COMP 633 - Parallel Computing

Lecture 2
August 24, 2017

PRAM (1): The PRAM model and complexity measures
First class summary

- This course is about parallel computing to achieve high(-er) performance on individual problems
  - start with high level PRAM model
    - study algorithms and asymptotic complexity
  - subsequently focus on more practical models from implementation point of view
    - shared memory, distributed memory, distributed computing
      - study hardware organization, programming models, performance prediction and analysis
      - examine various algorithms and case studies
Topics today

- **PRAM model**
  - execution model
  - programming model

- **Work-Time model**
  - programming model
  - complexity metrics
  - Brent’s theorem: translation to PRAM programs

- **Parallel prefix algorithm**
  - derivation
  - applications
PRAM model of parallel computation

- **PRAM** = Parallel Random Access Machine
  - $p$ processors
  - shared memory
  - each processor has a unique identity $1 \leq i \leq p$
  - SIMD operation (**synchronous PRAM**)
    - each processor may be active (✓) or inactive (✗)
    - each instruction executed by all active processors
    - each instruction completes in unit time
PRAM program

• PRAM program
  – sequential program
  – expressions involving processor id \( i \) have a unique value in each processor
    • \( i \) can be used as an array index
      \[ X[i] := i \]
    • conditionals specify active processors
      \[ \text{if } \text{odd}(i) \text{ then} \]
      \[ X[i] := X[i] + X[i+1] \]
      \[ \text{endif} \]
      \[ \text{if } i \leq 2 \text{ then} \]
      \[ X[i] := 1 \]
      \[ \text{else} \]
      \[ X[i] := -1 \]
      \[ \text{endif} \]
Concurrent memory access - Read

- Concurrent reads (CR)
  - all readers of a given location see the same value
    \[ X[i] := y \quad \text{value of } y \text{ read concurrently by all } p \text{ processors} \]
    \[ X[i] := B\left\lceil i/2 \right\rceil \quad \text{some locations in } B \text{ read concurrently by two processors} \]

- Eliminating bounded-degree concurrent reads
  - replace \( X[i] := B\left\lceil i/2 \right\rceil \) with
    \[
    \text{if } \text{odd}(i) \text{ then } \\
    X[i] := B\left\lceil i/2 \right\rceil \\
    \text{endif} \\
    \text{if } \text{even}(i) \text{ then } \\
    X[i] := B\left\lceil i/2 \right\rceil \\
    \text{endif}
    \]

concurrent read is eliminated but number of steps is doubled

Ex. \( p = 6 \)
Concurrent memory access - Write

- Concurrent writes (CW)
  - Stored value depends on write arbitration policy:
    - Arbitrary CW
      - nondeterministic choice among values written
    - Common CW
      - All processors that write a value must write the same value, else error
    - Priority CW
      - value written by processor with lowest processor id
    - Combining Write
      - all values combined using a specified associative operation (e.g. “+”)

- Example (p = 6)

\[
\begin{align*}
  y & := X[i] \\
  B\left[\ceil{i/2}\right] & := X[i]
\end{align*}
\]
Concurrent writes:

- Let $B[1:p]$ be an array of boolean values and define $c = B_1 \lor B_2 \lor \ldots \lor B_p$

  - use $p$ processors and concurrent writes to compute $c$ in a constant number of steps
    a) with combining CW

  b) with a CW policy other than combining CW (which?)
Concurrent memory access

• PRAM variants
  – EREW, CREW, ERCW, CRCW
  – differ in performance, not expressive power
    • EREW < CREW < CRCW
  – loosely reflect difficulty of model implementation

• The following are considered EREW
  – references to
    • processor id \(i\)
    • number of processors \(p\)
    • problem size \(n\)

  – references to local variables
    \texttt{local} \(h; \ h := 2*i + 1; \ X[h] := X[i]\

  – expression evaluation is synchronous, e.g.
    \(X[i] := X[i] + X[i+1]\\
    \text{is EREW}
A PRAM program

- Simple problem: vector addition
  - given V,W vectors of length n
  - compute Z = V + W

- PRAM program
  - constructed to operate with arbitrary
    - problem size n
    - number of processors p
  - work to be performed must explicitly be “scheduled” across processors
  - time complexity with p procs
    - $T_c(n,p) = \ldots$
  - PRAM model?

