Square Root Decomposition Data Structure

Daniel Enrico Cahall

October 8, 2017

1 Introduction

The idea behind this algorithm is to reduce the runtime when processing data that is stored in a statically sized array. While the original proposal was sensors, the concept is abstract enough to utilize in other applications. The algorithm itself is not new (see references 1, 2, 3), but as far as I know, there are no applications of the algorithm in the literature, and there is also room for modification/expansion.

2 Theory

Suppose we have an array A of simple numeric primitives (integers, bits, doubles, etc.) which is of size n.

$$A = [x_0, x_1, x_2, ...x_{n-1}]$$

We can create another array S which stores the sum of k-sized partitions of A which is of size $\frac{n}{k}$.

$$S = [\sum_{i=0}^{k-1} x_i, \sum_{i=k}^{2k-1} x_i, ... \sum_{i=n-(k+1)}^{n-1} x_i]$$

If we need to compute the sum of any subsection of the A, we no longer need to sum each individual element - we can now utilize S, which should reduce the computations down by a factor of k. However, we still need to select k such that we minimize the total computation time.

To derive this, let's first analyze the best case runtime. Assume we want to compute the sum of all elements within the array. We can just utilize S, and our computation time is reduced to $\frac{n}{k}$.

$$T(n, k) = \frac{n}{k}$$

Now, let's assume the worst case scenario: suppose we want to find the sum from elements *1* to *n - 2*. While we can use S for the sum of all the elements within partitions 1 to $\frac{n}{k} - 1$, the sum of the partition 0 and last partitions will need to be computed manually, which will require *k-1* computations each:

$$T(n, k) = \frac{n - 2k}{k} + (k - 1) + (k - 1) = \frac{n}{k} - 2 + 2(k - 1)$$

To minimize this function, we differentiate our function with respect to our parameter k, and set it to 0:

$$\frac{dT(n, k)}{dk} = \frac{-n}{k^2} + 2 = 0$$

Our optimal works out to be $\sqrt{(\frac{n}{2})}$. Plugging that back into our original equation, we get:

$$T(n) = 2\sqrt{(2n)} - 4$$

Which is $O(\sqrt{n})$ - as $n \to \inf$, the arbitrary constants become irrelevant.

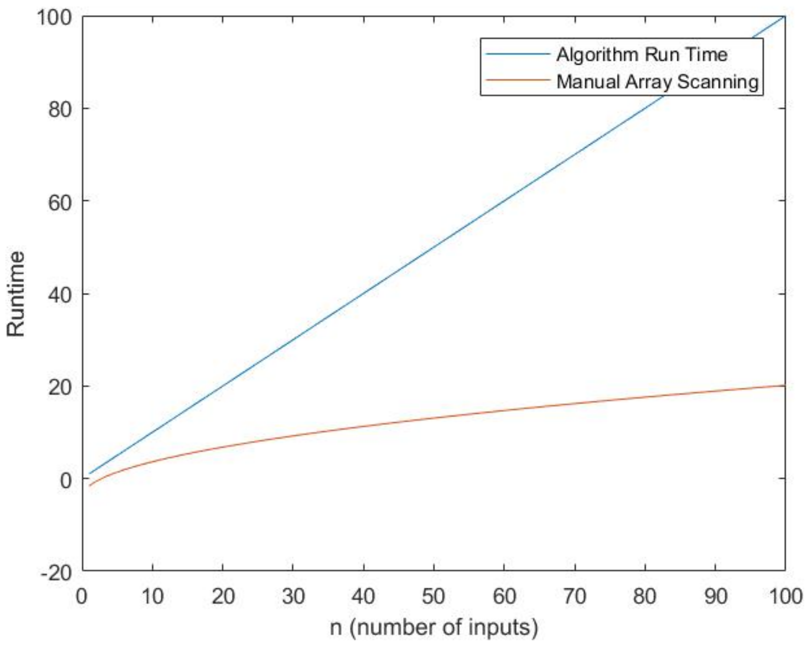

Figure 1: Comparison of runtimes for n from 0 to 1000

3 Algorithm Modifications and Applications

The first simple modification that can be made is instead of treating S as an accumulator array (storing the sum of a range in A), it can be treated as a compounder array (storing the product of a range of elements in A). Similarly, it can store the squared sum, the logical AND/OR results (in the case of a bit vector), or a variety of other quantities.

