CS3400 - Principles of Software Engineering

Software Engineering for Multicore Systems

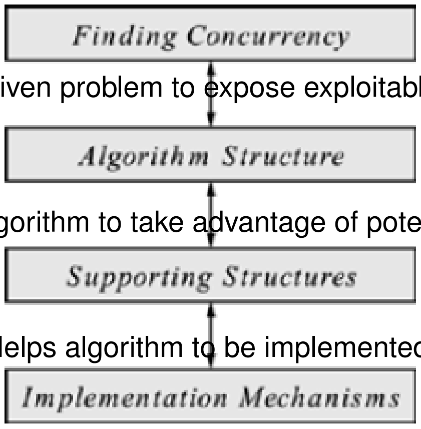
V. Krishna Nandivada

IIT Madras

Part I

Patterns

A pattern language for parallel programs



Structure the given problem to expose exploitable concurrency.

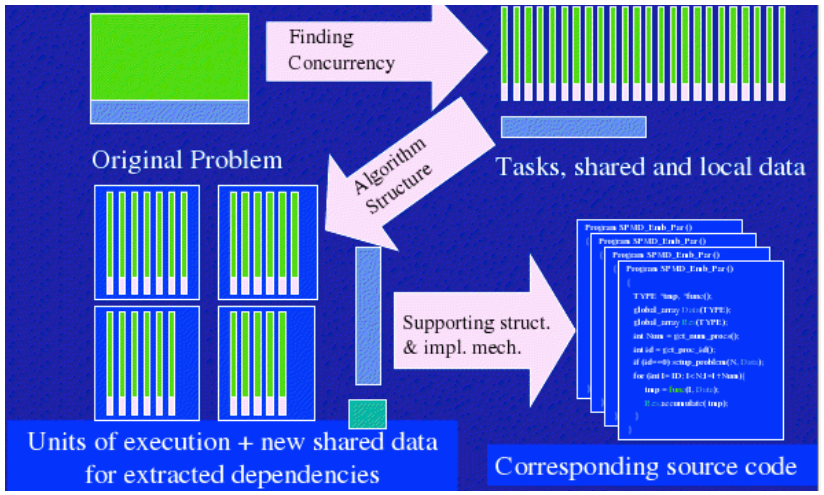
Structuring the algorithm to take advantage of potential concurrency.

Helps algorithm to be implemented.

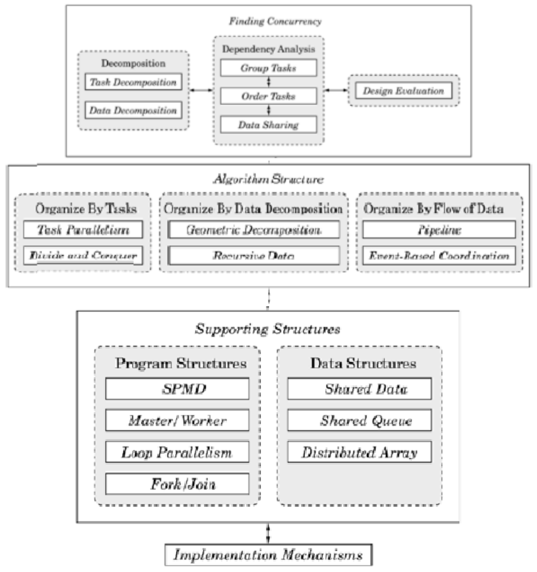
How the high level specifications are mapped.

Goal: Identify patterns in each stage.

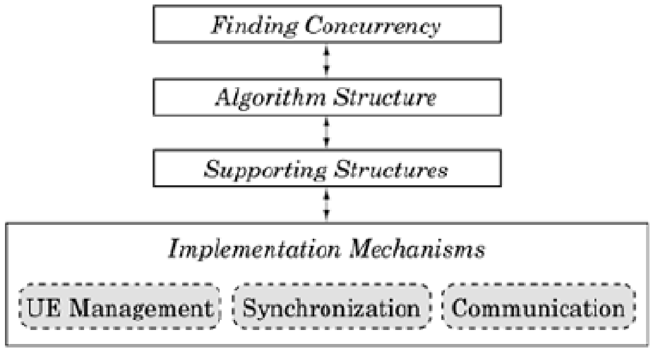
Overall big picture


Finding Concurrency
Original Problem
Tasks, shared and local data
Algorithm Structure
Supporting struct. & impl. mech.
Units of execution + new shared data for extracted dependencies
Corresponding source code

