for Mid2 - review

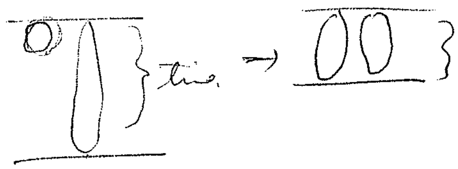
= MK book - parallel Software
Ch2

- shared-mem program — start a process → forks threads
- vs. distr.-mem program — start multiple processes

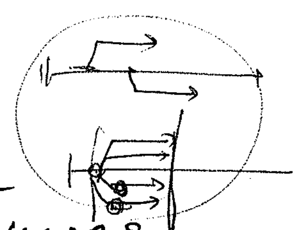
- SPMD — same program for all threads — for data parallel (also task par.)
- MPMD — running a different program on each processor (core)

- (data parallel — divide data ; task parallel — divide task) programming

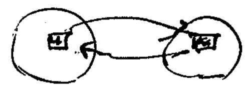
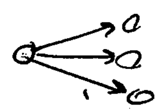
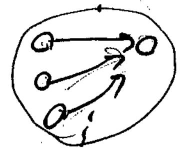
- process/threads coordination issues
  - load balancing
  - min comm.
  - synchronization



- shared-mem program



- vs.
  - dynamic threads — high cost for fork/join
  - static threads — all are forked at once
    - less efficient resource usage
- non-determinism on thread termination order
- C.S. and mutex control
  - mutex
  - semaphore
  - monitor
  - busy-waiting
- thread safety — some serial built-in func malfunctions in parallel prog.

- distr.-mem system/program issues
  - multiple processes ~~on~~, each on different cpu (and op)
    distr. system
  - MSG passing API
    - send / receive
      blocked until sender sends msg.
  - One-sided comm. 
  - Collective communication
    - broadcast 
    - reduction 
- hybrid system programming
  MPI + OpenMP — on cluster of multicore machines.

= I/O for parallel programming

suggested rules

- file access — only a single process/thread accesses any single file (not 2 processes open the same file)
- std I/O
  non determinism

= performance

$$\boxed{T_p = \frac{T_s}{p}}$$

— ideal case : ~~serial~~ linear speedup

but, ∃ overheads
- shared mem — C.S.
  serial
- distr-mem
  - data transfer
  - IN speed.

$$\boxed{\underset{\text{ideal}}{\text{Speedup}} = \frac{T_s}{T_p} = \frac{T_s}{\left(\frac{T_s}{p}\right)} = p}$$

↓

Efficiency (per processor) $\boxed{E = \frac{S}{p}} = \frac{\left(\frac{T_s}{T_p}\right)}{p} = \frac{T_s}{p \cdot T_p} = \frac{1}{p}\left(\frac{T_s}{T_p}\right)$

$\Rightarrow$ — effective $T_p = \frac{T_s}{p} + T_{overhead}$ — vs. — ideal $T_p = \frac{T_s}{p}$

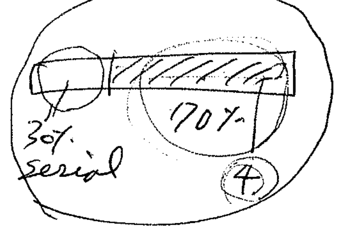
effective $S = p \cdot \underbrace{\left(\frac{T_s}{T_s + p \cdot T_o}\right)}_{<1}$ — ideal $S = p$

— see real example case graphs

— linear speed up (E is always 1)

- measuring parallel prog. time — should exclude I/O time
  - elapsed time — $\underbrace{\text{cpu time} + \text{I/O time}}$
  - Amdahl's law

$$\boxed{S = \frac{1}{(1 - Fe) + \left(\frac{Fe}{Se}\right)}} \quad Se = 4$$



④ — Scalability

- parallel prog. is scalable

  if p and $n \Rightarrow E$ }
  and $k*p$ and $k*n \Rightarrow E$ } same Efficiency

vs.

- strongly scalable — p and $n \Rightarrow E$, $k*p$, $n \Rightarrow E$ } same E — whatever n
- weakly scalable — p and $n \Rightarrow E$, $k*p$, $k*n \Rightarrow E$ } same E

— parallel program design steps

1. partition
2. Communication
3. aggregation — simplify — reduce comm. cost
4. mapping — assign tasks to process/threads

ex) serial prog → parallel prog.

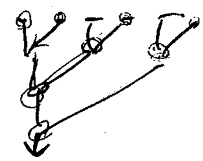
— histogram making

issues

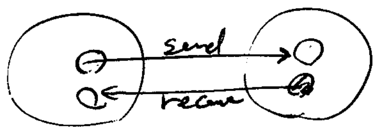
— race condition — C.S.

— reduction — by 1 process only

by full-tree reduction



— msg passing comm.



— shared-mem comm

— How to implement this in Java Threads?

⇒ each Thread holds shared-obj and read/write on the shared-obj

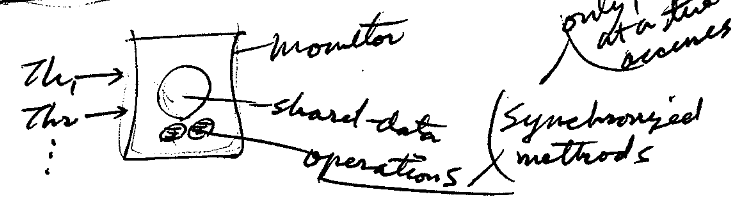
— Java Threads

concept

when create the thread object, pass shared-obj

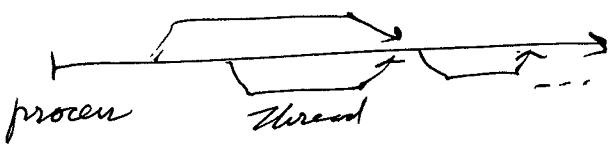
— readers/writers in Java Threads

— monitor mechanism

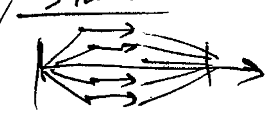


= MK. Ch4 — Shared-mem program with pThreads

issues
- C.S.
- Thread synchronization



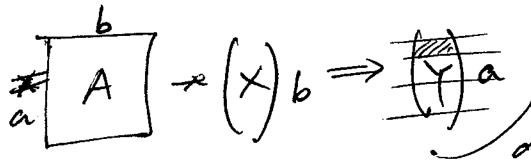
— dynamic / static



— each thread has its stack/heap (private)

