October 13 11:50 AM

hyper vs multi core

Multithreading

→ Dividing the task to multiple core

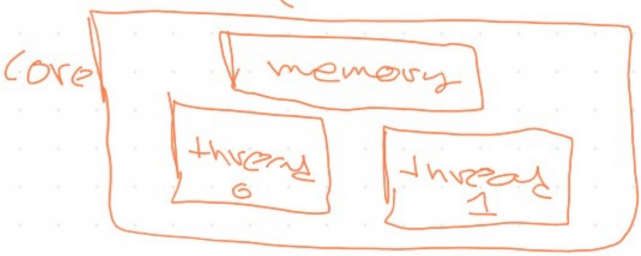
→ physical

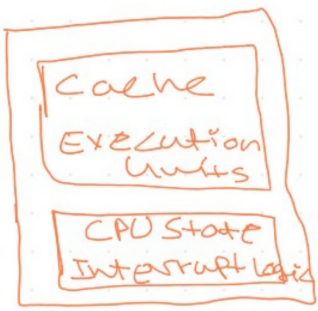
Hyperthreading

→ more than one threads on each core.

→ share same cache memory of same core

→ logical





October 13 1:35 PM

Performance measure

Performance measurement :-

Sequential execution Time = S

Parallel execution Time = T

Speed Up = S/T

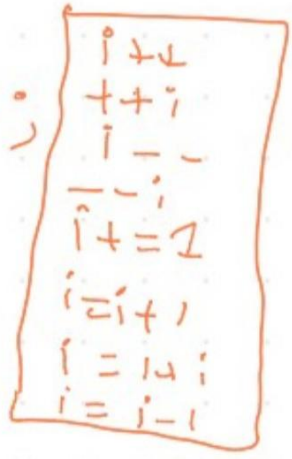
no. of core in Processor = P

Processor utilization = $\frac{\text{Speed Up}}{P}$

max P.U = 100%

In

#Pragma omp parallel for

for(i=start;  end; )

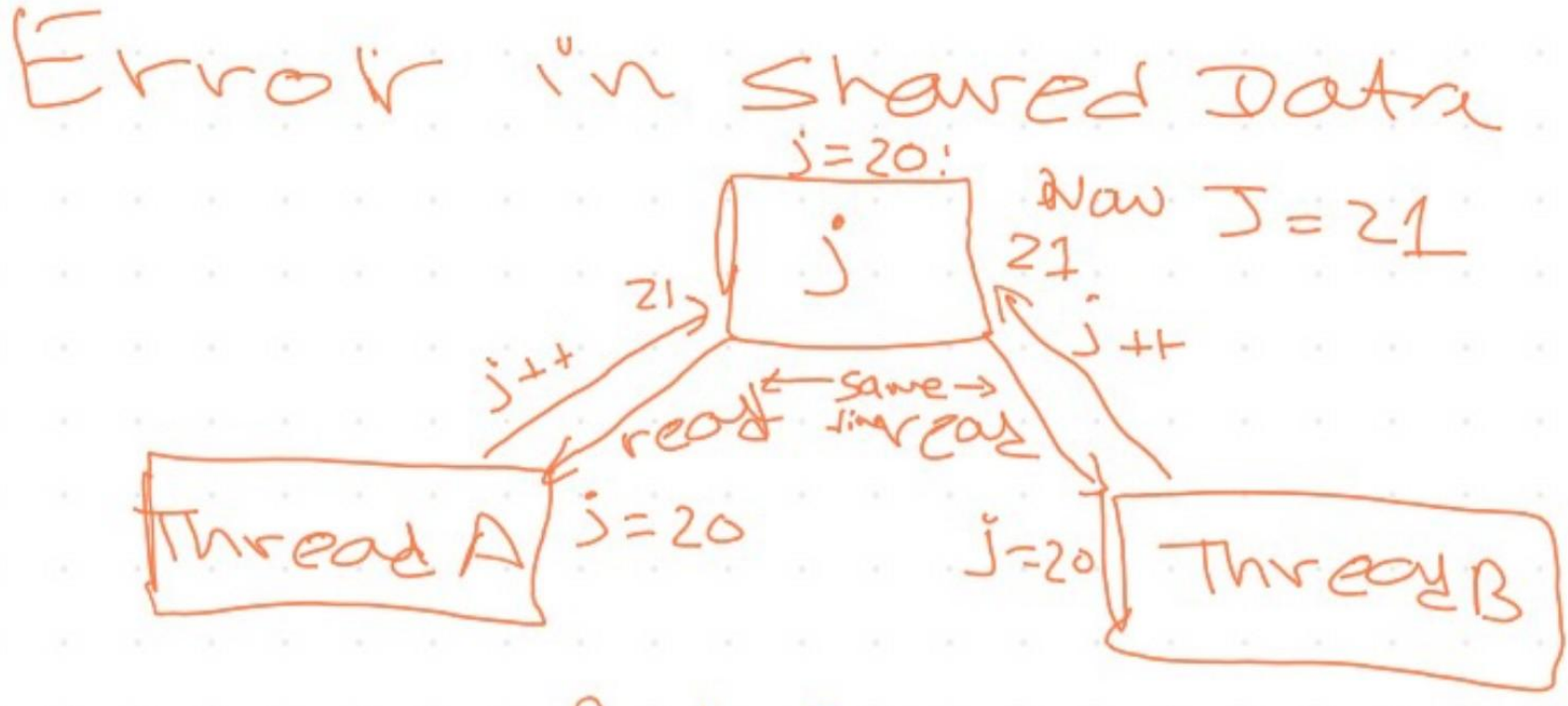
→ can't be increment Outside the for statement

