Solutions to Self Study 2 Exercises

CLRS 18-1

a. Obviously, with this primitive algorithm any sequence of n stack operations will require $\Theta(n)$ disk accesses and $\Theta(nm)$ CPU time.

b. Now you need to write a block only once every m PUSH operations. Thus, there are $\Theta(n/m)$ disk accesses and $\Theta(n)$ CPU time.

c. Here is an example worst-case sequence of stack operations: $m+1$ PUSH'es, 2 POP's, 2 PUSH'es, 2 POP's, 2 PUSH'es and so on. For the second POP, we will need to read the first disk page; for the second PUSH after that, we need to write that page back on disk again. Reading or writing of that page will happen every second operation. Thus, in the worst-case, a sequence of n stack operations will require $\Theta(n)$ disk accesses and $\Theta(nm)$ CPU time as in part *a*.

d. We maintain two pages in memory at the top of the stack. Let's call them *left* and *right*. In addition to the stack pointer p, we maintain a pointer c to the last word of the *left* page. We start with $p = 0$ and $c = m$. The invariant is maintained that $c - m \le p \le c + m$, that is, p always points to a word in one of the two pages, or to the element in the stack just before the first word of *left*.

```
PUSH(x)
 1  if p = c + m then        ▷ both memory pages are full, we can write the left one to disk
 2      DISK-WRITE(left)
 3      left ←right
 4      c ← c + m            ▷ move one page to the right
 5  p ← p + 1
 6  put x at the position in either left or right corresponding to p
```

```
POP()
 1  if p = c − m then        ▷ we emptied both pages; need to read in the page before left
 2      left ←DISK-READ(a page with words from p − m + 1 to p)
 3      c ← c − m            ▷ move one page to the left
 4  x ← a word at the position in either left or right corresponding to p
 5  p ← p − 1
 6  return x
```

It is quite obvious, that this has the required amortized costs, but let us analyze it formally. For the analysis of the amortized CPU cost of any sequence of operations on stack S, we can define a potential function $\Phi(S) = |c - p|$. It grows to m when p reaches either $c - m$ or $c + m$ and the CPU costs of disk reads or writes can be paid from the potential. After disk read or disk write the potential drops to 0 (but is increased to 1 in line 5 of either PUSH or POP).

Thus, if the i-th operation does not involve disk reading/writing, $\Delta\Phi_i = \pm 1$, depending on which operation is performed and whether p is smaller or larger than c. Then, $\hat{c}_i = 1 + \Delta\Phi_i = 1 \pm 1 = 0 \text{ or } 2$. If the i-th operation does involve disk reading/writing, $\Delta\Phi_i = \Phi_i - \Phi_{i-1} = 1 - m$ and $\hat{c}_i = m + 1 + \Delta\Phi_i = m + 1 + 1 - m = 2$.

In conclusion, the amortized CPU time is $O(1)$ per operation.

For the amortized I/O cost, we divide the potential function by m: $\Phi'(S) = |c - p|/m$. The rest of the analysis is analogous. The potential function grows from 0 to 1 and is 1 when disk read/write is performed, after which it drops to 0 (but is increased to $1/m$ in line 5).

If the i-th operation does not involve disk reading/writing, $\Delta\Phi'_i = \pm 1/m$. Then, $\hat{c}_i = 0 + \Delta\Phi'_i = \pm 1/m$. If the i-th operation does involve disk reading/writing, $\Delta\Phi'_i = \Phi'_i - \Phi'_{i-1} = 1/m - 1$ and $\hat{c}_i = 1 + \Delta\Phi'_i = 1 + 1/m - 1 = 1/m$. In conclusion, the amortized I/O cost is $O(1/m)$ per operation.

CLRS3 27-1 (CLRS4 26-1)

a. Here is the algorithm expressed using nested parallelism (initial call with $l = 1$ and $r = n$):

```
SUM-ARRAYS-MT(A, B, C, l, r)
 1  if l = r then
 2      C[l] ← A[l] + B[l]
 3  else
 4      q ← ⌊(l + r)/2⌋
 5      spawn SUM-ARRAYS-MT(A, B, C, l, q)
 6      SUM-ARRAYS-MT(A, B, C, q + 1, r)
 7      sync
```

The work of the algorithm is $\Theta(n)$, the span is $\Theta(\lg n)$, so the parallelism is $\Theta(n/\lg n)$.

b. When *grain-size* $= 1$, the parallelism is just 1, because the work and the span are both $\Theta(n)$.