Finding concurrency and Algorithm Structure



Implementation Mechanisms



UE management

- UE - unit of execution (a process / thread / activity)
- Difference between process / thread / activity.
- Management = Creation, execution, termination.
- Varies with different underlying languages.
- Go back to first few lectures for a recap.

Synchronization: Memory synchronization and fences

Synchronization: Enforces constraint among parallel events.

-

```
done=true;                          done = false;
while(done) ;
```

  - Value may be present in cache. cache coherence may take care.
  - Value may be present in a register - Culprit compiler.
  - Value may not be read. How?

-

x = y = 0

Thread 1	Thread 2
1: r1 = x	4: x = 1
2: y = 1	r3 = y
3: r2 = x	

r1 == r2 == r3 == 0. Possible?

Synchronization: Memory synchronization and fences

- A memory fences guarantees that the UEs will see a consistent view of memory.
- Writes performed before the fence will be visible to reads performed after the fence.
- Reads performed after the fence will obtain a value written no earlier than the latest write before the fence.
- Only for shared memory.
- Explicit management can be error prone. High level: OpenMP flush, shared, Java - volatile. *Read yourself.*

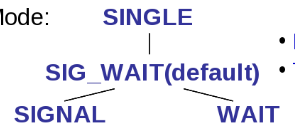
Syncrhonization: Barriers

Barrier is a synchronization point at which every member of a collection of UEs must arrive before any member can proceed.

- MPI_Barrier, join, finish, clocks, phasers
- Implemented underneath via passing messages.

Phasers[1]

- **Phaser allocation**
  - **Phaser ph = new Phaser(mode)**
    - Phaser **ph** is allocated with **registration mode**
    - Mode:



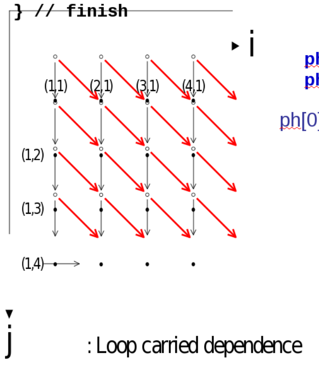
- Mode defines capability
- There is a lattice ordering of capabilities

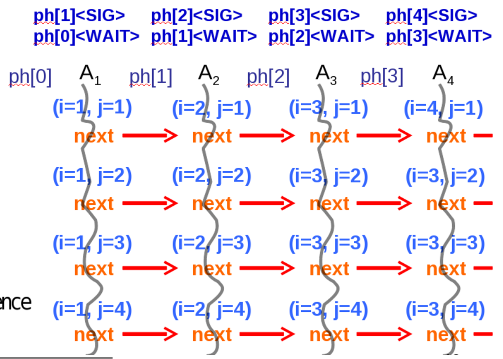
- **Activity registration**
  - **async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) {STMT}**
    - Spawned activity is registered with **ph_1** in **$mode_1$**, **ph_2** in **$mode_2$**, …
    - child activity's capabilities must be subset of parent's
- **Synchronization**
  - **next:**
    - Advance each phaser that activity is registered on to its next phase
    - Semantics depends on registration mode

[1]Thanks - Jun Shirako

Power of Phaser - pipeline parallelism[2]

```
finish {
  phaser [] ph = new phaser[m+1];
  foreach (point [i] : [1:m-1]) phased (ph[i]<SIG>, ph[i-1]<WAIT>)
    for (int j = 1; j < n; j++) {
      a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
      next;
    } // for
  } // foreach
} // finish
```