→ i become private variable. no need to define Private for i

→ other varible Outside the for loop become shared variable and shared by all threads.

October 13 1:46 PM

Shared Data error



But expected = 22

Data race condition

also int1 Problem

Solution → Locking time

Shared variable while modifi

& Unlock after done

October 13 2:00 PM

Locking

Types of locking :-

→ Single SpinLock

→ fine-grain Locking
(multiples Lock)

→ OpenMP.

```
void omp_set_lock(omp_lock_t *lock)

void omp_unset_lock(omp_lock_t *lock)

void omp_init_lock(omp_lock_t *lock);
```

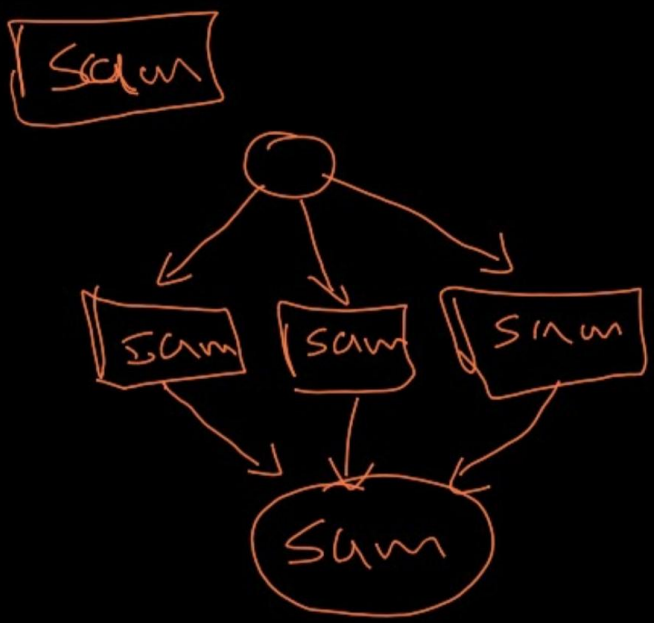
→ once each lock

October 13 2:30 PM

Reduction

Reduction :-

→ associative & commutative operation on shared variable



acts as Private inside thread

reduction (OP: variable)

⟨variable⟩ : any shared variable

⟨OP⟩ : operation

+
*
&&
||
(bit operations)

October 13 3:32 PM

Scheduling

Scheduling loops

→ Static
→ Dynamic
→ guided

Static

→ thread load balanced.
→ all iterations are allocated to thread before execute

schedule (static, chunk)

→ Low overhead → high load imbalance

Dynamic

↓ schedule (dynamic, c) determine which thread is free.

→ high overhead
→ reduce load imbalance

Default

→ chunk size is large
no of iteration / threadCount

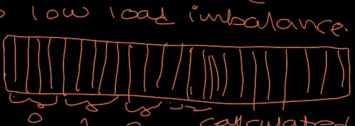
→ if loop iteration have equal duration then default is best.

