PARALLEL PROCESSING SYSTEMS

Chapter 3: Parallel Algorithm Complexity

Introduction

- Algorithms can be analyzed in two ways
  - precise
    - we count the number of operations performed in the worst or average case
      - (e.g., arithmetic, memory access, data transfer)
    - use these counts as indicators of algorithm complexity
    - is quite tedious and at sometimes impossible to perform.
  - approximate.
    - We resort to various approximate analysis methods
    - keeping in mind the error margin of the method applied
      - if such an approximate analysis indicates that Algorithm A is 1.2 times slower than Algorithm B
      - we may not be able to conclude with certainty that Algorithm B is better for the task at hand.

ASYMPTOTIC COMPLEXITY

- Suppose that
  - a parallel sorting algorithm requires $(\log_2 n)^2$ compare–exchange steps
  - another one $(\log_2 n)^2/ 2 + 2 \log_2 n$ steps
  - a third one $500 \log_2 n$ steps
- Ignoring lower-order terms and multiplicative constants
  - we may say that
    - the first two algorithms take on the order of $\log^2 n$ steps
    - the third one takes on the order of $\log n$ steps.
- The logic behind ignoring these details
  - when n becomes very large
    - eventually $\log n$ will exceed any constant value.
  - for such large values of n and any values of the constants c and c'
    - an algorithm with $c \log n$ is asymptotically better than an algorithm with $c' \log^2 n$

ASYMPTOTIC COMPLEXITY

- n must indeed be very large for log n to overshadow the constant 500
- In practice
  - we do not totally ignore the constant factors
  - We take a two-step approach
    - First, we determine which algorithm is likely to be better for large problem sizes
      - through asymptotic analysis
    - If we have reason to doubt this conclusion
      - we resort to an exact analysis to determine the constant factors involved

ASYMPTOTIC COMPLEXITY

- Some notations
  - Given two functions f(n) and g(n) we define the relationships
    - "O" (big-oh)

$$f(n) = O(g(n)) \text{ if } \exists c, n_0 \text{ such that } \forall n > n_0 \text{ we have } f(n) < c\, g(n)$$

    - "Ω" (big-omega)

$$f(n) = \Omega(g(n)) \text{ if } \exists c, n_0 \text{ such that } \forall n > n_0 \text{ we have } f(n) > c\, g(n)$$

    - "Θ" (theta)

$$f(n) = \Theta(g(n)) \text{ if } \exists c, c', n_0 \text{ such that } \forall n > n_0 \text{ we have } c\, g(n) < f(n) < c'\, g(n)$$

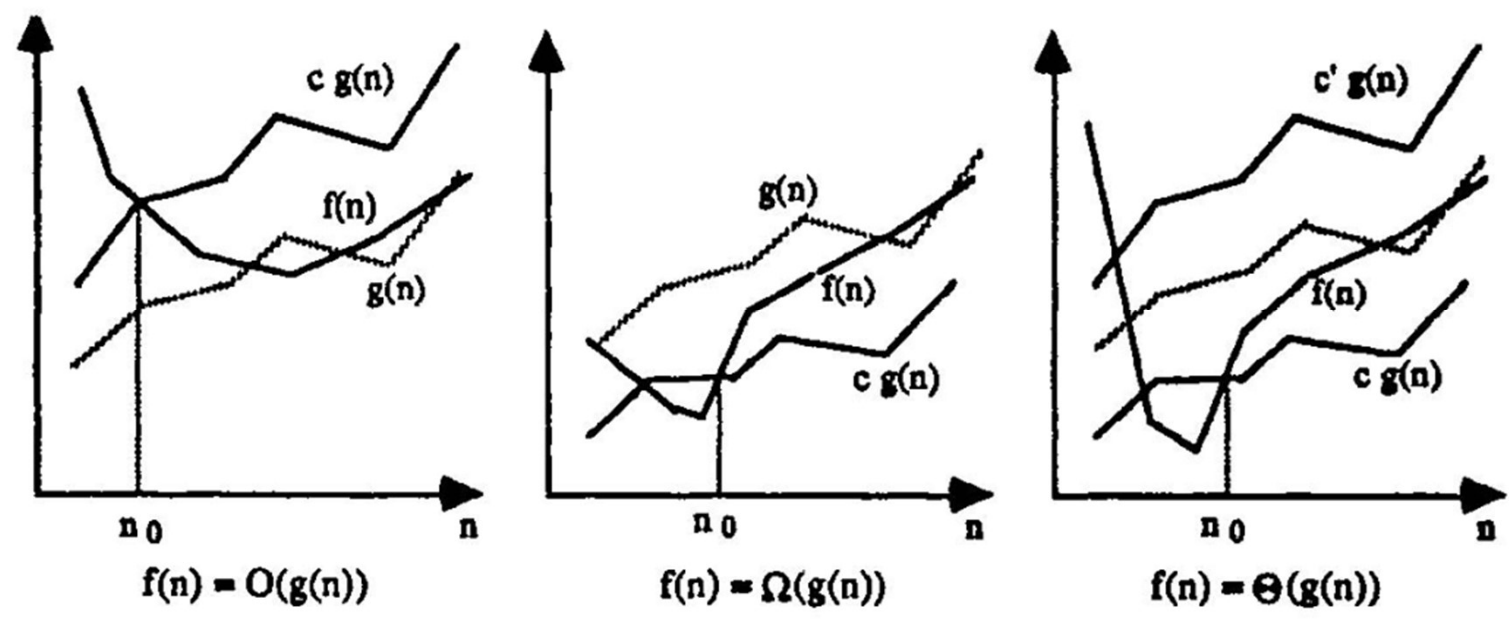
$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