Input: V[1:n], W[1:n] in shared memory
Output: Z[1:n] in shared memory

local integer h, k
for h := 1 to $\lceil n/p \rceil$ do
  k := $(h-1) \cdot p + i$
  if k ≤ n then
    $Z[k] := V[k] + W[k]$
  endif
endo
Work-Time paradigm

- **W-T parallel programming model**
  - high-level PRAM programming model
    - specifies available parallelism
    - no explicit scheduling of parallelism over processors
  - simplifies algorithm presentation and analysis
  - W-T programs can be mechanically translated to PRAM programs

- **W-T program**
  - sequential program
  - **forall** construct
    - specification of available parallelism
    - number of processors is not a parameter of the model!

WT program for vector addition

```
Input: V[1:n], W[1:n]
Output: Z[1:n]

forall i in 1:n do
  Z[i] := V[i] + W[i]
enddo
```
Programming notation for the W-T framework

- **standard sequential programming notation**
  - **statements**
    - assignment
    - statement composition
    - alternative construct (if ... then ... else ..endif)
    - repetitive construct (for, while)
  - **expressions**
    - arithmetic and logical functions
    - variable reference
    - (recursive) function and procedure invocation

- **forall** statement
  - specifies T may be executed simultaneously for each value of i in D
  - no restriction on T
    - can be a sequence of statements,
    - can invoke (recursive) functions

```plaintext
forall i in D do
  statement T depending on i
enddo
```
W-T complexity metrics

• Work complexity $W(n)$
  – total number of operations performed (as a function of input size $n$)

• Step complexity $S(n)$
  – number of parallel steps required (as a function of input size $n$)
  – assuming unbounded parallelism

• Inductively defined over constructs of W-T programming notation
W-T complexity measures: simple example

\[
\begin{array}{l}
\text{forall } i \text{ in } 2:n-1 \text{ do } \\
\quad \text{R[i] := } \frac{(R[i-1] + R[i] + R[i+1])}{3} \\
\text{enddo}
\end{array}
\]

\[
\begin{array}{l}
\text{for } h := 1 \text{ to } k \text{ do } \\
\quad \text{forall } i \text{ in } 2:n-1 \text{ do } \\
\quad \quad \text{R[i] := } \frac{(R[i-1] + R[i] + R[i+1])}{3} \\
\quad \text{enddo} \\
\text{enddo}
\end{array}
\]
Work and Step Complexity of the forall construct

- How to define work and time complexity of the `forall` construct?

P: 
\[
\text{forall } i \text{ in } D \text{ do } \\
\text{body } T \text{ depending on } i \\
\text{enddo}
\]

- assume we can determine \( W(T_i) \) and \( S(T_i) \) for each \( i \) in \( D \)

- \( W(P) = \)

- \( S(P) = \)
W-T complexity measures: vector summation

• let $n = 2^k$

```plaintext
forall i in 1:n/2 do
    S[i] := S[2i - 1] + S[2i]
enddo
```

```plaintext
for h := 1 to k do
    forall i in 1:n/2^h do
        S[i] := S[2i - 1] + S[2i]
    enddo
enddo
```

$n = 4$, $k = 2$
W-T complexity measures: vector summation

- **Vector summation** (sum - reduction)
  - given $V[1..n]$, $n = 2^k$
  - compute $s = \text{sum}(V[1:n])$
  - optimal sequential time $T_s(n) = \Theta(n)$

- **Complexity**
  - $W(n) =$
  - $S(n) =$

PRAM model needed?

