CSc 8530 Parallel Algorithms

Spring 2019

January 22nd, 2019

Divide and conquer

- There are many strategies for designing algorithms
- The most basic is **incremental**
  - We iteratively add one element to our partial solution at a time until we have a complete solution
  - Example: insertion sort
- **Divide and conquer** is another common approach
  - **Divide** the problem into smaller instances of the same problem
  - **Conquer** the subproblems by solving them *recursively*
    - **Base case:** if a subproblem is small enough, solve it directly
  - **Combine** the solutions to the subproblems to solve the original problem
  - Example: merge sort

Greedy algorithms

- Algorithms for optimizing a value (e.g., the minimum cost of a spanning tree) typically go through a sequence of steps, with a set of choices at each step
- An algorithm is **greedy** if, when faced with a set of possible actions, it always picks the one that looks best at the moment
    - It doesn't factor in how earlier choices influence later ones
- We make a *locally optimal choice* in the hope of getting a *globally optimal solution*

Dynamic programming

- **Dynamic programming** (DP) is a powerful optimization technique which breaks a problem into subproblems
  - Similar to divide-and-conquer, but DP caches intermediate results
    - Avoids solving the same subproblem twice
  - Similar to greedy algorithms, but applies to problems where we have to factor in the subsequent cost of an action
    - In the greedy case, we only care about the local, immediate cost
- **Note:** the term "programming" refers to scheduling, not code
- As in the phrases: "Today's reception has been programmed for 5:00pm" or "Get with the program"

Pointer arithmetic

- Pointers can be manipulated like other integers
  - Addition, subtraction, multiplication, etc.
- Pointers often point to the start of a data structure (e.g., an array)
- We usually add a constant to a pointer to access different parts of that structure
- Example: `p = p + 1` accesses the adjacent memory location
- **A point of caution:** Trying to access memory locations outside of your program's valid area will result in a segmentation fault

```
// Variable of type int
int x = 5;

// pnt points to x. & is called a
// derefencing (or address) operator
int *pnt = &x;

// y has the same value as x (5)
// The * operator access the value
// that the pointer points to
int y = *pnt;

// Pointer z points to x
int *z = pnt;

// How are x, y, and z affected?
*pnt = 10;
```

Dynamic memory allocation

```
// Static memory allocation
int array[10];

// Dynamic memory allocation
// with error checking
int *array = malloc(size*sizeof(int));
if (array == NULL) {
  fprintf(stderr, "malloc failed\n");
  return(-1);
}

// ... use the array

// Deallocate the memory
free(array);
```

Parallel speedup

- Let P be a computational problem with inputs of size n
- We denote the best-possible sequential (i.e., classic) complexity of P as $T^*(n)$
- Let A be a parallel algorithm that solves P in time $T_p(n)$ using p processors
- Then, the **speedup** achieved by A is:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- By construction, $S_p(n) \leq p$
- We would like $S_p(n) \approx p$
  - i.e., each processor should do around $1/p$ of the work of a single one
- In practice, inefficiencies in concurrency, synchronization, communication, etc. reduce the actual speedup

Parallel efficiency

- The **efficiency** of a parallel algorithm A is given by:

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

- $T_1(n)$ is the running time of the parallel algorithm with a single processor
    - Not necessarily equal to $T^*(n)$
- Efficiency measures how much bang for our buck we get per processor
- Ideally, $E_p(n) \approx 1$
- Again, inefficiencies reduce this value in practice

Computational models

- The **random-access memory** (RAM) model is the standard for sequential algorithms
  - Assumes a single processor and that memory can be accessed in constant time
- No parallel model has the same level of consensus
- Sucha a model faces two conflicting requirements:
  - **Simplicity:** easy to analyze and hardware-independent
  - **Implementability:** pseudocode should easily translate to implementable algorithms
- Here, we will introduce three simple parallel models:
  - Directed acyclic graphs (dags)
  - The shared-memory model
  - The network model
- In the bulk of this course, we will use the shared-memory model for analysis

