AD Aflevering 2

zts900, chs380

February 2025

1. Parallel Algorithms

We consider a simple, undirected graph $G = (V, E)$, where:
V is the set of vertices (nodes).
E is the set of edges.

We know that a triangle in a graph is a triplet (u, v, w) of vertices such that all three pairs of vertices are connected by edges, $\{u, v\}, \{u, w\}, \{v, w\} \in E$.

The incidence matrix of a graph is an $n \times n$ symmetric matrix $M(E)$, where:

$$M(E)_{ij} = \begin{cases} 1, & \text{if } \{v_i, v_j\} \in E \text{ (an edge exists between } v_i \text{ and } v_j) \\ 0, & \text{otherwise.} \end{cases}$$

Since the graph is undirected, the matrix $M(E)$ is symmetric, which means that $M(E)_{ij} = M(E)_{ji}$.

a)

We want to show that the elements of $P = M(E)^2$ represent the number of paths of length exactly 2 between any two nodes.
We consider the matrix product:

$$P_{ij} = (M(E)^2)_{ij} = \sum_{k=1}^{n} M(E)_{ik} \cdot M(E)_{kj}$$

We see that each term in the sum contributes:

- $M(E)_{ik} = 1$ if v_i is connected to v_k.
- $M(E)_{kj} = 1$ if v_k is connected to v_j.

Summing over all possible intermediate nodes v_k gives the total number of paths of length 2 from v_i to v_j.
Therefore we conclude that $P = M(E)^2$ encodes the number of paths of length exactly 2 between nodes.

b)

Using the result from a), we can design a parallel algorithm for counting triangles.
An important result in graph theory states that:

$$\text{Total number of triangles} = \frac{\text{sum of diagonal elements of } M(E)^3}{6}$$

This works because of the following reasons

- $(M(E)^3)_{ii}$ (diagonal elements of $M(E)^3$) count the number of cycles of length 3 starting and ending at v_i.
- Summing all diagonal elements gives the total number of triangle traversals.
- Each triangle is counted six times (once for each permutation of its three vertices), so we divide by 6.

We can design a parallel algorithm by using parallel matrix multiplication:

1. We Compute $M(E)^3$:
   - We use parallel matrix multiplication, where each entry of the resulting matrix can be computed independently.
2. Sum the diagonal elements:
   - Since only the diagonal elements of $M(E)^3$ are needed, this summation can also be performed in parallel.
3. We divide by 6 to obtain the final triangle count.

c)

We analyze the work and span assuming we use a parallel matrix multiplication algorithm with:

- Serial time T_1 (work complexity).
- Span T_∞ (Longest sequence of dependent operations).

Since matrix multiplication can be performed using stressens method we use the following:
Work for matrix multiplication complexity:

$$T_1 = O(n^\omega)$$

where $\omega = 2.37$ for fast matrix multiplication
Span for matrix multiplication:

$$T_\infty = O(\log n)$$

using divide and conquer parallelism
For the triangle counting algorithm we have the following:

1.Computing $M(E)^3$: Work: $O(n^\omega)$; Span: $O(\log n)$

2. Summing diagonal elements: Work: $O(n)$; Span: $O(\log n)$ (using parallel reduction).

3. Division by 6 Constant time, $O(1)$, which is negligible.

The overall Complexity is therefore: Total Work: $O(n^\omega)$ which is T_1 ; Total Span: $O(\log n)$ which is T_∞.

We conclude that the algorithm is highly parallelizable, as it can execute in $O(\log n)$ time using a large number of processors.

---

2 Amortised analysis

Based on CLRS 16.4.1: We have a dynamic table datastructure, T. Which has some nr elemets T.num at a given time step. The table also has some memory allocation T.size. When calling TABLE_INSERT we have two possible actions with different computational cost, (1) table is not full, we insert the element O(1), (2) table is full, we find a new memory spot of double the size and reinsert all elements and also adding our last element O(n).

Adressing overall time complexity with the potential function we want to pick an external metric of progress that we can tie to "credits" in order to pay for the rare but expensive operation. Another criteria is that this metric can never be negative. We know when our expensive operation happens, that is when T.num == T.size. CLRS proposes the following function $\Phi = 2(T.num - T.size/2)$ When T.num == T.size the list size doubles, the potential right after the size doubles is $\Phi = 2(T.num - T.size/2) = \Phi = 2(0) = 0$, and we have T.size/2 nr elements, then the next element potential will rise by 1*2, 2*2 etc. We take times two so we have $\Phi = T.size$ when we need to do the expansion, reseinrting $T.size = T.num$ elements. We have that $T.num \geq T.size/2$ meaning its always non negative.

