CSc 8530
Parallel Algorithms

Spring 2019

January 15th, 2019

Dramatis personae

- **Instructor:** Rolando Estrada
  - Email: `restrada1@gsu.edu`
  - Office: 1 Park Place, Suite 634
  - Office hours: Tues. 11:30am – 12:30pm or by appointment
- **Grader:** Saeid Motevalialamoti
  - Email: smotevalialamoti1@student.gsu.edu
  - Office: 1 Park Place, Suite 633
  - Office hours: TR 11:00am – 12:00pm or by appointment

Course prerequisites

- CSc 4520/6520 – Design & Analysis of Algorithms
- CSc 4310/6310 – Parallel & Distributed Computing
- Equivalent senior-level course

Conceptual prerequisites

- Standard algorithm analysis
  - Big-O notation, complexity classes, asymptotic bounds, etc.
  - Design techniques, e.g., divide and conquer, greedy algorithms, etc.
- Basic graph theory
  - Graph algorithms: BFS, DFS, Dijkstra, Kruskal, etc.
  - Graph properties: sparse vs. dense, subtypes (regular, small-world, etc.), explicit vs.implicit representations, etc.
- Some programming experience
  - Preferably C/C++
  - Hardware-level a plus

Required textbooks

- *An Introduction to Parallel Algorithms, 1st edition* by Joseph JáJá, Addison-Wesley, 1992
- *Programming Massively Parallel Processors: A Hands-on Approach*, **3rd edition** by David B. Kirk and Wen-mei W. Hwu, Morgan Kaufmann, 2016
- Both books are available on Amazon and similar retailers

Course content (general overview)

- Asymptotic analysis of parallel algorithms
- Parallel data handling and message passing
- Parallel algorithm design techniques (e.g., divide and conquer)
- Parallel graph algorithms
- CUDA/GPU programming

Course Grade

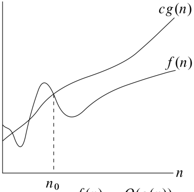
- **Rubric**
  - Homework 20%
  - Midterm 20%
  - Paper presentation 15%
  - Final project 45%
- The lowest homework grade will be dropped.
- Grades on group projects will be assigned individually
- Letter grades will be assigned relative to class performance.

Policies

- Late/missing assignments will not be graded
- Complete academic honesty is expected
  - No cheating
  - No plagiarism
- Cell phones must be off/silent during class
  - No texting, social media, etc.

O-notation

- $O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
- $g(n)$ is an *asymptotic upper bound* for $f(n)$
- If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$
  - **Abuse of notation for convenience**
  - Similarly for the other notations

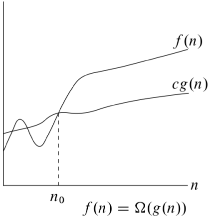


$f(n) = O(g(n))$

(b)

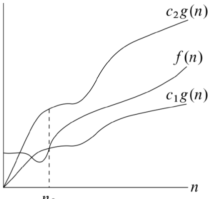
Ω-notation

- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \leq n_0\}$
- $g(n)$ is an *asymptotic lower bound* for $f(n)$



Θ-notation

- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \le n_0\}$
- $g(n)$ is an *asymptotically tight bound* for $f(n)$
- **Theorem:** $f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - Means $2n^2 + f(n)$, with $f(n) \in \Theta(n)$

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - Means $2n^2 + f(n)$, with $f(n) \in \Theta(n)$
  - Here, $f(n) = 3n + 1$

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - Means $2n^2 + f(n)$, with $f(n) \in \Theta(n)$
  - Here, $f(n) = 3n + 1$
  - But, $f(n) = 4n + 5$ or $f(n) = 10000n + 1000$, etc. are all equivalent from an asymptotic perspective

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - Means $2n^2 + f(n)$, with $f(n) \in \Theta(n)$
  - Here, $f(n) = 3n + 1$
  - But, $f(n) = 4n + 5$ or $f(n) = 10000n + 1000$, etc. are all equivalent from an asymptotic perspective
- **On left-hand side:** there is a way to choose the corresponding anonymous functions on the *right-hand side* to make the equation valid
- Interpret $2n^2 + \Theta(n) = \Theta(n^2)$

