ASYNC
FORK

FINISH
JOIN

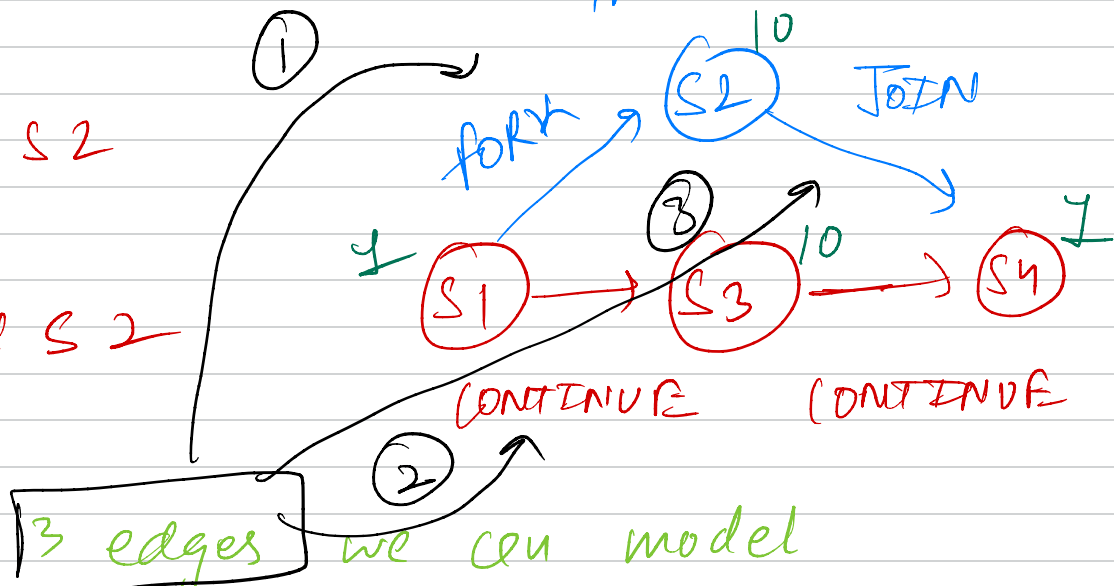
JAVA API
JAVA PARALLEL

Same thing

Task Int

S1
FORK S2
S3
JOIN S2
S4

GRAPH



→ With these 3 edges we can model any parallel program

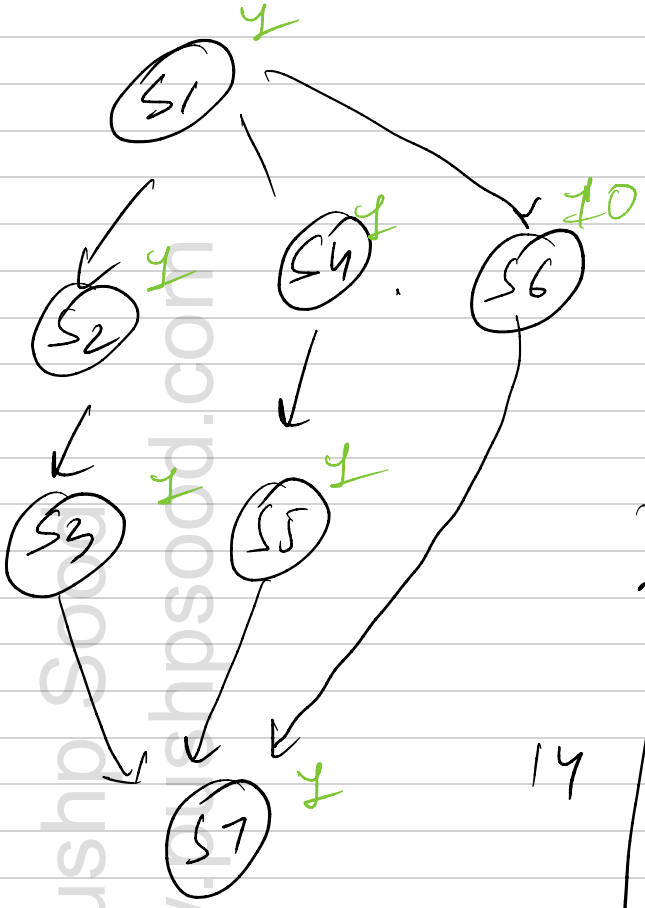
→ If there is a path directly b/w two nodes in the graph they cannot run in parallel for eg (S2) & (S3) can run in parallel, but (S2) & (S4) cannot