---

- **Input**: $V[1:n]$ vector of integers, $n = 2^k$
- **Output**: $s = \text{sum}(V[1:n])$

**P1**: 

```plaintext
forall i in 1:n do
    B[i] := V[i]
enddo
```

**P2**: 

```plaintext
for h := 1 to k do
    forall i in 1:n/2^h do
    enddo
enddo
```

**P3**: 

```plaintext
s := B[1]
```
Brent’s theorem and $T_c(n,p)$

- Brent’s theorem schedules a W-T program for a $p$-processor PRAM
  - idea
    - simulate each parallel step in W-T program using $p$ processors
    - the work $W_i(n)$ to be performed in step $i$ can be completed using $p$ processors in time
      $$\left\lfloor \frac{W_i(n)}{p} \right\rfloor$$
  - bound concurrent runtime $T_c(n,p)$ of resultant PRAM program
    - by summing over all $S(n)$ steps
      $$T_c(n, p) = \sum_{i=1}^{S(n)} \left\lfloor \frac{W_i(n)}{p} \right\rfloor \leq \sum_{i=1}^{S(n)} \left( \left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \leq \left\lfloor \sum_{i=1}^{S(n)} \frac{W_i(n)}{p} \right\rfloor + S(n) = \left\lfloor \frac{W(n)}{p} \right\rfloor + S(n)$$

$$\left\lfloor \frac{W(n)}{p} \right\rfloor = \left\lfloor \sum_{i=1}^{S(n)} \frac{W_i(n)}{p} \right\rfloor \leq \sum_{i=1}^{S(n)} \left\lfloor \frac{W_i(n)}{p} \right\rfloor = T_c(n, p)$$
Scheduling W-T vector summation algorithm

W-T vector summation algorithm

*Input*: $V[1:n]$ vector of integers, $n = 2^k$

*Output*: $s = \sum(V[1:n])$

**P1**:\texttt{forall i in 1:n do}
\hspace{1em} $B[i] := V[i]$
\hspace{1em} \texttt{enddo}

**P2**:\texttt{for h := 1 to k do}
\hspace{1em} \texttt{forall i in 1:n/2^h do}
\hspace{1em} \texttt{enddo}
\hspace{1em} \texttt{enddo}

**P3**: $s := B[1]$

PRAM vector summation algorithm

*Input*: $V[1:n]$ vector of integers, $n = 2^k$

*Output*: $s = \sum(V[1:n])$

$p > 0$ processor *PRAM*; processor index $i$

local integer $j$, $r$;

**P1**:\texttt{for j := 1 to \left\lceil n/p \right\rceil do}
\hspace{1em} $r := (j-1) \cdot p + i$
\hspace{1em} \texttt{if } $r \leq n$ \texttt{then } $B[r] := V[r]$ \texttt{endif}
\hspace{1em} \texttt{enddo}

**P2**:\texttt{for h := 1 to k do}
\hspace{1em} \texttt{forall i in 1:n/2^h do}
\hspace{1em} \texttt{enddo}
\hspace{1em} \texttt{enddo}

**P3**: \texttt{if i \leq 1 then } $s := B[1]$ \texttt{endif}
Performance of translated W-T program

- Count steps needed to perform the additions
  - Brent’s theorem predicts
    \[ T_c(n, p) = O\left(\left\lceil \frac{n-1}{p} \right\rceil + \log n \right) \]
  - counts for various \( p \)
    \[
    \begin{array}{ll}
    p & T_c(n, p) \\
    p = 1 & (n-1)/p \\
    p > n & \log n \\
    p = 3, n = 2^k, k \text{ even} & \approx \left\lceil (n-1)/p \right\rceil + \frac{1}{2} \log n \\
    \end{array}
    \]

- Upper bound is tight (for this program)
- translation retains EREW model

**PRAM vector summation algorithm**

**Input:** \( V[1:n] \) vector of integers, \( n = 2^k \)

**Output:** \( s = \text{sum}(V[1:n]) \)

\( p > 0 \) processor PRAM; processor index \( i \)

**local integer** \( j, r; \)

**P1:** for \( j := 1 \) to \( \lceil n/p \rceil \) do
  \( r := (j-1)\cdot p + i \)
  if \( r \leq n \) then \( B[r] := V[r] \) endif
enddo

**P2:** for \( h := 1 \) to \( k \) do
  for \( j := 1 \) to \( \lceil (n/2^h)/p \rceil \) do
    \( r := (j-1)\cdot p + i \)
    if \( r \leq n/2^h \) then
    endif
  enddo
enddo

