Slide 1

Administrivia

- Homework 1 posted; due end of this week.

Slide 2

Divide and Conquer Revisited

- "Divide and conquer", in context, is a strategy for problem-solving that consists of three steps:
- *Split* the problem into one or more smaller problems that are instances of the same problem, unless it's so small and simple that it can be solved directly (in which case it's called a *base case*).
- *Solve* the subproblems by solving them recursively.
- *Merge* the solutions to the subproblems into a solution to the whole problem.
- Note that of course this can only work if this recursive splitting eventually always reaches a base case.

Slide 3

Analyzing Divide and Conquer Algorithm

- To analyze execution time of a divide-and-conquer algorithm, we use a *recurrence* — an equation or inequality that describes a function using the function itself. As in the mergesort example, this should seem like a logical way to approach analyzing such an algorithm.
- Generally we'll be interested in recurrences that are *algorithmic*, i.e.,:
  - $T(n)$ is $\Theta(1)$ (i.e., constant-time) for sufficiently small n.
  - Every path of recursion ends in a defined base case within a finite number of recursive invocations.

Slide 4

Solving Recurrences

- Several ways to solve recurrences:
- *Substitution method* — guesses a solution and proves that it is correct via induction.
- *Recursion-tree method* — models the recurrence as a tree, as the textbook does for MERGE-SORT.
- *Master method* — applies to recurrences of the form

  $$T(n) = aT(n/b) + f(n)$$

  by applying a "master theorem".
- *Akra-Bazzi method* — a general method involving calculus.

Slide 5

Examples — Multiplying Square Matrices

- Matrix multiplication occurs frequently in numerical work. You may remember how this works, and if not, review summary in Appendix D. Given two N by N matrices A and B, their product is also N by N, and its elements are defined as:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

(I find it useful to model this as a dot product of the i-th row of A and the j-th column of B.)

Generally we'll look at *dense* matrices, in which most elements are nonzero. (In *sparse* matrices, most elements are zero; such matrices can be more compactly represented.)

Slide 6

Block-Based Matrix Multiplication

- We'll first observe that another way to compute the product $C = A \cdot B$ is by first partitioning each of the matrices into submatrices ("blocks", also identified by two indices, e.g., C_{11} for the top left block) and applying the basic idea of the algorithm — sum of products — to these blocks:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

- (Showing that this works is not difficult — basic algebra — though tedious.)
- To put this to use, it's helpful to define and implement an algorithm to compute

$$C = C + A \cdot B$$

Slide 7

Multiplying Square Matrices, Continued

- As noted, it's more generally useful to define an algorithm for computing $C = C + A \cdot B$ (where A, B, and C are n by n dense matrices). The naive algorithm is just:

```
MATRIX-MULTIPLY$(A, B, C, n)$
for $i = 1$ to $n$
    for $j = 1$ to $n$
        for $k = 1$ to $n$
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
```

  It should be fairly clearly that this algorithm's running time is $\Theta(n^3)$.

Slide 8

Recursive Matrix Multiplication, Take 1

- A simple way to approach matrix multiplication recursively is based on dividing each matrix into 4 submatrices, each $n/2$ by $n/2$. Then we can write the desired product as on the next slide . . .

Slide 9

Recursive Matrix Multiplication, Take 1, Continued

- In matrix notation:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- In equation form, one for each submatrix of C:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Slide 10

Recursive Matrix Multiplication, Take 1, Algorithm

- An algorithm based on this approach (assuming C is initialized to 0):

```
MATRIX-MULTIPLY-RECURSIVE($A, B, C, n$)
if $n == 1$
        $c_{11} += a_{11} \cdot b_{11}$ // Base case
        return
MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{11}, C_{11}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{12}, C_{12}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{11}, C_{21}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{12}, C_{22}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{21}, C_{11}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{22}, C_{12}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{21}, C_{21}, n/2$)
MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{22}, C_{22}, n/2$)
```

Slide 11

Recursive Matrix Multiplication, Take 1, Implementation Details

- It's worth noting that in order to make access to the submatrices work we have to either copy the submatrices to and from temporary storage, incurring a runtime cost of $\Theta(n^2)$, or manipulate indices appropriately, which takes constant time. The latter is more efficient and practical, though the details are a little tedious to work out.

Slide 12

Sidebar: Block-Based Matrix Multiplication, Revisited

- Total tangent, but I think maybe interesting:
- Matrix multiplication turns out to be an operation that lends itself to "parallelization" (solving in a way that uses multiple threads or processes at the same time), and often block-based algorithms work well for such implementation.
- So . . . I teach an elective (CSCI 3366) in parallel computing, and I use matrix multiplication as an example. I used to do research on design patterns for parallel computing, and one of my long-term research collaborators claimed that block-based matrix multiplication was faster even without multiple processes or threads, because it "made better use of the memory hierarchy" (cache etc.). I was skeptical, but when I wrote code for it . . . (Short demo. Note that code is available on course Web site under "sample programs".)

Slide 13

Sidebar, Continued

- Sample code accesses submatrices using the method of index calculation, based on a slight extension to how straight C represents multidimensional arrays:
- Straight C represents an m by n 2D array `A` in "row-major" order, i.e., as a 1D array `AA` of size $m \cdot n$ consisting of row 0 of `A` followed by row 1, then row 2, etc. `A[i][j]` is then accessed as `[(i*n)+j]`. (A diagram helps here!)

  How then to access elements of submatrices? well, each row of a submatrix is contiguous in memory, just as a full row is, but while the rows of the full matrix are laid out one after another, the rows of a submatrix aren't, but are spaced out at regular intervals, based on the dimensions of the full matrix. An index relative to a submatrix is not too hard to compute using the dimensions of the full matrix and a "stride" variable representing the distance between submatrix rows in the full 1D array. Here too a diagram is helpful, and see example code.

Slide 14

Algorithm Correctness

- The base case is okay — in this case each of the submatrices is a single element.
- It might not be 100% obvious why this algorithm correctly translates the block-based equations for the multiplication, but: Note that each of the equations does two multiplications and adds the result, for a total of 8 calculations of the form $C_{ij} += A_{ik} \cdot B_{kj}$. (This also explains why it's useful to have as a basic operation $C = C + A \cdot B$.)

Slide 15

Algorithm Execution Time

- If we let $T(n)$ be execution time of the algorithm:
- First we note that the base case just performs one scalar multiplication and one addition, so it is solved in constant time.
- In the recursive case, we call MATRIX-MULTIPLY-RECURSIVE a total of eight times, each with a problem size of $n/2$, for a total of $8T(n/2)$. The cost to split the problem is the cost to partition the matrix into submatrices, which if we do it by manipulating indices is constant, and there's no cost to combine (since we update C in place). This gives a recurrence of

$$T(n) = 8T(n/2) + \Theta(1)$$

Slide 16

Algorithm Execution Time, Continued

- The textbook uses the "master method" (more later) to get an order of growth of $\Theta(n^3)$ for this recurrence. It's at least a bit reassuring that this is the same as the order of growth of the naive algorithm, since this does seem like it's doing the same basic operations, just in a different order, no?

Slide 17

Recursive Matrix Multiplication, Take 2

- It probably seems intuitively obvious that execution time for matrix multiplication would be at best $\Theta(n^3)$, and this was widely believed for a long time, but then a mathematician named Strassen came up with al algorithm that runs in $\Theta(n^{\log 7})$ time, which isn't as dramatic an improvement as mergesort over insertion sort but is an improvement!
- It does this by reducing the number of recursive steps from eight to seven. More next time . . .

Slide 18

Minute Essay

- Questions?