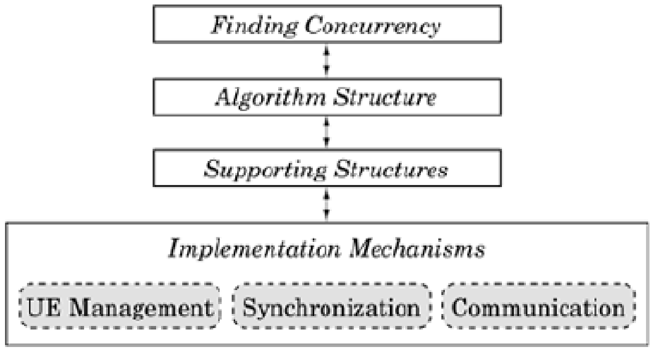


[2]Thanks - Jun Shirako

Syncrhonization

- Memory fence
- Barriers
- Mutual exclusion: Java `synchronized`, `omp_set_lock`, `omp_unset_lock`.

Implementation Mechanisms



Communication

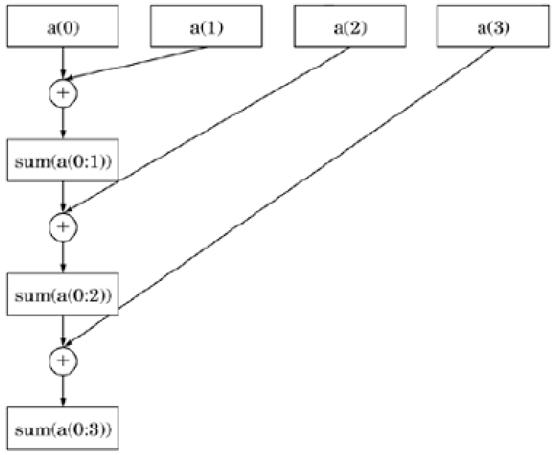
- UEs need to exchange information.
  - Shared memory - easy. Challenge - synchronize the memory access so that results are correct irrespective of scheduling.
  - distributed memory - not much need for synchronization to protect the resources. $\rightarrow$ Communication plays a big role.
- One to one communication :
- Between all UEs in one event: Collective communication.

Collective communication

When multiple UEs participate in a single communication event, the event is called a collective communication operation. Examples:

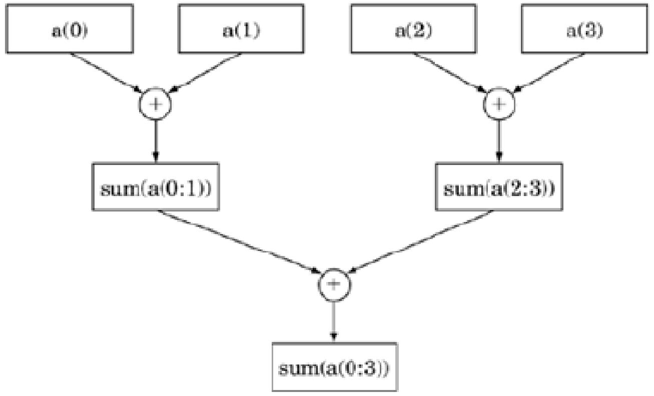
- Broadcast: a mechanism to send single message to all UEs.
- Barriers : a synchronization point.
- Reduction: Take a collection of objects, one from each UE, and "combine" into a single value;
  - combined value present only on one UE?
  - combined value present on all UEs?

Serial reduction



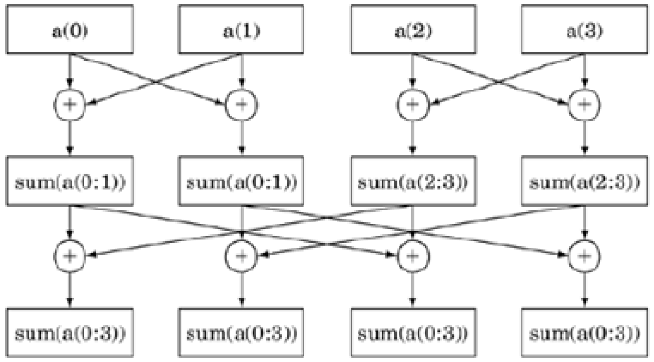
- Reduction with n items takes n steps.
- Useful especially if the reduction operator is not associative.
- Only one UE knows the result.

