Two Pivot Block Quicksort and Mergesort Comparison

Valerian Salim 2106630012
Kode asdos 3
DAA C

November 2023

1 Pakta Integritas

Dengan ini saya menyatakan bahwa TE ini adalah hasil pekerjaan saya sendiri.

2 Pendahuluan

Pada eksperimen ini, akan dibandingkan algoritma yang memanfaatkan konsep **Two-Pivot-Block-Quicksort** (Block Quicksort dengan dua pivot) dengan **Mergesort**.

2.1 Two-Pivot-Block-Quicksort

Two-Pivot-Block-Quicksort merupakan variasi dari algoritma **Quicksort dengan Lomuto's Partitioning**. Algoritma Block Quicksort dikembangkan untuk memperbaiki kekurangan Quicksort yaitu *branch misprediction*. Branch Prediction merupakan konsep yang digunakan CPU modern untuk mempercepat eksekusi program pada CPU dengan mengeksekusi cabang instruksi secara spekulatif. Sifat spekulatif tersebut merupakan masalah yang muncul pada algoritma Quicksort, karena dengan terjadinya *branch misprediction*, proses eksekusi yang telah dijalankan oleh CPU secara spekulatif akan dibuang dan diulangi prosesnya dari awal lagi. *Branch misprediction* sendiri merupakan tantangan inheren dari desain CPU modern pada *branching statement* seperti if.

Untuk elaborasi yang lebih lanjut terhadap masalah *branch misprediction* pada partisi Quicksort yang biasanya digunakan, akan digunakan pseudocode di bawah.

Algorithm 1 Quicksort Partition

```
1: for each element in the array do
2:     if element < pivot then
3:         move element to the 'less than pivot' side
4:     else
5:         move element to the 'greater than pivot' side
6:     end if
7: end for
```

Konsep *branch prediction* akan mencoba menjalankan salah satu state yang ada meskipun *guard element* $<$ *pivot* belum selesai terevaluasi. Pada beberapa kasus input, *branch misprediction* akan secara konsisten terjadi terus menerus, contohnya jika data yang diproses telah *sorted* secara menaik maupun menurun. Hal tersebut juga akan menyebabkan performa dari Quicksort menurun secara signifikan.

Algoritma Block Quicksort sendiri berusaha untuk mengurangi terjadinya *branch misprediction* dengan menghilangkan statement if pada algoritma Quicksort Partition diatas. Agar partitioning dapat dilakukan pada Block Quicksort, semua element pada data akan dipisah ke masing-masing block dengan ukuran konstan. Kemudian, setiap bilangan pada block tersebut akan dibandingkan dengan pivot, dan hasil dari perbandingan tersebut akan disimpan pada buffer. Setelah semua bilangan telah dievaluasi, akan diiterasi buffer tersebut dan dilakukan penyusunan kembali.

Perhatikan pada algoritma tersebut, tidak terjadi *branching* dikarenakan evaluasi masing-masing bilangan tersebut memanfaatkan fakta bahwa statement seperti $eval = (element \leq pivot)$ yang pada tingkat *low-level* tidak menimbulkan *branching* pada CPU.

2.2 Mergesort

Mergesort merupakan algoritma yang populer digunakan. Mergesort bekerja dengan konsep *Divide and Conquer (DnC)* yang akan memisahkan array menjadi dua bagian yang sama panjang, yang masing-masing akan dipisahkan kembali dan pada akhirnya akan digabung dan diurutkan.

3 Analisis Algoritma

3.1 Mergesort

Secara *time complexity*, Mergesort akan memiliki running time rekurens berupa,

$$T(n) = 2 \cdot T(n/2) + f(n), \text{ dengan kompleksitas } f(n) = O(n),$$

yang dapat dianalisa menggunakan *Master's Theorem* maupun *Recurrence Tree* menghasilkan running time complexity sebesar $\Theta(n \log n)$.

Secara *memory complexity*, jika diimplementasikan dengan rekursi, Mergesort akan menggunakan sebanyak $O(\log n)$ stack rekursi. Namun untuk proses *merging*, diperlukan alokasi memori sebesar $O(n)$. Sehingga dapat dikatakan space complexity Mergesort adalah $O(n)$.

3.2 Two Pivot Block Quicksort

Secara *time complexity*, Two Pivot Block Quicksort akan menghasilkan *worst case running time* sebesar $O(n^2)$ yang akan terjadi ketika elemen-elemennya telah terurut secara menaik maupun menurun. Hal tersebut terjadi karena pemilihan pivot pada algoritma tersebut yang serupa dengan Lomuto. Namun pada elemen-elemen dengan urutan random, Two Pivot Block Quicksort akan memiliki kompleksitas $O(n \log n)$.

Secara *space complexity*, Two Pivot Block Quicksort yang diimplementasikan dengan rekursi, akan menggunakan sebanyak $O(n)$ stack rekursi. Quicksort sendiri merupakan algoritma *in-place* sehingga tidak diperlukan alokasi memori baru. Sehingga space complexity Two Pivot Block Quicksort adalah $O(n)$.

4 Hasil Eksperimen

Pada percobaan ini terdapat 9 dataset yang diuji, yaitu dataset berjumlah 2^9 bilangan, 2^{13} bilangan, dan 2^{16} bilangan, dengan 3 variasi dari masing-masing dataset, yaitu sorted, reversed, dan random.