Guided

→ first chunk size is large & set
schedule (guided, c)

→ chunk size progressively smaller size assigned dynamically to threads as need.

→ high overhead
→ low load imbalance.



October 13 3:48 PM

Data dependency

Data dependency

→ if S1 & S2 statements refere to same memory location L

→ S1 refere L occurs before S2 refere L.

Flow Dependency

S2 depend upon S1

S1 → write to L

S2 → read from L

```
a[i] = a[i-1] + b[i]
```

Anti-dependency

S2 depend upon S1

S1 → read from L

S2 → write to L

```
a[i] = a[i+1] + b[i]
```

Output Dependency

two statement write the same memory,

```
a[i] = i
a[i+1] = a[i] + 1
```

a0 a1 a2

Loop-independent dependencies

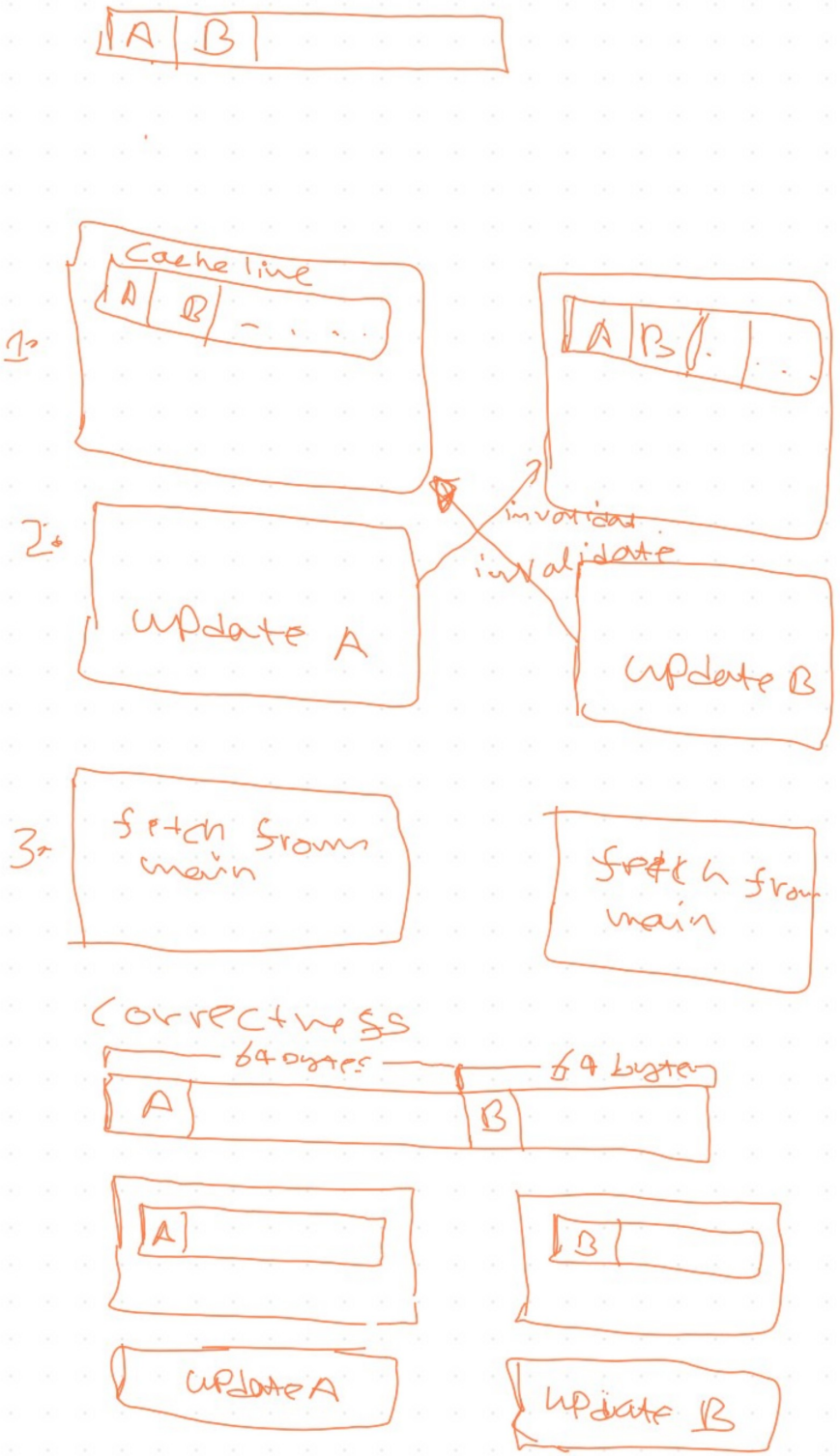
```
S1:  a[i] = i
S2:  b[i] = a[i] + 2
```

a0 a0 b2 .

a1 a1 b1 .