HERE → WORK = 22
(Total sum of all nodes)

→ SPAN = 12
(sum of longest path)

IDEAL PARALLISM =
$\frac{WORK}{SPAN} = \frac{22}{12}$

Multiprocessor scheduling, parallel speedup



T_p = Execution time on p processes

p is processor (2 in our case) done by H/W

P_0	P_1
S1	IDLE
S2	S4
S3	S5
S6	IDLE
S7	IDLE

14

SCHEDULE 1

Here, $T_2 = 14$

P_0	P_1
S4	IDLE
S6	S2
	S3
	S4
	S5
S7	

10 | 4

Here $T_2 = 12$

If $p = 1$, $T_p = T_1$ = WORK $T_1 = 16$
(1 processor)

Here p is ← T_∞ = SPAN (length of longest path) = 12

$$T_\infty \leq T_p \leq T_1$$

SPEEDUP = $\frac{T_1}{T_p}$ (How much faster the parallel version was able to run)

BOUND ON SPEEDUP

SPEEDUP $\leq p$

SPEEDUP $\leq \frac{\text{WORK}}{\text{SPAN}}$ = IDEAL PARALLELISM

Amdahl's Law

(Interesting observation made by Amdahl 50 yrs back)

$$\text{SPEEDUP} \leq \frac{\text{WORK}}{\text{SPAN}}$$

q = FRACTION (SEQUENTIAL)

$$\text{SPEEDUP} \leq 1/q$$

$q = 0.5 \Rightarrow$ SPEED UP ≤ 2

↓

No Need for large parallelism

$q = 0.1 \Rightarrow$ SPEED UP ≤ 10

PROOF

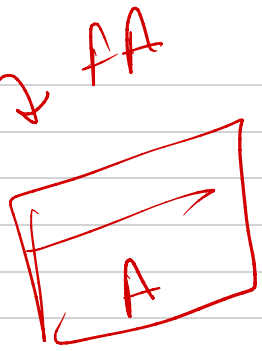
SPAN $\geq$ $q \times$ WORK

$q \times$ WORK must be included in the computational graph (Lower bound)

SPEED UP $\leq$ WORK / SPAN

$$\text{SPEEDUP} \leq \frac{\text{WORK}}{q \times \text{WORK}}$$

futures



$A = f(B)$

$C = g(A)$ → only dependent on A.

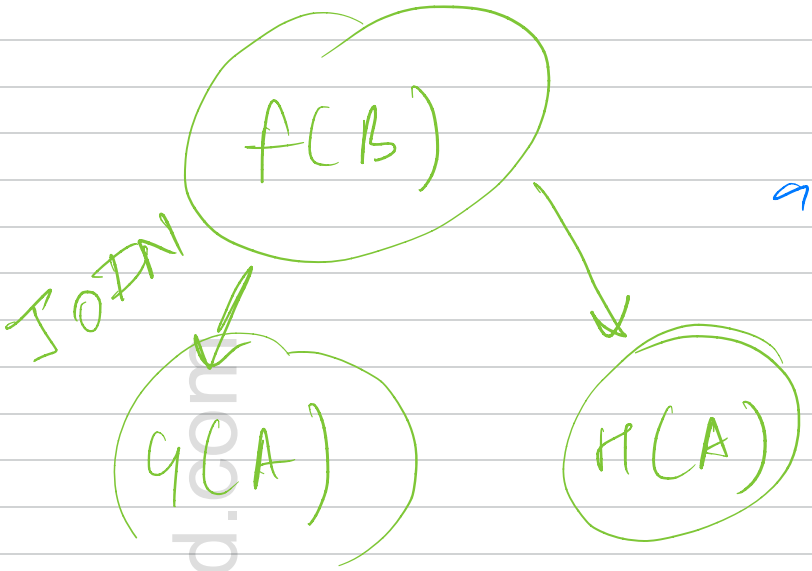
$D = H(A)$

$FA = FUTURE(f(B))$

$FC = FUTURE(g(f(A).get()))$

$FD = FUTURE(H(f(A).get()))$

BLOCK / WAIT until A is Available



future provide a very natural way to convert functional program to parallel program

```
fSUM1 = FUTURE { LOWER };

fSUM2 = FUTURE { UPPER };

SUM = fSUM1.get() + fSUM2.get()
```

