Parallel Processing + Models (Ch. 1)

Course Goals:
- Learn how to design "good" parallel algo
- Analyze & Select best algo for machine/problem
  (alg good for one arch not necessar. good for others)

Sequential Processing
- one CPU, performs one oper at a time
- traditional RAM model studied in 311, 629

Parallel Processing
- multiple processing elements, can perform operations simultaneously (in parallel)
- complicating factors (over sequential computing)
  • inter-processor communication
  • synchronization among processors
  • scheduling (processor allocation)
- no generally accepted theoretical model suitable for all architectures

Remark: I see no conceptual difference between parallel + distributed computing. Others disagree.
(including text). Usually, parallel means procs "close" + distributed means pross "far" apart

Why Parallel Processing?
- solve problem faster
- solve larger problems than possible w/ sequent. machine
Many Divide-And-Conquer Algs have "natural" parallel vers.

Ex: Merge Sort

\[
\text{mergesort} \left( A \right) \\
\text{mergesort} \left( 1^{\text{st}} \text{ half } A \right) \\
\text{mergesort} \left( 2^{\text{nd}} \text{ half } A \right) \\
\text{merge} \left( 1^{\text{st}} + 2^{\text{nd}} \text{ halves } A \right)
\]

end

\textbf{Time of parallel Alg (assume n procs)}

\[
T_n(n) = T_{n/2} \left( \frac{n}{2} \right) + M(n)
\]

Suppose \( M(n) = n \)

\[
T_n(n) = T_{n/2} \left( \frac{n}{2} \right) + n \\
= T_{n/4} \left( \frac{n}{4} \right) + \frac{n}{2} + n \\
= \sum_{i=0}^{\log n} \frac{n}{2^i} = O(n) \quad \left\langle \text{faster than sequential} \right\rangle
\]

\textbf{Work of parallel Alg}

\[
\leq p \cdot T_p(n) \quad \left( \text{if all procs work whole time} \right)
\]

\[
= O(n \cdot T_n(n)) = O(n^2) \quad \left\langle \text{does more work than seq.} \right\rangle
\]
Performance of Parallel Algorithms

Suppose have problem Q of size n and a PRAM algorithm for solving Q that runs in time $T(n)$ using $P(n)$ processors.

Equivalent Performance Measures

1. $T(n)$ time and $P(n)$ processors
2. $T(n)$ time and $C(n) = T(n) \cdot P(n)$ cost
3. $O(T(n) \cdot P(n)/p)$ time for $p \leq P(n)$ processors
   - can "slow" down algorithm keeping cost the same
   - have each processor simulate multiple processors
4. $O((C(n)/p + T(n))$ time for any # $p$ of processors
   - need at least $T(n)$ time but maybe more if $p<P(n)$

Note: For (3) + (4) need to take care of scheduling issues, i.e. which proc does what when (aka processor allocation)

Work-Time Framework

$W(n) =$ work of parallel alg = total # operations performed

Fact: $W(n) = O(T(n) \cdot P(n))$

i.e., in worst-case every processor is busy during every time step of algorithm
def: pardo statement

\[
\text{for } 1 \leq i \leq u \text{ Pardo Statements for iter } i \text{ end pardo}
\]

- don't have prog. for each separate processor
- oper. in iteration depend on index i
- all iterations independent and can be run in parallel

- time for pardo: time for each iteration
- work for pardo is \( O((u-1) \text{ (time for iteration)}) \)

Algorithm Design w/ Pardos

⇒ describe algorithm as series of pardo statements

Time of algorithm = sum of time for each pardo \( (T(n)) \)
Work of algorithm = sum of work for each pardo \( (W(n)) \)

NOTE: Can generally implement an algorithm using \( T(n) \) time and \( W(n) \) work on a p processor PRAM in \( O(W(n)/p + T(n)) \) time (see text)

- Basic idea, make 1 physical processor simulate more than one "virtual" processor
- Again, we have neglected scheduling issues, i.e., how to schedule each pardo on p processors (aka processor allocation problem)
Problem P of size n

Speedup
\[ T_{s}(n) = \text{Time taken by best known sequential alg } A_s \text{ for } P \]
\[ T_{p}(n) = \text{Time of parallel alg } A_p \text{ to solve } P \text{ w/ } p \text{ procs} \]
\[ S_{p}(n) = \frac{T_{s}(n)}{T_{p}(n)} \]

- Best case, \( S_{p}(n) = p \)
- In reality, may not be achieved
  - insufficient concurrency in computation (limitation imposed by problem)
  - overhead of synchronization, communication, scheduling, etc.