ASYMPTOTIC COMPLEXITY

- notations allow us to compare the growth rates of different functions
  - $f(n) = O(g(n))$ means that $f(n)$ grows no faster than $g(n)$
    - for n sufficiently large and a suitably chosen constant c
      - $f(n)$ always remains below c g(n)
  - $f(n) = \Omega(g(n))$ means that $f(n)$ grows at least as fast as $g(n)$
    - Eventually $f(n)$ will exceed c g(n) for all n beyond n0
  - $f(n) = \Theta(g(n))$ means that $f(n)$ and $g(n)$ grow at about the same rate
    - value of f(n) is always bounded by c g(n) and c' g(n) (for $n > n0$)

ASYMPTOTIC COMPLEXITY

- notations allow us to compare the growth rates of different functions



ASYMPTOTIC COMPLEXITY

- In other words

"The rate of growth of $f(n)$ is ___ that of $g(n)$."

we can fill in the blank with the relational symbol ($<$, $\leq$, $=$, $\geq$, $>$) to the left of the defined relations shown below:

$<$	$f(n) = o(g(n))$	$\lim_{n\to\infty} f(n)/g(n) = 0$ {read little-oh of $g(n)$}
$\leq$	$f(n) = O(g(n))$	{big-oh}
$=$	$f(n) = \Theta(g(n))$ or $\theta(g(n))$	{theta}
$\geq$	$f(n) = \Omega(g(n))$	{big-omega}
$>$	$f(n) = \omega(g(n))$	$\lim_{n\to\infty} f(n)/g(n) = \infty$ {little-omega}

ASYMPTOTIC COMPLEXITY

- At a very coarse level

Sublinear	$O(1)$	constant-time
	$O(\log \log n)$	double logarithmic
	$O(\log n)$	logarithmic
	$O(\log^k n)$	polylogarithmic; k is a constant
	$O(n^a)$	$a < 1$ is a constant; e.g., $O(\sqrt{n})$ for $a = 1/2$
	$O(n / \log^k n)$	k is a constant
Linear	$O(n)$	
Superlinear	$O(n \log^k n)$	
	$O(n^c)$	polynomial; $c > 1$ is a constant; e.g., $O(n\sqrt{n})$ for $c = 3/2$
	$O(2^n)$	exponential
	$O(2^{2^n})$	double exponential

Complexity classes

- Problems are divided into several complexity classes
  - Based on their running times on a single-processor
- Problems said to belong to the P class
  - running times are upper bounded by polynomials in n
  - generally considered to be tractable.
    - Even if the polynomial is of a high degree
  - there is still hope that a reasonable running time may be obtained
    - by improvements in the algorithm or in computer performance