futures in JAVA FORK & JOIN

* NOT RECURSIVEACTION

CLASS ASUM EXTEND RECURSIVETASK

FIELDS : ARR, LO, HI

COMPUTE() {

if (low > HI) RETURN 0;

else if (low == HI) RETURN ARR[LO]

else {

MID = (low + HIGH) / 2

L = NEW ASUM (LOW, MID, ARR)

R = NEW ASUM (MID+1, HI, ARR)

R.FORK(); // FUTURE

RETURN L.COMPUTE() + R.JOIN();

}

Join for future implements future get operation

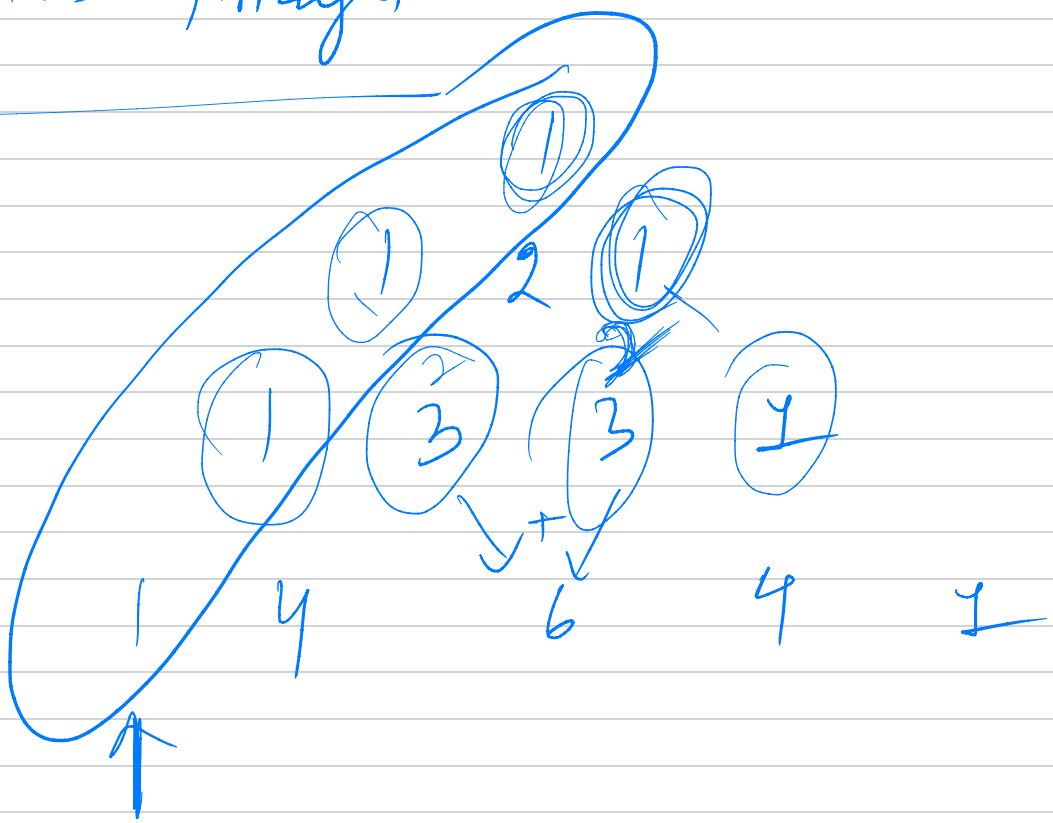
Memoization

y1 = g(x1)
INSERT(g, x1, y1)
y2 = g(x2)

y3 = g(x1) → LOOKUP(g, x1)

fy1 = FUTURE{ g(x1) }
INSERT(g, x1, fy1)
y2 = g(x2)
wasted work in sequential
y3 = LOOKUP(g, x1).get()

