Contents

10.6 Scatter Pattern

10.6.1 What is a Scatter?

The **Scatter Pattern** is a data movement pattern where **elements from a source array are distributed (or scattered) to various locations in a destination array based on specified indices**. Essentially, it's about writing data to random locations.

⭐ Key Features

In general, the **input** of the scatter pattern is:

- **Source data array**: value to be written.
- **Index array**: specifies where each value should be written in the destination.

Each element from the source is written to the position in the target array specified by the corresponding index. The **output is a target array where the data is scattered across the positions**.

⚖ Gather vs. Scatter

The main differences between gather and scatter are:

- **Gather** (read focused): Involves **random reads** where the read **locations are provided as input**.
- **Scatter** (write focused): Involves **random writes** with write **locations provided as input**, which can lead to race conditions.

🛠 Serial implementation

The following pseudocode shows a serial implementation of the scatter pattern:

```
template<typename Data, typename Idx>
void scatter(
    size_t n,      // Number of elements
                   // in the output data collection
    size_t m,      // Number of elements
                   // in the input data and index collection
    Data a[],      // Input data collection (m elements)
    Data A[],      // Output data collection (n elements)
    Idx idx[]      // Input index collection (m elements)
) {
    // Loop through each input element
    for (size_t i = 0; i < m; ++i) {
        // Get the target index from the index array
        size_t j = idx[i];
        // Ensure the index is within output array bounds
        assert(0 <= j && j < n);
        // Perform the scatter: write a[i] to A[j]
        A[j] = a[i];
    }
}
```

The method is characterized by:

- **Loop** (`for`): Iterates through each element in the input array `a[]` (length `m`).
- **Indexing** (`j = idx[i]`): Determines **where to place** `a[i]` in the output array `A[]`.
- **Boundary Check** (`assert`): Ensures the index `j` is **valid** (avoids out-of-bounds errors).
- **Write Operation** (`A[j] = a[i]`): **Scatters** the data from `a[i]` to `A[j]`.

The possibilities to parallelize the code are interesting, but instead of the gather pattern, here we have a problem with race conditions on write operations. So the code can be parallelized (**parallelize over for loop to perform random write**), but we need to find some strategies to avoid race conditions.

Race Conditions in Scatter

Unfortunately, the scatter pattern can lead to race conditions because of the way it handles writes to memory. As we know, a race condition occurs when two or more operations try to access and modify the same data at the same time, and the final result depends on the order in which the operations are performed.

In the scatter pattern, we're performing random writes to different locations in memory. The **problem arises when multiple write operations target the same location at the same time**. Because the writes happen in parallel, we can't predict:

1. Which write will happen first
2. Which value will be stored last

Example 48: Race Condition in Scatter

Consider this:

- Source Data:

```
A = [A, B, C, D, E, F]
```

- Index Array (write locations):

```
idx = [1, 5, 0, 2, 2, 4]
```

This means that the two threads that will handle the 3rd and 4th positions on the index array will suffer from a race condition. Because if they try to write at the same time, the result will depend on which thread finishes last:

- If thread A (writing `D`) finishes last, position `2` will be equal to `D`.
- If thread B (writing `E`) finishes last, position `2` will be equal to `E`.