October 13 4:41 PM

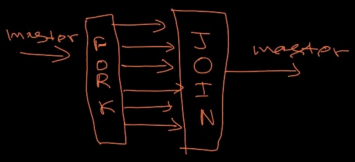
False sharing



October 13 5:16 PM

Parallel Regions

Parallel Regions




```
#Pragma omp parallel {

}
```

for construct

```
#Pragma omp parallel {
  #Pragma omp for {
    for( ) { }
  }
}
```

→ data parallelism

→ Shares iterations of a loop across a team

Section construct

```
#Pragma omp sections {
    #Pragma omp section{ }
    #Pragma omp section{ }
}
```

→ functional parallelism

→ each section executed by a thread. (assign new section after completion of previous thread)

Single:-

```
#Pragma omp single{ }
```

→ Serialize a section of code

→ only executed by one of the threads

→ other waits for finished this block

Critical

→ enclosed code become an atomic operation

→ replaces the use of locks

```
#Pragma omp critical{

}
```

Java Thread

Creating Java Threads.

1. extends Thread

```
class MyThread extends Thread {
    Public void run() { }
}
```

2. Runnable Interface

```
class ThreadAction implements
                    Runnable {
  Public void run() { }
}

Thread th = new Thread (
    new ThreadAction());
```

Run Thread

```
th.start()
```

Join Thread

```
th.join();
```

→ main thread wait to complete th thread.

```
th.join(35);
```

→ Put current thread for wait to complete th thread for 35 milliseconds. or until current thread terminated.

→ join used for both synchronized & non-synchronized operations.

LOCK

```
Lock L = new ReentrantLock();
```

java.util.concurrent.locks

→
```
L.lock();
a[i]++;
L.unlock();
```

→ Operation same as openMP lock.

→ Use Array of Lock for fine-grained Lock.

October 13 7:12 PM

Locking in Java

Synchronized keyword

→ Every object has an associated lock

→ Synchronized block makes the compiler append instructions to acquire the lock on the specific object,

→ synchronized (obj) {

}

or method lock

synchronized void method() {

}

lock on all the statements & objects in method using synchronized (this) {

}

→ this is mutual exclusion lock

October 13 7:18 PM

Read Write Lock

Read-Write Locking

→ greater level of concurrency accessing shared data.

→ only a single thread at a time (writer thread) can modify data

→ any number of thread can concurrently read the data (reader threads)

```
ReadWriteLock L = new
        ReentrantReadWrite
        Lock();
Lock wL = L.writeLock();
rL = L.readLock();
wL.lock();
wL.unlock();
```

October 13 7:42 PM

Lock Fairness

Lock Fairness :-

- Reentrant Lock accept optional fairness parameter.
- when set true, under contention, locks favour granting access to the longest-waiting thread.
- might be lower overall throughput.
- but guarantee lack of starvation.
- new ReentrantLock(true);

Read-Write Fairness

- threads contend for entry using an approximately arrival-order policy.
- longest-waiting single write lock or group of reader threads waiting longer than all waiting writer threads.
- read lock block if
  - write lock held, or waiting
  - until oldest currently waiting writer threads has acquired and released the write lock.
- write lock block if already read or write lock is present

October 13 8:01 PM

Lock timeout

Lock Timeout

```
Lock L = new ReentrantLock();
L.tryLock( 100,
           TimeOut.MILLISECONDS);
```

→ true if the lock is acquired with in specified time.

Improve Performance

→ reduce the amount of shared data

→ reduce duration of lock

→ use fine-grain locking

October 13 8:12 PM

Java Barrier

Java Barriers

```
CyclicBarrier barrier =
    new CyclicBarrier(5);

barrier.await();
```

Barrier Action

```
Runnable action =
new CyclicBarrier(5,
action);
```