule"
   else
      Record the state of C as the set of messages
      received along C after $p_{j's}$ state was recorded
      and before $p_j$ received the marker along C
```

Correctness

To prove the correctness of the algorithm, we show that a recorded snapshot satisfies conditions C1 and C2. Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot. Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel. Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition C2 is satisfied.

When a process pj receives message mij that precedes the marker on channel Cij, it acts as follows: if process pj has not taken its snapshot yet, then it includes mij in its recorded snapshot. Otherwise, it records mij in the state of the channel Cij. Thus, condition C1 is satisfied.

Complexity

The recording part of a single instance of the algorithm requires O(e) messages and O(d) time, where e is the number of edges in the network and d is the diameter of the network