eg multiple threads call same func → each thread has its own copy of func's para, local vars.

eg matrix-vector multiplication



divide Y by p threads (a/p each)

- Thread (slave) func
  using rank (thread-id), computes first/last index

— issues
- C.S. — sol: busy-waiting ×, mutex — 1/0, Semaphore ✓
- synchro.
- producer/consumer
- condition var
- read/write locks
- false-sharing

— c.s. { multiple threads try to access shared-mem
→ race condition

e.g.) π computation $\pi = 4*(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots)$

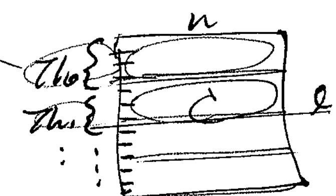
global-sum update → sol: { mutex ✓
busy-waiting — too much overheads
— semaphore

— prg3 matrix multi
with pthreads

$$\boxed{C_{ij} = \sum_{k=0}^{n-1} A_{ik} * B_{kj}}$$ — nested for-loops

serial algo → parallel algo. — $\boxed{\text{block divide way}}$
— rotational way

A × B → C



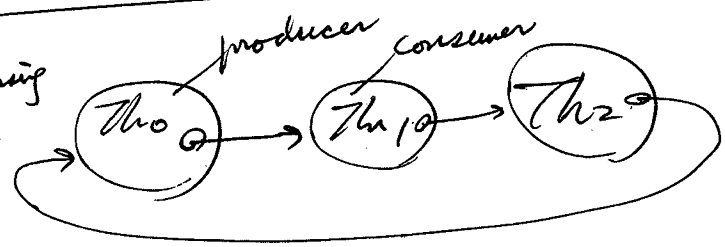
l/p each Th.

load balancing
(Some threads are assigned more

1001/4
Th0 — start-idx
— end-idx

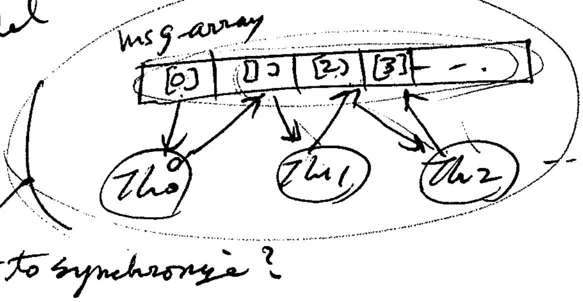
§4.7
— producer-consumer synchronization.

— msg-passing model



case
{ Thread-ordering is important

— shared-mem model



{ each ~~thread~~ sends to next,
then reads from prev. Th.

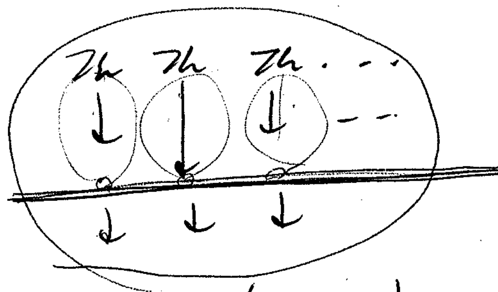
issue
How to synchronize?

↓ producer-Consumer Synchro.

Sol:
- busy-waiting — high cost
- mutex — X
- semaphore — V (best)

---

§4.8 — Barrier and Condition var



barrier (align)

— implementation of barrier
- — busy-waiting — X
- — mutex — X

  — problem ~~on~~ reusing for the 2nd barrier
- — Semaphore — X — may cause race condition
- V condition var — pthread supports.

actually, condition var & mutex

Concept: lock mutex;

$$\boxed{\begin{array}{l} \text{if condition,} \\ \quad \text{signal thread(s)} \\ \text{else} \\ \quad \text{unlock mutex;} \\ \quad \text{block} \end{array}}$$

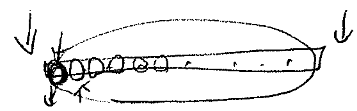
— c.s.

wait

unlock mutex;

— §4.9 — read/write locks — pthread supports

— multiple threads r/w big data structure



Sol:
1. one mutex for entire data structure
2. one mutex per each ele. — worst
3. using pthread r/w locks — best time

CREW

~~read lock~~

ops:
- rdlock()
- wrlock()
- unlock()

- if one is reading, any writer should wait;
- if one is writing, any reader/writer wait;

— §4.10 — Cache coherence and false sharing.

ex) matrix-vector multiplication

matrix $A_{m \times n} * (x)_{8000} = (y)_8$

(ex) each cache line = 64B (8 doubles)

1. A is 8,000,000 × 8 — worst in cache-W miss
   Y[i]=0.0; — initialize Y

   (Y) 8,000,000
2. A is 8000 × 8000
3. A is 8 × 8,000,000 — worst in cache-R miss

```
for (j=0~n)
    Y[i] += A[i][j] * X[j]
```

(X) 8,000,000

reading X n times

Mid2