Secondly, I believe this can be applied to sensor data. Suppose you have a microcontroller which controls and monitors many sensors. It sends commands to the sensors, or maybe it just pings them to make sure they're alive. If you send a known number of n sequential orders with associated IDs/hash, A can be a bit vector, initialized to all 0s. Whenever a response is received (i.e; the sensor was queried for data, pinged to ensure it's still running, etc.), the response should have the same ID as the order which executed it. If there is a bit in the bit vector which represents that ID, we can flip it to a 1 to indicate that a valid response was received. After a certain amount of time has passed, we assume that all orders should have had a response (all the sensors will respond in some finite amount of time), and we can check the bit vector. Ideally, each element in S should be equal to k (that would indicate that each bit within that range has been flipped). In the case it's not k, one or more orders were not responded to, and therefore we would be detecting faults within the sensor network. From there, we can navigate back to the A and determine the order which did not have a response.

In this application, the runtime equation looks a bit different:

$$T(n,k) = \frac{n}{k} - c + ck$$

Where c is the number of elements in S which have faults, assuming we run through all k elements in the subsection of A searching for the index (or indices) of the fault. In the ideal case, all sensors in the network are functioning and responded to all tasks, c = 0, and our runtime is:

$$T(n,k) = \frac{n}{k}$$

In the worst case scenario, all sensors are broken and not responding to any tasks, $c = \frac{n}{k}$::

$$T(n,k) = \frac{n}{k} - \frac{n}{k} + \frac{n}{k}k = n$$

For all values of c in between ($1 \leq$ c $\leq \frac{n}{k}$-1) , we can find the optimal k by setting the derivative of our runtime function equal to 0:

$$\frac{dT(n,k)}{dk} = \frac{-n}{k^2} + c = 0$$

It can be seen that the optimal $k = \sqrt{\frac{n}{c}}$. Plugging that back into the original function:

$$T(n) = 2\sqrt{cn} - c$$

If we assume c is low (few to no faults within our network), we can approximate our optimal k as $\sqrt{n}$.

3.1 Benefits

1. Even in the worse case scenario, the algorithm will run in the same amount of time as the naive solution (checking each individual bit), and in the best case/average case, there should be a noticeable performance improvement. This can be beneficial to time critical applications
2. If the A and S array are populated in real time (the bits are flipped as the sensor data is acquired) the algorithm preparation time is nonexistent. And within the algorithm, there are no computationally heavy tasks - only array assignments, some logical operators, and addition.
3. Using a bit vector is also memory efficient, which can be critical when optimizing code for a microcontroller. This becomes especially relevant if the controller needs to collect large amounts of sensor data, or if it is to be deployed in a location where changes cannot be made easily.
4. Implementation should be fairly simple in C using an unsigned char array and a uint8 array.
5. With additional information about the sensors and the tasks that need to be executed, further optimizations can be made. The parameter c is really a function of sensor failure rates, and is also determined by how the task IDs are partitioned.

3.2 Detriments

1. There is additional memory which needs to be allocated for creating an additional array. However, S should only occupy $\sqrt{n}$ bits. Even if 1 million measurements were being taken (n = 1,000,000), only 1000 bits (125 bytes) would be used.
2. Although in a majority of scenarios there will be performance improvements, the algorithm doesn't prevent the worst case scenario. Additional logic can be added to improve on that worst case, although currently I don't think it can be brought below *O(n)*.
3. Whether there is one fault or k faults in the network for each section of the array A, the runtime will be the same, assuming we check every index. Additional logic can be placed in there to break out once all faults have been found. In the case that all k tasks of a subsection of A failed, the corresponding location in S would be 0, and there would be no need to check each individual index.

4 Algorithm Outline

Algorithm 2 Process Arrays

```
for $i \in \{0, \ldots, \sqrt{n} - 1\}$ do
  if S[i] != $\sqrt{n}$ then
    for $j \in \{(i)\sqrt{n}, \ldots, (i+1)\sqrt{n}\}$ do
      if A[j] != 1 then
        //Do something with A[j]
      end if
    end for
  end if
end for
```

Algorithm 1 Populate Arrays

```
1: Orders[n] = $[o_0, o_1, o_2, ...o_{n-1}]$
2: A[n] = [0, 0, 0, .... 0]
3: S[$\sqrt{n}$ ] = [0, 0, 0, .... 0]
4: assignIDsToOrders()//assign IDs to orders based on the sequence they should be sent, the
   responses will have the same associated ID
5: while $taskingExecutes()$ do
6:     if (response $r_x$ to an order $o_x$ is received) then
7:         i = IDToIndexConversion($r_x$) //use this value to index into array
8:         //NOTE: for simplicity,assign order ID to index values
9:         A[i] = 1 //Indicate a response was received in the bit vector
10:        S[floor($\frac{i}{\sqrt{n}}$)] += 1 //Add to sum array
11:    end if
12: end while
```

5 References

[1] http://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/
[2] https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/
[3] http://www.infoarena.ro/blog/square-root-trick