Advanced Algorithms and Parallel Programming

Matteo Secco

March 5, 2021

Contents

1 Algorithm recap

Algorithm any well defined computational procedure that takes some values as input, and produces some values as output. It must terinate in a finite number of steps

Algorithm analysis Theoretical study of computer-program performance and resource usage Performance is not the only thing that matters. For example

- modularity
- maintainability
- user-friendlyness

are some important aspects as well

Why do we study algorithms and performance?

- to better understand scalability
- to understand what is feasible and what is not
- to have a formal language to talk about a program behaviour

Algorithm 1 Insertion sort

Require: A, n
Require: $len(A) = n$
Ensure: A is sorted
  for $j \leftarrow 2$ **to** $j \leq n$ **do**
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i > 0$ $A[i] > key$ **do**
      $A[i+1] \leftarrow A[i]$
      $i \leftarrow i - 1$
    end while
    $A[i+1] = key$
  end for

Obvious observations

- Running time depends on the input
- Running time has to be parametrized on the length of the input
- We generally look for upper limits because those are more interesting in real world

Kind of analysis

Worst case $T(n)$ =maximum time of the algorithm on any input of size n

Average case $T(n)$ =expected time over all inputs of size n. Requires assumption on statistical distribution of the inputs

Best case Easy to be cheated with slow algorithms that works very well on specific inputs

Asympthotic Analysis Ignore machine-dependent constraints and look at the behaviour of $T(n)$ as $n \to \infty$

1.1 Asymptotic notations

O-notation provides upper bounds to execution times

$$O(g(n)) = \{f(n) : \quad \exists c > 0, n_0 > 0 : 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\}$$

Ω-notation provides lower bounds to execution times

$$\Omega(g(n)) = \{f(n) : \quad \exists c > 0, n_0 > 0 : 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0\}$$

Θ-notation provides tight bounds to execution times

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Algorithm 2 Merge-Sort

Require: A, n
Require: $len(A) = n$
  if $n = 1$ **then**
    return A
  end if
  $L \leftarrow$ Merge-Sort $\left(A\left[1..\left\lceil\frac{n}{2}\right\rceil\right]\right)$
  $R \leftarrow$ Merge-Sort $\left(A\left[\left\lceil\frac{n}{2}\right\rceil + 1..n\right]\right)$
  return Merge(L, R)

Merge sort

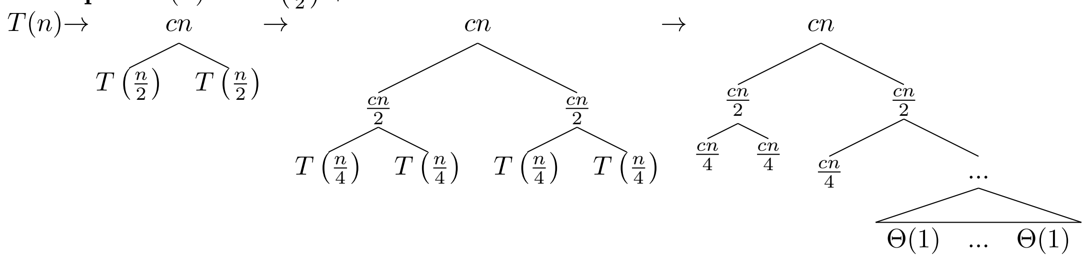
2 Recurrence analysis

2.1 Recursion tree analysis

applied to merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Example $T(n) = 2T\left(\frac{n}{2}\right) + cn$



Recursion conplexity	**Base case complexity**	**Total Complexity**
$h = \log(n)$ each level adds up to cn	#leaves$= n$ $\Theta(1)$ per leave	
$h \cdot cn = cn\log(n)$	n	$O(n\log(n)$

2.2 Analysis by substitution

Guess the form of the solution

Verify by induction

Solve for constraints

Example $T(n) = 4T\left(\frac{n}{2}\right) + n$ (and $T(1) = \Theta(1)$)

- Guess $T(n) = O(n^3)$
- Find some $k < n$ such that $T(k) \leq ck^3$
- Prove $T(n) \leq cn^3$ by induction

2.3 Master theorem

Applies to recurrencies of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
where $a \geq 1, b > 1, f > 0$ for $n \to \infty$

$$
\begin{aligned}
f(n) &= O(n^{\log_b a - \epsilon}) && \to T(n) = \Theta(n^{\log_b a}) \\
f(n) &= \Theta(n^{\log_b a} \log^k n) && \to T(n) = \Theta(n^{log_b a} \log^{k+1} n) \\
f(n) &= \Omega(n^{\log_b a + \epsilon}) && \to T(n) = \Theta(f(n))
\end{aligned}
$$

3 Divide and Conquer

Divide the problem into subproblems

Conquer the subproblems recursively

Combine subproblem solutions

Merge sort

Divide split in half

Conquer Sort the 2 subarrays

Combine Linear-time merge

Binary search

Divide Check middle element

Conquer Search 1 subarray

Combine Return result up

Compute a^n

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if n is even} \\ a \cdot a^{(n-1)/2} \cdot a^{(n-1)/2} & \text{if n is odd} \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log n)$$