Untuk hasil dari masing-masing algoritma dapat dilihat pada tabel berikut,

4.1 Running Time

Algoritma	$N = 2^9$ (ms)			$N = 2^{13}$ (ms)			$N = 2^{16}$ (ms)		
	sorted	rand	reversed	sorted	rand	reversed	sorted	rand	reversed
Mergesort	0.77	0.82	0.73	15.71	25.54	17.88	153.26	252.00	165.40
Two-Pivot-Block	21.10	1.38	19.70	6427.44	44.72	6845.22	410633.67	314.82	397054.00

Table 1: Running Time Benchmark

Dapat dilihat dari tabel tersebut, bahwa *running time* dari Two-Pivot-Block-Quicksort masih lebih lambat dari Mergesort. Perbedaannya tidak begitu signifikan untuk dataset yang random, namun untuk dataset yang sudah terurut, perbedaannya sangat signifikan. Hal tersebut terjadi karena algoritma Quicksort memiliki *running time* terburuk pada dataset terurut, yang menyebabkan kompleksitasnya menjadi $O(n^2)$.

4.2 Memory Usage

Algoritma	$N = 2^9$ (MiB)			$N = 2^{13}$ (MiB)			$N = 2^{16}$ (MiB)		
	sorted	rand	reversed	sorted	rand	reversed	sorted	rand	reversed
Mergesort	22.25	22.25	22.25	22.25	22.25	19.28	23.92	25.32	24.93
Two-Pivot-Block	22.25	22.25	22.25	22.25	22.25	19.28	24.00	25.32	24.96

Table 2: Memory Usage Benchmark

Dari tabel diatas, terlihat bahwa memory usage dari kedua algoritma relatif sama. Hal tersebut sesuai dengan analisis *space complexity* pada subbab ke-3, yang mana masing-masing algoritma memiliki *space complexity* sebesar $O(n)$.

5 Implementasi

Untuk implementasi masing-masing algoritma tersebut, tercantum di github.com/valeelim/tep-1.

5.1 Mergesort

5.2 Two-Pivot-Block-QuickSort

Pada eksperimen ini, implementasi Two-Pivot-Block-Quicksort dilakukan menggunakan stack, bukan rekursi. Hal tersebut dilakukan karena pada dataset terurut, Quicksort tersebut memanggil sebanyak $O(n)$ stack rekurens yang menyebabkan *segmentation fault.*

Algorithm 2 Mergesort

```
1: function MERGESORT(arr, left, right)
2:     if left == right then
3:         return arr
4:     end if
5:     middle ← ⌊(left + right)/2⌋
6:     left_arr ← MERGESORT(arr, left, middle)
7:     right_arr ← MERGESORT(arr, middle + 1, right)
8:     return MERGE(left_arr, right_arr)
9: end function
```

Algorithm 3 Two Pivot Block Lomuto Quicksort

```
1: function TWOPIVOTBLOCKLOMUTO(arr, l, r)
2:     rec_stack ← [(l, r)]
3:     while len(rec_stack) > 0 do
4:         (l, r) ← rec_stack.pop()
5:         (i, j) ← PARTITION(arr, l, r)
6:         if l < i − 1 then
7:             rec_stack.append((l, i − 1))
8:         end if
9:         if i + 1 < j − 1 then
10:            rec_stack.append((i + 1, j − 1))
11:        end if
12:        if j + 1 < r then
13:            rec_stack.append((j + 1, r))
14:        end if
15:    end while
16:    return arr
17: end function
```

Algorithm 4 Partition Procedure for Two Pivot Block Lomuto

```
1: function PARTITION(arr, l, r)
2:     n ← r − l + 1
3:     if n ≤ 1 then
4:         return (l, r)
5:     end if
6:     if arr[l] > arr[r] then
7:         arr[l], arr[r] ← arr[r], arr[l]
8:     end if
9:     p, q ← arr[l], arr[r]
10:    block ← new array (BLOCK_SIZE)
11:    i, j ← l + 1
12:    k ← 1
13:    nlp, nlq ← 0
14:    while k < n − 1 do
15:        t ← min(BLOCK_SIZE, n − k − 1)
16:        for c ← 0 to t − 1 do
17:            block[nlq] ← c
18:            nlq ← nlq + (q ≥ arr[k + c + l])
19:        end for
20:        for c ← 0 to nlq − 1 do
21:            arr[j + c], arr[k + block[c] + l] ← arr[k + block[c] + l], arr[j + c]
22:        end for
23:        k ← k + t
24:        for c ← 0 to nlq − 1 do
25:            block[nlp] ← c
26:            nlp ← nlp + (p > arr[j + c])
27:        end for
28:        for c ← 0 to nlp − 1 do
29:            arr[i], arr[j + block[c]] ← arr[j + block[c]], arr[i]
30:            i ← i + 1
31:        end for
32:        j ← j + nlq
33:        nlp, nlq ← 0
34:    end while
35:    arr[i − 1], arr[l] ← arr[l], arr[i − 1]
36:    arr[j], arr[r] ← arr[r], arr[j]
37:    return (i − 1, j)
38: end function
```