Concurrent Programming

Romolo Marotta

Data Centers and
High Performance Computing

SAPIENZA
UNIVERSITÀ DI ROMA

Amdahl Law—Fixed-size Model (1967)

- The workload is fixed: it studies how the behaviour of the *same* program varies when adding more computing power

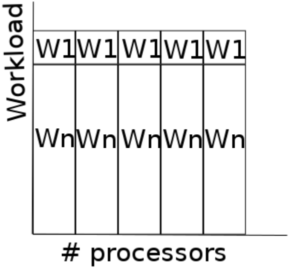
$$S_{Amdahl} = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + (1-\alpha)\frac{T_s}{p}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

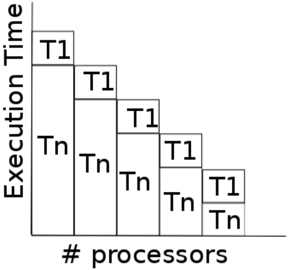
- where:

  $\alpha \in [0,1]$: Serial fraction of the program
  $p \in \mathbb{N}$: Number of processors
  T_s : Serial execution time
  T_p : Parallel execution time

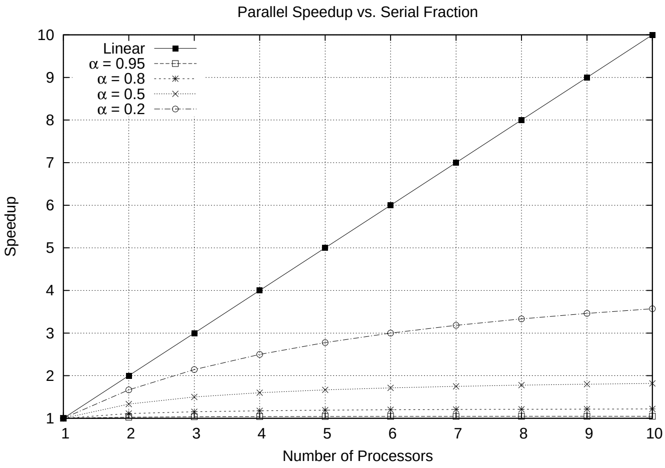
- It can be expressed as well vs. the *parallel fraction* $P = 1 - \alpha$

Fixed-size Model





Speed-up According to Amdahl



How Real is This?

$$\lim_{p \to \infty} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha}$$

How Real is This?

$$\lim_{p\to\infty} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha}$$

- So if the sequential fraction is 20%, we have:

$$\lim_{p\to\infty} = \frac{1}{0.2} = 5$$

- Speedup 5 using *infinte* processors!

Gustafson Law—Fixed-time Model (1989)

- The execution time is fixed: it studies how the behaviour of a *scaled* program varies when adding more computing power

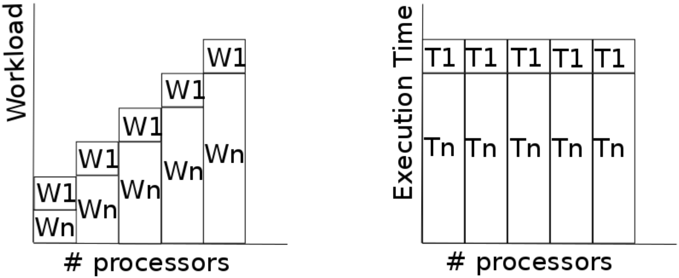
$$W' = \alpha W + (1-\alpha)pW$$

$$S_{Gustafson} = \frac{W'}{W} = \alpha + (1-\alpha)p$$

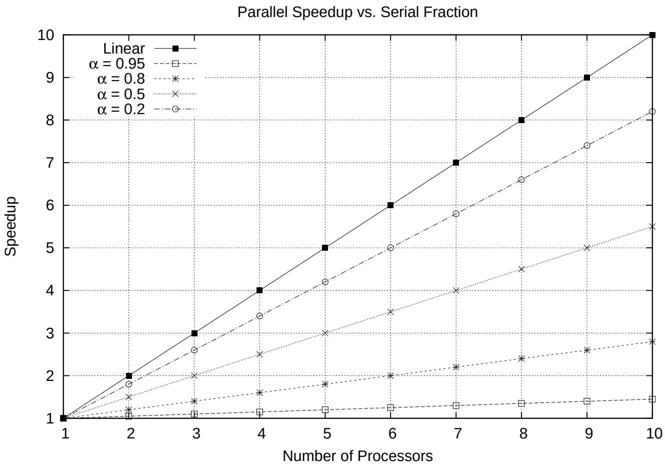
- where:

