Assignment 7, Flowcharts

Oleg Sivokon

<2017-12-29 Fri>

Contents

1 Sorting Lines Synchronously

This is an extremely simple task, since all the code was, essentially, written before me. All I need is to compose the following methods: `io.IOBase.readlines()` to `sorted()` to `io.IOBase.writelines()` in this exact order. First will read lines, second will sort them and the third will write them sorted.

Unfortunately, the example input file contains some junk (incorrect line delimiters and empty lines). Thus we also need to sanitize and filter what we read from `readlines()`.

1.1 Algorithm

Algorithm 1 Read and sort lines in a file

Require: *input* exists and is readable

1: **function** SYNC-READ-SORT($input, output$)
2: **return** $write(sort(filter(read(file))), output)$
3: **end function**

1.2 Implementation

```
# Assuming 'source' parameter is an already opened file,
# and the writing to file will be taken care of by another
# procedure:

def simple_sort_lines(source):
    return sorted(line for line in source if line.strip())
```

Some explanation may be due.

1. `sorted()` accepts iterators as well as lists, and in such case, will only read the values once, so no extra reading will be performed.

2. `sorted()` produces a list. It is possible, to write an algorithm which generates sorted lines on-line, without having to deal with the entire list at a time, but this would take too much space to write about here, and the implementation would be quite complex. However, I note that `nth_order_statistic()` could be used for such task.

3. File-like objects in Python are also iterators, and it just so conveniently happened that the iterator implemented by file-like objects iterates by line.

4. The lines thus produced have `\n` character appended to them. This is intentional, since `io.IOBase.writelines()` doesn't append it on its own.

2 Sorting Lines Asynchronously

This is a more interesting task. But the idea is still very simple: I can split the work between multiple workers, where each worker will sort its chunk of the lines it reads, then I could collect all chunks and sort them together.

The algorithm thus uses a popular in concurrent programming scheme: "Scatter-Join".

The non-trivial parts of this algorithm deal with ensuring that no worker reads the input intended for another worker, and that merging doesn't do too much work. First is ensured by that each worker will skip ahead to the first line break after seeking to an arbitrary position in file. This is while each worker will also read only full lines, while it is allowed to read more than the chunk allocated to it. This means that some parts of the file will be read more than once, and that the worker who gets to read the very beginning of the file is not allowed to skip the first line it reads.

2.1 Algorithm

I will only illustrate the concerns raised in the previous chapter:

We can easily see that we can pre-compute all the necessary information to make inner `while` loops independent, and so we can split the work between independent workers.

Algorithm 2 Ensure workload is distributed evenly and all lines are read

Require: $length(f)$ gives the length of the file f
Require: $seek(f, n)$ seeks to the position n in file f
Require: $position(f)$ gives the position at which f is being read
Require: $append(a, b)$ appends lists a and b
Require: $merge(lists)$ produces a sorted list with elements drawn from each list in $lists$

```
1: function DISTRIBUTE-WORK(input, workers)
2:     size ← length(input)
3:     lines ← Empty List
4:     for i ← 0; i < workers; i ← i + 1 do
5:         seek(input, i * size/workers)
6:         lines_i ← Empty List
7:         if i ≠ 0 then ▷ Skip line that will be read by the previous worker
8:             readline(input)
9:         end if
10:        while position(file) < (i + 1) * size/workers do
11:            lines_i ← append(lines_i, readline(input))
12:        end while
13:        lines ← append(lines, lines_i)
14:    end for
15:    return merge(lines)
16: end function
```

Algorithm 3 Merges sorted arrays A and B to obtain third sorted array C

Require: $length(X)$ gives the length of array X

```
1: function MERGE(A, B)
2:     alength ← length(A)
3:     blength ← length(B)
4:     apos ← 0
5:     bpos ← 0
6:     clength ← alength + blength
7:     C ← array(clength)
8:     cpos ← 0
9:     while alength > apos ∧ blength > bpos do
10:        if A_apos ≤ B_bpos then
11:            C_cpos ← A_apos
12:            apos ← apos + 1
13:        else
14:            C_cpos ← B_bpos
15:            bpos ← bpos + 1
16:        end if
17:        cpos ← cpos + 1
18:    end while
19:    if apos = alength then
20:        remainder ← B
21:        rpos ← bpos
22:    else
23:        remainder ← A
24:        rpos ← apos
25:    end if
26:    while rpos < length(remainder) do
27:        C_cpos ← remainder_rpos
28:        cpos ← cpos + 1
29:        rpos ← rpos + 1
30:    end while
31:    return C
32: end function
```

The `merge()` function is the same one used in textbook `merge-sort()` procedure. For completeness, the pseudocode is given below:

2.2 Implementation

The implementation of the algorithm given above:

```
def merge_sync(left, right):
    left_pos, right_pos = 0, 0
    left_size, right_size = len(left), len(right)
    result, remainder = [None] * (left_size + right_size), None
    result_pos, remainder_pos = 0, 0

    while left_pos < left_size and right_pos < right_size:
        if left[left_pos] <= right[right_pos]:
            result[result_pos] = left[left_pos]
            left_pos += 1
        else:
            result[result_pos] = right[right_pos]
            right_pos += 1
        result_pos += 1

    if left_pos < left_size:
        remainder = left
        remainder_pos = left_pos
    else:
        remainder = right
        remainder_pos = right_pos

    while result_pos < len(result):
        result[result_pos] = remainder[remainder_pos]
        result_pos += 1
        remainder_pos += 1
    return result


async def sort_chunk(queue, source, start, end):
    chunk = []
    source.seek(start)
    source.readline()
    while source.tell() < end:
        line = source.readline().strip()
        if line:
            chunk.append(line + '\n')
    await queue.put(sorted(chunk))


async def merge(queue, expected_count):
    result = []
    for _ in range(expected_count):
        result = merge_sync(result, await queue.get())
    return result
```

```
def async_merge_sort_lines(source, fsize, coroutines_count=0):
    coroutines_count = coroutines_count or multiprocessing.cpu_count() - 1

    with closing(asyncio.new_event_loop()) as loop:
        queue = asyncio.Queue(loop=loop)
        chunk_size = (fsize // coroutines_count) + 1
        readers = [
            sort_chunk(queue, source, x, min(x + chunk_size, fsize))
            for x in range(0, fsize, chunk_size)
        ]
        writer = merge(queue, coroutines_count)
        future = asyncio.gather(*(readers + [writer]), loop=loop)
        loop.run_until_complete(future)
        return future.result()[-1]
```