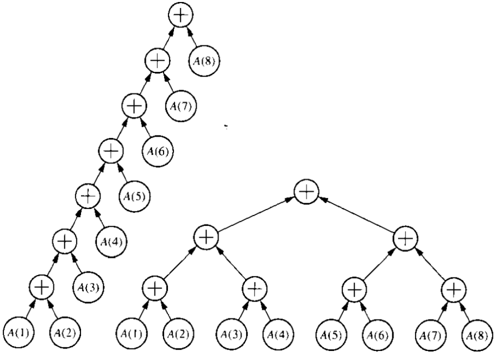
Directed acyclic graphs

- Directed acyclic graphs (dags) are a special case of directed graphs
    - Have **no** directed cycles
        - i.e., it is not possible to return to a node (go backwards)
    - Suitable for modeling **causal processes** and **precedence relations**
- **Some terminology:**
    - *in-degree:* number of incoming edges to a node
    - *out-degree:* number of outgoing edges
    - node u is the *parent* (*ancestor*) of v iff there is an edge (path) from u to v
    - Conversely, v is called the child (descendant) of u

Dags for parallel processing

- We can represent computations using dags
  - Nodes with zero in-degree are inputs (also called **leafs**)
  - Nodes with zero out-degree are outputs (also called **roots** or **sinks**)
  - For simplicity, here we assume all internal vertices have in-degree ≤ 2
- Each node represents an $O(1)$ (constant-time) operation
- This model is best-suited for numerical computations
- For simplicity, we will assume no loops (what the book strangely calls branching)
  - We can always unroll a loop by duplicating it the appropriate number of times
- Node order represents **precendence**
  - What operations must come before and after

Parallel sums



- Two dags for computing the sum S of the $n = 2^k$ elements of an array A
- Note how the depths (max distance from leafs to root) differ significantly between the two choices: $O(n)$ vs $O(\log(n))$

Dag model

- In the dag model, we assume that:
  - *Every processor can access the data computed by any other processor without incurring additional cost*
- A particular implementation is defined by **scheduling** each node for execution on a processor
- Given p processors, we associate a pair (j_i, t_i) with each internal node i:
  - j_i is the processor used for node i
  - t_i is the time at which we process node i
- The following two conditions must hold:
  - If $t_i = t_k$, for some $i \neq k$, then $j_i \neq j_k$
    - Each processor can only process one node at a time
  - If (i, k) is an edge, then $t_k \geq t_i + 1$
    - Node i has to be processed before node k

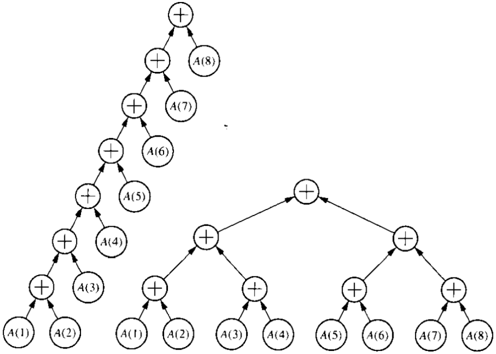
Dag model

- Input nodes have $t_i = 0$ and no processor is allocated to them
- The sequence $\{(j_i, t_i) \mid i \in N\}$ is an execution **schedule**
    - With p processors
    - N is the number of nodes in the dag
- The time to execute a particular schedule is $\max_{i \in N} t_i$
- The **parallel complexity** is

$$T_p(n) = \min\{\max_{i \in N} t_i\}$$

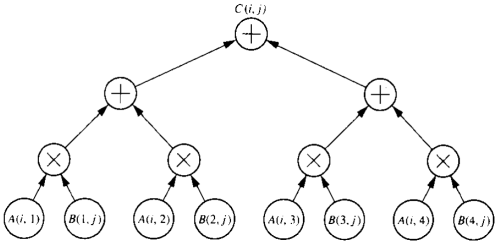
- The minimum is taken over all possible schedules with p processors
- The depth of the dag is a lower bound on $T_p(n)$, for any p

Parallel sums revisited



- The best schedule for the leftmost algorithm is $O(n)$, regardless of p
- The best schedule for the rightmost one is $O(\log(n))$ with $n/2$ processors

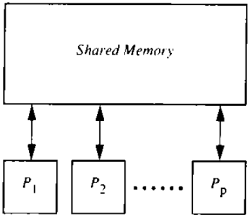
Another example: matrix multiplication



- Let A and B be two $n \times n$ matrices
- The standard algorithm for $C = AB$ computes the terms $C(i,j) = \sum_{l=1}^{n} A(i,l)B(l,j)$
- A parallel algorithm with n^3 processors can compute C in $O(\log(n))$