**P3:** if \( i \leq 1 \) then \( s := B[1] \) endif
Parallel prefix-sum

- **Inclusive prefix sum**
  - **Input**
    - Sequence \( X \) of \( n = 2^k \) elements, binary associative operator +
  - **Output**
    - Sequence \( S \) of \( n = 2^k \) elements, with \( S_i = x_1 + ... + x_i \)
  - **Example:**
    - \( X = [1, 4, 3, 5, 6, 7, 0, 1] \)
    - \( S = [1, 5, 8, 13, 19, 26, 26, 27] \)
    - \( T_S(n) = \Theta(n) \)

- **Uses of prefix sum**
  - Efficient parallel implementation of sequential “scan” through consecutive actions
    - **ex:** Given series of bank transactions \( T[1:n] \), with \( T[i] \) positive or negative, and \( T[1] \) the opening deposit > 0
      - Was the account ever overdrawn?
  - Explicit or implicit component of many parallel algorithms
Prefix sum algorithm

- **Recursive solution**
  - $X_i$ stands for $X[i]$ and $X_{ij}$ stands for $X[i] + X[i+1] + \ldots + X[j]$

![Recursive prefix sum diagram]
Parallel prefix sum algorithm – WT model

**Input:** $X[1..n]$ vector of integers

**Output:** $S[1..n]$

```
par_prefix_sum( X[1..n] ) =

var Y[1..n/2], Z[1..n/2], S[1..n];
S[1] := X[1];
if n > 1 then
    forall 1 ≤ i ≤ n/2  do
        Y[i] := X[2i-1] + X[2i]
    enddo
    Z[1..n/2] := par_prefix_sum(Y[1..n/2]);
    forall 2 ≤ i ≤ n  do
        if even(i) then
            S[i] := Z[i/2]
        else
            S[i] := Z[(i-1)/2] + X[i]
        endif
    enddo
endif
return S[1..n]
```
Balanced trees in arrays

• Balanced Tree Ascend / Descend
  – Key idea
    • view input data as balanced binary tree
    • sweep tree up and/or down
  – “Tree” not a data structure but a control structure (e.g., recursion)

• Example
  – vector summation
In-place prefix sum

- $S(n)$
- $W(n)$
- Space
- PRAM model
In-place prefix-sum algorithm – WT model

\begin{align*}
\text{Input: } & X[1..n] \text{ vector of values, } n = 2^k \\
\text{Output: } & S[1..n] \text{ vector of prefix sums} \\
\end{align*}

\[
\text{parallel\_prefix\_sum}( X[1..n] ) = \\
\quad \text{forall } i \text{ in } 1:n \text{ do} \\
\quad \quad S[i] := X[i] \\
\quad \text{enddo} \\
\quad \text{for } h = 1 \text{ to } k \text{ do} \\
\quad \quad \text{forall } i \text{ in } 1:n/2^h \text{ do} \\
\quad \quad \quad S[2^hi] := S[2^hi - 2^{h-1}] + S[2^hi] \\
\quad \quad \text{endo} \\
\quad \text{endo} \\
\quad \text{for } h = k \text{ downto } 1 \\
\quad \quad \text{forall } i \text{ in } 2:n/2^{h-1} \text{ do} \\
\quad \quad \quad \text{if odd}(i) \text{ then} \\
\quad \quad \quad \quad S[2^{h-1}i] := S[2^{h-1}i - 2^{h-1}] + S[2^{h-1}i] \\
\quad \quad \quad \endif \\
\quad \quad \text{endo} \\
\quad \text{endo}
\]
Scan-based primitives

- Scan operations (parallel prefix operations) can be used to implement many useful primitives
  - Suppose we are given SCAN to compute prefix sum of integer sequences
    \[
    \text{seq<int> } \text{SCAN(seq<int>)}
    \]
    - step complexity is $\Theta(\lg n)$
    - work complexity is $\Theta(n)$
    - PRAM model is EREW

- The next three examples have the same complexity as SCAN
COPY (or DISTRIBUTE)

```c
seq<int> COPY(int v, int n) {

  seq<int> V[1:n];
  V[1] = v;
  forall i in 2 : n do
    V[i] := 0;
  enddo
  return SCAN(V);
}
```

v = 5
n = 7
V = 5 0 0 0 0 0 0
Res = 5 5 5 5 5 5 5 5 5
seq<int> ENUMERATE(seq<bool> Flag){

    seq<int> V[1:#Flag];
    forall i in 1 : #Flag do
        V[i] := Flag[i] ? 1 : 0;
    enddo
    return SCAN(V);
}