PASCALS Triangle



Java Streams

①
```
FOR ( S : STUDENTS )
    PRINT S

STUDENTS.STREAM().( S → PRINTS)
```

②
```
FOR ( S : STUDENTS )
    if ( S is ACTIVE )
        ACTIVE.ADD( S )

FOR ( A : ACTIVE )
    AGESUM += A.AGE ;

AVG = AGESUM / ACTIV.SIZE()
```

USING STREAMS

```
STUDENTS.STREAM().
        FILTER(S -> S.ISACTIVE).
        MAP(S -> S.AGE).
        AVERAGE();
```

TAKEAWAYS

→ ITS EASY TO CONVER SEQUENITIAL STREAM TO PARALLEL

→ USE PARALLEL STREAM instead of STREAM

→ AVG & FOREACH ARE TERMINAL OPERATIONS

→ FILTR & MAP ARE INTERMEDIATE OPERATONS

DATA RACES & 2 TYPES DETERMINISM

We are interested in parallel programs that are both structurally & functionally deterministic but not always!

FUNCTIONAL DET

Same input will always lead to same output

has no data race

STRUCTURAL DETERMINISM

SAME INPUT lead to SAME COMPUTATIONAL GRAPH (structure of parallel program)

Benign Non Determinism (Next Page)

FUTURE ASYNC

Read

Write

SUM1 = SUM of LOWER HALF

SUM2 = SUM of upper HALF

SUM = SUM1 + SUM2

.get()

BUG IN PARALLEL PROG (DATA RACE)

Read & Write OP in parallel

WRITE & WRITE in parallel.

(vice versa if code is having DRF it is not S Deterministic)

DRF → Data Race Freedom

vv

(can be achieved when we achieve functional & Struct determinism)

⇒ FUNCTIONAL + STRUCTURAL Determinism

(BENIGN NON DETERMINISM)

Cases when we get diff output but they are all accepted

eg Search → PARALLEL LOOP

~~FINISH~~ &

FOR (i : [0 : M-N])

FOUND = TRUE

ASYNC FOR (j : [0 : N-1]) {

IF (TEXT(i+j) != PAT[j])

FOUND = FALSE

value of index is non deterministic but the answer is acceptable

}

IF (FOUND = TRUE) index = i;

If multiple Cousins have same string then any index is the right example;

PARALLEL LOOPS

When the task that is being performed inside the loop is independent of each other we can easily use parallelism to compute them independently

eg1

```
finish {
    for (p = head ; p != null ;
          p = p.Next)
    {
        async p.compute
    }
}
```

eg2

Any for all → This will convert it to a parallel stream

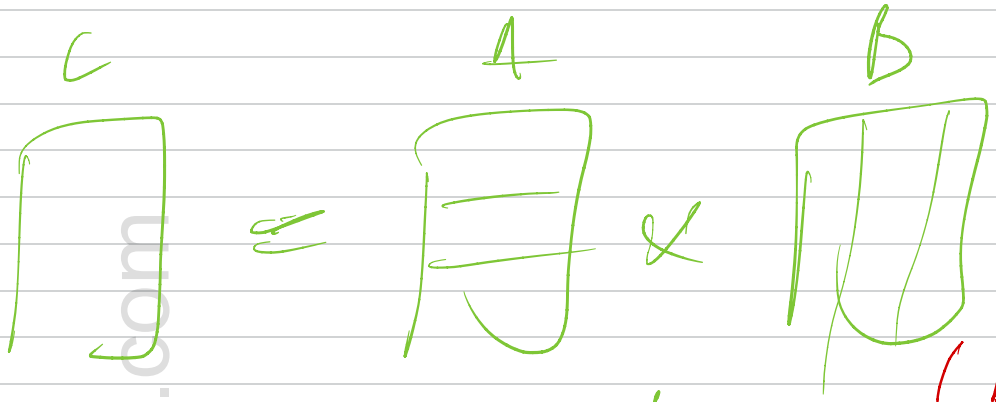
```
forall (i : [0-n-1])
    a[i] = b[i] + c[i])
```