Efficiency
\[ E_{p}(n) = \frac{T_{s}(n)}{p \cdot T_{p}(n)} \]

- Measures effective utilization of processors
- Best case \( E_{p}(n) = 1 \), i.e., all processors work all the time
- Some upper bound on #procs can use efficiently fastest parallel running time is \( T_{\infty}(n) \)
  - Efficiency of alg degrades as \( p > T_{s}(n)/T_{\infty}(n) \)
Shared Memory Model - PRAM, Parallel RAM
- natural extension of sequential RAM
- p processors, each w/ own local memory
- communicate via shared (global) memory
- in one time step a proc can read/write global mem or perform one arithmetic operation

- processors have unique id (they know it)
- synchronous (usually), all procs have same clock
- MIMD - multiple-instruction multiple-data
  (each proc can execute its own program on own data)

Example: Sum on PRAM

```
sum (A)
i := get-my-id() 
B(i) := A(i) 
for h := 1 to logn do 
  if (i ≤ n/2^h) then 
    x := B(2i-1) 
    y := B(2i) 
    B(i) := x + y 
  endif 
endfor 
sum := B(1) 
end
```

- Why synchronous?
- Concurrent read/write?

Time: T_n(n) = O(log n)

Work: O(n · T_n(n)) = O(n log n)

Each proc checks in every iter.
Variations of the PRAM

**EREW PRAM** Exclusive-Read Exclusive-Write
no simultaneous access to any shared memory cell

**CREW PRAM** Concurrent-Read Exclusive-Write
simultaneous reads, but not writes, to shared memory cell

**CRCW PRAM** Concurrent-Read Concurrent-Write
simultaneous reads & writes to shared memory cells
• How to resolve concurrent writes?

  **Common CRCW PRAM** - all procs must write same value (if they write)

  **Arbitrary CRCW PRAM** - an arbitrary proc (unknown) will win (write value "last")

  **Priority CRCW PRAM** - proc with min (max) index (proc id) wins

**Power:** EREW < CREW < CRCW

Some probs known to require $\Omega(\log n)$ time on EREW or CREW, but can be done in O(1) time on CRCW
• compute OR of $n$ bits (w/ $n$ procs)
• compute MAX, MIN of $n$ values (w/ $n$ procs)

**Class NC** (high level) problems that can be solved efficiently w/ PRAM (like Class P)
• time polylog., e.g. $O(\log^c n)$ constant $c$
• polynomial # procs, e.g. $O(n^{c_2})$ procs, constant $c_2$
like NP-Complete probs for $P$, we have P-Complete probs for NC.
Example: Computing OR on PRAM

\textit{EREW PRAM} ⇒ like sum alg, but replace "+" with "OR"

- \(O(\log n)\) time, \(n\) proc, \(O(\log n)\) work
- matching lower-bound on time \(\Omega(\log n)\) (time optimal)
- can be smarter & use only \(\log n\) proc in same time ⇒ gives \(O(n)\) work, which is optimal

\textit{CRCW PRAM} - common model (when all write same value)

- use auxiliary variable \(x\) initialized to zero
- use \(n\) proc, proc \(i\) reads \(A[i]\)
  - if \(A[i] = 1\), proc \(i\) sets \(x\) to 1
  - if \(x = 1\), then result of OR is 1
  - if \(x = 0\), then result of OR is 0

The \textit{EREW} algorithm can be modified to work for \(\text{Min + Max}\) too (replace "+" by \(\text{Min + Max}\))

A more complex approach, using \(n^2\) proc, can be used to solve \(\text{Min + Max}\) on \textit{CRCW PRAM} in constant time.
Networks

- set of processors (w/ local mem)
- connected by interconnection network (viewed as a graph)
- synchronous or asynchronous
- message passing (maybe)

Generally, in one time step, a proc can communicate w/ one of its neighbors in graph
- lower bound on most probs (in worst-case) is the diameter of graph (max dist between any pair of nodes).
- need routing alg/mehc to determine how any pair of procs communicate
- usually have set of basic communication primitives such as broadcast (one proc sends to all).

