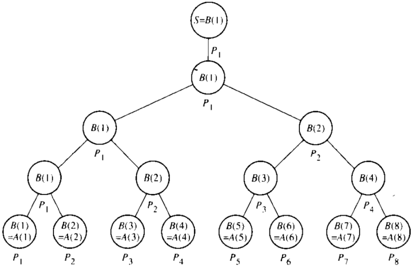
CSc 8530
Parallel Algorithms

Spring 2019

January 31st, 2019

Example: sum on the PRAM model

- Given an array A with $n = 2^k$ values and a PRAM with p processors
- We wish to compute $S = A[1] + A[2] + \ldots A[n]$
- A parallel implementation will run fastest with n processors
- However, not all processors are needed at every iteration



PRAM variations

- PRAM variants differ in how they handle simultaneous access to the same location in shared memory
  - **Exclusive read exclusive write (EREW)**
  - **Concurrent read exclusive write (CREW)**
  - **Concurrent read concurrent write (CRCW)**
- Furthermore, we have three subtypes of CRCW:
  - **Common** CRCW PRAM
    - Allows concurrent writes only when all processors attempt to write *the same value*
  - **Arbitrary** CRCW PRAM
    - Allows an arbitrary processor to succeed
  - **Priority** CRCW PRAM
    - Assumes processors have a priority (based on their ids)
    - The lowest id wins
- EREW, CREW, and, CRCW differ slightly in their computational power
  - i.e., in the space of functions they can theoretically compute

The network model

- A **network** is a graph $G = (V, E)$
  - The nodes V are the processors
  - The edges E are two-way communication links between processors
- There is **no** shared memory
  - Each processor does have local memory
- The model can be either **synchronous** or **asynchronous**
- **send**(X, i) instruction: sends X to processor P_i (and continue executing the next instruction immediately)
- **receive**(Y, j) operation: wait for Y from processor P_j (and suspend execution until data is received)

Linear array – matrix-vector multiplication

We split the computations as follows (for $p = n/2$):

$$
\begin{aligned}
y_1 &= a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + a_{1,4}x_4 + \ldots + a_{1,n-1}x_{n-1} + a_{1,n}x_n \\
y_2 &= a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + a_{2,4}x_4 + \ldots + a_{2,n-1}x_{n-1} + a_{2,n}x_n \\
&\vdots \\
y_n &= a_{n,1}x_1 + a_{n,2}x_2 + a_{n,3}x_3 + a_{n,4}x_4 + \ldots + a_{n,n-1}x_{n-1} + a_{n,n}x_n
\end{aligned}
$$

for processors P_1, P_2, ..., P_p resp.

Linear array – matrix-vector multiplication

We split the computations as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} z_{1,1} \\ z_{2,1} \\ \vdots \\ z_{n,1} \end{bmatrix} + \begin{bmatrix} z_{1,2} \\ z_{2,2} \\ \vdots \\ z_{n,2} \end{bmatrix} + \ldots + \begin{bmatrix} z_{1,p} \\ z_{2,p} \\ \vdots \\ z_{n,p} \end{bmatrix}$$

for processors P_1, P_2, ..., P_p resp.

Linear array – matrix-vector multiplication

- **Computation time:**
  $T_{comp} = O(n^2/p)$
  - Approx $\alpha(n^2/p)$ for some constant α
- However, P_1 has to wait until the $p-1$ partial sums have been transmitted to execute the last instruction
- **Communication time:**
  $T_{comm} = p * comm(n)$
  - $comm(n)$ is the time needed to transmit n numbers between adjacent processors

ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix* $B = A(1:n, (i-1)r+1:ir)$ *of size* $n \times r$, *where* $r = n/p$; *(4) the ith subvector* $w = x((i-1)r+1:ir)$ *of size r.*