A better way is by using the JAVA API's parallel stream

```
a = IntStream.rangeClosed(0, n-1).
        parallel.
        .toArray( i -> b[i] + c[i]);
```

Matrix Multiplication

This is used to measure speed of fastest computers in the world (benchmarking)



$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] * B[k][j]$$

PORALL → Parallelism

```
FOR([i,j] : [0:N-1, 0:N-1])
{
    C[i][j] = 0;
    FOR(k:[0-N-1]) {
        C[i][i] += A[i][k] *
        B[k][j];
    }
}
```

→ Can we run this in parallel

No! This will lead to data race

→ It will remain as it is

Barriers in parallel loop

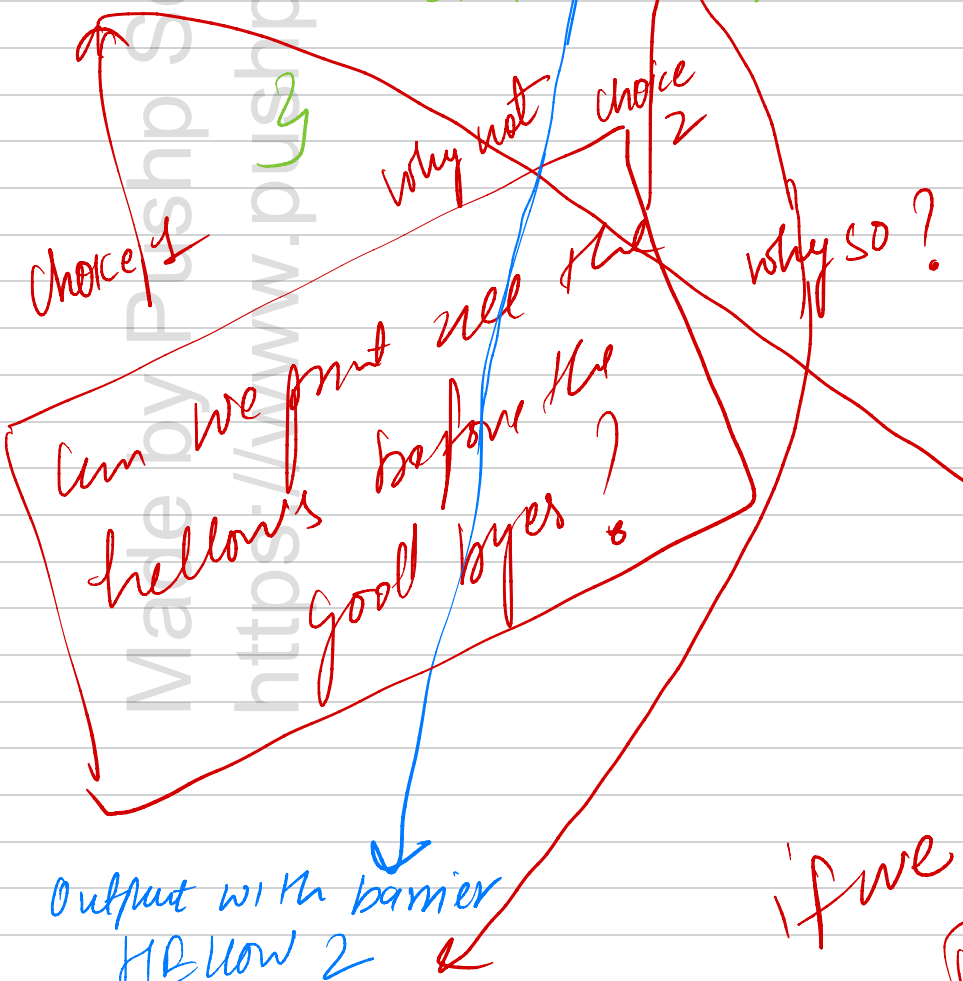
```
FORALL (I : [0 : N-1])
    S = LOOKUP(I);
    PRINT = "Hellow" + S
    BARRIER
    PRINT = "BYE" + S
```

