CME 213, Introduction to parallel computing
Eric Darve

Radix Sort

1 Radix Sort

1.1 Some background

Before detailing the Radix Sort algorithm, we mention a couple of properties of this sort:

- It is not a comparison sort. This means that contrary to Merge Sort, Quick Sort, Insertion Sort, ..., the sorting does not only rely on a comparison operator (i.e., the less-or-equal operator), instead it compares each element bit-wise.
- It is a stable sort. It means that the sorting maintains the relative order of elements that have equal values.
- The sorting is not done in place. This implies that we will need a temporary array to store partial results.
- The complexity of the algorithm is $\mathcal{O}(kn)$ where k is the average element length.

You may find this online video useful. There are many online tutorials that you can check as well.

1.2 The algorithm

Radix Sort[1] sorts an array of elements in several passes. To do so, it examines, starting from the least significant bit, a group of `numBits` bits, sorts the elements according to this group of bits, and proceeds to the next group of bits.
More precisely:

1. Select the number of bits `numBits` you want to compare per pass.
2. Fills a histogram with numBuckets $= 2^{\text{numBits}}$ buckets, i.e., make a pass over the data and count the number of elements in each bucket.
3. Reorder the array to take into account the bucket to which an element belongs.
4. Process the next group of bits and repeat until you have dealt with all the bits of the elements (in our case 32 bits).

For instance let us say we want to sort:

$$\texttt{keys} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0010 & 1011 & 0111 & 0000 & 0101 & 1111 & 1101 & 1001 \\ \hline\end{array}$$

<u>Step 1</u>: We choose `numBits` = 2.
<u>Step 2</u>:

$$\texttt{buckets} = \underbrace{\boxed{1}}_{00}\underbrace{\boxed{3}}_{01}\underbrace{\boxed{1}}_{10}\underbrace{\boxed{3}}_{11}$$

[1] In this programming assignment, we will implement LSD (least significan digit) Radix Sort.

Step 3:

$$\texttt{new\_keys} = \underbrace{\boxed{0000}}_{00}\underbrace{\boxed{0101}\boxed{1101}\boxed{1001}}_{01}\underbrace{\boxed{0010}}_{10}\underbrace{\boxed{1011}\boxed{0111}\boxed{1111}}_{11}$$

Step 4: Repeat with the two most significant bits.

1.3 A detailed example

We want to sort the following array:

001	101	011	000	010	111	110	100

`keys` = the array above.

Let us say that `numBits` = 1, i.e., we process one bit at a time.

First pass

The first thing we do is computing the histogram: in our case we will have `numBuckets` = $2^{\texttt{numBits}}$ = 2 and:

`histogramRadixFrequency` = | 4 | 4 |

We scan this histogram (i.e., we create a cumulative sum of the elements, starting at zero and ignoring the last element since it's equal to the number of elements):

`exScanHisto` = | 0 | 4 |

The next step is to fill the temporary array. To do this, we need to keep track of the local offset in each of the bucket (the local offset will be used to compute the global offset, i.e., the position in the (partially) sorted array). We do this using:

`localOffsets` = | 0 | 0 |

We now can fill the (partially) sorted array by reading keys and placing the elements in `temp_keys` (this step is called scattering). We start with:[2]

`temp_keys` = | | | | | | | | |

After reading the first element in `keys` we have:

`temp_keys` = | | | | | 001 | | | |

`localOffsets` = | 0 | 1 |

After the second:

`temp_keys` = | | | | | 001 | 101 | | |

`localOffsets` = | 0 | 2 |

After the third:

`temp_keys` = | | | | | 001 | 101 | 011 | |

`localOffsets` = | 0 | 3 |

[2] An empty cell means that the cell does not contain relevant information.

After the fourth:

$$\begin{aligned}\texttt{temp\_keys} &= \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & & & & 001 & 101 & 011 & \\ \hline\end{array}\\ \texttt{localOffsets} &= \begin{array}{|c|c|}\hline 1 & 3 \\ \hline\end{array}\end{aligned}$$

After the fifth:

$$\begin{aligned}\texttt{temp\_keys} &= \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 010 & & & 001 & 101 & 011 & \\ \hline\end{array}\\ \texttt{localOffsets} &= \begin{array}{|c|c|}\hline 2 & 3 \\ \hline\end{array}\end{aligned}$$

After the sixth:

$$\begin{aligned}\texttt{temp\_keys} &= \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 010 & & & 001 & 101 & 011 & 111 \\ \hline\end{array}\\ \texttt{localOffsets} &= \begin{array}{|c|c|}\hline 2 & 4 \\ \hline\end{array}\end{aligned}$$