The sum of the amortised cost for n insets gives an upper bound of the actual costs. Since we build up to approximately $\Phi = 2(T.num - T.num/2) = T.num$ before going to 0 again, we can upper bound this potential function with $T.num$, in other words time complexity is $O(i)$ for i elements. Lets evaluate the options:

Evaluating the alternatives:

1. $\phi_i = \Omega(1)$: False: Meaning there is some constant c that our function never falls below $0 < c \leq \phi_i$. Since the potential after expansion is 0, this does not hold.

2. $\phi_i = O(1)$: False: Meaning there is some cosntat c st $\phi_i \leq c$ for all i. We have as discussed approxiately linear time complexity, hence false.

3. $\phi_i = \Omega(i)$: Meaning our function does not drop below some constant function, $\phi_i \geq c * i$ for all i. This is false as function can drop to 0.

4. $\phi_i = O(i)$: Meaning our function doesnt grow faster than some cosntat function, this is true, the worst case for any given timestep is O(i) when having to resize the last i, otherwise smaller.

5. $\phi_i \geq \phi_{i-1}$: False, we can have high potential saved up f.ex in the step before we expand the table, that is then used leaving us with 0 and then +2 potential in the following time step which could be less than previously.

6. $\phi_i \geq 0$: True, per definition we can not have a negative potential. We make sure that this holds for this particular case by knowing we have 0 potential after every expansion, and inserting can only add to that. Making it strictly posstivie.

---

3 Red black nodes

We have two relevant invariants for RB trees in this scenario:

1. Every path from root to leaves (simple) have the same nr of black nodes

2. The tree is sorted, meaning leftChild.key ¡ parent.key ¡ rightChild.key.

Analysing the questionmarks:

- Looking at the left most question mark we need (1) a black node, that fits in the ranges (2) 27 ¡ ? ¡31 and 24 ¡ ?. Based on those criteria we select: Nr 5. Black node with key 30.

- Looking at the right most question mark we need (1) a red node, (2) in the range 56 ¡ ? : We select opetion 3, a red node with key 58.

4 Insertion into red-black search trees

The root is 10, and since $9 < 10$, we go to the left. The node 3 is the next choice, but since $9 > 3$, we go to the right. The node 8 is the next choice, but since $9 > 8$, we go to the right. The node 8 has no children, so we insert 9 as the right child of 8.
The inserted node is always colored red initially to maintain the black-height balance in the tree, therefore 9 is inserted as red.
Therefore, **1** is the correct answer.

---

5. Running time for red-black search trees

We know that a red-black tree is a kind of balanced binary search tree, where the operations Insert, Delete and Search all have a time complexity of $O(log\ n)$, on average and worst case. This is because the trees depth always is $O(log\ n)$, which means that operations that require searching down the tree are limited to this depth.

1

True. Each of the operations Insert, Delete and Search require at least a constant time in the smallest case. This means that the total time is at least propotional to the number of operations.

2

This is true. We know that insert and delete take each $O(log\ n)$ over $n_1 + n_2$: $O((n_1 + n_2)log\ n)$.
We also know that the search operation takes $O(log\ n)$ as well so the total time for n_3 searches is $O(n_3\ log\ n)$ Therefore we have that the total time is $O(n_3\ log\ n + (n_1 + n_2)\ log\ n) = O(N\ log\ N)$

3

This is false because it would need the search operation to take $O(n_3)$ instead of $O(n_3\ log\ N)$, which isn't correct, because each search operation still needs $O(log\ n)$

4

This is true, because we know of a fact (from the book where proof follows) that the red-black trees have a deep of no more than 2 $log(n_1 + 1)$

5

This is false because the leaves are placed in a depth between $log\ n$ and 2 $log\ n$, but the n_3 refers to the searches, which arent necesarily the trees structure, so this statement isn't always true.

6

This is true because in the red-black trees, all the paths from the root to the leaves have the same number of black nodes(black-height). Therefore all blacj nodes muts be equal bepths.