Parallel Algorithms

Lecture 4: Consistency & Protocols

Ronald Veldema

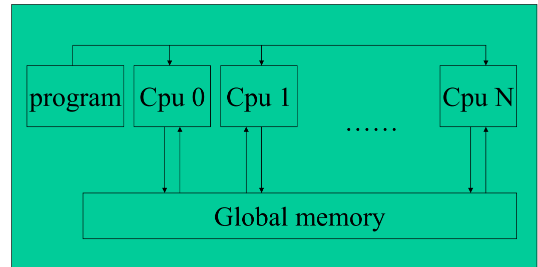
Generic Parallel Machines

- SPMD model
  - PRAM machine
  - A write by one cpu is immediately seen by all others
- MDMD model
  - SMP machines
  - A write _can_ be seen much later

PRAM model (1)

- 1 program (SPMD programming model)
- N times local memory & cpu
- 1 memory access unit
- 1 global memory

PRAM model (2)



PRAM model (3)

- Normally, cpu 0 runs program
  - Makes control flow decisions (i.e. call foo)
- ‘par [range] <statement>
  - Each cpu runs <statement>
  - If semantic:
    - If <condition> then <statement> else nops
      - Enough nops are run to cover run time of <statement>

```
max = -infinite
par i=0..N
        if x[i] > max then
              max = x[i]
```

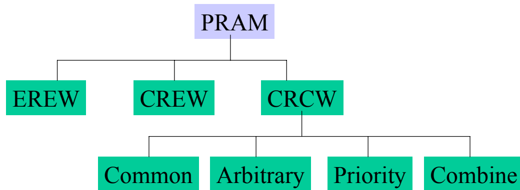
Is this correct ?

NO: each cpu overwrites other found cpu’s max !

PRAM model (3)

- In par block
  - Each cpu reads the insn’s data in parallel
    - concurent reads allowed ?
  - each cpu does writes to memory in parallel
    - concurrent writes allowed ?
      - priority write (value in memory is of most valued cpu)
      - common write (each cpu writes same value to location X)
      - arbitrary write (value in memory is arbitrary)
      - combining write (value is 'combine' (min, max, add) of values
  - each cpu immediately 'sees' other cpus' writes.
- This model is often used to proof parallel alg. qualities

PRAM model (4)



PRAM model (4)

- Good for proving algorithm properties but not implementable
  - Assumes each processor has constant time access to all memory
  - Concurrent reads/writes to a memory cell
  - Single clock signal for all processors

PRAM model (5)

- Proof example:
  - Prio-CRCW can be simulated on EREW with a log(#processors) penalty in performance
    - (U vishkin: simultanious access in models that forbid it *Journal of algorithms vol. 4*)
  - Rj = 54
    - Write 54 at address j in memory

PRAM model (6)

```
// assume: num cpus = N
//          'temp', 'a' are arrays of length N
//          'temp' of type (address,int)
//          'a' of type boolean
function set_memory(address j, int value)
```

1	par p = 0 .. N temp[p] = (j, value)
log	sort_ascending(temp)
1	par p = 1 .. N if temp[p].address == temp[p-1].address then a[p] = 0 if temp[p].address > temp[p-1].address then a[p] = 1 if p = 1 then a[p] = 1
1	par p = 0 .. N if a[p] == 1 then memory[temp[p].address] = temp[p].value

PRAM model

- Every PRAM can be simulated by every other
- P-CRCW by EREW
  - O(log(p))
- P-CRCW by C-CRCW
  - O(log(p) / (log(p)log(p)))
- P-CRCW by A-CRCW
  - O(log(p)log(p))

PRAM model

```
par I = 0 .. N
    m[I] = 1
par I = 0 .. N
    par J = 0 .. N
        if x[I] < x[J] then
            m[I] = 0
par I = 0 .. N
    if m[I] == 1 then
        max := x[I]
```

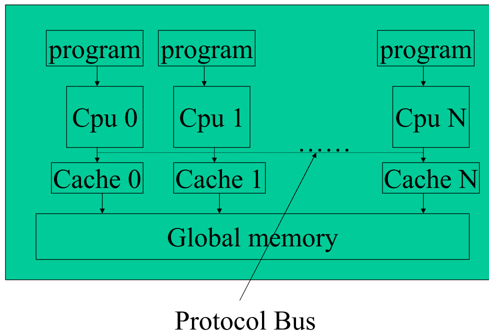
Question: what is the time Complexity ?