$\alpha \in [0,1]$: Serial fraction of the program
$p \in \mathbb{N}$: Number of processors
W : Original Workload
$W^{'}$: Scaled Workload

Fixed-time Model



Speed-up According to Gustafson



Amdahl vs. Gustafson—a Driver's Experience

Amdahl Law:

A car is traveling between two cities 60 Kms away, and has already traveled half the distance at 30 Km/h. No matter how fast you drive the last half, it is impossible to achieve 90 Km/h average speed before reaching the second city. It has already taken you 1 hour and you only have a distance of 60 Kms total: Going infinitely fast you would only achieve 60 Km/h.

Gustafson Law:

A car has been travelling for some time at less than 90 Km/h. Given enough time and distance to travel, the car's average speed can always eventually reach 90 Km/h, no matter how long or how slowly it has already traveled. If the car spent one hour at 30 Km/h, it could achieve this by driving at 120 Km/h for two additional hours.

Sun, Ni Law—Memory-bounded Model (1993)

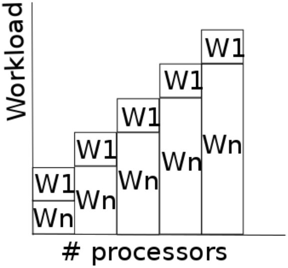
- The workload is scaled, bounded by *memory*

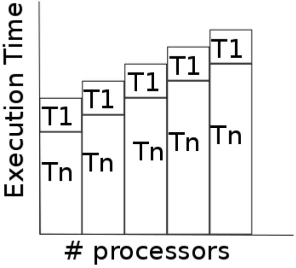
$$S_{Sun-Ni} = \frac{\text{sequential time for Workload } W^*}{\text{parallel time for Workload } W^*} =$$

$$= \frac{\alpha W + (1-\alpha)G(p)W}{\alpha W + (1-\alpha)G(p)\frac{W}{p}} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- where:
  - $G(p)$ describes the workload increase as the memory capacity increases
  - $W^* = \alpha W + (1-\alpha)G(p)W$

Memory-bounded Model





Speed-up According to Sun, Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

Speed-up According to Sun, Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If G(p) = 1

$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

Speed-up According to Sun, Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If G(p) = 1

$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- If G(p) = p

$$S_{Gustafson} = \alpha + (1-\alpha)p$$

Speed-up According to Sun, Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If G(p) = 1

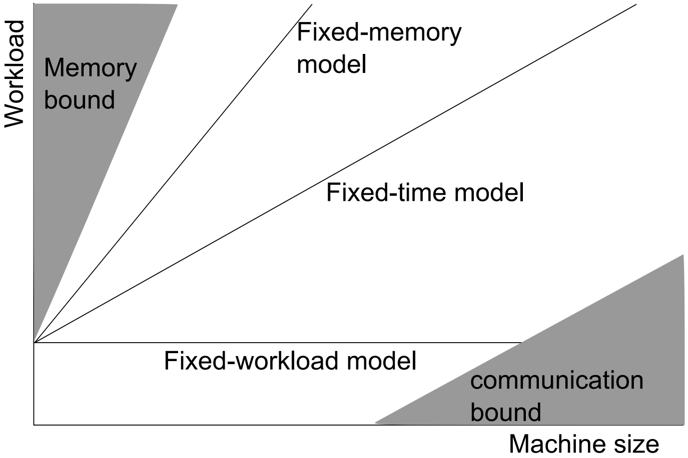
$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- If G(p) = p

$$S_{Gustafson} = \alpha + (1-\alpha)p$$

In general $G(p) > p$ gives a higher scale-up

Application Model for Parallel Computers



Scalability

- Efficiency $E = \frac{\text{speed-up}}{\text{number of processors}}$
- **Strong Scalability**: If the efficiency is kept fixed while increasing the number of processes and maintainig fixed the problem size
- **Weak Scalability**: If the efficiency is kept fixed while increasing at the same rate the problem size and the number of processes

Superlinear Speedup

- Can we have a Speed-up $> p$?