Tree based reduction



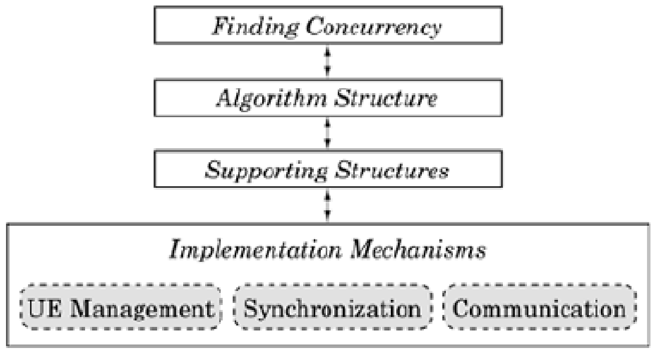
- Reduction with 2^n items takes n steps.
- What if number of UEs $<$ number of data items?
- Only one UE knows the result.
- Associative + Commutative or don't care (example?)

Recursive doubling



- Reduction with $2 \times n$ items takes n steps.
- What if number of UEs $<$ number of data items?
- All UEs know the result.

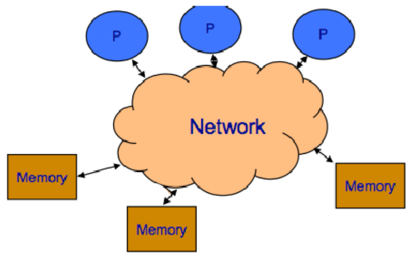
Implementation Mechanisms



Part II

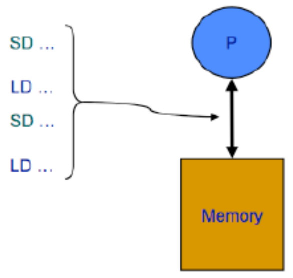
Memory Models

Memory Consistency Models



- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a *read* return?
  - a contract between programmer and system.
  - Determines what optimizations can be performed for correct programs.
- Affects : Ease of programming, and performance

Uniprocessor Memory model

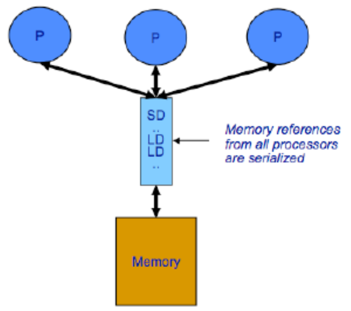


- *Memory value requirement*: Memory operations occur in program order: read returns the value of the last write in program order.
- Simple to reason about.
- Compiler optimizations preserve these semantics.
- Independent operations can execute in parallel.

Strict consistency

- Strictest memory model.
- Requires that the ‘read’ should get the value written by the last ‘write’.
- Requires a *Global* clock $\equiv$ Halting problem.

Sequential consistency



[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Sequential consistency

Result of an execution appears as if:

- All operations executed in some sequential order.
- Memory operations of each process in program order.
- Nothing specified about caches, write buffers.

Understanding Program Order. Dekker's Algorithm

```
Initially Flag1 = Flag2 = 0

P1                              P2
Flag1 = 1                       Flag2 = 1
if (Flag2 == 0)             if (Flag1 == 0)
   critical section                critical section
```

```
Execution:

P1                              P2
(Operation, Location, Value)  (Operation, Location, Value)
Write, Flag1, 1                 Write, Flag2, 1

Read, Flag2, 0                  Read, Flag1, ___
```

Reads of 1 by `Flag1` `Flag2` *are valid.*

Problematic situation

- Write buffers with read bypassing.
- Overlap or reorder writes/reads by compiler / hardware.
- Values in registers

Understanding Program Order. Ex 2

```
Initially A = Flag = 0
P1                              P2
A = 23;                         while (Flag != 1) {;}
Flag = 1;                       ... = A;
```

```
P1                              P2
Write, A, 23                    Read, Flag, 0
Write, Flag, 1
                                Read, Flag, 1
                                Read, A, _____
```

Problematic situation

- Overlap or reorder writes/reads by compiler / hardware.

