ch1 (basic book)

- ILP → TLP processor design migration
- global_sum computation in parallel
  - Compute local start/end indexes
  - Compute local sums
  - Combine local sums into global sum
    - serial vs. tree-reduction way
    - $O(p)$ $O(\log_2 p)$
- task parallel vs. data parallel
- communication
  synchronization
  load balancing
- parallel programming paradigms
  - HL parallel Lang.
  - vs. explicit (e.g. OpenMP, MPI, ..) pThread
- process/Thread
- demo — Unix fork/join — output trace
  - #include <unistd.h> — for fork()
  - #include <sys/wait.h> — for wait() ≡ join

Ch2 — (basic book) parallel HW and SW

— von Neumann bottleneck : cpu speed ≠ mem speed (<)

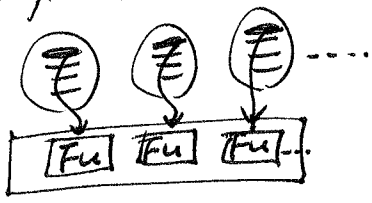
sols:
1. cache
2. V.M.
3. ILP — fine grain
4. TLP — coarse "

— cache access pattern affects performance
C/C++ row-major vs. col-major access
more misses

— ILP — [scalar pipeline
multiple-issue machines
[VLIW
superscalar) SW vs. HW speculation

— TLP — [multi threading (e.g., SMT)
multicore processor

SMT — fine grained



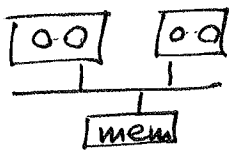
1 instr. per thread → compete for resources.

— processor design migration

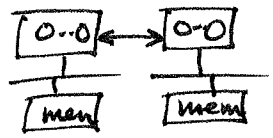
scalar processor
↓
scalar pipelined "
↓
serial control flow
(ILP) — multi-issue "
(VLIW/superscalar)
↓
(TLP) — TLP processors
(multi threading/multicore)
multiple control flows

- multicore processor design issues
  - IN — should be scalable
  - fault tolerency
  - low power consumption
  - efficient mem. hierarchy
- building shared-mem system with multicore processors

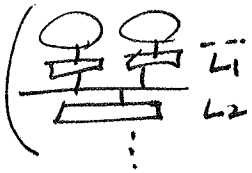
UMA



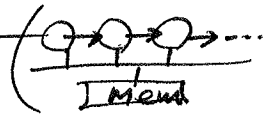
NUMA



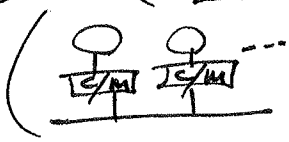
- multicore chip arch.
  - hierarchical design — e.g., Core i5, i7



  - pipelined "



  - network "



Springer book

Parallel Arch. and taxonomy

- Flynn's taxo.
  - SIMD
  - MIMD
- memory organization of para. comp.
  - shared-mem
  - distributed mem.
  - virtual shared mem. (DSM)

— Kai Hwang's taxo. of MIMD systems

pvp, smp > UMA
DSM — NUMA
] shared mem.

MPP, COW > NORMA — distr. mem.

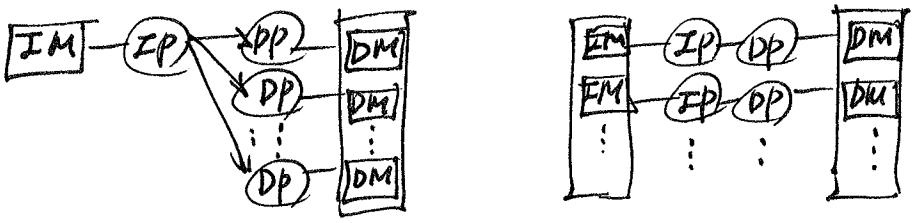
— SIMD mode computation

vs. [Vector processors
array processors]

eg. $(A) + (B) \Rightarrow (C)$ $\quad C_i = A_i + B_i \ (1 \le i \le n)$

performance on SISD vs. SIMD
[vector processor way
array processor way]

— SIMD vs. MIMD — characteristics



— data-parallel applications (fine-grained)

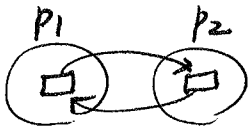
— coarse-grained parallel applications
— data-parallel / task-parallel.