Superlinear Speedup

- Can we have a Speed-up $> p$? Yes!
  - Workload increases more than computing power ($G(p) > p$)
  - Cache effect: larger accumulated cache size. More or even all of the working set can fit into caches and the memory access time reduces dramatically
  - RAM effect: enables the dataset to move from disk into RAM drastically reducing the time required, e.g., to search it.
  - The parallel algorithm uses some search like a random walk: the more processors that are walking, the less distance has to be walked in total before you reach what you are looking for.

Parallel Programming

- Ad-hoc concurrent programming languages
- Development Tools
  - Compilers try to optimize the code
  - MPI, OpenMP, Libraries...
  - Tools to ease the task of debugging parallel code (gdb, valgrind, ...)
- Writing parallel code is for artists, not scientists!
  - There are approaches, not prepackaged solutions
  - Every machine has its own singularities
  - Every problem to face has different requisites
  - The most efficient parallel algorithm is **not** the most intuitive one

Ad-hoc languages

Ada	Alef	ChucK	Clojure	Curry
Cω	E	Eiffel	Erlang	Go
Java	Julia	Joule	Limbo	Occam
Orc	Oz	Pict	Rust	SALSA
Scala	SequenceL	SR	Unified Parallel C	XProc

Classical Approach to Concurrent Programming

- Based on blocking primitives
  - Semaphores
  - Locks acquiring
  - ...

PRODUCER

```
Semaphore p, c = 0;
Buffer b;

while(1) {
  <Write on b>
  signal(p);
  wait(c);
}
```

CONSUMER

```
Semaphore p, c = 0;
Buffer b;

while(1) {
   wait(p);
   <Read from b>
   signal(c);
}
```

Parallel Programs Properties

- **Safety**: *nothing wrong happens*
  - It's called **Correctness** as well

Parallel Programs Properties

- **Safety**: *nothing wrong happens*
  - It's called **Correctness** as well
- **Liveness**: *eventually something good happens*
  - It's called **Progress** as well

Correctness

- What does it mean for a program to be *correct*?
  - What's exactly a concurrent FIFO queue?
  - FIFO implies a strict temporal ordering
  - *Concurrent* implies an ambiguous temporal ordering
- Intuitively, if we rely on locks, changes happen in a non-interleaved fashion, resembling a sequential execution
- We can say a concurrent execution is *correct* only because we can associate it with a sequential one, which we know the functioning of
- A *concurrent execution* is correct if it is *equivalent* to a correct *sequential execution*

A simplyfied model of a concurrent system

- A concurrent system is a collection of sequential *threads* that communicate through shared data structures called *objects*.
- An object has a unique name and a set of primitive *operations*.
- An invocation of an operation *op* of the object x is written as

```
A op(args*) x
```

  where A is the invoking thread and $args*$ the sequence of arguments A
- A response to an operation invocation on x is written as

```
A ret(res*) x
```

  where A is the invoking thread and $res*$ the sequence of results

A simplyfied model of a concurrent execution

- A *history* is a sequence of *invocations* and *replies* generated on an *object* by a set of threads

A simplyfied model of a concurrent execution

- A *history* is a sequence of *invocations* and *replies* generated on an *object* by a set of threads
- A *sequential history* is a history where all the invocations have an immediate response

Sequential

```
H': A op()  x
    A ret() x
    B op()  x
    B ret() x
    A op()  y
    A ret() y
```