Flag = T T F T F F F T
V = 1 1 0 1 0 0 0 1
Res = 1 2 2 3 3 3 3 4
PACK

seq<T> PACK(seq<T> A, seq<bool> Flag) {

  seq<T> R[1:#A];
  P := ENUMERATE(Flag);
  forall i in 1 : #Flag do
    if Flag[i] then R[P[i]] := A[i] endif;
  enddo
  return R[1:P[#Flag]];
}

A = ! @ # $ % ^ &
Flag = T T F T F F T T
P = 1 2 2 3 3 3 4
R = ! @ $ &
Radix Sort

Input: \( A[1:n] \) with \( b \)-bit integer elements
Output: \( A[1:n] \) sorted

\[
\begin{align*}
&\text{for } h := 0 \text{ to } b-1 \text{ do} \\
&\quad \text{forall } i \text{ in } 1:n \text{ do} \\
&\quad \quad \text{FL}[i] := (A[i] \text{ bit } h) == 0 \\
&\quad \quad \text{FH}[i] := (A[i] \text{ bit } h) != 0 \\
&\quad \text{enddo} \\
&\quad \text{BL} := \text{PACK}(A, FL) \\
&\quad \text{BH} := \text{PACK}(A, FH) \\
&\quad m := \#BL \\
&\quad \text{forall } i \text{ in } 1:n \text{ do} \\
&\quad \quad A[i] := \text{if } (i \leq m) \text{ then } BL[i] \text{ else } BH[i-m]\text{endif} \\
&\quad \text{enddo} \\
&\text{enddo}
\end{align*}
\]

\[S(n) = \]
\[W(n) = \]
Complexity measures for W-T algorithms

• Asymptotic time complexity measures
  – (optimal) sequential time complexity $T_s(n)$
  – parallel time complexity $T_c(n,p)$

• Speedup
  – definition
    $$SP(n, p) = \frac{T_s(n)}{T_c(n, p)}$$
  – limitation
    $$SP(n, p) = \frac{T_s(n)}{T_c(n, p)} \leq \frac{T_s(n)}{W(n) / p} = \frac{pT_s(n)}{W(n)} = O(p)$$

• Average available parallelism
  – definition
    $$AAP(n) = \frac{W(n)}{S(n)}$$
Objectives in the design of W-T algorithms

- **Goal 1:** construct work efficient algorithms
  - a W-T algorithm is work efficient if \( W(n) = \Theta(T_s(n)) \)

- work-inefficient parallel algorithms have limited appeal on a PRAM with a fixed number of processors \( p \)

\[
\lim_{n \to \infty} SP(n, p) \leq \lim_{n \to \infty} \frac{pT_s(n)}{W(n)} = p \lim_{n \to \infty} \frac{T_s(n)}{W(n)} = 0
\]
Objectives in the design of W-T algorithms

- **Goal 2:** minimize step complexity
  - get optimal speedup using $AAP(n) = \frac{T_s(n)}{S(n)}$ processors

$$SP(n, AAP(n)) = \Theta\left(\frac{T_s(n)}{T_c(n, AAP(n))}\right) = \Omega\left(\frac{T_s(n)}{AAP(n) + S(n)}\right)$$

$$= \Omega\left(\frac{T_s(n)}{S(n) + S(n)}\right) = \Omega(AAP(n))$$

- when $S(n)$ is decreased, $AAP(n)$ is increased
  - with fixed problem size
    - can use more processors to get greater speedup
  - with fixed number of processors
    - reach optimal speedup at smaller problem size
W-T model advantages

- Widely developed body of techniques

- Ignores scheduling, communication and synchronization
  - “easiest” parallel programming

- Source-level complexity metrics
  - Work and step complexity
  - related to running time via Brent’s theorem

- Good place to start
  - many “real-world” algorithms can be derived starting from W-T algorithms