O(1)

Question: how many Processors did I use ?

N^2

Generic SMP machine



What is a consistency model ?

- Consistency model = when is it guaranteed that others (cpus/threads) see my writes

```
Thread1() {
  lock();      // flush cache here ?
  A = 4;       // or write A to directly to memory ?
  unlock();    // or flush cache here ?
}

Thread2() {
   Print A;
}
```

Why should I care ?

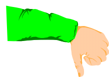
- When writing a parallel algorithm, you need to know what you can depend on
- High performance ?
  – Exploit memory model characteristics !
- Very subtle bugs…
- Multiple coherence 'views' in a single computer:
  – View presented by architecture to compiler
  – View presented by compiler to middleware
  – View presented by middleware to programmer

Why should I care ?

- Sparc pre-version 9: Total Store Order
  – Load can be performed before outstanding stores complete
  – Sparc version 9 and above: Relaxed Memory Order
  – Loads and stores can be completely reordered as long as self consistency is ensured
    - write issued is immediately visible by writee, not by other reading cpus

Why should I care ?

p.x = 1
Print p.x -> "0"!



p.x = 1;
Nop
Nop
Print p.x -> "1"

Why should I care ?

Cpu 0	Cpu 1
x.f = 1 x.g = 2	
	tmp1 = x.f tmp2 = x.g tmp3 = x.f

What is a consistency protocol ?

- protocol implements a given consistency model
- Examples:
  - Only a single writer at a time to a variable
    - For example by letting other writers wait
  - Multiple concurent writers allowed
    - For example by merging modified data from all writers

Sequential consistency (1)

- From lamport
- Serializability of operations
  - a series of operations H is equal to _some_ series of serial operations.

Sequential consistency (2)

cpu 0	cpu 1
A = B D = H	C = D G = A

in serial:
A = B
C = D
D = H
G = A
or in serial:
C = D
G = A
A = B
D = H
etc

Bad:
G = A
C = D
A = B
D = H

Sequential consistency (3)

- All operations of each processor happen in order of the program that specified them.
- Every write operation becomes instantaneously visible throughout the system.

Sequential consistency (4)

- pro: very much what the programmer wants to see.
- con: slow as no operations can be (observably) reordered
- con: writes must be 'flushed' immediately.
- con: modern (SMP) machines have own caches, perform write reordering etc.
  - An programming language implementor would have to perform a lot of work...

Release consistency (1)

- paper: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors (1990)
- conflicting accesses:
  - two processors read/write the same variable X with at least one of them writting X.
- synchonization accesses:
  - flags set to inform another processor of an event. (ackquire and release for example).

Release consistency (2)

- compiler/programmer sets flags to tell if an access is special or ordinary.
  - Special = synchonization or competing.
- before an ordinary access, all previous ackuires must have been performed
- before a release, all ordinary reads+writes must have been performed
- special accesses are sequentially consistent in respect to one another.

Release consistency (3)

example:

```
// 'flag' = special (synchronized,
                    conflicting)
// 'X,Y' = ordinary (non-conflicting)

if (flag == true) { // release
        X++;
        Y++;
}
```

Note: flag can't be cached
Note: read of flag acts as 'release'
Note: X,Y access can't be moved over flag access
Note: release of 'flag' acts as 'flush thread/cpu cache'
Note: X,Y accesses can be reordered, flag accesses cannot

Eager release consistency

- writes immediately trigger protocol actions: invalidate caches
- release encountered while still have outstanding writes.

Lazy release consistency

- At release send write notices to processors that hold a copy of the data.
- Those processors can then invalidate, etc.

Processor consistency

- writes issued by a processor may not be observed in any other order.
- writes from two processors are not neccesarily in order.