Output: *Processor* P_i *computes the vector* $y = A_1x_1 + \cdots + A_ix_i$ *and passes the result to the right. When the algorithm terminates,* P_1 *will hold the product Ax.*

begin
1. *Compute the matrix vector product* $z = Bw$.
2. **if** $i = 1$ **then** *set* $y := 0$
   else receive$(y, left)$
3. *Set* $y := y + z$
4. **send**$(y, right)$
5. **if** $i = 1$ **then receive**$(y, left)$

end

Linear array – matrix-vector multiplication

- $comm(n) \approx \sigma + n\tau$
  - σ: startup time
  - τ: transfer rate
- Total execution time:

$$T = T_{comp} + T_{comm}$$
$$\approx \alpha(n^2/p) + p(\sigma + n\tau)$$

- There is a trade-off between the two terms
- The sum is minimized when $\alpha(n^2/p) = p(\sigma + n\tau)$
  - Such that $p = n\sqrt{\alpha/(\sigma + n\tau)}$

ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix* $B = A(1:n, (i-1)r+1:ir)$ *of size* $n \times r$, *where* $r = n/p$; *(4) the ith subvector* $w = x((i-1)r+1:ir)$ *of size r.*

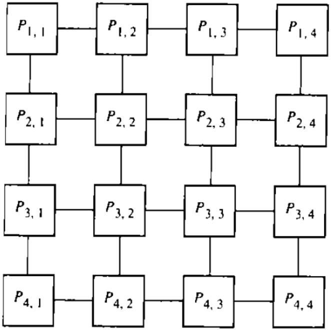
Output: *Processor* P_i *computes the vector* $y = A_1x_1 + \cdots + A_ix_i$ *and passes the result to the right. When the algorithm terminates,* P_1 *will hold the product Ax.*

begin

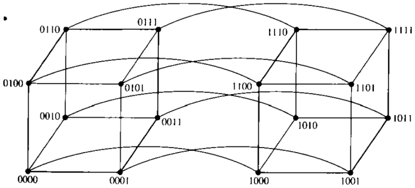
1. *Compute the matrix vector product* $z = Bw$.
2. **if** $i = 1$ **then** *set* $y := 0$
   else receive$(y, left)$
3. *Set* $y := y + z$
4. **send**$(y, right)$
5. **if** $i = 1$ **then receive**$(y, left)$

end

Other topologies



2D mesh



Hypercube

- Both topologies are **sparse**
  - The number of connections $k << p$
- Routing becomes more important, the more complex the network

PRAM justification

- Dags, shared-memory, and network models capture parallel processing at different levels of abstraction
- However, both dags and networks have significant drawbacks:
  - Dags can be hard to analyze
  - Dags require additional scheduling specifics
  - Dags have no formalism for memory management
  - Algorithms for the network model are very hard to analyze
  - The network model is highly dependent on the underlying topology
    - Different topologies may require completely different algorithms
- Thus, we will focus on the synchronous shared-memory model (PRAM)

PRAM justification

- The PRAM model has a number of strengths:
  - Well-developed body of techniques
  - Removes algorithmic details concerning synchronization and communication
  - Captures the allocation of jobs to processors over time
  - Many network algorithms can be mapped directly to the PRAM architecture
  - If needed, synchronization and communication can easily be added to the formalism

Worst-case analysis

- Let Q be a problem that we can solve in $T(n)$ with $P(n)$ processors
- **Parallel cost:** $C(n) = T(n)P(n)$
- The parallel algorithm can be converted to a sequential algorithm that runs in $O(C(n))$
- More generally, we can simulate a single step in $O(P(n)/p)$ sub-steps:
  - In sub-step 1: simulate processors $[1, p]$
  - In sub-step 2: simulate processors $[p + 1, 2p]$, etc.
- We can simulate the entire process in $O(T(n)P(n)/p)$

