Fast sorting data from the Unified Host and Network Data Set

Sergey V.

September 7, 2024

1 Introduction

This document describes a simple approach for rapidly sorting many rows ($\geq 10K$) of data from the Unified host and network data set [1]. Modern cyberattacks often involve a complex interplay between malicious actions on individual hosts (computers, servers) and anomalous activity on the network. A unified host and network dataset allows security analysts to connect the dots, track attack progressions across different systems, and gain a more holistic understanding of the threat landscape. By correlating host-based events (e.g., process creation, file modifications, login attempts) with network traffic patterns (e.g., unusual connections, data exfiltration), security tools and analysts can identify suspicious activity that might go unnoticed when analyzing each dataset in isolation. This enables faster and more accurate threat detection and response. Sorting allows for quick and efficient searching and filtering of data based on specific criteria (e.g., timestamps, IP addresses, event types). This is necessary for real-time threat detection and incident response, where analysts need to quickly identify relevant information in a massive stream of data. The data set consists of rows with the following fields, some text based (such as device names) and the rest numerical:

Field Name	**Description**
Time	The start time of the event in epoch time format.
Duration	The duration of the event in seconds.
SrcDevice	The device that likely initiated the event.
DstDevice	The receiving device.
Protocol	The protocol number.
SrcPort	The port used by the *SrcDevice*.
DstPort	The port used by the *DstDevice*.
SrcPackets	The number of packets the *SrcDevice* sent during the event.
DstPackets	The number of packets the *DstDevice* sent during the event.
SrcBytes	The number of bytes the *SrcDevice* sent during the event.
DstBytes	The number of bytes the *DstDevice* sent during the event.

Figure 1: Data fields in each row

2 Representation

We must take rows of such data and transform it to a form which can be readily sorted. We should recognize that in most cases, we can enumerate all the device names. This allows to map them to a set of integers. In fact, we can represent the entire data set with only integer values. In order to generate rows of such data, we first generate a set of device names, then use direct mappings (dictionaries) to convert device names to integers and back.

```
devices = [f"Device_{i}" for i in range(1, num_devices)]
device_to_int = {name: i for i, name in enumerate(devices)}
int_to_device = {i: name for name, i in device_to_int.items()}

data = {
    "Time": generate_epoch_time(num_samples),
    "Duration": np.random.randint(1, 1800, size=num_samples),  # Duration (sec.)
    "SrcDevice": [random.choice(devices) for _ in range(num_samples)], # Src dev string
    "DstDevice": [random.choice(devices) for _ in range(num_samples)], # Dst dev string
    "Protocol": np.random.choice([6, 17, 1], num_samples),  # TCP, UDP, ICMP, etc
    "SrcPort": np.random.randint(1024, 65535, num_samples),
    "DstPort": np.random.randint(1024, 65535, num_samples),
    "SrcPackets": np.random.randint(1, 1000, num_samples),
    "DstPackets": np.random.randint(1, 1000, num_samples),
    "SrcBytes": np.random.randint(20, 1000000, num_samples),
    "DstBytes": np.random.randint(20, 1000000, num_samples)
}

# Create DataFrame
df = pd.DataFrame(data)

# Generate the full string representation of each row
df['FullString'] = df.apply(lambda row: ' '.join(str(x) for x in row), axis=1)
```

Next, we can write the transform functions and generate the interger representation of the data:

```
# replace specific string fields with their integer mappings
def full_string_to_integer_string(full_string):
    parts = full_string.split()
    parts[2] = str(device_to_int[parts[2]])  # SrcDevice from string to int ID
    parts[3] = str(device_to_int[parts[3]])  # DstDevice from string to int ID
    return ' '.join(parts)

# replace integer IDs with their string representations
def integer_string_to_full_string(integer_string):
    parts = integer_string.split()
    parts[2] = int_to_device[int(parts[2])]  # SrcDevice ID back to string
```

```
    parts[3] = int_to_device[int(parts[3])]  # DstDevice ID back to string
    return ' '.join(parts)

# Generate the integer string representation of each row using the conversion function
df['IntString'] = df['FullString'].apply(full_string_to_integer_string)
```

Here is how an original and transformed row of the data looks like:

```
Full string: 1686469509 711 Device_74 Device_71 17 13414 53370 294 49 985314 944662
Invertible integer rep.: 1686469509 711 73 70 17 13414 53370 294 49 985314 944662
```

With the simple approach above, we can generate rows of the all integer string representation.

3 Sorting

We can consider two different approaches to sorting: one that uses a merge-based sorting algorithm applied to string representations of data and another that directly operates on integer arrays. To sort many rows of the Unified host and network data set, it is best to first convert all the rows to integer form, as discussed above. This is the case even if string sorting is to be used. Notice that we actually can encode every row as one big integer value (essentially, an int64). We can do this by recording where the spaces occur in the string and then removing all the spaces. It is probably best to avoid converting to a single integer per row, as that would necessitate operations with int64 values, which may be slower than handling multiple int32s. First, we consider an iterative parallel merge sort for strings with C++. This leverages an optimized library function for the in place merge. Before we can effectively use the iterativeParallelMerge() or iterativeParallelMergeInteger() functions, we need to ensure that the data within individual segments or "buckets" is pre-sorted. This pre-sorting stage is critical because the merge process relies on each segment being already sorted to function correctly. The sorting can be done by individual threads, sorting individual parts of the data (sets of rows). Essentially, for both string and integer data, we employ the following bucket sort and merge approach. Initially, the array is divided into several "buckets" (subarrays), each of which is sorted independently, in parallel. After all buckets are sorted, iterativeParallelMerge() takes over to merge these sorted subarrays. Initially, it merges small subarrays that are just a few elements each, but as the iterations progress, it merges larger and larger subarrays, until the entire array is sorted.

```
// Sort a portion of a vector
void sortChunk(vector<string>& lines, int start, int end) {
  sort(lines.begin() + start, lines.begin() + end);
}

// Sort buckets of the data
void sortStringBuckets(vector<string>& lines, int numThreads) {
```

```
    int n = lines.size();
    int chunkSize = n / numThreads; // Calculate chunk size
    vector<thread> threads; // To hold all threads

    for (int i = 0; i < numThreads; i++) {
        int start = i * chunkSize;
                                // Handle last chunk
        int end = (i == numThreads - 1) ? n : (start + chunkSize);
                                // Create thread to sort the chunk
        threads.emplace_back(sortChunk, ref(lines), start, end);
    }

    for (auto& thread : threads) {
                                // Wait for threads to complete
        thread.join();
    }
}
```