A simplyfied model of a concurrent execution

- A *history* is a sequence of *invocations* and *replies* generated on an *object* by a set of threads
- A *sequential history* is a history where all the invocations have an immediate response
- A *concurrent history* is a history that is not sequential

Sequential

```
H': A op()  x
    A ret() x
    B op()  x
    B ret() x
    A op()  y
    A ret() y
```

Concurrent

```
H: A op()  x
   B op()  x
   A ret() x
   A op()  y
   B ret() x
   A ret() y
```

A simplyfied model of a concurrent execution (2)

- A *process subhistory* $H|P$ of a history H is the subsequence of all events in H whose process names are P

```
H: A op()  x
   B op()  x
   A ret() x
   A op()  y
   B ret() x
   A ret() y
```

A simplyfied model of a concurrent execution (2)

- A *process subhistory* $H|P$ of a history H is the subsequence of all events in H whose process names are P

```
H: A op()  x

   A ret() x
   A op()  y

   A ret() y
```

A simplyfied model of a concurrent execution (2)

- A *process subhistory* $H|P$ of a history H is the subsequence of all events in H whose process names are P

```
H: A op()  x
   B op()  x
   A ret() x
   A op()  y
   B ret() x
   A ret() y
```

```
H|A: A op()  x
     A ret() x
     A op()  y
     A ret() y
```

A simplyfied model of a concurrent execution (2)

- A *process subhistory* $H|P$ of a history H is the subsequence of all events in H whose process names are P

```
H: A op()  x
   B op()  x
   A ret() x
   A op()  y
   B ret() x
   A ret() y
```

```
H|A: A op()  x
     A ret() x
     A op()  y
     A ret() y
```

- Process subhistories are always sequential

A simplyfied model of a concurrent execution (3)

- An *object subhistory* $H|x$ of a history H is the subsequence of all events in H whose object names are x

```
H: A op()  x
   B op()  x
   A ret() x
   A op()  y
   B ret() x
   A ret() y
```

A simplyfied model of a concurrent execution (3)

- An *object subhistory* $H|x$ of a history H is the subsequence of all events in H whose object names are x

```
H: A op()  x
   B op()  x
   A ret() x

   B ret() x
```

A simplyfied model of a concurrent execution (3)

- An *object subhistory* $H|x$ of a history H is the subsequence of all events in H whose object names are x

```
H: A op()   x
   B op()   x
   A ret()  x
   A op()   y
   B ret()  x
   A ret()  y
```

```
H|x:  A op()   x
      B op()   x
      A ret()  x
      B ret()  x
```

A simplyfied model of a concurrent execution (3)

- An *object subhistory* $H|x$ of a history H is the subsequence of all events in H whose object names are x

```
H: A op()   x              H|x:  A op()   x
   B op()   x                    B op()   x
   A ret()  x                    A ret()  x
   A op()   y                    B ret()  x
   B ret()  x
   A ret()  y
```

- Object subhistories are not necessarily sequential

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H: A op()  x        H': B op()  x
   B op()  x            B ret() x
   A ret() x            A op()  x
   A op()  y            A ret() x
   B ret() x            A op()  y
   A ret() y            A ret() y
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H: A op()   x         H':

   A ret()  x                 A op()   x
   A op()   y                 A ret()  x
                              A op()   y
   A ret()  y                 A ret()  y
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H: A op()   x          H':                   H|A:
                                           H'|A: A op()   x
   A ret()  x               A op()   x             A ret()  x
   A op()   y               A ret()  x             A op()   y
                            A op()   y             A ret()  y
   A ret()  y               A ret()  y
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H: A op()  x        H': B op()  x        H|A:
   B op()  x            B ret() x       H'|A: A op()  x
   A ret() x            A op()  x             A ret() x
   A op()  y            A ret() x             A op()  y
   B ret() x            A op()  y             A ret() y
   A ret() y            A ret() y
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H:                     H': B op()  x        H|A:
   B op()  x               B ret() x       H'|A: A op()  x
                                                 A ret() x
                                                 A op()  y
   B ret() x                                     A ret() y
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H:                    H': B op()   x        H|A:
   B op()   x             B ret()  x       H'|A: A op()   x
                                                 A ret()  x
                                                 A op()   y
   B ret()  x                                    A ret()  y

