Radix Sort

Sammy El Ghazzal (selghazz@stanford.edu)

April 12, 2013

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 single comparison operator (*i.e.* the less-or-equal operator).
- 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.

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` $= 2^{\texttt{numBits}}$ 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>:

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

<u>Step 3</u>:

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

<u>Step 4</u>: Repeat with the two most significant bits.

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

1.3 A detailed example

We want to sort the following array:

`keys` =

001	101	011	000	010	111	110	100

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:

`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

After the fourth:

`temp_keys` =

000				001	101	011	

`localOffsets` =

1	3

After the fifth:

`temp_keys` =

000	010			001	101	011	

`localOffsets` =

2	3

After the sixth:

`temp_keys` =

000	010			001	101	011	111

`localOffsets` =

2	4

After the seventh:

`temp_keys` =

000	010	110		001	101	011	111

`localOffsets` =

3	4

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

At the end of the first pass:

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

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

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. We give the result[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 `sizeBlock` (here `sizeBlock` = 2).

$$\texttt{keys} = \underbrace{\boxed{001}\boxed{101}}_{\text{block 0}}\underbrace{\boxed{011}\boxed{000}}_{\text{block 1}}\underbrace{\boxed{010}\boxed{111}}_{\text{block 2}}\underbrace{\boxed{110}\boxed{100}}_{\text{block 3}}$$

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

$$\texttt{blockHistograms} = \underbrace{\boxed{0}\boxed{2}}_{\text{block 0}}\underbrace{\boxed{1}\boxed{1}}_{\text{block 1}}\underbrace{\boxed{1}\boxed{1}}_{\text{block 2}}\underbrace{\boxed{2}\boxed{0}}_{\text{block 3}}$$

We then combine these histograms into a global one using `reduceLocalHistoToGlobal`:

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

We scan this global histogram:

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

We compute the offset for each of the local histograms using `computeBlockExScanFromGlobalHisto`:

$$\texttt{blockExScan} = \underbrace{\boxed{0}\boxed{4}}_{\text{block 0}}\underbrace{\boxed{0}\boxed{6}}_{\text{block 1}}\underbrace{\boxed{1}\boxed{7}}_{\text{block 2}}\underbrace{\boxed{2}\boxed{8}}_{\text{block 3}}$$

We populate the (partially) sorted array using `populateOutputFromBlockExScan`:

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