Write Atomicity

```
Initially A = B = C = 0
 P1       P2       P3                  P4
 A = 1;   A = 2;   while (B != 1) ;    while (B != 1) ;
 B = 1;   C = 1;   while (C != 1) ;    while (C != 1) ;
                   tmp1 = A;           tmp2 = A;
```

Q: What are the possible values of tmp1 and tmp2?
Q: Can tmp1 = 1 and tmp2 = 2 be possible? How?

- Cache coherence protocol must serialize writes to same location.
- Writes to same location should be seen in same order by all.

Atomicity Ex 2

```
Initially A = B = 0
P1              P2                  P3
A = 1           while (A != 1) ;    while (B != 1) ;
                B = 1;                  tmp = A
```

```
P1              P2                  P3
Write, A, 1
                Read, A, 1
                Write, B, 1
                                    Read, B, 1
                                    Read, A, ~~0~~
```

- if ‘read’ returns a new value before all copies see it.
- *Read others’-write early* optimization is unsafe.

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order
- Write Atomicity : All writes appear to be instantaneous (no buffer).
- All processors must see all write operations in the same order (cache coherence).
- Easier to implement in architectures with no cache, no write buffers, blocking reads, .

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others’ write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.
- Makes it very hard to modify a parallel program (automatic and manual)
  - Processor reordering for performance - write buffers, overlapped writes, non-blocking reads
  - Compiler transformations - scalar replacement, register allocation, instruction scheduling.
  - Programmer reordering code for aesthetics/SE requirements.

Sequential consistency - too strict

- Many architectures do not give SC.
- Compiler optimizations on SC are limited.
- Sofwtware engineering issues.

- Give up!
- Use weaker models - relax the program order requirement and write atomicity requirement.

Sequential consistency (English)

- Memory operations of each process happens in program order.
- any valid interleaving of read and write operations is OK.
- all processes must see the same interleaving.

Sequential consistency examples

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)2	R(x)1

Sequentially consistent - as both P3 and P4 see writes in the same sequential order.

Sequential consistency (counter) example

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)1	R(x)2

Sequentially inconsistent - as both P3 and P4 see writes in the two different sequential orders.

Sequential consistency (counter) example

P1	P2	P3
x = 1;	y = 1	z = 1
print(y,z)	print (x,z)	print (x,y)

Inconsistent execution:

1. x = 1
2. print (y, z);
3. print (x, z);
4. y = 1;
5. z = 1;
6. print (x, y);

Sequential consistency

Result of an execution appears as if:

- All operations executed in some sequential order.
- Memory operations of each process in program order.
- Nothing specified about caches, write buffers.

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

```
P1                          P2
Flag1 = 1                   Flag2 = 1
if (Flag2 == 0)         if (Flag1 == 0)
    critical section            critical section
```

Execution:

P1	P2
(Operation, Location, Value)	*(Operation, Location, Value)*
Write, Flag1, 1	Write, Flag2, 1
Read, Flag2, 0	Read, Flag1, ___

Reads of 1 by `Flag1` `Flag2` *are valid.*

Problematic situation

- Write buffers with read bypassing.
- Overlap or reorder writes/reads by compiler / hardware.
- Values in registers

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor or access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.
- Hence such memory operations must be seen in same order by all processors.
- Here, write atomicity has been slightly weakened.
- weaker than sequential consistency, which requires that all nodes see all writes in the same order.

Causal consistency (example)

P1	W(x)1			W(x)3		
P2		R(x)1	W(x)2			
P3		R(x)1			R(x)3	R(x)2
P4		R(x)1		R(x)2	R(x)3	

Causally consistent, but not sequentially/strict consistent.

- Processors may see different order.
- All orders respect causal order (program order and read-write order).
- Has no global order, partial order for each processor.

Causal consistency (counter) Example

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- Violates causal consistency.
- Removing the Read from the P2 – makes the execution causally consistent.

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.
- no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline.

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict
- (Also known as, FIFO consistency, or Processor consistency)

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - All other accesses may be seen in different order on different processes
- Illusion of write atomicy has to be maintained.
- Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed.

