ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Algorithms

- Informally, an algorithm can be viewed as
  - a well-defined computational procedure that
    - takes a set of values as input
    - produces a set of values as output
  - a solution to a computational problem
    - whose statement specifies the intended relationship between inputs and outputs
    - and the algorithm being the specific computational procedure that achieves this relationship
- the prototypical computational problem is *sorting*
  - problem specification
    - input: a sequence S of n numbers $(s_0, s_1, \ldots, s_n)$
    - output: a permutation S' of the input sequence $(s'_0, s'_1, \ldots, s'_n)$
    - constraint: the elements of the output sequence must satisfy

$$s'_0 \leq s'_1 \leq \ldots \leq s'_n$$

  - problem instance: $S = (0, \pi, 1, e, 2, 16)$
  - invalid input: $(0, i, 1)$
    - why is this bad? which implicit property of S does it violate? what is the set of valid inputs?

Correctness

- once again informally, an algorithm is *correct* if
  - it terminates for all valid input
  - upon termination on valid input, the output satisfies the constraints expressed in the problem statement
- equivalently, we say that the algorithm *solves* the computational problem
- after correctness has been established, algorithms are classified according to their demands on computational resources
  - running time complexity
    - a measure of the number of instructions necessary to solve the problem
  - and, occasionally
    - amount of auxiliary storage
    - network bandwidth or other communication infrastructure requirements
    - for parallel algorithms: speedup and efficiency
- algorithms are often specified using *pseudoce*
  - a loose language with mostly notational constraints
  - a mixture of reasonable looking code with whatever expressive method makes the point clear
  - hence, the use of human languages to convey meaning that might be too difficult to code up, or would obscure the point, is perfectly acceptable

A sorting algorithm

Algorithm 1: INSERTION-SORT(S)

```
for j ← 2 to length[S] do
    key ← S[j]
    i ← j − 1
    while i > 0 and S[i] > key do
        S[i + 1] ← S[i]
        i ← i − 1
    S[i + 1] ← key
```

- valid inputs:
  - empty sequence, singlet, other sequences of finite length
  - what kinds of objects in S?
- walk through it by hand with $S = (5, 2, 4, 6, 1, 3)$

Pseudocode conventions

- the symbol "$\triangleright$" indicates a comment through to the end of the line
- block structure is indicated by the indentation level
- all variables are local; no global variables, unless explicitly marked
- $i \leftarrow j \leftarrow k$ assigns the rightmost expression to all the other variables
- indexing: $S[i]$; slicing: $S[i..j]$
- conditionals, looping constructs, function calls should be familiar
- compound objects have attributes or fields that are referenced using indexing, e.g. $length[S]$
- variables assigned to objects or containers are references
- parameters passed to procedures *by assignment*

Python implementation

- direct translation of pseudocode in python, with no attempt to improve

```python
def insertion_sort(S):
   for j in range(1, len(S)):
      key = S[j]
      i = j-1
      while i>=0 and S[i]>key:
         S[i+1] = S[i]
         i = i-1
      S[i+1] = key
```

- minor adjustments to loop indices are required since python lists are zero based

Analyzing algorithms

- algorithm analysis is the computation of resource requirements
  - memory, communication bandwidth, *computational time*
- need a model for the implementation environment
  - RAM: *random access machine*
  - an abstraction of a single processor sequential execution machine that has access to a single block of memory with uniform access cost
  - even though the model is extermely simple, algorithm analysis remains a hard problem, full of subtleties
- in general, we seek to relate the running time to input size
  - definition of input size is problem dependent – could be number of items to sort, or number of grid points in a mesh, etc.
  - running time counts the number of primitive steps executed
  - different lines have different costs
  - but each execution of a given line is assumed to cost the same
- *exercise*: decorate Alg. 1 with the number of times each line is executed

Runtime complexity of INSERTION-SORT

- summing up the number of times each line of Alg. 1 get executed:

$$T(n) = c_2 n^2 + c_1 n + c_0$$

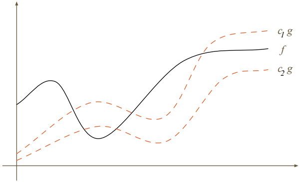
  where the c_i are constants related to the cost of the various lines
- note that it is a quadratic function of n
- best case: input S is already sorted
- worst case: input S is reverse-sorted
- average case: assume a *random* S and compute an expectation value for the number of executions of each line
  - still a quadratic function of n
- we quantify the run time complexity of INSERTION-SORT by saying that it is asymptotically bound by n^2
  - concentrating on the highest power of n
  - disregarding the multiplicative constants that are strongly dependent on the execution model, rather than the quality of the algorithm

Asymptotic bounds

- most often, run time complexity analysis is reduced to constructing asymptotic bounds on execution time as the input size $n \to \infty$
  - i.e., finding a simpler function of the input size with similar behavior for large n
- we say that $f = \Theta(g)$ if there are constants c_1, c_2 such that

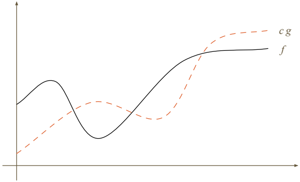
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

  for sufficiently large n

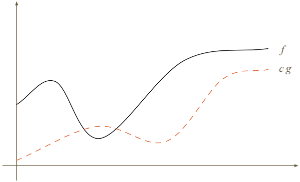


Upper and lower bounds

- bounded from above: we say that $f = O(g)$ if there is a constant c such that $f(n) \leq cg(n)$ for sufficiently large n



- bounded from below: we say that $f = \Omega(g)$ if there is a constant c such that $cg(n) \leq f(n)$ for sufficiently large n



Designing algorithms

- INSERTION-SORT is *incremental*:
  - having sorted $S[i..j]$, put $S[j]$ in its proper place
  - how would you break this up into tasks that can be executed in parallel?
- one alternative is *divide-and-conquer*: MERGE-SORT
  - *divide*: split S into two parts of roughly equal length
  - *conquer*: sort the subsequences recursively
  - *combine*: merge the two sorted subsequences to produce the sorted output

Sorting by divide-and-conquer

Algorithm 2: MERGE-SORT(S, p, r)

```
if p < r then
    q ← ⌊(p + r)/2⌋
    MERGE-SORT(S, p, q)
    MERGE-SORT(S, q + 1, r)
    MERGE(S, p, q, r)
```

- *exercise*: write MERGE; can be done in $\Theta(r - p + 1)$
- analysis of running time:
  - involves solving a recurrence relation
  - worst case:

$$T(n) = \left\{ \begin{array}{ll} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{array} \right\} \rightarrow \Theta(n \log n)$$

- is this a better candidate for parallel sorting?