The shared-memory model

- A natural extension of the sequential RAM model
- Many processors have access to a single, **shared memory** unit (also called **global memory**)
- Each processor also has its own **local memory**
- Processors communicate by exchanging data through the shared memory
- Each processor is indexed by a unique id



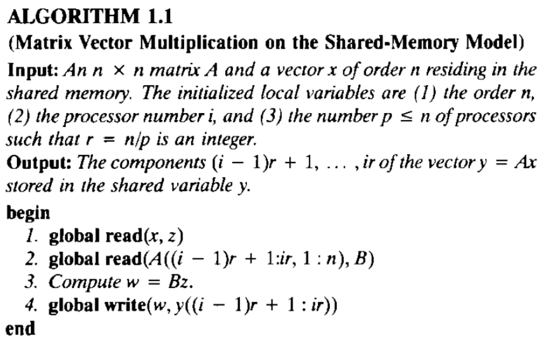
The shared-memory model

- In **synchronous** mode, all processors operate in lock-step, under a common clock
- In **asynchronous** mode, each has a independent clock
- Synchronous mode is called the **parallel random-access machine (PRAM)** model
  - It it the model we will primarily study in this class
- Asynchronous mode requires additional checks to make sure the data is up-to-date when accessed
- Both models are **multiple instruction multiple data (MIMD)**
- The amount of **communication** is given by the size of data transferred via the shared memory
- A **global read** (X, Y) moves the variable X into the local memory Y
- A **global write** (U, V) does the opposite

Example: matrix-vector multiplication

- Let A be an $n \times n$ matrix and let x be an n-dimensional vector
- For simplicity, assume we have $p \leq n$ processors, such that $r = n/p$ is an integer
- We also assume *asynchronous* operation
- We partition $A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix}$
- Each block A_i is of size $r \times n$
- We can compute $y = Ax$ as follows:
  - Each processor P_i reads A_i and x
  - Computes $z = A_i x$
  - Writes z (r values) to the appropriate location in the shared memory

Example: matrix-vector multiplication – pseudocode



ALGORITHM 1.1

(Matrix Vector Multiplication on the Shared-Memory Model)

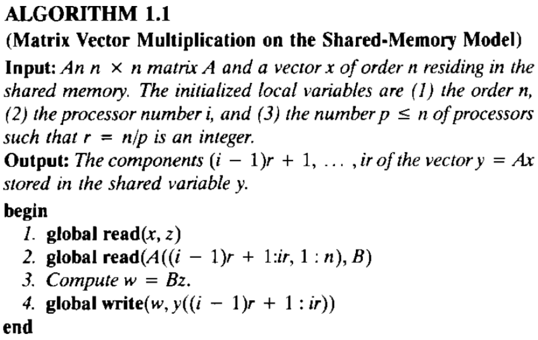
Input: *An $n \times n$ matrix A and a vector x of order n residing in the shared memory. The initialized local variables are (1) the order n, (2) the processor number i, and (3) the number $p \le n$ of processors such that $r = n/p$ is an integer.*

Output: *The components $(i-1)r+1, \ldots, ir$ of the vector $y = Ax$ stored in the shared variable y.*

```
begin
  1. global read($x, z$)
  2. global read($A((i-1)r+1:ir, 1:n), B$)
  3. Compute $w = Bz$.
  4. global write($w, y((i-1)r+1:ir)$)
end
```

- The above algorithm is specified for a single processor
  - We execute it in parallel on each available one
- The processors can operate asynchronously, because they do not write to overlapping variables or memory locations

Example: matrix-vector multiplication – pseudocode



ALGORITHM 1.1

(Matrix Vector Multiplication on the Shared-Memory Model)

Input: *An $n \times n$ matrix A and a vector x of order n residing in the shared memory. The initialized local variables are (1) the order n, (2) the processor number i, and (3) the number $p \leq n$ of processors such that $r = n/p$ is an integer.*

Output: *The components $(i-1)r+1, \ldots, ir$ of the vector $y = Ax$ stored in the shared variable y.*

```
begin
   1. global read($x$, $z$)
   2. global read($A((i-1)r+1:ir, 1:n)$, $B$)
   3. Compute $w = Bz$.
   4. global write($w$, $y((i-1)r+1:ir)$)
end
```

- Analysis:
  - Steps 1 and 2 transfer $O(n^2/p)$ values from the shared memory into each processor
  - Step 3 requires $O(n^2/p)$ arithmetic operations
  - Step 4 stores n/p numbers from local to shared memory