A = B = 0	
A = 5	print B ("0")
B = 4	print A ("5")

Commit-Reconcile & Fences (CRF)

- Each thread/cpu has a cache
  - 'commit' writes a value from the cache to memory
  - 'Reconcile' removes stale values from cache
  - 'Fence' prevents recordering
    - Argument of fence is list of dependent variables
    - Waits until every dependent value is stored in memory
- Other memory models are 'mapped' to CRF
- Compiler/architecture can optimize by removing C,R,F actions…

Commit-Reconcile & Fences (CRF)

- Sequential consistency on top of CRF:
  - 'R=A.F' = reconcile(A); load(R, A.F); $fence_w$(A)
  - 'A.F=V' = Store(A.F, V); commit(A); fencew(A)
- Release consistency on top of CRF

```
// fence_rw(flag,X,Y); reconcile(flag); load(flag)
if (flag == true) {
        X++;  // reconcile(X); load(X); store(X+1)
        Y++;  // reconcile(Y); load(Y); store(Y+1)
}
```

Scope consistency

- flush cached items in cache that we're pulled into the cache at when exiting a scope.

```
enter_scope();
   A = 2;
enter_scope();
   B = 3
exit_scope(); // flush B
exit_scope(); // flush A
```

Single Writer Protocols

- Single Writer Protocols
- update protocols
- directory based systems
- start/end-read/write protocols
- protocol state diagrams, protocol-provers

Single writer protocols (2)

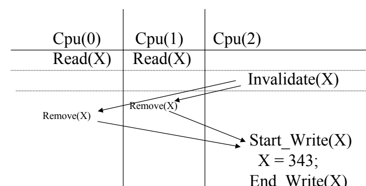
```
write_lock(X);
     X++;
write_unlock(X);

read_lock(X)
   print X
read_unlock(X);
```

Single writer Protocols (3)

- Invalidation based protocols
  - When writing to a data item, inform all current readers that the data item is
    - Not to be used anymore
    - Should not remain cached

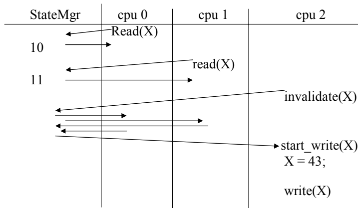
Single Writer Protocols (4)

- Broadcast to all current readers
  - How do you know who is a reader ?
    - Broadcast to everyone
    - Someone maintains a bitvector who is a reader
    - Everyone maintains a bitvector of current readers



Single Writer Protocols (5)

- Broadcast to all current readers
  - How do you know who is a reader ?
    - Broadcast to everyone
    - Someone maintains a bitvector who is a reader
    - Everyone maintains a bitvector of current readers



Multiple writer protocols (1)

- each thread/cpu may concurently write to a memory location.
- the final result is some form of 'merge'.

Multiple writer protocols (2)

```
read_into_cache(X)
read_into_cache(Y)
read_into_cache(Z)
    X++;
    Y++;
    Z++;
flush_cache();
```

Java Memory Model (1)

- Each thread has own cpu
- Each thread has own working memory
- Working memory is initially empty
- Working memory is flushed at each lock/unlock
  - Lock/unlock = entry/exit synchronized block

Java Memory Model (2)

```
synchronized(LOCK) {
        X++;
        Y++;
        Z++;
    }
```

```
read_into_working_memory(X)
read_into_working_memory(Y)
read_into_working_memory(Z)

read_into_cache(X)
X++;
write_to_cache(X)

read_into_cache(Y)
Y++;
write_to_cache(Y)

read_into_cache(Z)
Z++;
write_to_cache(Z)

store_cache_in_memory(X)
store_cache_in_memory(Y)
store_cache_in_memory(Z)

flush_cache();
```

Java Memory Model (3)

- Double Locking Protocol failure

```
// Try 1

class A {
        Data d = null;

        void foo() {
                if (d == null)
                        d = new Data()
        }
}

// what happens under the JMM if two threads simultaniously enter 'foo' ?
```

Java Memory Model (4)

- Double Locking Protocol failure

```
// Try 2

class A {
        Data d = null;

        synchronized void foo() {
                if (d == null)
                        d = new Data()
        }
}

// Correct, but the cost if the synchronized is not trivial and not needed if d != null…
```

Java Memory Model (5)

- Double Locking Protocol failure

```
// Try 3

class A {
        Data d = null;

        void foo() {
                if (d == null) {
                        synchronized (this) {
                                d = new Data();
                        }
                }
        }
}

// NOT Correct,  two threads concurrectly test d == null and think its true…
// (result = two Data allocations ISO one)
```

Java Memory Model (6)

- Double Locking Protocol failure

```
// Try 4

class A {
        Data d = null;
        void foo() {
                if (d == null) {
                        synchronized (this) {
                                if (d == null)
                                        d = new Data();
                        }
                }
        }
}
// NOT Correct,  as assignments in the Data constructor may be moved to after the 'd=new'
// assignment. The other thread would assume the object to be initialized (as d != null)
// while in reality the constructor is still running…
//   (in short: an optimizing compiler/processor makes this example incorrect)
```

Java Memory Model (7)

- Be careful when programming threads !
  - You (or compiler) optimizes code slightly...