4 Parallel Random Access Machine

RAM: abstract device having

- Unbounded number of memory cells
- Unbounded size for each memory cell
- Instruction set including simple operations, data operations, comparations, branches
- All operations take unitary time
- Time complexity = # instructions executed
- Space complexity = # memory cells used

PRAM: abstract device for designing parallel algorithms. M' is a system $< M, x, y, A >$ of infinitely many

- RAMs M_i called processors. Each is assumed to be itentical to the others and recognize its own index
- Input cells X_i
- Output cells Y_i
- Shared memory cells A_i

Computation step consists of 5 phases. In parallel, each processor:

- Reads a value from one of X_i
- Reads a value from one of A_i
- Performs some internal computation
- May write into one of the Y_i
- May write into one of the A_i

Some peculiarities to highlight:

- Some processors may remain idle
- More processor can safely read the same memory cell
- When more processor write the same cell at the same time a **write conflict** occours

4.1 Conflict Management

PRAM are classified based on how they manage conflicts.

Exclusive Read (ER) processors can read only from distinct memory cells

Concurrent Read (CR) processors can simultaneously read from any memory cell

Exclusive Write (EW) processors can symultaneously write to distinct memory cells

Concurrent Write (CR) processors can simultaneously read from any memory cell

Four possible combination may happen, but only three (EREW, CREW, CRCW) are interesting. This is because in real applications, it is pointless to have read capabilities stricter than write ones.

Concurrent Writes allows three models to determine what will be actually written in case of a conflict:

Priority CW each processor is assigned a (unique) priority. The value from the processor with higher priority is the one actually written

Common CW the write completes $\iff$ all the values to be written are equal

Arbitrary/Random CW one randomly chosen processor is allowed to complete the write

4.2 Strenghts of PRAM

Natural: the number of operations per cycle having p processors is at most p

Strong: any processor can read/write any shared cell in unit time

Simple: abstracts from communication/synchronization overheads, making it easier to evaluate complexity and correctness

Can be used as benchmark: if a problem has no feasible solution in PRAM, it neither has on any parallel machine

4.3 Computational power

A is computationally stronger than $B \iff$ any algorithm written for B will run unchanged on A in the same parallel time and with the same basic properties.

Most powerful
Least realistic Priority $\geq$ Arbitrary $\geq$ Common $\geq$ CREW $\geq$ EREW *Least powerful*
Most realistic

4.4 Definitions

$T^*(n)$ Time to solve on <u>one</u> processor, using the best sequential algorithm

$T_p(n)$ Time to solve on p processors

$SU_p(n) = \frac{T^*(n)}{T_p(n)}$ Speedup on p processors

$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)}$ Efficiency (time on 1 / time that could be used on p)

$T_\infty(n)$ Shortest run time for any value of p

$C(n) = P(n) \cdot T(n)$ Cost (in terms of time and processors

$W(n)$ Work=total #operations

4.5 Amdahl's vs Gustafson's law

4.5.1 Amdahl's law

Computation model consists in interleaved segments. Segments can either be serial (no speedup from parallelization) or parallelizable (allowing for speedup)

$$T_p > \frac{T_1}{P} \implies SU > P$$

The length of the parallelizable part is a **fixed** fraction f, and the sequential length is $1 - f$.

$$SU(P, f) = \frac{T_1}{T_p} = \frac{T_1}{\underbrace{T_1 \cdot (1-f)}_{\text{sequential part}} + \underbrace{\frac{T_1 \cdot f}{P}}_{\text{parallel part}}} = \frac{1}{(1-f) + \frac{f}{P}}$$

$$\lim_{P \to \infty} SP(P, f) = \frac{1}{1-f}$$

4.5.2 Gustafson's law

Differences in the model

- f is not fixed
- Absolute serial time is fixed
- Parallel problem size is increased to exploit more processors
- Serial time s is fixed
- Parallel time per processor $1 - s$ is fixed

$$SU(P) = \frac{T_1}{T_p} = \frac{s + \overbrace{P \cdot (1-s)}^{\text{full parallel computation}}}{s + \underbrace{(1-s)}_{\substack{\text{parallel computation} \\ \text{on each processor}}}} = s + P \cdot (1-s) \qquad \Longrightarrow \textbf{Linear speedup}$$

5 Complexity classes

5.1 P complexity

Complexity class P contains decision problems which can be solved by a deterministic Turing machine in polynomial time

P-complete A problem is P-complete if it is in P and any problem in P can be reduced to it by an appropriate reduction

5.2 NC complexity

Nick's Class is the set of problem decidable in polylogarithmic time on a parallel computer with a polynomial number of processors

$$\exists c, k : \text{ the problem can be solved in time } O(log^c n) \text{ using } O(n^k) \text{ processors}$$