Once this is done, the merge function, can merge the individual sorted chunks into a fully sorted array.

```
void iterativeParallelMerge(vector<string>& lines) {
    int n = lines.size();
    vector<string> temp(n);
    for (int size = 1; size < n; size *= 2) {
        #pragma omp parallel for schedule(static, 1)
        for (int i = 0; i < n - size; i += 2 * size) {
            int left = i;
            int mid = min(i + size - 1, n - 1);
            int right = min(i + 2 * size - 1, n - 1);
            if (size > 256) {
                inplace_merge(lines.begin() + left, lines.begin() + mid + 1,
lines.begin() + right + 1);
            } else {
                int k = left, j = mid + 1;
                for (int l = left; l <= right; ++l) {
                    if (k > mid) temp[l] = move(lines[j++]);
                    else if (j > right) temp[l] = move(lines[k++]);
                    else if (lines[k] < lines[j]) temp[l] = move(lines[k++]);
                    else temp[l] = move(lines[j++]);
                }
                move(temp.begin() + left, temp.begin() + right + 1,
lines.begin() + left);
            }
        }
    }
}
```

The function uses a loop where the size of the segments to be merged doubles with each iteration (size = 2). This doubling is characteristic of bottom-up merge sort, where you start with small segments (initially considering individual elements as sorted segments) and progressively merge them into larger sorted segments. Inside the loop, a parallel for loop (facilitated by OpenMP) iterates through the array in chunks defined by the current segment size (2 x size). The merging of these segments is the computationally intensive part that is parallelized to improve performance. For larger segments (size greater than a threshold, e.g. 256), the function uses the STL library function std::inplace_merge, that efficiently merges two consecutive sorted ranges into a single sorted range, in-place. Operations like std::inplace_merge would be more efficient with integers, due to their fixed and small size which enhances CPU cache utilization. The manual merge step would also be faster because moving integers is cheaper in terms of processing time and memory bandwidth.

The alternate version, is designed specifically for integer data:

```
void mergeIntegers(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }
    while (i <= mid) {
        temp[k++] = arr[i++];
    }
    while (j <= right) {
        temp[k++] = arr[j++];
    }
    for (i = left, k = 0; i <= right; ++i, ++k) {
        arr[i] = temp[k];
    }
}


void iterativeParallelMergeInteger(vector<int>& numbers) {
    int n = numbers.size();
    const int inplaceMergeThreshold = 1024;
    for (int size = 1; size < n; size *= 2) {
        #pragma omp parallel for schedule(static, 1)
        for (int i = 0; i < n - size; i += 2 * size) {
            int left = i;
            int mid = std::min(i + size - 1, n - 1);
```

```
            int right = std::min(i + 2 * size - 1, n - 1);
            if (size > inplaceMergeThreshold) {
                std::inplace_merge(numbers.begin() + left, numbers.begin() + mid + 1,
numbers.begin() + right + 1);
            } else {
                mergeIntegers(numbers, left, mid, right);
            }
        }
    }
}
```

The function iterativeParallelMergeInteger applies a parallel merge sort to an array of integers, leveraging OpenMP for concurrency. As before, the function is structured to handle large datasets efficiently by parallelizing the merge steps and using an optimized library function, std::inplace_merge, for larger segments. The for loop increases the size of the segments to merge exponentially. It starts by merging individual elements and progressively merges larger segments. The function mergeIntegers merges two contiguous subarrays within an array. It is a classic merge operation from the merge sort algorithm. A temporary vector temp is created to hold the merged result of the two subarrays defined from left to mid and from mid+1 to right. Two pointers i and j are initiated to traverse the two subarrays. i starts at left and goes up to mid, while j starts at mid+1 and goes up to right. A third pointer k is used to index into temp. The elements from the two subarrays are compared: if arr[i] is less than or equal to arr[j], arr[i] is placed into temp[k] and i is incremented. Otherwise, arr[j] is placed into temp[k], and j is incremented. After one of the subarrays is exhausted, the remaining elements of the other subarray are copied directly into temp. Finally, the sorted elements in temp are copied back into arr, replacing the elements from left to right with their newly sorted order.

This two-stage sort process (bucket sorting followed by merging) helps in efficiently sorting large arrays by taking advantage of parallel processing and reducing the overall time complexity compared to a simple merge sort implementation.

3.1 References

[1]. Turcotte, Melissa JM, Alexander D. Kent, and Curtis Hash. "Unified host and network data set." In *Data science for cyber-security*, pp. 1-22. 2019.