Need for Concurrency

Consider two processors:
- in 1997 the Pentium P55C had 4.5M transistors
- in 2006 the Itanium 2 had 1700M transistors

~ Intel could have built a processor with 256 Pentium cores in 2006

⚠️ However:
- most programs are not inherently parallel
  ~ parallelizing a program is between difficult and impossible
- correctly communicating between different cores is challenging
  ~ correctness of concurrent communication is very hard
  ▶ low-level aspects: locking algorithms must be correct
  ▶ high-level aspects: program may deadlock
- a program on $n$ cores runs $m \leq n$ times faster
  ~ all effort is voided if program runs no faster
  ▶ distributing work load is application specific

The free lunch is over

Single processors cannot be made much faster due to physical limitations.
The free lunch is over

Single processors cannot be made much faster due to physical limitations.

But Moore's law still holds for the number of transistors:
- they double every 18 months for the foreseeable future
- may translate into doubling the number of cores
- programs have to become parallel

Concurrency for the Programmer

How is concurrency exposed in a programming language?
- spawning of new concurrent computations
- communication between threads

Communication can happen in many ways:
- communication via shared memory (this lecture)
- atomic transactions on shared memory
- message passing

Learning Outcomes
- Happened-before Partial Order
- Sequential Consistency
- The MESI Cache Model
- Weak Consistency
- Memory Barriers
**Strict Consistency**

Assuming `foo` and `bar` are started on two cores operating in lock-step. Then **one** of the following may happen:

- A unique order between memory accesses is unrealistic in reality:
  - each conditional (and loop iteration) doubles the number of possible lock-step executions
  - processors use caches — lock-step assumption is violated since cache behavior depends on data

---

**Strict Consistency**

Assuming `foo` and `bar` are started on two cores operating in lock-step. Then **one** of the following may happen:

---

**Events in a Distributed System**

A process as a series of events [Lam78]: Given a distributed system of processes `P, Q, R, . . .`, each process `P` consists of events \( \bullet p_1, \bullet p_2, \ldots \)

---

**Strict Consistency**

Assuming `foo` and `bar` are started on two cores operating in lock-step. Then **one** of the following may happen:

- A unique order between memory accesses is unrealistic in reality:
  - each conditional (and loop iteration) doubles the number of possible lock-step executions
  - processors use caches — lock-step assumption is violated since cache behavior depends on data

- strict consistency is too strong to be realistic

Idea: state correctness in terms of what event may happen before another one
Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P, Q, R, \ldots$, each process $P$ consists of events $p_1, p_2, \ldots$.

Example:

**Definition**

If an event $p$ happened before an event $q$ then $p \rightarrow q$.

The Happened-Before Relation

Concurrency in Process Diagrams

Let $a \not< b$ abbreviate $\neg (a \rightarrow b)$.

**Definition**

Two distinct events $p$ and $q$ are said to be concurrent if $p \not< q$ and $q \not< p$.

Observe:

- $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q \land q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric (if $p \rightarrow q$ then $\neg (q \rightarrow p)$)
- $\rightarrow$ the $\rightarrow$ relation is a strict partial order

$p_1 \rightarrow r_4$ in the example

$p_2$ and $q_3$ are, in fact, concurrent since $p_2 \not< q_3$ and $q_3 \not< p_2$
Ordering

Let \( C \) be a **logical clock** that assigns a time-stamp \( C(p) \) to each event \( p \).