Worst-case analysis

- The following four measures are asymptotically equivalent:
  1. $P(n)$ processors and $T(n)$ time
  2. $C(n) = P(n)T(n)$ cost and $T(n)$ time
  3. $O(T(n)P(n)/p)$ for $p \leq P(n)$ processors
  4. $O(\frac{C(n)}{p} + T(n))$ time for any p
- PRAM example: sum of n elements
  1. n processors and $O(\log(n))$ time
  2. $O(n\log(n))$ cost and $O(\log(n))$ time
  3. $O(\frac{n\log(n)}{p})$, for $p \leq P(n)$
  4. $O(\frac{n\log(n)}{p} + \log(n))$, for all p
- Note that the $O(n+m)$ notation really means $O(\max(n,m))$
  - Depending on the input, one term may be bigger than the other

Work-time (WT) paradigm

- The **work-time (WT) paradigm** provides a two-level description of parallel algorithms
  - Upper level suppresses specific details
  - Lower level follows a general **scheduling principle**
- **Upper Level:** Describe the algorithm in terms of a sequence of time units
  - Each time unit may include any number of concurrent operations
- **Work:** total number of operations
- For convenience, at this level we can use a **pardo** statement
  - **for** $l \leq i \leq u$ **pardo** $\{$statement(s)$\}$
  - All the statements, for all valid indices, are executed concurrently

Memory-explicit vs. WT pseudocode

ALGORITHM 1.2
(Sum on the PRAM Model)
Input: *An array A of order $n = 2^k$ stored in the shared memory of a PRAM with n processors. The initialized local variables are n and the processor number i.*
Output: *The sum of the entries of A stored in the shared location S. The array A holds its initial value.*

```
begin
  1. global read(A(i), a)
  2. global write(a, B(i))
  3. for h = 1 to log n do
        if (i ≤ n/2^h) then
          begin
               global read(B(2i - 1), x)
               global read(B(2i), y)
               Set z: = x + y
               global write(z, B(i))
          end
  4. if i = 1 then global write(z, S)
end
```

Memory-explicit pseudocode

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$

```
begin
  1. for 1 ≤ i ≤ n pardo
        Set B(i): = A(i)
  2. for h = 1 to log n do
        for 1 ≤ i ≤ n/2^h pardo
            Set B(i): = B(2i - 1) + B(2i)
  3. Set S: = B(1)
end
```

WT pseudocode

WT pseudocode

- The WT pseudocode makes no mention of number of processors or allocation
- Stated only in terms of time units
- Each time unit may contain any number of concurrent operations
- **Time units**:
- **Breakdown:**

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
  1. **for** $1 \le i \le n$ **pardo**
    Set $B(i) := A(i)$
  2. **for** $h = 1$ **to** $\log n$ **do**
    for $1 \le i \le n/2^h$ **pardo**
      Set $B(i) := B(2i - 1) + B(2i)$
  3. *Set* $S := B(1)$
end

WT pseudocode

WT pseudocode

- The WT pseudocode makes no mention of number of processors or allocation
- Stated only in terms of time units
- Each time unit may contain any number of concurrent operations
- **Time units**:
  - $\log(n) + 2$
- **Breakdown:**

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
  1. **for** $1 \leq i \leq n$ **pardo**
    Set $B(i) := A(i)$
  2. **for** $h = 1$ **to** $\log n$ **do**
    for $1 \leq i \leq n/2^h$ **pardo**
      Set $B(i) := B(2i - 1) + B(2i)$
  3. *Set* $S := B(1)$
end

WT pseudocode

WT pseudocode

- The WT pseudocode makes no mention of number of processors or allocation
- Stated only in terms of time units
- Each time unit may contain any number of concurrent operations
- **Time units**:
  - $\log(n) + 2$
- **Breakdown:**
  - Step 1: n operations
  - Step j: $n/2^{j-1}$ operations
  - Last step: one operation

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
1. **for** $1 \leq i \leq n$ **pardo**
   Set $B(i) := A(i)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \leq i \leq n/2^h$ **pardo**
   Set $B(i) := B(2i - 1) + B(2i)$
3. *Set* $S := B(1)$

end

WT pseudocode

WT pseudocode

- **Total work:**
- **Running time:**

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
  1. **for** $1 \le i \le n$ **pardo**
      Set $B(i) := A(i)$
  2. **for** $h = 1$ **to** $\log n$ **do**
      for $1 \le i \le n/2^h$ **pardo**
        Set $B(i) := B(2i - 1) + B(2i)$
  3. *Set* $S := B(1)$
end