Ch2 — basic book

- parallel programming on MIMD
  - shared-mem system
    - a process → forks threads
  - distributed-mem system
    - multiple processes
  - SPMD — same program for all threads — (for both data/task parallel
  - MPMD — running a separate prog. on each processor.

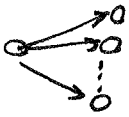
- shared-mem. programming
  - dynamic threads — when needed
  - static threads — at the same time (at begining)
  - non-determinism — on thread termination
  - critical section and mutex control — (mutex, semaphore, monitor, busy-waiting)
  - thread safety — (some serial, built-in func., malfunctions in parallel prog.

- distributed-mem programming
  - multiple processes, each on different cpu/opsys.
  - MSG-passing API — send/receive — synchronization
  - one-sided comm.

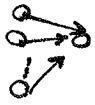


controlled by 1 process (P1) ex)

  - collective comm.
    - broadcasting



    - reduction



— on COW (cluster of multicore nodes)

hybrid programming is possible

eg) MPI + OpenMP model

— I/O for parallel programming

suggested rules:

- file access: a single process/thread should access any single file (not 2 processes open the same file)
- std io
  - non-determinism

= performance

ideal

$T_p = \frac{T_s}{p}$ — ideal case (linear speed up) ($S=p$)

ideal speedup (S) $= \frac{T_s}{T_p} = \frac{T_s}{\left(\frac{T_s}{p}\right)} = p$

E is always 1

$E = \frac{S}{p} = 1$

efficiency $E = \frac{S}{p} = \frac{\left(\frac{T_s}{T_p}\right)}{p} = \frac{T_s}{p \cdot T_p} = \frac{1}{p}\left(\frac{T_s}{T_p}\right)$

(speedup per processor)

effective

$T_p' = \frac{T_s}{p} + T_{overhead}$

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

effective speedup

$S' = \frac{T_s}{T_p'}$

$T_p' = \left(\frac{T_s}{p} + T_o\right)$

— maximum speedup from parallelization

Amdahl's law

$$S_p = \frac{1}{(1-F_e) + \left(\frac{F_e}{S_e}\right)}$$

ex) 10% serial, 90% parallel
⇒ max. speedup achievable?

— <u>Scalability</u> — of parallel program

problem size n, # of processors p.

$\left(\begin{array}{l} n \to k*n \\ p \to k*p \end{array}\right)$ — yielding $\underbrace{\text{same } E}_{\text{with } n,p}$ — <u>weakly scalable</u>

only <u>$p \to kp$</u> — yields $\underbrace{\text{same } E}_{\text{with } n,p}$ — <u>strongly scalable</u>.

— parallel program design steps

1. partition
2. determine communication
3. aggregation — reduce comm. cost
4. mapping — assign tasks to processes/threads

ex) histogram making program

vs.
- shared-bin way → race condition (mutex)
- local-bin and then combine way
  - data parallel
  - send/receive

Advanced book ch1.

- speed up
- Efficiency
- cost : $c = T_p * P$ — ideal case, $c = T_s$
- scalability

- Computation-to-comm. ratio → the higher the better.

- runtime analysis example — with global sum computation

  $\sum_{i=0}^{n} A[i]$

  serial time $T_s(1, n) = n-1$

  parallel time $T_p(P, n) = T_p(2^q, 2^k) = \underbrace{(2^{k-q}-1)}_{\text{comp. time}} + \underbrace{7q}_{\text{comm. time}}$

  ⇒ weakly scalable program.

  (proof)

- parallel program design
  - partitioning
  - Communication
  - synchronization
  - Load balancing

- prefix_sum example — concept
  - step1: local_sum in each processor
  - step2: prefix_sum comp. with right-most values only from each local array
  - step3: addition of resulting value from step2 to right neighbor.

- image classifier example

  task-vs. data-parallel approaches.

Advanced book ch2.

