CS 433 (2) HW1 Report

Boris Nikulin

2019-02-11

Contents

1 Data Analysis

1.1 Data Import

First we load the benchmark data and give the data a once over.

```
library(readr)
library(dplyr)
library(magrittr)

data <- read_csv('./bench_ready_queue.csv',
    col_types = cols(
        run = col_integer(),
        time = col_integer()
    )
)

data %<>% mutate(run = as.factor(run))
```

```
glimpse(data, width = 60)
```

```
## Observations: 20
## Variables: 2
## $ run  <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,...
## $ time <int> 52, 50, 51, 49, 51, 51, 52, 50, 52, 52, 52...
```

First we compute a summary and standard deviation of the times. The times are in milliseconds.

```
summary(data$time)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    49.0    50.0    51.0    51.2    52.0    55.0
```

```
sd(data$time)
```

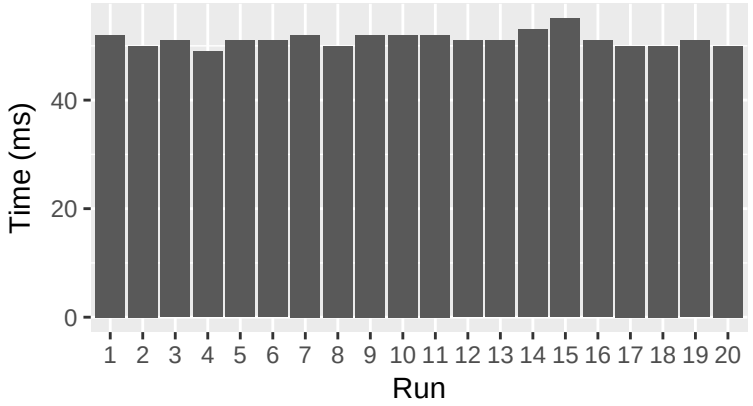
```
## [1] 1.321881
```

1.2 Data Visualization

Next we plot the data as a bar plot, a box plot, and as a density plot.

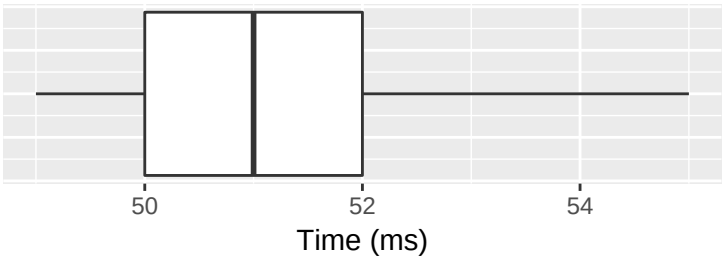
```
library(ggplot2)

ggplot(data, aes(run, time)) +
    geom_bar(stat = 'identity') +
    labs(x = 'Run', y = 'Time (ms)') +
    theme(aspect.ratio = 0.5)
```



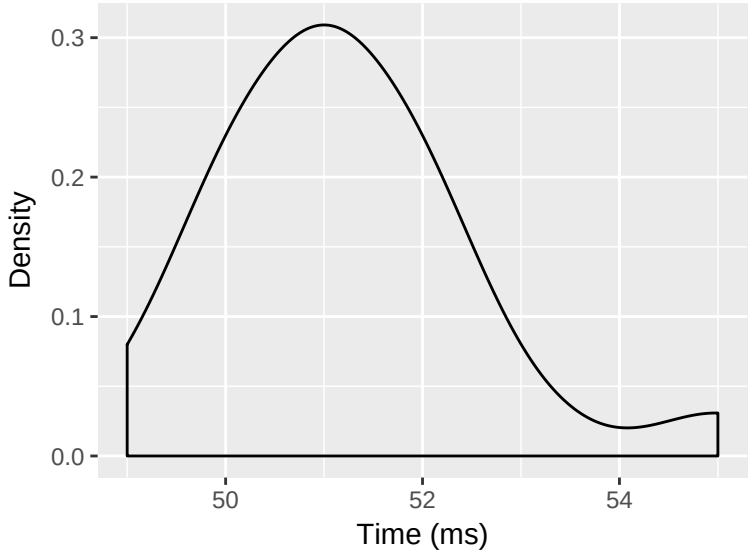
The box plot displays the raw data. The raw data shows that the runs do not vary by much.

```
ggplot(data, aes(1, time)) +
    geom_boxplot() +
    coord_flip() +
    labs(y = 'Time (ms)') +
    theme(aspect.ratio = 0.25,
        # x axis was being cut off on the right
        plot.margin = margin(r = 16),
        axis.title.y = element_blank(),
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank()
   )
```



The box plot shows very stable benchmark results.

```
ggplot(data, aes(time)) +
    geom_density() +
    labs(x = 'Time (ms)', y = 'Density')
```



The density plot further shows the uniformness of the data.

1.3 Conclusion

The median is likely to be more resistant to benchmark computer slow downs and is 51 ms. For completeness it is important to note the average of 51.2 ms with a standard deviation of 1.32 ms.

2 Data Structure Choice

2.1 Rationale

I chose a binary heap for the ready queue. A binary heap is a good choice for a priority queue because insertion and deletion is $\mathcal{O}(\log n)$. Once inserted, finding the max or min element is $\mathcal{O}(1)$. These two properties are why a binary heap make good priority queues.

2.2 Implementation

For my implementation I tried to model the Standard Template Library (STL), except for allocators. The data structures I made can be used in functions of the C++ `algorithm` header, where applicable, primarily by giving my structures STL compatible iterators.

The priority queue backing the ready queue is made similarly to the STL `priority_queue`, where I implemented the priority queue as an adaptor on top of an existing container through random access iterators. The priory queue

is backed, by default, by my `Vector`. The elements are inserted according to a max heap through the use of iterator aware heap algorithms. The priority queue is a max binary heap.

3 Empirical and Theoretical Results Compared

Since the ready queue is backed by a binary heap, I do not expect the time to be too high. Using "Latency Numbers Every Programmer Should Know" with the year 2019, one can expect an L1 cache hit to be 1 ns. Taking into account large modern L1 cache sizes and the fact that the benchmark is only using 20 PCBs, it is reasonable to expect most, if not all, accesses of the data to be an L1 cache hit. Adding or removing a node requires following a pointer so there are at least a million L1 cache hits or 1 ms. Adding to the ready queue requires, at worst, traversing through $\log_2 20 = 4.32$ elements and potentially branching. A branch miss prediction is 7 ns. If all 4.32 comparisons are miss predicted one million times, then that would be $7 \cdot 4.32 = 30.24$ ms plus the L1 cache hits of 4.32 ms. To find which PCB to add, I do a linear search of at most 20 elements. This adds twenty million L1 cache hits roughly or 20 ms. The current total is 55.56 ms. Although only 50% of the time one adds, this still shows that the rough estimate of the time should be in the 50's of ms. The other half of the time we remove which is similar to adding, but does include a linear search into the PCB table to find something to add. This comes to the same $7 \cdot 4.32 + 4.32 = 34.56$ ms.

This very rough ballpark estimate shows that around 50 ms should be expected and is in the same ball park as the benchmark.

4 Post Script

The test data was much more interesting on my laptop (around 50 ms with standard deviations of 0.5 ms to 5 ms forming a right skewed distribution). The graphs now seem pointless …