Complexity classes

- problems said to belong to NP (nondeterministic polynomial) class
  - best known deterministic algorithm runs in exponential time
    - But the correctness of the solution can be verified in polynomial time
  - They are intractable
  - E.g., subset-sum problem
    - Given a set of n integers and a target sum s
    - determine if a subset of the integers in the given set add up to s

Complexity classes

- problems said to belong to NP-complete class
  - any problem in NP can be transformed to any one of these problems
    - by a computationally efficient process
  - The subset-sum problem is known to be NP-complete
  - if one ever finds an efficient solution one of these problems
    - this proves P = NP
  - are the “hardest” problems in the NP class
    - proving that a problem is NP-complete removes any hope of finding an efficient algorithm

Complexity classes

- problems said to belong to NP-hard class
  - problems that are not even in NP
  - verifying that a claimed solution to such a problem is correct is currently intractable
  - we do not know to be in NP
    - but do know that any NP problem can be reduced to it by a polynomial-time algorithm
  - are at least as hard as any NP problem

Complexity classes

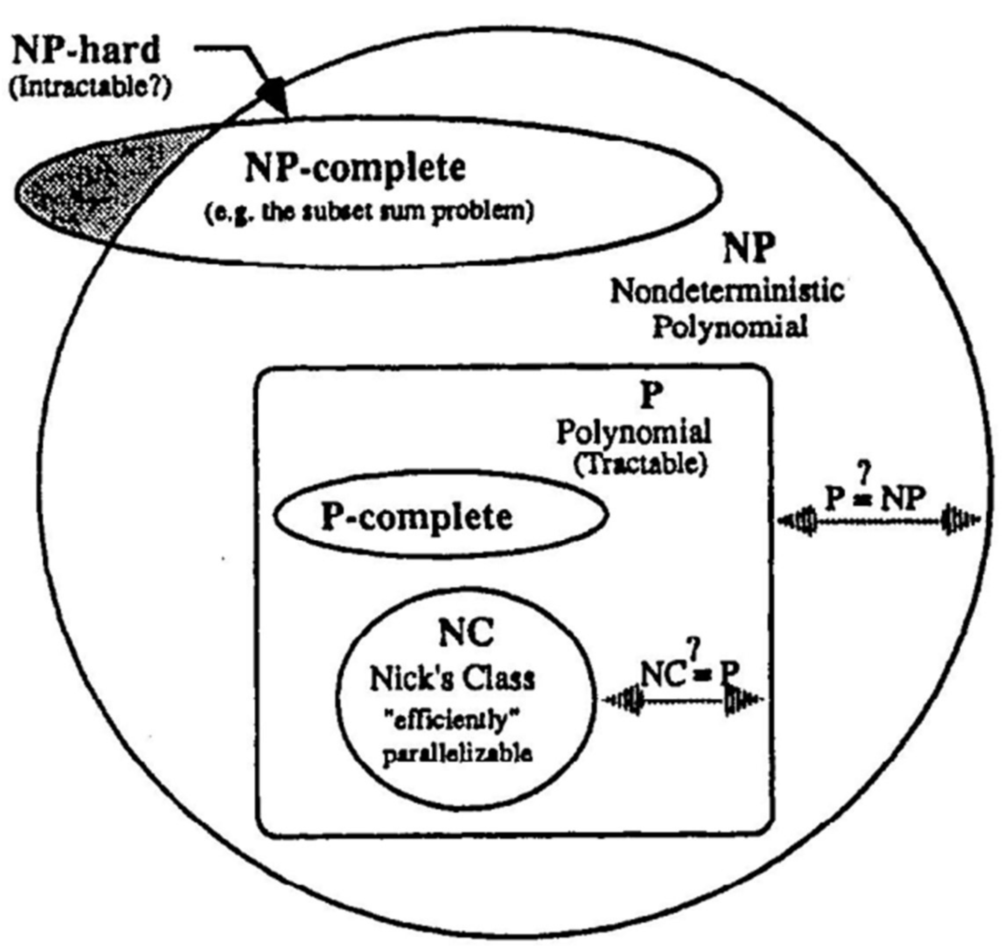


Figure 3.4. A conceptual view of complexity classes and their relationships.