P = Q; P.X = 0

Thread 0	Thread 1
M = P.X	P.X = 3
N = Q.X	
Z = P.X	

Thread 0	Thread 1
M = P.X	P.X = 3
N = Q.X	
Z = M	

May return M=Z=0, N = 3 !

Java Memory Model (8)

- Possible swap

```
class Swapper {
        int a = 1, b = 2;

        void hither() {
                a = b;
        }

        void yon() {
                b = a;
        }
}
```

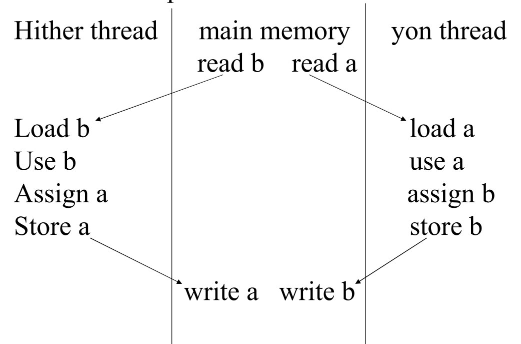
*note: hither and yon not synchronized !

*note: a thread's cache may be flushed at any time

*note: what happens when hither & yon are concurrently called ?

Java Memory Model (8)

- Possible swap



Java Memory Model (8)

- Possible swap
  - Write a precedes read a, read b precedes write b
    - Ha=2,hb=2, ma=2,mb=2,ya=2,yb=2
  - Read a precedes write a, write b precedes read b
    - Ha=1,hb=1,ma=1,mb=1,ya=1,yb=1
  - Read a precedes write a, read b precedes write b
    - Ha=2,hb=2,ma=1,mb=2,ya=1,yb=1

Java Memory Model (9)

- Possible swap with synchronized ???????

```
class Swapper {
        int a = 1, b = 2;

        synchronized void hither() {
                a = b;
        }

        synchronized void yon() {
                b = a;
        }
}
```

Java Memory Model (9)

- Possible swap with synchronized: NO !

```
class Swapper {
        int a = 1, b = 2;

        synchronized void hither() {
                a = b;
        }

        synchronized void yon() {
                b = a;
        }
}
```

No !
- hither runs first
  a=b=2

- yon runs first
  b=a=1

Java Memory Model (10)

- Out Of Order writes: what are the possible print results ?

```
Class Example {
        int a = 1, b = 2;

        // called from thread 1
        synchronized void foo() {
                a = 3;
                b = 4;
        } // flush a & b and then unlock this object

        // called from thread 2: note: not synchronized !
        void print() {
                System.out.println(“a=“+a+”, b=“+b);
        }
}
```

Java Memory Model (10)

- Out Of Order writes: what are the possible print results ?

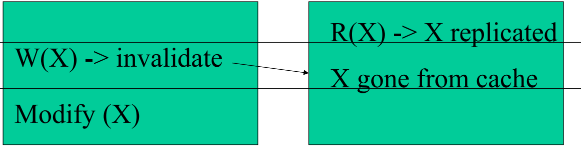
```
Class Example {
        int a = 1, b = 2;

        // called from thread 1
        synchronized void foo() {
                a = 3;
                b = 4;
        } // flush a & b and then unlock this object

        // called from thread 2:
        synchronized void print() {
                System.out.println(“a=“+a+”, b=“+b);
        }
}
```

Exploiting the Java Memory Model

- Java software DSM systems
- Allow variables to be pulled into registers until a synchronization point is reached

Invalidation Protocols

- Whenever a piece of data is updated, remove all invalid replicas held by other cpus by sending explicit invalidate messages
  - Question: broadcast invalidate or maintain lists of replicas ?



Update Protocols

- Whenever a piece of data is updated, broadcast modications made
  - Question: broadcast updates or maintain lists of replicas ?
  - Broadcast every write or wait a little and save on communication overheads ?