c. Here is the analysis. The work is $\Theta(n)$, independent of the *grain-size*. The span is $n/$*grain-size* $+$ *grain-size*. To minimize the span, we differentiate it and find when the derivative is equal to 0. This gives that *grain-size* $= \sqrt{n}$ minimizes the span. The span is then $n/\sqrt{n} + \sqrt{n} = \sqrt{n} + \sqrt{n} = \Theta(\sqrt{n})$. The parallelism is work divided by span. Thus, when *grain-size* $= \sqrt{n}$, the parallelism is:

$$P = \frac{n}{\sqrt{n}} = \Theta(\sqrt{n}).$$

Comparing this with the parallelism in part *a*, we see that $n/\lg n = \omega(\sqrt{n})$ (To see it, observe that $\sqrt{n} = n/\sqrt{n}$ and remember that $\sqrt{n}$, as a polynomial function, dominates a logarithm). So the moral is that the standard "implementation" of parallel loops (as in part *a*) is the best way to parallelize loops using nested parallelism.

CLRS3 27.3-3 (CLRS4 26.3-3)

The idea is to do a divide-and-conquer algorithm which divides an array into two equal halves. In the combining step, one has to merge the two partitions. This is done by temporarily copying one part from each or the two produced partitions into a temporary array and then copying back into the original array.

Here is the algorithm (which assumes that $n > 1$). If q is returned, all elements in $A[1..q-1]$ are smaller than elements in $A[q..n]$.

```
PARALLEL-PARTITION(A[1..n])
 1  x ← A[n]        ▷ a pivot; after partitioning, A[n] will naturally be the last element of the right partition
 2  return PARALLEL-PARTITION-AUX(A, H, 1, n − 1, x)        ▷ H[1..n] is a temporary array
```

```
PARALLEL-PARTITION-AUX(A, H, l, r, x)
          ▷ if q is returned, ∀a ∈ A[l..q − 1] : a < x and ∀a ∈ A[q..r] : a ≥ x
 1    if l = r then
 2        if A[r] ≥ x then return r
 3        else return r + 1
 4    else
 5        k ← ⌈(l + r)/2⌉
 6        spawn q1 ←PARALLEL-PARTITION-AUX(A, H, l, k − 1, x)
 7        q2 ←PARALLEL-PARTITION-AUX(A, H, k, r, x)
 8        sync
 9        parallel for i = k to q2 − 1
10            H[q1 + i − k] ← A[i]        ▷ copy elements smaller than x from the right side
11        q ← q1 + (q2 − k)
12        parallel for i = q1 to k − 1
13            H[q + i − q1] ← A[i]        ▷ copy elements larger or equal than x from the left side
14        parallel for i = q1 to q2 − 1
15            A[i] ← H[i]        ▷ copy elements back to A
16        return q
```

Note that after the recursive calls in lines 6 and 7, we have the following four consecutive groups of elements in the array A between indexes l and r (some of the groups are possibly empty):

PartL1($A[l..q_1 - 1] < x$), PartL2($A[q_1..k - 1] \geq x$), PartR1($A[k..q_2 - 1] < x$), PartR2($A[q_2..r] \geq x$)

Lines 9–15 swap the places of parts $L2$ and $R1$ using the temporary array H. Parts $L1$ and $R2$ are already in their correct places in the array A.

Alternatively, instead of lines 9–15, one could write one parallel loop that reverses the sub-array $A[q_1..q_2 - 1]$ (parts $L2$ and $R1$). The loop swaps the elements using H as an array of temporary variables. Remember, you can not use one temporary variable for multiple parallel threads (race condition!).

```
 9    parallel for i = 1 to ⌊(q2 − q1)/2⌋
10        H[q1 − 1 + i] ← A[q1 − 1 + i]
11        A[q1 − 1 + i] ← A[q2 − i]
12        A[q2 − i] ← H[q1 − 1 + i]
```

The worst-case work of the algorithm can be described by the recurrence $W(n) = 2W(n/2) + \Theta(n)$ which solves to $\Theta(n \lg n)$ and is worse than for the non-parallel version ($\Theta(n)$).

The span of lines 9–15 of PARALLEL-PARTITION-AUX is $\Theta(\lg n)$. Thus, the recurrence for the total span of the algorithm is $S(n) = S(n/2) + \Theta(\lg n)$. The solution of this recurrence is $\Theta(\lg^2 n)$.

Thus, the parallelism is $W(n)/S(n) = \Theta(n/\lg n)$.

Actually, it is possible to achieve the same parallelism, but reduce the work to $\Theta(n)$ (the same as for the serial version) and the span to only $\Theta(\lg n)$ (using $\Theta(n)$ extra space—the same as for this solution). It requires some clever tricks, such as the technique described in part *d* of CLRS3 27-4 assignment.