OUTPUT

This order will be retained

HELLOW TWO
HELLOW ZERO
BYE TWO
HELLOW ON
⋮

Different Combination



We have to repeat the LOOKUP

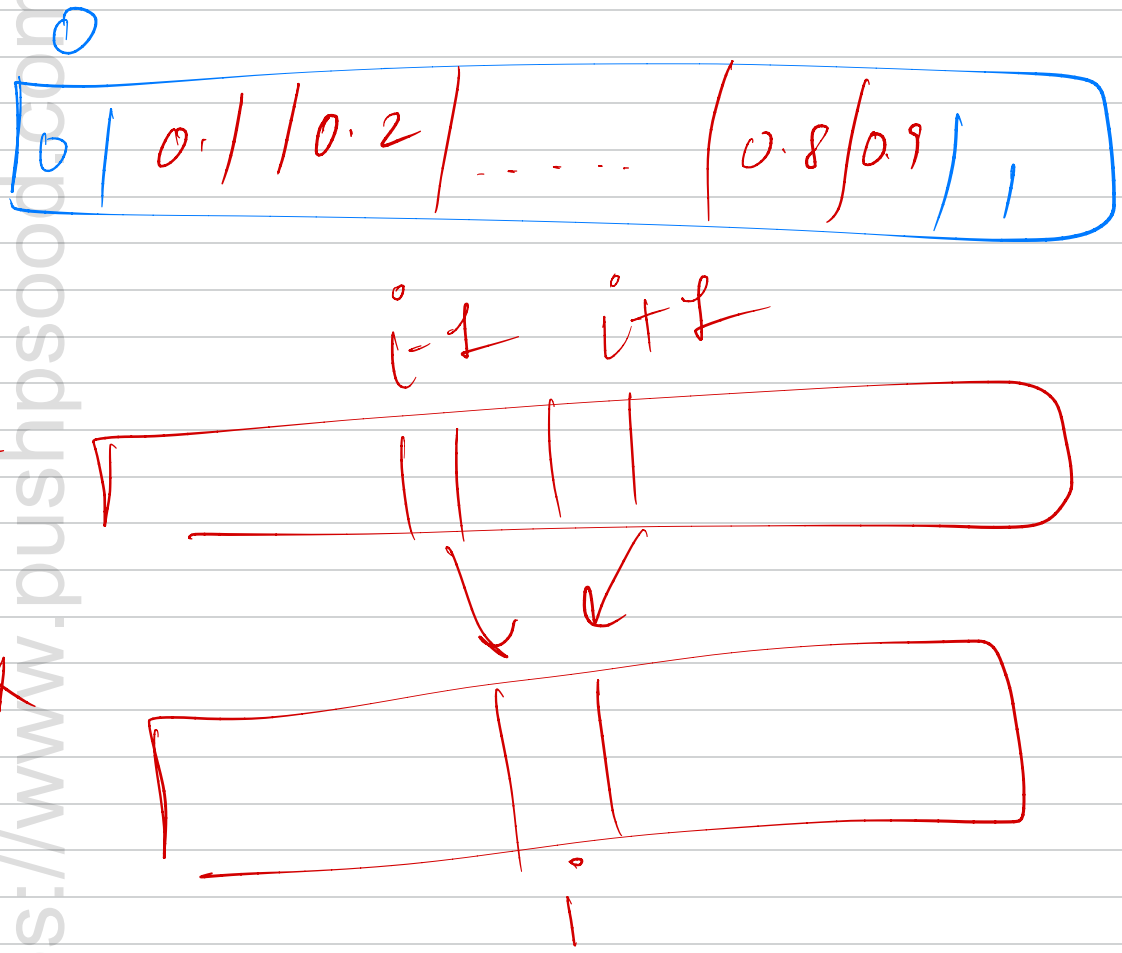
if we have another FORALL for GOOD BYE which can be done concurrently