E.g. Mesh (2d array)

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

\[\text{diameter} = \sqrt{p}\]

Hypercube \((p = 2^k)\)

\[
\begin{array}{ccc}
0 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 0 \\
\end{array}
\]

\[\text{diameter} = K\]

3d hypercube
Meshes

- **1d-mesh**: linear array; \( p \) procs in array
  - diameter = \( p-1 \), max degree = 2
  - ring is linear array w/ \( P \) connected to \( P \)
    diameter = \( \sqrt{P} \),

- **2d-mesh**: \( p \) procs in \( \sqrt{p} \times \sqrt{p} \) array
  - diameter = \( \sqrt{p} \), max degree = 4

**Sum on Linear Array (message passing)**
- assumes \( p = n \) (How to modify for \( p < n \)?)
- initially, \( A(i) \) stored on \( P_i \)
- at end have sum on \( P_p \)
- each proc runs following program

```
sum (A) /* assume p=n*/
local var: id, y
id := get-my-id()
if id = 1
  then y := 0
  else y := receive (y, left = P_{id-1})
  y := y + A(id)
  if id < p
    then send(y, right = P_{id+1})
  else sum := y
end /* sum */
```

Time: \( O(p) = O(n) \) (no better than sequential)
Work: \( O(n) \)
Sum on 2d Mesh

- assume \( p = n \), and \( \sqrt{n} \) is integer
- number proc \( P_{ij} \) \( i \)th row, \( j \)th col \( 1 \leq i \leq \sqrt{n}, 1 \leq j \leq \sqrt{n} \)
- initially \( P_{ij} \) holds \( A((i-1)\sqrt{n} + j) \)

- at end sum is in \( P_{\sqrt{n}, \sqrt{n}} \)

1. in first phase compute sum each row \( i \) and \( \Rightarrow \) put sum in \( P_{i, \sqrt{n}} \) (last col).
   - use alg for linear array sum (\( O(\sqrt{n}) \) time)

2. in second phase compute sum of col \( \sqrt{n} \) and \( \downarrow \) put sum in \( P_{\sqrt{n}, \sqrt{n}} \)
   - use alg for linear array sum (\( O(\sqrt{n}) \) time)

Time \( O(n) \) in total

Common Mesh technique \( \Leftrightarrow \) Systolic Paradigm (ex in text)
- feed input array in synchronous fashion in rows of mesh (col. \( i \) of Array to col. \( i \) of mesh)
  - often feed els in in skewed fashion, e.g.
    - col. \( 1 \) started at timestep 1, col. \( 2 \) at step 2, ...
    - col. \( i \) at timestep \( i \).
Hypercubes

- Diameter of 2^d-hypercube is d, max degree is d.
- Build (d+1)-dimensional hypercube by combining two d-dimensional hypercubes.

add edges between nodes w/ same index.

Sum on Hypercube (synchronous alg)

- Each entry $A(i)$ is initially stored on $p_i$ of an $n = 2^d$ processor (d-dimen) hypercube.
- At end put sum in $P_0$.

$$i := \text{get-my-id}()$$

$$\text{for } l = d-1 \text{ down to } 0 \text{ do}$$

$$\text{if } (0 \leq i \leq 2^l - 1) \text{ then}$$

$$A(i) := A(i) + A(i^{\text{compl}})$$

$$\text{end for}$$

$\text{ex}$

$$d = 3$$

1 iteration 1 ($l = 2$)

$$A(0) := A(0) + A(4) = 1 + 7 = 8$$

$$A(1) := A(1) + A(5) = 2 + 8 = 10$$

$$A(2) := A(2) + A(6) = 4 + 5 = 9$$

$$A(3) := A(3) + A(7) = 3 + 6 = 9$$

Time $d = \log_2 n$ parallel steps

$O(\log_2 n)$ (same as PRAM)

2 iteration 1 ($l = 1$)

$$A(0) := A(0) + A(2) = 8 + 9 = 17$$

$$A(1) := A(1) + A(3) = 10 + 9 = 19$$

3 iteration 1 ($l = 0$)

$$A(0) := A(0) + A(1) = 17 + 19 = 36$$