WT pseudocode

WT pseudocode

- **Total work:**
  - $W(n) = n + \sum_{j=1}^{\log(n)} (n/2^j) + 1$
- **Running time:**

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin

```
  1. for 1 ≤ i ≤ n pardo
         Set B(i) := A(i)
  2. for h = 1 to log n do
         for 1 ≤ i ≤ n/2^h pardo
             Set B(i) := B(2i − 1) + B(2i)
  3. Set S := B(1)
```

end

WT pseudocode

WT pseudocode

- **Total work:**
  - $W(n) = n + \sum_{j=1}^{\log(n)} (n/2^j) + 1$
  - $W(n) = O(n)$
- **Running time:**

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
1. **for** $1 \le i \le n$ **pardo**
  Set $B(i) := A(i)$
2. **for** $h = 1$ **to** $\log n$ **do**
  for $1 \le i \le n/2^h$ **pardo**
    Set $B(i) := B(2i - 1) + B(2i)$
3. *Set* $S := B(1)$
end

WT pseudocode

WT pseudocode

- **Total work:**
  - $W(n) = n + \sum_{j=1}^{\log(n)} (n/2^j) + 1$
  - $W(n) = O(n)$
- **Running time:**
  - $T(n) = O(\log(n))$

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
1. **for** $1 \leq i \leq n$ **pardo**
Set $B(i) := A(i)$
2. **for** $h = 1$ **to** $\log n$ **do**
for $1 \leq i \leq n/2^h$ **pardo**
Set $B(i) := B(2i - 1) + B(2i)$
3. Set $S := B(1)$
end

WT pseudocode

WT pseudocode

- **Total work:**
  - $W(n) = n + \sum_{j=1}^{\log(n)} (n/2^j) + 1$
  - $W(n) = O(n)$
- **Running time:**
  - $T(n) = O(\log(n))$
- Parallelization can reduce the running time, but not the total work
  - We do more operations at once, but not fewer operations in total

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$

begin
1. **for** $1 \le i \le n$ **pardo**
   Set $B(i) := A(i)$
2. **for** $h = 1$ **to** $\log n$ **do**
   for $1 \le i \le n/2^h$ **pardo**
   Set $B(i) := B(2i-1) + B(2i)$
3. *Set* $S := B(1)$

end

WT pseudocode

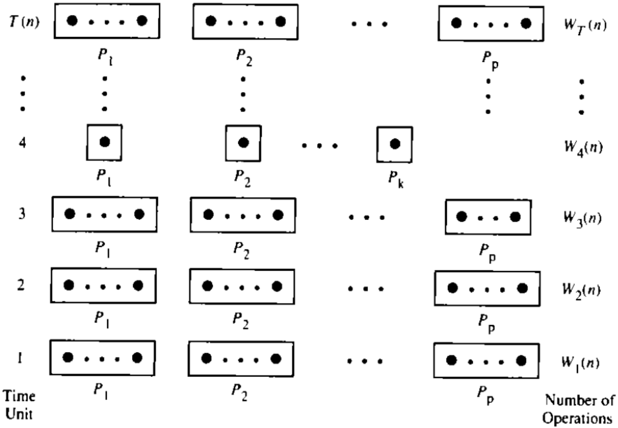
Work-time (WT) paradigm

- **Lower Level:** Suppose an upper-level description yields an algorithm with $T(n)$ running time and $W(n)$ work
  - We can almost always adapt this algorithm to run in $\lfloor \frac{W(n)}{p} + T(n) \rfloor$ parallel steps
- **WT Scheduling Principle:** let $W_i(n)$ be the number of operations in time unit i, $1 \leq i \leq T(n)$
  - Simulate each $W_i(n)$ in $\leq \lceil \frac{W_i(n)}{p} \rceil$
- The p-processor PRAM takes

$$\leq \sum_i \lfloor \frac{W(n)}{p} \rfloor \leq \sum_i (\lfloor \frac{W(n)}{p} \rfloor + 1) \leq \lfloor \frac{W(n)}{p} \rfloor + T(n)$$

  parallel steps
  - We assume that we calculate $W_i(n)$ for each i
  - We also assume each processor knows what instruction it needs to execute

