EEECE7205: Fundamentals of Computer Engineering

Randomized and Parallel Algorithms

Introduction

- ***Probabilistic analysis*** is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm.
- In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the **distribution** of the inputs.
- Then we analyze our algorithm, computing an ***average-case running time*** (vs. best or worst cases), where we take the average over the distribution of the possible inputs.
- In many cases, we know very little about the input distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally.
- Yet we often can use probability and randomness as a tool for algorithm *design* and *analysis*, by making the behavior of part of the algorithm random.

Randomized Algorithms (1 of 2)

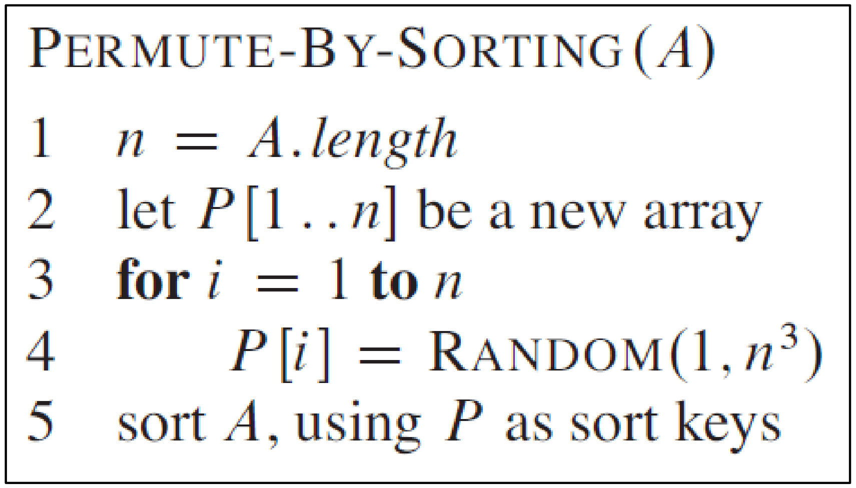
- We call an algorithm ***randomized*** if its behavior is determined not only by its input but also by values produced by a ***random-number generator*** (as part of its design).
- When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator.
- We distinguish these algorithms from those in which the input is random by referring to the running time of a randomized algorithm as an ***expected running time***.
  - The ***average-case running time*** when the probability distribution is over the inputs to the algorithm.
  - The ***expected running time*** when the algorithm itself makes random choices as part of its design (e.g., choosing the pivot randomly in Quicksort).

Randomized Algorithms (2 of 2)

- Many times, we do not have the distribution on the inputs, thus preventing an average-case analysis.
- Instead of assuming a distribution of inputs, we impose a distribution. In particular, before running the algorithm, we randomly permute the elements in the input array in order to enforce the property that every permutation is equally likely.
- *Example*: We are given an array *A* which contains the elements 1 through *n*. Our goal is to produce a ***uniform random permutation***, that is, it is equally likely to produce every permutation of the elements 1 through *n*.
  - Two methods to solve this problem: *Permute by Sorting* and *Randomize in Place*.

Permute by Sorting

- Assign each element $A[i]$ of the array a random priority $P[i]$, and then sort the elements of A according to these priorities.
- For example, if our initial array is A = (Red, Black, Blue, Green) and we choose random priorities P = (36, 3, 62, 19), we would produce an array B = (Black, Green, Red, Blue).
- Line 4 chooses a random number between 1 and n^3.
- We use a range of 1 to n^3 to make it likely that all the priorities in P are unique.
- The time-consuming step in this procedure is the sorting in line 5 that can take $\Theta(n \lg n)$



```
PERMUTE-BY-SORTING(A)
1  n = A.length
2  let P[1..n] be a new array
3  for i = 1 to n
4      P[i] = RANDOM(1, n^3)
5  sort A, using P as sort keys
```

Randomize in Place

- In its *i*th iteration, the procedure chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$.

```
RANDOMIZE-IN-PLACE(A)
1  $n = A.length$
2  for $i = 1$ to $n$
3      swap $A[i]$ with $A[\text{RANDOM}(i, n)]$
```

- This method takes $O(n)$ time.

Randomized Quicksort (1 of 2)

- The original Partition procedure of the Quicksort algorithm always uses *A*[*r*] as the pivot.

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

- A randomization version of the procedure selects as a pivot a randomly chosen element from the subarray *A*[*p* .. *r*].

```
RANDOMIZED-PARTITION($A, p, r$)
1  $i = $ RANDOM($p, r$)
2  exchange $A[r]$ with $A[i]$
3  return PARTITION($A, p, r$)
```

Randomized Quicksort (2 of 2)

- By randomly sampling the range $p, \ldots, r$, we ensure that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray.

- Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average and hence decrease the chance of having the worst-case scenario of Quicksort.

Parallel Algorithms

- The algorithms we studied so far are ***serial algorithms*** suitable for running on a uniprocessor computer in which only one instruction executes at a time.

- ***Parallel algorithms***, which can run on a multiprocessor computer that permits multiple instructions to execute concurrently.

Parallel Processing

- Nowadays processors in inexpensive desktop and laptop contain a single ***multicore*** integrated-circuit chip that houses multiple processing "cores". Each core is a full processor that can access a common memory.
- At an intermediate price/performance level are **clusters** built from individual computers—often simple PC-class machines—with a dedicated network interconnecting them.
- The highest-priced machines are **supercomputers**, which often use a combination of custom architectures and custom networks to deliver the highest performance in terms of instructions executed per second.

Dynamic Multithreaded Programming

- One important class of concurrency platform is ***dynamic multithreading***.
- It is a simple extension of the serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just three "concurrency" keywords: **parallel**, **spawn**, and **sync**.
- A parallel loop is like an ordinary **for** loop, except that the iterations of the loop can execute concurrently.
- A subroutine can be "spawned," where the caller is allowed to proceed while the spawned subroutine is computing its result.
- The keyword **sync** indicates that the procedure must wait for all its spawned children to complete before proceeding to the statement after the **sync**.

Dynamic Multithreading Example

- Recurrence definition of Fibonacci numbers:

$$
\begin{aligned}
F_0 &= 0 , \\
F_1 &= 1 , \\
F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2
\end{aligned}
$$

<u>Serial Algorithm</u>

```
FIB(n)
1  if $n \leq 1$
2      return $n$
3  else $x = \text{FIB}(n-1)$
4      $y = \text{FIB}(n-2)$
5      return $x + y$
```

<u>Dynamic Multithreading Algorithm</u>

```
P-FIB(n)
1  if $n \leq 1$
2      return $n$
3  else $x$ = spawn P-FIB$(n-1)$
4      $y = \text{P-FIB}(n-2)$
5      sync
6      return $x + y$
```

Multithreaded Merge Sort

- Because merge sort already uses the divide-and-conquer paradigm, it seems like a terrific candidate for multithreading using nested parallelism.
- We can modify the pseudocode so that the first recursive call is spawned:

```
MERGE-SORT'$(A, p, r)$
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    spawn MERGE-SORT'$(A, p, q)$
    MERGE-SORT'$(A, q + 1, r)$
    sync
    MERGE$(A, p, q, r)$
```