Introduction

What is a parallel or multiprocessor system?

Why parallel architecture?

Performance potential

Flynn classification

Communication models

Architectures

Centralized shared memory

Distributed shared memory

Parallel programming

Synchronization

**Memory consistency models**
Example shared-memory program

Initially all locations = 0

*Processor 1*          *Processor 2*

Data = 23           while (Flag != 1) {;}
Flag = 1            … = Data

Execution (only shared-memory operations)

*Processor 1*          *Processor 2*

Write, Data, 23
Write, Flag, 1

Read, Flag, 1
Read, Data, ___
Memory consistency model

Order in which memory operations will appear to execute

⇒ What value can a read return?

Affects ease-of-programming and performance
The Uniprocessor Model

Program text defines total order = *program order*

Uniprocessor model

- Memory operations appear to execute one-at-a-time in program order
  - Read returns value of last write

BUT uniprocessor hardware

- Overlap, reorder operations

Model maintained as long as

- maintain control and data dependences

⇒ Easy to use + high performance
Implicit Memory Model

Sequential consistency (SC) [Lamport]

Result of an execution appears as if

- All operations executed in some **sequential order** (i.e., atomically)
- Memory operations of each process in **program order**
Initially Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)
    critical section

P2
Flag2 = 1
if (Flag1 == 0)
    critical section

Execution:

P1
(Operation, Location, Value)
Write, Flag1, 1

P2
(Operation, Location, Value)
Write, Flag2, 1

Read, Flag2, 0

Read, Flag1, ___
Understanding Program Order – Example 1

P1
Write, Flag1, 1
Read, Flag2, 0

P2
Write, Flag2, 1
Read, Flag1, 

Can happen if
- Write buffers with read bypassing
- Overlap, reorder write followed by read in h/w or compiler
- Allocate Flag1 or Flag2 in registers
Understanding Program Order - Example 2

Initially \( A = \text{Flag} = 0 \)

P1
\[ A = 23; \]
\[ \text{Flag} = 1; \]

P1
Write, \( A, 23 \)
Write, \( \text{Flag}, 1 \)

P2
while (\( \text{Flag} \neq 1 \)) {;

\[ \ldots = A; \]

P2
Read, \( \text{Flag}, 0 \)
Read, \( \text{Flag}, 1 \)
Read, \( A, \_\_\_ \)
Initially $A = Flag = 0$

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {};
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0
Read, Flag, 1
Read, A, 0

Can happen if
Overlap or reorder writes or reads in hardware or compiler
Understanding Program Order: Summary

SC limits program order relaxation:

Write → Read
Write → Write
Read → Read, Write
A mechanism needed to propagate a write to other copies

⇒ Cache coherence protocol
Cache Coherence Protocols

How to propagate write?

*Invalidate* -- Remove old copies from other caches

*Update* -- Update old copies in other caches to new values
Understanding Atomicity - Example 1

Initially $A = B = C = 0$

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A = 1;$</td>
<td>$A = 2;$</td>
<td>while $(B != 1)$</td>
<td>while $(B != 1)$</td>
</tr>
<tr>
<td></td>
<td>$B = 1;$</td>
<td>$C = 1;$</td>
<td>while $(C != 1)$</td>
<td>while $(C != 1)$</td>
</tr>
<tr>
<td></td>
<td>tmp1 = A;</td>
<td></td>
<td>tmp1 = A;</td>
<td>tmp2 = A;</td>
</tr>
</tbody>
</table>
### Understanding Atomicity - Example 1

*Initially $A = B = C = 0*

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1;$</td>
<td>$A = 2;$</td>
<td>while ($B != 1$) {}</td>
<td>while ($B != 1$) {}</td>
</tr>
<tr>
<td>$B = 1;$</td>
<td>$C = 1;$</td>
<td>while ($C != 1$) {}</td>
<td>while ($C != 1$) {}</td>
</tr>
<tr>
<td>tmp1 = A;</td>
<td></td>
<td>tmp2 = A;</td>
<td>tmp2 = A;</td>
</tr>
</tbody>
</table>