After the seventh:

$$\begin{aligned}\texttt{temp\_keys} &= \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 010 & 110 & & 001 & 101 & 011 & 111 \\ \hline\end{array}\\ \texttt{localOffsets} &= \begin{array}{|c|c|}\hline 3 & 4 \\ \hline\end{array}\end{aligned}$$

At the end of the first pass:

$$\begin{aligned}\texttt{temp\_keys} &= \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 010 & 110 & 100 & 001 & 101 & 011 & 111 \\ \hline\end{array}\\ \texttt{localOffsets} &= \begin{array}{|c|c|}\hline 4 & 4 \\ \hline\end{array}\end{aligned}$$

At the end of the pass, we copy the result[3] of `temp_keys` into `keys`.

Second pass

The second pass now focuses on the second bit. The result is:[4]

$$\texttt{keys} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 100 & 001 & 101 & 010 & 110 & 011 & 111 \\ \hline\end{array}$$

Third (and last) pass

$$\texttt{keys} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline\end{array}$$

1.4 Parallel implementation

We first divide the array into blocks of size `blockSize` (here `blockSize` = 2).

$$\texttt{keys} = \underbrace{\begin{array}{|c|c|}\hline 001 & 101 \\ \hline\end{array}}_{\text{block 0}}\underbrace{\begin{array}{|c|c|}\hline 011 & 000 \\ \hline\end{array}}_{\text{block 1}}\underbrace{\begin{array}{|c|c|}\hline 010 & 111 \\ \hline\end{array}}_{\text{block 2}}\underbrace{\begin{array}{|c|c|}\hline 110 & 100 \\ \hline\end{array}}_{\text{block 3}}$$

Instead of creating a global histogram, we will create `numBlocks` local histograms using `computeBlockHistograms`:

$$\texttt{blockHistograms} = \underbrace{\begin{array}{|c|c|}\hline 0 & 2 \\ \hline\end{array}}_{\text{block 0}}\underbrace{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline\end{array}}_{\text{block 1}}\underbrace{\begin{array}{|c|c|}\hline 1 & 1 \\ \hline\end{array}}_{\text{block 2}}\underbrace{\begin{array}{|c|c|}\hline 2 & 0 \\ \hline\end{array}}_{\text{block 3}}$$

[3] In fact, there is a way of avoiding this copy by storing alternatively the result in `keys` and `temp_keys`. If the number of passes is odd, there is a final copy from `temp_keys` to `keys`. This is sometimes called *ping-ponging*.

[4] Pay attention to the crucial role played by the stability property of Radix Sort!

We then combine these histograms into a global one using `reduceLocalHistoToGlobal` (notice that this produces the same result as the sequential version):

$$\texttt{globalHisto} = \begin{array}{|c|c|}\hline 4 & 4 \\ \hline\end{array}$$

We scan this global histogram using `ScanGlobalHisto`:

$$\texttt{globalHistoExScan} = \begin{array}{|c|c|}\hline 0 & 4 \\ \hline\end{array}$$

We then compute the offset for each of the local histograms using `computeBlockExScanFromGlobalHisto`. For this, we start by appending `globalHistoExScan` to the front of `blockHistograms` and discarding the final block as follows

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 4 & 0 & 2 & 1 & 1 & 1 & 1 \\ \hline\end{array}$$

We then scan this bucket-wise to get `blockExScan`.

$$\texttt{blockExScan} = \underbrace{\begin{array}{|c|c|}\hline 0 & 4 \\ \hline\end{array}}_{\text{block 0}}\underbrace{\begin{array}{|c|c|}\hline 0 & 6 \\ \hline\end{array}}_{\text{block 1}}\underbrace{\begin{array}{|c|c|}\hline 1 & 7 \\ \hline\end{array}}_{\text{block 2}}\underbrace{\begin{array}{|c|c|}\hline 2 & 8 \\ \hline\end{array}}_{\text{block 3}}$$

Each block in `blockExScan` represents the bucket-wise offsets in `temp_keys` for that block. The elements in `keys` should be written starting from this offset. As an example, the value 001 is the first value in block 0 and belongs to the second bucket (since its lsd is 1), it should therefore be written to position 4 in `temp_keys`. The value 101 follows the same pattern and must be written to the next position in `temp_keys`. In this case, that position is 5 (i.e., 4 + 1). We populate the (partially) sorted array using `populateOutputFromBlockExScan`:

- block 0 will write at positions 4 and 5
- block 1 will write at positions 6 and 0
- block 2 will write at positions 1 and 7
- block 3 will write at positions 2 and 3