                                           H|B:
                                           H'|B: B op()   x
                                                 B ret()  x
```

Equivalence between histories

- Two histories H and H' are equivalent if for every process P, $H|P = H'|P$

```
H: A op()   x
   B op()   x
   A ret()  x
   A op()   y
   B ret()  x
   A ret()  y
```

```
H': B op()   x
    B ret()  x
    A op()   x
    A ret()  x
    A op()   y
    A ret()  y
```

```
 H|A:
H'|A: A op()   x
      A ret()  x
      A op()   y
      A ret()  y

H|B:
H'|B: B op()   x
      B ret()  x
```

Correctness Conditions

- A *concurrent execution* is correct if it is *equivalent* to a correct *sequential execution*

⇒ A *history* is correct if it is *equivalent* to a *sequential history* which satisfies a set of correctness criteria

- A *correctness condition* specifies the set of correctness criteria

⇒ In order to implement correctly a concurrent object wrt a correctness condition, a programmer have to guarantee that every possible history on his implementation satisfies the correctness criteria

Sequential Consistency [Lamport 1970]

- A history is *sequentially consistent* if it is *equivalent* to a sequential history which *is correct according to the sequential definition of the objects*
- An *object* is sequentially consistent if every valid history associated with its usage is sequentially consistent

Sequential Consistency [Lamport 1970] (Example 1)

- x is a FIFO queue with Enqueue (Enq) and Dequeue (Deq) operations

Sequential Consistency [Lamport 1970] (Example 1)

- x is a FIFO queue with Enqueue (Enq) and Dequeue (Deq) operations
- Is the history H sequentially consistent?

```
H:  A Enq(1) x
    A ret()  x
    B Enq(2) x
    B ret()  x
    B Deq()  x
    B ret(2) x
```

Sequential Consistency [Lamport 1970] (Example 1)

- x is a FIFO queue with Enqueue (Enq) and Dequeue (Deq) operations
- Is the history H sequentially consistent? Yes!

```
H:  A Enq(1) x        H': B Enq(2) x
    A ret()  x            B ret()  x
    B Enq(2) x            A Enq(1) x
    B ret()  x            A ret()  x
    B Deq()  x            B Deq()  x
    B ret(2) x            B ret(2) x
```

Sequential Consistency [Lamport 1970] (Example 2)

```
H:  1.  A Enq(1) x
    2.  A ret()  x
    3.  A Enq(1) y
    4.  A ret()  y
    5.  B Enq(2) y
    6.  B ret()  y
    7.  B Enq(2) x
    8.  B ret()  x
    9.  A Deq()  x
   10.  A ret(2) x
   11.  B Deq()  y
   12.  B ret(1) y
```

Sequential Consistency [Lamport 1970] (Example 2)

```
H:  1.  A Enq(1) x
    2.  A ret()  x
    3.  A Enq(1) y
    4.  A ret()  y
    5.  B Enq(2) y
    6.  B ret()  y
    7.  B Enq(2) x
    8.  B ret()  x
    9.  A Deq()  x
   10.  A ret(2) x
   11.  B Deq()  y
   12.  B ret(1) y
```

```
H|x:  A Enq(1) x
      A ret()  x
      B Enq(2) x
      B ret()  x
      A Deq()  x
      A ret(2) x
```

Sequential Consistency [Lamport 1970] (Example 2)

```
H:  1.  A Enq(1) x
    2.  A ret()  x
    3.  A Enq(1) y
    4.  A ret()  y
    5.  B Enq(2) y
    6.  B ret()  y
    7.  B Enq(2) x
    8.  B ret()  x
    9.  A Deq()  x
   10.  A ret(2) x
   11.  B Deq()  y
   12.  B ret(1) y
```

```
H|x:  A Enq(1) x
      A ret()  x
      B Enq(2) x
      B ret()  x
      A Deq()  x
      A ret(2) x
```