Can happen if updates of $A$ reach P3 and P4 in different order

Coherence protocol must serialize writes to same location

(Writes to same location should be seen in same order by all)
Understanding Atomicity - Example 2

Initially $A = B = 0$

P1
A = 1
Write, A, 1

P2
while (A != 1) ;
Read, A, 1
Write, B, 1
Read, B, 1

P3
B = 1;
tmp = A

Can happen if read returns new value before all copies see it
SC limits

Program order relaxation:
  Write → Read
  Write → Write
  Read → Read, Write

When a processor can read the value of a write
Unserialized writes to the same location

Alternative

(1) Aggressive hardware techniques proposed to get SC w/o penalty
    using speculation and prefetching
    But compilers still limited by SC

(2) Give up sequential consistency
    Use relaxed models
Classification for Relaxed Models

Typically described as system optimizations - system-centric

Optimizations

Program order relaxation:

Write → Read
Write → Write
Read → Read, Write
Read others’ write early
Read own write early

All models provide safety net
All models maintain uniprocessor data and control dependences, write serialization
## Some System-Centric Models

<table>
<thead>
<tr>
<th>Relaxation:</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R → RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 370</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PC</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBARs</td>
</tr>
<tr>
<td>PowerPC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>

- **W → R Order**: Write to Read order
- **W → W Order**: Write to Write order
- **R → RW Order**: Read to Write order
- **Read Others’ Write Early**: Enables reading before writing
- **Read Own Write Early**: Enables writing before reading
- **Safety Net**: Various synchronization techniques and instructions

Example: IBM 370 requires serialization instructions.
System-Centric Models: Assessment

System-centric models provide higher performance than SC

BUT 3P criteria

Programmability?
Lost intuitive interface of SC

Portability?
Many different models

Performance?
Can we do better?

Need a higher level of abstraction
One source of consensus

Programmers need SC to reason about programs

But SC not practical today

How about the next best thing…
Specify memory model as a contract
  System gives sequential consistency
  IF programmer obeys certain rules

+ Programmability
+ Performance
+ Portability
The Data-Race-Free-0 Model: Motivation

Different operations have different semantics

\[
\begin{align*}
P1 & \quad P2 \\
A &= 23; & \text{while} (\text{Flag} \neq 1) \{;\} \\
B &= 37; & \ldots = B; \\
\text{Flag} &= 1; & \ldots = A;
\end{align*}
\]

Flag = Synchronization; \(A, B = \text{Data}\)

Can reorder data operations

Distinguish data and synchronization

Need to

- Characterize data / synchronization
- Prove characterization allows optimizations w/o violating SC
Data-Race-Free-0: Some Definitions

Two operations conflict if

– Access same location
– At least one is a write
Data-Race-Free-0: Some Definitions (Cont.)

(Consider SC executions ⇒ global total order)

Two conflicting operations race if

– From different processors
– Execute one after another (consecutively)

P1
Write, A, 23
Write, B, 37
Write, Flag, 1

P2
Read, Flag, 0
Read, B, ___
Read, A, ___

Races usually “synchronization,” others “data”

Can optimize operations that never race
**Data-Race-Free-0 (DRF0) Definition**

Data-Race-Free-0 Program

- All accesses distinguished as either *synchronization* or *data*
- All *races* distinguished as *synchronization*
  
  (in any SC execution)

Data-Race-Free-0 Model

- Guarantees SC to data-race-free-0 programs

It is widely accepted that data races make programs hard to debug

independent of memory model (even with SC)
Distinguishing/Labeling Memory Operations

Need to distinguish/label operations at all levels
  • High-level language
  • Hardware

  Compiler must translate language label to hardware label

Java: volatiles, synchronized
C++: atomics
Hardware: fences inserted before/after synchronization
The idea

Programmer writes data-race-free programs

System gives SC

For programmer

Reason with SC

Enhanced portability

For hardware and compiler

More flexibility

Finally, convergence on hardware and software sides

(BUT still many problems…)

Data-Race-Free Summary