$x_0 = 0 \qquad x_N = y$

$$x_i^0 = (x_{i-1} + x_{i+1})/2$$

$0 < i < N$

$n = 10$



ONE DIMENTIONAL ITERATIVE AVERAGING APPROACH

sequential

FOR (ITER : [1 : NSTEPS]) {

No of time we want to repeat

NSTEPS

FORALL (I : [1 : N-2]) {

(N-2 TASKS)

NEWX[i] = AVG (OLDX[I-1], OLDX[I+1])

}

SWAP POINTERS

}

Quadratic

using barriers

(N-1) TASKS

(N-1) x NSTEPS

FORALL (I : [1 : N-2]) { BARRIER

FOR (ITER : [1 : NSTEPS]) {

NEW x[I] = AVG (OLDX[i-1] OLDX[i+1])

BARRIER

SWAP POINTERS

}

}

FORALL (I : [0 : N-1])

A[I] = B[i] + C[i]

Too Many ! N TASKS

Solution

No of groups

Total task will be NG

FORALL (g : [0 : NG-1])

How to determine?

Not always required to create these many tasks

FOR [i : MyGroup (g, NG, [0 : N-1])]

{

A[I] = B[i] + C[I]

}

fix to too Many task is chunking or iteration grouping

CHUNKS of EQUAL SIZE

I=0 1 2 3 4 5 ..

BLOCK

G=0 G=1

NORMAL GROUPING : BLOCK

(1)

i 0 1 2 3 4 5

G1 G3 G2

CYCLIC

Group = i MOD(ng)

(2)

Split phase Barriers with Java PHASERS

~~CPL SPAN = 200~~

```
forall (i : [1:N]) {
    print "HELLO" + i;
    ARRIVE
    myid = lookup(i);
    AWAIT ADVANCE
    print "GOODBYE" + myid;
}
```