Asymptotic notation in equations

- **On right-hand side:** $O(g(n))$ stands for some *anonymous function* in the set $O(g(n))$
- E.g.: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - Means $2n^2 + f(n)$, with $f(n) \in \Theta(n)$
  - Here, $f(n) = 3n + 1$
  - But, $f(n) = 4n + 5$ or $f(n) = 10000n + 1000$, etc. are all equivalent from an asymptotic perspective
- **On left-hand side:** there is a way to choose the corresponding anonymous functions on the *right-hand side* to make the equation valid
- Interpret $2n^2 + \Theta(n) = \Theta(n^2)$
  - For all functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$, such that $2n^2 + f(n) = g(n)$

Important classes of algorithms

- Most algorithms you'll encounter in practice are either:
  - **Constant time:** $\Theta(1)$
    - Running time is independent of input size
  - **Logarithmic time:** $\Theta(\log(n))$
    - Running time is proportional to the *number of bits* needed to encode the input
  - **Linear time:** $\Theta(n)$
    - Running time is proportional to the input size
  - **Log-linear time:** $\Theta(n \log(n))$
    - How many times we execute an $\Theta(n)$ operation depends on the input size's number of bits
  - **Polynomial time:** $\Theta(n^p)$
    - Common cases: $\Theta(n)$, $\Theta(n^2)$ (quadratic), $\Theta(n^3)$ (cubic)
    - Running time is proportional to a number of subsets (e.g., pairs for quadratic, triples for cubic, etc.)
  - **Exponential time:** $\Theta(2^{n^p})$
    - Common case: $\Theta(2^n)$
    - Running time *doubles* every time the input size grows by one
    - Practical only for small inputs

Graph definition

- A graph $G = (V, E)$ is defined by two sets:
  - A set of n vertices V (also called nodes)
  - A set of m edges E (also called links)
- All the elements in both V and E are unique (i.e., no repeated values)
- Every edge $e = (u, v) \in E$ is a tuple (i.e., two *ordered* values), such that $u, v \in V$
  - In other words, each edge is defined by its starting and ending vertices
- In general, $m = O(n^2)$ (why?)

Graph terminology

- A graph is **undirected**, if and only if

$$e = (u, v) \in E \Rightarrow (v, u) \in E$$

  - In other words, in an undirected graph, if it is possible to go from u to v, then it is also possible to go from v to u.
- A graph is **directed** if the above condition does **not** hold
- The vertex v is a **neighbor** of the vertex u if and only if $(u, v) \in E$.
  - For an undirected graph: if v is a neighbor of u, then u is also a neighbor of v.
- A **self-loop** is an edge where the starting and ending vertices are the same, i.e., $e = (v, v)$
- A **path** is a sequence of vertices $(v_1, v_2, ..., v_k)$, such that v_{i+1} is a neighbor of v_i.

Graph representations

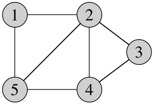
- There are two standard ways of representing graphs in a computer. For both, we assume that the vertices are indexed $1, ..., n$ in an arbitrary manner
  - **Adjacency list:** Each vertex $v \in V$ has an associated array Adj_v that lists all of its neighbors (randomly or ordered by index)
  - **Adjacency matrix:** We define an *adjacency matrix* A, such that

$$a_{i,j} = \begin{cases} 1 & \text{if}(i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

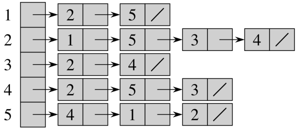
Graph representations

- Adjacency lists are usually better for **sparse** graphs (graphs where the number of edges is similar to the number of vertices, $m = \Theta(n)$)
- Adjacency matrices are sometimes more convenient for **dense** graphs, where $m = \Theta(n^2)$.
- More generally, though, some operations are easier on one representation than another. For example:
  - Determining if v is a neighbor of u takes $\Theta(1)$ for adjacency matrices, but $\Theta(n)$ for adjacency lists
  - Conversely, finding the number of neighbors of u takes $\Theta(1)$ for adjacency lists, but $\Theta(n)$ for adjacency matrices
- **The best representation will depend on your problem**

Undirected graph



(a)

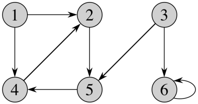


(b)

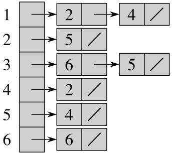
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Directed graph



(a)



(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)