```
H|y:  A Enq(1) y
      A ret()  y
      B Enq(2) y
      B ret()  y
      B Deq()  y
      B ret(1) y
```

Sequential Consistency [Lamport 1970] (Example 2)

- The composition of sequentially consistent histories is not necessarily sequential consistent

```
H:   1.  A Enq(1) x
     2.  A ret()  x
     3.  A Enq(1) y
     4.  A ret()  y
     5.  B Enq(2) y
     6.  B ret()  y
     7.  B Enq(2) x
     8.  B ret()  x
     9.  A Deq()  x
    10.  A ret(2) x
    11.  B Deq()  y
    12.  B ret(1) y
```

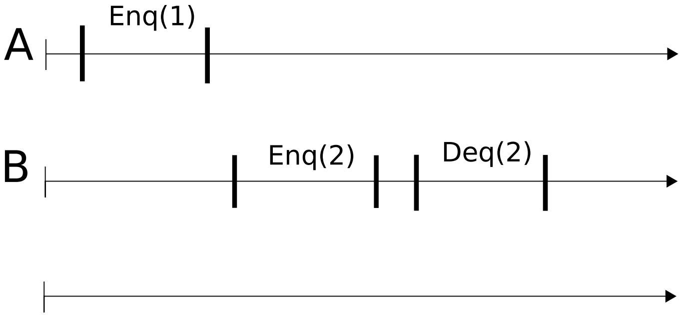
```
H|x:  A Enq(1) x
      A ret()  x
      B Enq(2) x
      B ret()  x
      A Deq()  x
      A ret(2) x
```