~ 100 unit of time

~ 100 unit of time

Barrier

Do local work

NEXT

W/ PHASER CPL = 100

SPAN

- ARRIVE AND AWAIT ADVANCE

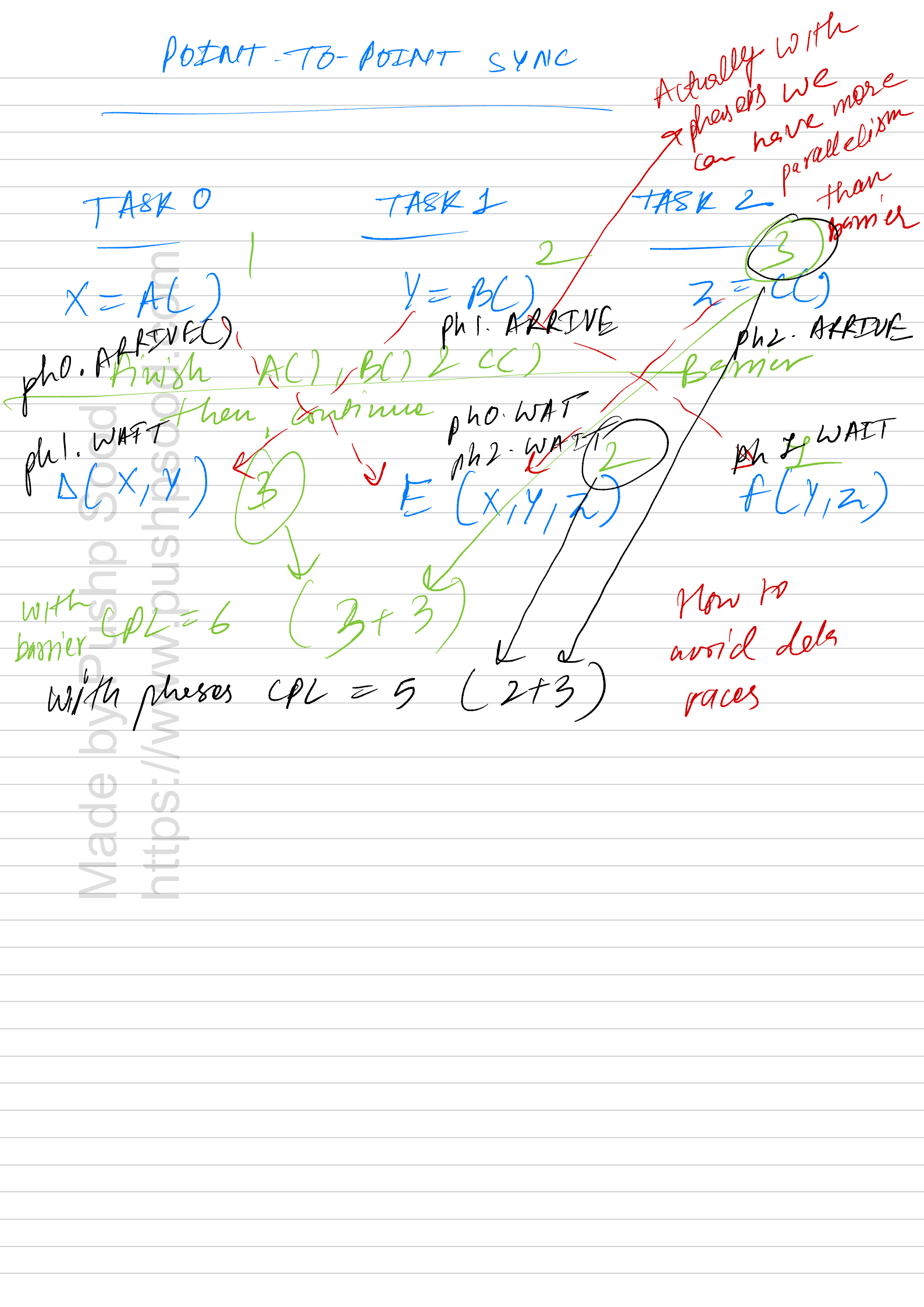
  it is actually a ~~Barrier~~

- ARRIVE

  LOCAL WORK → Not associated can be executed

- AWAIT ADVANCE

POINT-TO-POINT SYNC



1-D Iterative Averaging with Phasers

```
ph[] = new Phaser[N+2];
forall (i : [0 : N-1]) {
  for (iter : [0 : nsteps-1]) {
    newX[i] = avg(oldX[i-1], oldX[i], oldX[i+1]);
    ph[i].arrive();
    ph[i-1].wait();
    ph[i+1].wait();
  } // Barrier
  swap newX, oldX
} // used earlier
```

PIPELINING

MRI have 3 main stages → P pipelining stages

while(){ DENOISING TASK 0 PHO.ARRIVE

while(){ pho.AWAIT() REGISTRATION (TASK 1) ph1.ARRIVE() }

while(){ ph1.AWC() SEGMENTATION TASK 2 }

IMG 1 → D_1

IMG 2 → D_2

D_N

R_1

R_2

R_N

S_1

S_2

S_N

time

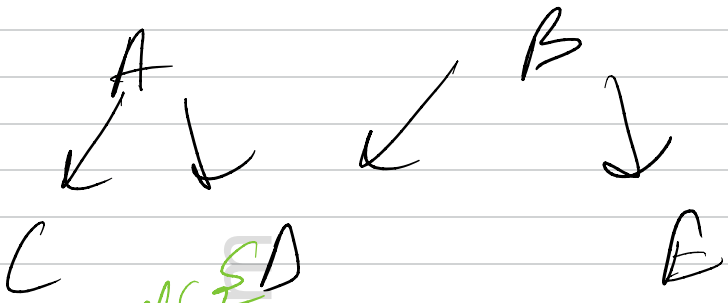
$$WORK = N * P$$

$$CPL = N + P - 1$$

$$PARALLELISM = N+P / (N+P-1)$$

$$\approx P$$

DATA FLOW




```
ASYNC {
    A.GET()
    B.GET()
    D.WORK
    ...
}
```

possibility 1

works but implicit dependency!

```
ASYNC {A.WORK; A.PUT()}
ASYNC {B.WORK; B.PUT()}
```

explicit

```
ASYNC AWAIT (A) {
    C.WORK
}
```

precondition

```
ASYNAWAIT (A, B) {
    D.WORK
}
ASYNC AWAIT (B) {
    E.WORK
}
```

possibility 2