Parallelizable tasks and the NC class

- parallel processing
  - is generally of no avail for solving NP problems
    - A problem that takes 400 billion centuries on a uniprocessor
      - if it can be perfectly parallelized over 1 billion processors
        - It would still take 400 centuries
  - is useful for speeding up the execution time of the problems in P
    - even a factor of 1000 speed-up can mean the difference between practicality and impracticality

Parallelizable tasks and the NC class

- The NC class
  - efficiently parallelizable problems in P
    - defined as
      - problems that can be solved
        - By at most polylogarithmic in the problem size n, i.e., T(p) = $O(\log^k n)$ for some constant k
        - using no more than a polynomial number p = $O(n^l)$ of processors
  - Sorting is a good example

Parallel programming paradigms

- Divide and Conquer
  - Decompose the problem of size n into two or more "smaller" subproblems.
    - takes $T_d(n)$ time when done in parallel
  - Solve the subproblems independently and obtain the corresponding results
    - the time T_s to solve them will likely be less than T(n)
  - Finally, combine the results of the subproblems to compute the answer to the original problem.
    - If the combining can be done in time $T_c(n)$,
      - the total computation time is given by $T(n) = T_d(n) + T_s + T_c(n)$.

Parallel programming paradigms

- Divide and Conquer
  - is perhaps the most important tool for devising parallel algorithms
  - E.g., sorting a list of n keys
    - decompose the list into two halves
    - sort the two sublists independently in parallel
    - merge the two sorted sublists into a single sorted list

Parallel programming paradigms

- Randomization
  - Often balanced divide and conquer is impossible, or computationally difficult
    - Obstacles for achievable effective speed-up
      - Large decomposition and combining overheads
      - wide variations in the solution times of the subproblems
    - it might be possible to use random decisions
      - lead to good results with very high probability.
  - Has led to the solution of many otherwise unsolvable problems.

Parallel programming paradigms

- Randomization
  - Again, sorting provides a good example
    - each of p processors begins with a sublist of size n/p
    - each processor selects a random sample of size k from its local sublist
    - The kp samples from all processors form a smaller list that can be readily sorted
      - on a single processor
      - or using an efficient parallel algorithm for small lists.
    - this sorted list of samples is now divided into p equal segments
    - the beginning values in the p segments used as thresholds to divide the original into p sublists
      - the lengths of these sublists will be approximately balanced with high probability

Parallel programming paradigms

- Randomization
  - Again, sorting provides a good example
    - The sorting problem has thus been transformed into
      - an initial random sampling
      - a small sorting problem for the kp samples
      - broadcasting of the p threshold values to all processors
      - permutation of the elements among the processors according to the p threshold values
      - p independent sorting problems of approximate size n/p
    - The average case running time can be quite good
    - However, there is no useful worst-case guarantee on its running time.

Parallel programming paradigms

- Randomization
  - can be applied in several other ways
    - Input randomization
      - used to avoid bad data patterns that a particular algorithm
      - is efficient on the average
      - might have close to worst-case performance
    - Random search
    - Control randomization
    - Symmetry breaking

Parallel programming paradigms

- Randomization
  - can be applied in several other ways
    - Input randomization
    - Random search
      - Searching a large space for an abundant element
        - random search can lead to very good average-case performance
        - A deterministic linear search can lead to poor performance
          - if all of the desired elements are clustered together
    - Control randomization
    - Symmetry breaking

Parallel programming paradigms

- Randomization
  - can be applied in several other ways
    - Input randomization
    - Random search
    - Control randomization
      - Randomly choosing the algorithm or an algorithm parameter
      - avoid consistently experiencing close to worst-case performance
        - with one algorithm
        - For some unfortunate distribution of inputs
    - Symmetry breaking

Parallel programming paradigms

- Randomization
  - can be applied in several other ways
    - Input randomization
    - Random search
    - Control randomization
    - Symmetry breaking
      - deterministic processes may exhibit a cyclic behavior that leads to deadlock
        - Randomization can be used to break the symmetry and thus the deadlock

Parallel programming paradigms

- Approximation
  - Iterative numerical methods often use approximation
    - begin with some rough estimates for the answers
    - successively refine these estimates using numerical calculations
  - Advantage
    - fairly precise results can be obtained rather quickly
    - additional iterations may be used to increase the precision if desired
  - It is a powerful method for time/cost/accuracy trade-offs because
    - the computations for each iteration can be easily parallelized over any number p of processors
    - the computation can still be performed at lower precision in case of deadline limitations