ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

A sorting algorithm

Algorithm 1: INSERTION-SORT(S)

for $j \leftarrow 2$ **to** *length[S]* **do**
  $key \leftarrow S[j]$
  $i \leftarrow j - 1$
  while $i > 0$ **and** $S[i] > key$ **do**
    $S[i+1] \leftarrow S[i]$
    $i \leftarrow i - 1$
  $S[i+1] \leftarrow key$

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

- only minor adjustments to loop indices 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

Designing algorithms

- INSERTION-SORT is *incremental*:
  - having sorted $A[i..j]$, put $A[j]$ 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 A 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**
    $q \leftarrow \lfloor (p+r)/2 \rfloor$
    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?