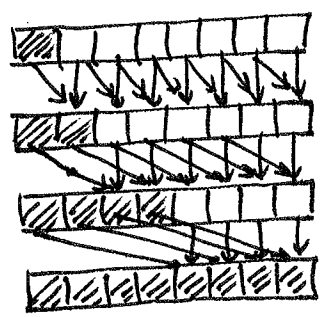
- PRAM models (shared-mem)
  - EREW
  - CREW
  - CRCW
    1. priority CW — highest priority processor writes.
    2. arbitrary CW — random selection
    3. Common CW — when all values are equal, write.
    4. Combining CW — all values are combined into a single value. (e.g. sum, min, ...)

- parallel prefix Computation (on PRAM)

$$\begin{cases} S_0 = X_0 \\ S_1 = X_0 \circ X_1 \\ \vdots \\ S_{n-1} = X_0 \circ X_1 \circ \cdots \circ X_{n-1} \end{cases}$$

($\circ$ — binary operation)

1. recursive doubling with n processors



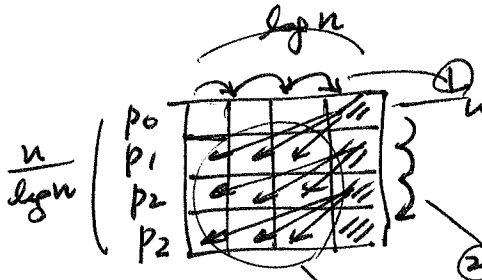
$\log_2 n$ steps — (in each step, all ops are parallel — using n processors.)

$$\underset{\text{cost}}{C(n)} = T(p,n) * p = \Theta(\log n) * n = \underbrace{\Theta(n * \log n)}_{\neq \Theta(n)}$$

→ not cost optimal.

2. cost optimal approach with $p = n/\log n$ processors.

↓



① in each processor, $\log n$ steps

② using right-most values, $\log_2 n$ steps.

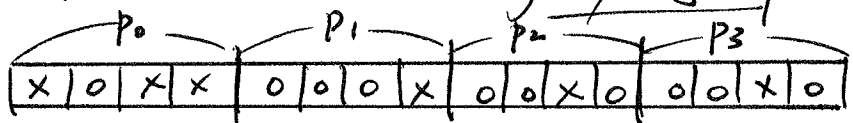
③ adding rightmost value to next processor's elements. $\log_2 n$ steps

$$\Rightarrow C(n) = T(p,n) * p = \Theta(\log_2 n) * \frac{n}{\log_2 n} = \underbrace{\Theta(n)}_{= T_s}$$

cost-optimal

— sparse array compaction example (on PRAM)

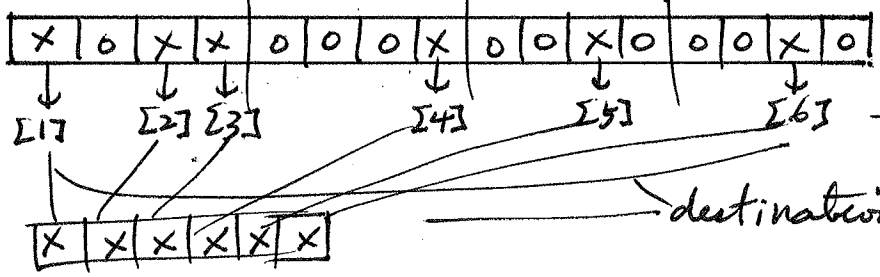
— cost-optimal approach using $n/\log n$ processors



1. mask: 1 0 1 1 | 0 0 0 1 | 0 0 1 0 | 0 0 1 0 — $\log n$ (in each processor)

2. compute parallel prefix sum on mask array — $\log n$ (in each processor)

3. ① 1 ② ③ | 3 3 3 ④ | 4 4 ⑤ 5 | 5 5 ⑥ 6



— $\log n$ (in each processor)

destination array (compact)

$$\Rightarrow C(n) = T(p,n) * p = \Theta(\log n) * \frac{n}{\log n} = \underbrace{\Theta(n)}_{= T_s}$$

cost-optimal

— Foster's parallel algo. design method.

partition → comm. → agglomeration → mapping

ex) Jacobi iteration