**Definition (Clock Condition)**

Function \( C \) satisfies the **clock condition** if for any events \( p, q \)

\[
p \rightarrow q \implies C(p) < C(q)
\]

For a distributed system the **clock condition** holds iff:

1. \( p_i \) and \( p_j \) are events of \( P \) and \( p_i \rightarrow p_j \) then \( C(p_i) < C(p_j) \)
2. \( p \) is the sending of a message by process \( P \) and \( q \) is the reception of this message by process \( Q \) then \( C(p) < C(q) \)

\( \rightarrow \) a logical clock \( C \) that satisfies the clock condition describes a **total order** \( a < b \) (with \( C(a) < C(b) \)) that **embeds** the strict partial order \( \rightarrow \)
Defining \( C \) Satisfying the Clock Condition

Given:

![Diagram]

<table>
<thead>
<tr>
<th>( e )</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C(e) )</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( e )</th>
<th>( q_1 )</th>
<th>( q_2 )</th>
<th>( q_3 )</th>
<th>( q_4 )</th>
<th>( q_5 )</th>
<th>( q_6 )</th>
<th>( q_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C(e) )</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( e )</th>
<th>( r_1 )</th>
<th>( r_2 )</th>
<th>( r_3 )</th>
<th>( r_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C(e) )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summing up Happened-Before Relations

We can model concurrency using processes and events:

- There is a happened-before relation between the events of each process
- There is a happened-before relation between communicating events
- Happened-before is a strict partial order
- A clock is a total strict order that embeds the happened-before partial order

Sequential Consistency on Multi-Processor Machines
Moving Away from Strict Consistency

Idea: use process diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:
- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
- ~~~ concisely represent some interleavings

We obtain a model for sequential consistency.

Definition: Sequential Consistency

Definition (Sequential Consistency Condition [Lam78])
The result of any execution is the same as if
- the operations of all the processors were executed in some sequential order and
- the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential Consistency applied to Multiprocessor Programs:
Given a program with \( n \) threads,
1. for fixed operation sequences \( p_{0}^{1}, p_{1}^{1}, \ldots, \) and \( p_{0}^{2}, p_{1}^{2}, \ldots \) and \( p_{0}^{3}, p_{1}^{3}, \ldots \) keeping the program order
2. executions obey the clock condition on the \( p_{i}^{j} \)
3. all executions have the same result

Yet, in other words:
- \( 1 \) defines the execution path of each thread
- each execution mentioned in \( 1 \) is one interleaving of processes
- \( 1 \) declares that the result of running the threads with these interleaveings is always the same.

Disproving Sequential Consistency

Sequential Consistency in Multiprocessor Programs:
Given a program with \( n \) threads,
1. for fixed operation sequences \( p_{0}^{1}, p_{1}^{1}, \ldots, \) and \( p_{0}^{2}, p_{1}^{2}, \ldots \) and \( p_{0}^{3}, p_{1}^{3}, \ldots \) keeping the program order
2. executions obey the clock condition on the \( p_{i}^{j} \)
3. all executions have the same result

Idea for showing that a system is not sequentially consistent:
- pick a result obtained from a program run on a SC system
- pick an execution and a total ordering of all operations
- add extra processes to model other system components
- the original order becomes a partial order
- show that total orderings \( C' \) exist for \( \rightarrow \) for which the result differs
Weakening the Model

There is no observable change if calculations on different memory locations can happen in parallel.

Idea: model each memory location as a different process

Sequential consistency still obeyed:
- the accesses of foo to a occurs before b
- the first two read accesses to b are in parallel to a = 1

Benefits of Sequential Consistency

Benefits of the sequential consistency model:
- concisely represent all interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- a good model for correct behaviors of concurrent programs
- programs results besides SC results are undesirable (they contain races)

It is a realistic model for older hardware:
- sequential consistency model suitable for concurrent processors that acquire exclusive access to memory
- processors can speed up computation by using caches and still maintain sequential consistency

Benefits of Sequential Consistency

Benefits of the sequential consistency model:
- concisely represent all interleavings that are due to variations in speed
- synchronization using time is uncommon for software
- a good model for correct behaviors of concurrent programs
- programs results besides SC results are undesirable (they contain races)

It is a realistic model for older hardware:
- sequential consistency model suitable for concurrent processors that acquire exclusive access to memory
- processors can speed up computation by using caches and still maintain sequential consistency

Not a realistic model for modern hardware with out-of-order execution:
- what other processors see is determined by complex optimizations to caching
- need to understand how caches work