```
H|y:  A Enq(1) y
      A ret()  y
      B Enq(2) y
      B ret()  y
      B Deq()  y
      B ret(1) y
```

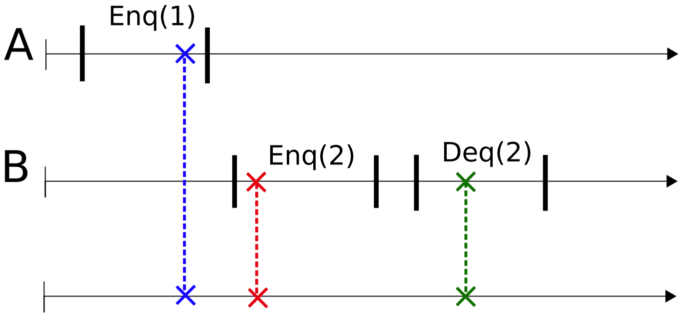
Linearizability [Herlihy 1990]

- A concurrent execution is *linearizable* if:
  - Each procedure appears to be executed in an indivisible point (*linearization point* between its invocation and completition
  - The order among those points is correct according to the sequential definition of objects

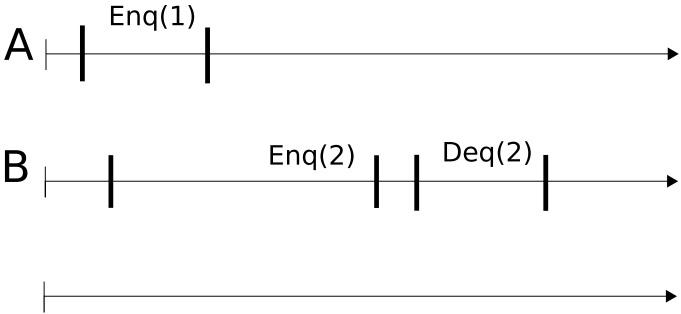
Linearizability [Herlihy 1990] (Example 1)



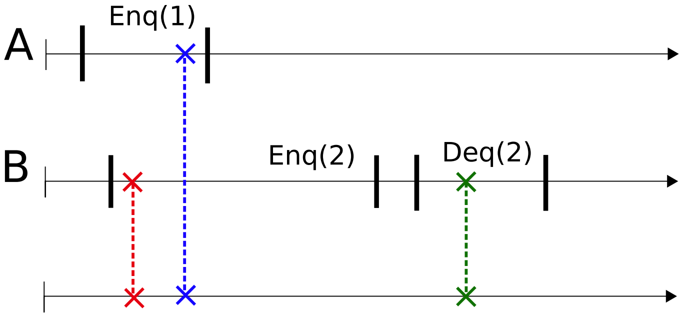
Linearizability [Herlihy 1990] (Example 1)



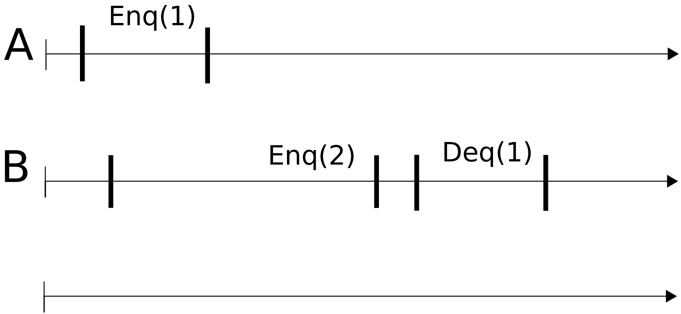
Linearizability [Herlihy 1990] (Example 1)



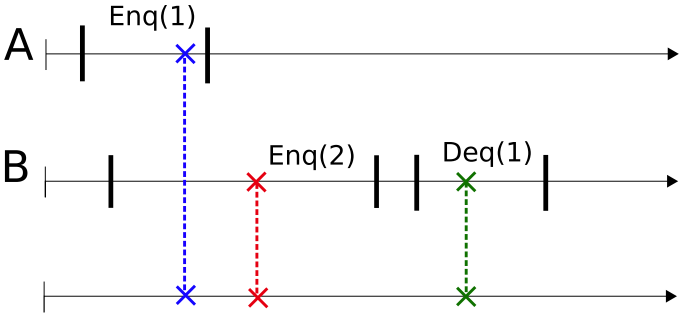
Linearizability [Herlihy 1990] (Example 1)



Linearizability [Herlihy 1990] (Example 1)



Linearizability [Herlihy 1990] (Example 1)



Linearizability [Herlihy 1990] (2)

- A history H is *linearizable* if it is equivalent to sequential history S such that:
  - S is correct according to the sequential definition of objects (H is sequential consistent)
  - If a response precedes an invocation in the original history, then it must precede it in the sequential one as well
- An *object* is linearizable if every valid history associated with its usage can be linearized

Linearizability [Herlihy 1990] (Example 2)

- Is the history H is linearizable?

```
H:  A Enq(1) x
    A ret()  x
    B Enq(2) x
    B ret()  x
    B Deq()  x
    B ret(2) x
```

Linearizability [Herlihy 1990] (Example 2)

- Is the history *H* is linearizable? No!

```
H:  A Enq(1) x
    A ret()  x
    B Enq(2) x
    B ret()  x
    B Deq()  x
    B ret(2) x
```

Linearizability [Herlihy 1990] (Example 2)

- Is the history H' is linearizable?