WT Scheduling Principle



WT vs. lower-level pseudocode

ALGORITHM 1.7
(Sum)
Input: $n = 2^k$ numbers stored in an array A.
Output: The sum $S = \sum_{i=1}^{n} A(i)$
begin
  1. **for** $1 \le i \le n$ **pardo**
    Set $B(i) := A(i)$
  2. **for** $h = 1$ **to** $\log n$ **do**
    for $1 \le i \le n/2^h$ **pardo**
      Set $B(i) := B(2i - 1) + B(2i)$
  3. *Set* $S := B(1)$
end

WT pseudocode

ALGORITHM 1.8
(Sum Algorithm for Processor P_s)
Input: *An array A of size $n = 2^k$ stored in the shared memory. The initialized local variables are (1) the order n; (2) the number p of processors, where $p = 2^q \le n$, and (3) the processor number s.*
Output: *The sum of the elements of A stored in the shared variable S. The array A retains its original value.*
begin
  1. **for** $j = 1$ **to** $l \left(= \frac{n}{p}\right)$ **do**
    Set $B(l(s-1)+j) := A(l(s-1)+j)$
  2. **for** $h = 1$ **to** $\log n$ **do**
    2.1. **if** $(k - h - q \ge 0)$ **then**
      for $j = 2^{k-h-q}(s-1)+1$ **to** $2^{k-h-q}s$ **do**
        Set $B(j) := B(2j-1) + B(2j)$
    2.2. **else** {**if** $(s \le 2^{k-h})$ **then**
        Set $B(s) := B(2s-1) + B(2s)$}
  3. **if** $(s = 1)$ **then** *set* $S := B(1)$
end

Lower-level pseudocode

Work vs. cost

- If a parallel algorithm runs in $T(n)$ with a total of $W(n)$ operations
  - Can be simulated in $O(\frac{W(n)}{p} + T(n))$ on a p-processor PRAM
  - The cost is $C_p(n) = T_p(n)p = O(W(n) + T(n)p)$
- Work and cost coincide asymptotically for $p = O(\frac{W(n)}{T(n)})$
- Otherwise they differ:
  - Work is independent of the number of processors
  - Cost is measured relative to the number of available processors
  - Cost $\geq$ Work due to inefficient processor utilization
- For computing the sum of n numbers:
  - Work: $O(n)$, running time: $O(\log(n))$
  - Cost: $C_p(n) = O(n + p\log(n))$
  - With n processors, the cost is $O(n\log(n))$, not $O(n)$ (Why?)

Work vs. cost

- If a parallel algorithm runs in $T(n)$ with a total of $W(n)$ operations
  - Can be simulated in $O(\frac{W(n)}{p} + T(n))$ on a p-processor PRAM
  - The cost is $C_p(n) = T_p(n)p = O(W(n) + T(n)p)$
- Work and cost coincide asymptotically for $p = O(\frac{W(n)}{T(n)})$
- Otherwise they differ:
  - Work is independent of the number of processors
  - Cost is measured relative to the number of available processors
  - Cost $\geq$ Work due to inefficient processor utilization
- For computing the sum of n numbers:
  - Work: $O(n)$, running time: $O(\log(n))$
  - Cost: $C_p(n) = O(n + p\log(n))$
  - With n processors, the cost is $O(n\log(n))$, not $O(n)$ (Why?)
  - We cannot use all the processors at all time steps, so the cost is higher than the total work