Weak Ordering

- Example 1:

```
P1  W(x)1 W(x)2 Sync
P2                    R(x)1 R(x)2 Sync
P3                    R(x)2 R(x)1 Sync
```

- Example 2:

```
P1  W(x)1 W(x)2 Sync
P2                    SyncR(x)2
```

- The programmer has to manage synchronization explicitly.
- Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Weak consistency (counter) example

```
P1  W(x)1 W(x)2 Sync
P2                    SyncR(x)1
```

- P2 will observe the most recent write of the variable x, which has the value 2. Thus, it's not a valid sequence.

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after 'release' in program order do not have to wait for release (unless protected by another acquite).
- do "acquite" = that writes on other processors to protected variables will be known
- do "release" = that writes to protected variables are exported
- and will be seen by other machines when they do a lock (lazy release consistency) or immediately (eager release consistency)
- Total order among all synchronization instructions must be maintained.

Weak and Release comparison

- Weak: Shared data can be counted on to be consistent only after a synchronization is done.
- Release: Shared data are made consistent when a critical region is exited.

Release Consistency - example

- Example:
```
P1:   L W(x)1 W(x)2 U
P2:                       L R(x)2 U
P3:                                   R(x)1
```
- RC $\leq$ Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.
  - if an object is modified, during the short period of time following its modification, the read may not be consistent.
  - after a fixed time period, the modification is propagated and the read will be consistent.
- **Eventual Consistency Model** : The writes propagates eventually (we cannot have a fixed bound on the delay)

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models
  - How to use the safety nets minimally, to get the desired semantics from program
- Even Sequential Consistency is not simple enough.
- We need models which is simple for the programmer, but provides enough information about program to apply optimization and get efficiency.

Programmer centric models

Programmers understand their code:

- Different operations have different semantics

P1	P2
A = 23;	while (Flag != 1) ;
B = 37;	. . . = B;
Flag = 1;	. . . = A;

- Flag = Synchronization; A, B = Data
- Can reorder data operations
- Distinguish data and synchronization

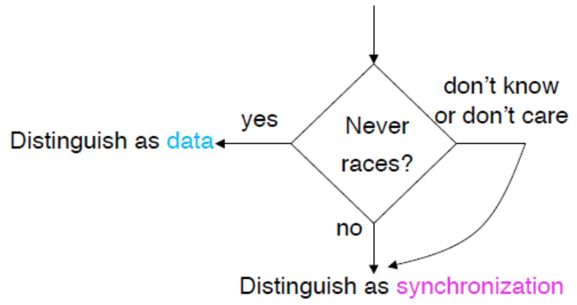
Data Race Free 0 - DRF0

- **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**
  - **—(For others, reads return value of some write to the location)**

Programming with Data Race Free 0 - DRF0

- **Information required:**
  - **— *This operation never races* (in any SC execution)**

1. **Write program assuming SC**
2. **For every memory operation specified in the program do:**



Problems with data race free model

- It does not define any semantics for programs with data races.
- A concern for safe languages like Java, which provide safety for any program and cannot let the behavior of a program to be ambiguous.
- Either define safe semantics for such programs or identify them and prevent their execution.
- Define higher abstractions for programmers which are inherently data race free
- Expensive for hardware to implement

Goals of Memory model

- Programmability? - Lost intuitive interface of SC
- Portability? - Many different models
- Performance? - Can we do better?

Future:

- Parallel programs today are inherently non deterministic
- We need deterministic outcomes from our parallel programs.
- Deterministic Outcomes from Inherent non determinism. Possible?

Sources

- Patterns for Parallel Programming: Sandors, Massingills.
- multicoreinfo.com
- Wikipedia
- fixstars.com
- Jernej Barbic slides.
- Loop Chunking in the presence of synchronization.
- Vivek Sarkar's slides.
- Sarita Adve's slides.
- Nimit's Singhania's presentation.
- http://regal.csep.umflint.edu/ swturner/Classes/csc577/Online/Chapter06/Chapter06.html
- Java Memory Model JSR-133: "Java Memory Model and Thread Specification Revision"