```
H:  A Enq(1)  x
    B Enq(2)  x
    A ret()   x
    B ret()   x
    B Deq()   x
    B ret(2)  x
```

Linearizability [Herlihy 1990] (Example 2)

- Is the history H' is linearizable? Yes!

H:		H':
	A Enq(1) x	B Enq(2) x
	B Enq(2) x	B ret() x
	A ret() x	A Enq(1) x
	B ret() x	A ret() x
	B Deq() x	B Deq() x
	B ret(2) x	B ret(2) x

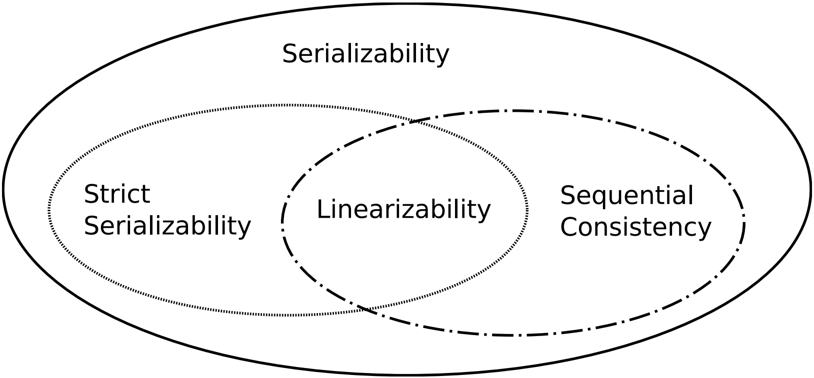
Linearizability Properties

- Linearizability requires:
  - Correctness with objects semantic (as Sequential Consistency)
  - Real-time order
- Linearizability $\Rightarrow$ Sequential Consistency
- The composition of linearizable histories is still linearizable

Quick look on transaction correctness conditions

- We can see a transaction as a set of procedures on different object that has to appear as atomic
- Serializability requires that transactions appear to execute sequentially, i.e., without interleaving.
  - A sort of sequential consistency for multi-object atomic procedures
- Strict-Serializability requires the transactions' order in the sequential history is compatible with their precedence order
  - A sort of linearizability for multi-object atomic procedures

Quick look on transaction correctness conditions (2)



Correctness Conditions (Incomplete) Taxonomy

	Sequential Consistency	**Linearizability**	**Serializability**	**Strict Serializability**
Equivalent to a sequential order	Y	Y	Y	Y
Respects program order in each thread	Y	Y	Y	Y
Consistent with real-time ordering	-	Y	-	Y
Can touch multiple objects atomically	-	-	Y	Y
Locality	-	Y	-	-

Progress Conditions

- **Deadlock-free**:
  Some thread acquires a lock eventually
- **Starvation-free**:
  Every thread acquires a lock eventually
- **Lock-free**:
  Some method call completes
- **Wait-free**:
  Every method call completes
- **Obstruction-free**:
  Every method call completes, if they execute in isolation

Maximum and Minimum Progress

- **Minimum** Progress:
  - *Some* method call completes eventually
- **Maximum** Progress:
  - *Every* method call completes eventually

Maximum and Minimum Progress

- **Minimum** Progress:
  - *Some* method call completes eventually
- **Maximum** Progress:
  - *Every* method call completes eventually

- Progress is a per-method property:
  - A real data structure can combine *blocking* and *wait-free* methods
  - For example, the Java Concurrency Package:
    - Skiplists
    - Hash Tables
    - Exchangers

Progress Taxonomy

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free		Deadlock-free

Scheduler's Role

Progress conditions on **multiprocessors**:

- Are not about guarantees provided by a method implementation
- Are about the *scheduling support* needed to provide maximum of minimum progress

Scheduler Requirements

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free		Deadlock-free

Scheduler Requirements

	Non-Blocking		Blocking
For everyone	Nothing	Thread executes alone	No thread locked in CS
For some	Nothing		No thread locked in CS

Dependent Progress

- A progress condition is said **dependent** if maximum (or minimum) progress requires scheduler support

- Otherwise it is called **independent**

Dependent Progress

- A progress condition is said **dependent** if maximum (or minimum) progress requires scheduler support

- Otherwise it is called **independent**

- Progress conditions are therefore not about guarantees provided by the implementations
- Programmers develop lock-free, obstruction-free or deadlock-free algorithms implicitly assuming that modern schedulers are benevolent, and that therefore every method call will eventually complete, as they were wait-free

Progress Taxonomy

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free		Deadlock-free

Progress Taxonomy

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free	Clash-Free	Deadlock-free

Progress Taxonomy

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free	Clash-Free	Deadlock-free

- The *Einsteinium* of progress conditions: it does not exists in nature and has no value
- It is known that clash freedom is a strictly weaker property than obstruction freedom

Concurrent Data Structures

- Developing data structures which can be concurrently accessed by more threads can significantly increase programs' performance
- Synchronization primitives must be avoided
- Result's correctness must be guaranteed (recall linearizability)
- We can rely on atomic operations provided by computer architectures