University of Magdeburg

School of Computer Science

Bachelor Thesis

Optimizations for Massively Parallel Sort-Merge Join

Author:

Artur Sitnikov

May 29, 2016

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical & Business Information Systems

M.Sc David Broneske

Department of Technical & Business Information Systems

Sitnikov, Artur:
Optimizations for Massively Parallel Sort-Merge Join
Bachelor Thesis, University of Magdeburg, 2016.

Acknowledgements

I would like to thank my advisors David Broneske and Prof. Gunter Saake for giving me the opportunity to write this thesis at the Department of Technical & Business Information Systems. David always found the time for giving me feedback and discuss my questions. I would like to express my sincere appreciation for his outstanding support in correction of the thesis and advices that significantly improved my writing and the result of this work.

Furthermore, I would like to thank Ralf M. Henke and Hannes Kurth from Quinsol AG for their understanding and comprehensive support. I am truly grateful to Marilena Nalli for managing my irregular schedule during this period. I also thank the whole Quinsol team for their support and for being such a great colleagues.

I am very grateful to my parents and my girlfriend, who endured this long process with me. This thesis would not be be possible without their help.

Finally, I thank all my friends for being tolerant for my limited time during this work.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

In this day and age it is not uncommon to see the computers with multiple cores. Having multiple cores is a big advantage for the algorithms, which are able to work in parallel, because we can execute multiple tasks in parallel and thus, significantly increase the performance of the algorithms. That is why it is so important to use such algorithms, especially if they work with a huge amount of data.

In the year 2012 the massively parallel sort-merge (named MPSM) join algorithm was introduced. It represents a new look on the sort-merge join, which works based on partial partition-based sorting [AKN12a]. In this work we concentrate on the optimization of this algorithm in terms of performance.

Goal of this Thesis

The goal of the thesis is to evaluate how MPSM algorithm performs and find the possibilities to improve the performance of the algorithm. Therefore, we provide the following contributions to reach this goal:

1. Investigate and review each single algorithm the MPSM consists of, thereby examining the implementations of these algorithms. Then, find possible improvements for these algorithms.

2. Investigate and review the MPSM join itself, to see, how all single algorithms work together, and look for possible improvements for it.

3. Finally, evaluate the performance of MPSM algorithm under different improvements.

Structure of the Thesis

This thesis is structured as follows. We begin with the background chapter, that introduces the necessary to understand algorithms and their implementation, used in MPSM join. In chapter 3, we present the MPSM algorithm itself. This chapter also reviews the possible improvements for the MPSM algorithm. In chapter 4 we will evaluate the presented improvements for the MPSM algorithm. This is followed by the conclusion and future work.

2. Background

In this chapter we are going to present sorting algorithms as well as a sort-merge join algorithm which are directly used in ”MPSM”. In the end of this chapter we will have a look on the ”MPSM” algorithm itself, recognizing its structure, capabilities and used technologies.

2.1 Sorting Algorithms

In this chapter, we present different approaches of sorting algorithms. Sorting algorithms take an array or a list of elements in an undefined initial order and outputs the input collection with all elements following a certain order. We will start with a simple sorting algorithm, and, step by step introduce more complicated ones. Our goal is to present every algorithm which is used directly in MPSM. For characterizing the sorting algorithm, we discuss some criteria for every algorithm to make the qualification of them more clear. Those criteria are:

1. Stability: The sorting algorithm can be described as stable, if it is not changing the relative order of the elements with equal keys [SS06]. For example, we take a list of students, whose ids are sorted in ascending order. Now we sort this list by another criteria, for example, the age of the student. Using the stable algorithm would mean, two students have the same age, their relative order will be preserved, so the student with the lower id appears before the student with the higher id.
2. Online vs. Offline: The offline algorithm requires the whole data input at the beginning, performing action on the complete data input. In contrast, an online algorithm receives the data input piece-by-piece and the input data is continuously supplemented during the execution time of the algorithm. We can say, an offline algorithm knows exactly, what happens next, while the online algorithm does

not [Kar92]. In case of sorting algorithms, the algorithm is called offline, if it requires the complete input array before the algorithm can start. For example, selection sort, an offline sorting algorithm, needs the whole input array to perform the first step: finding the lowest entry. Insertion sort, by contrast, is online, because it does not need this information in the beginning. It is requesting the information during run time.

3. Time complexity: Here we are describing the time complexity for an algorithm in big O Notation [Bac94]. The time complexity shows us, how efficient the algorithm is. The algorithm performance can also vary in different situation. For example, insertion sort needs less time if the input array is already sorted. In such cases we need to distinguish between best and worst case scenarios and specify both of them. In contrast, some algorithms perform independently of input, like selection sort. Those algorithms always have the same time complexity.

4. Storage consumption: The additional space used by a sorting algorithm is also an important criterion. The sorting algorithms can be divided into two basic groups according to the storage consumption. Those, that sort in place and use no extra space or a small amount of extra space, and those that need enough extra memory to hold another copy of the array to be sorted [SW11]. In other words, if the algorithm uses less than $O(n)$ of extra memory space it is called in place, otherwise it is a not in place algorithm.

There is also a sixth criterion, namely, its adaptivity [ECW92]. We do not specify this criterion in the description, as it depends on the time complexity criterion. So, if the algorithm performs different in best and worst case scenarios, it is an adaptive algorithm and, vice versa, if it has the same time complexity in best and worst case, then it is a non adaptive algorithm.

2.1.1 Insertion sort

The first sorting algorithm that we introduce is insertion sort. Insertion sort algorithm is an efficient algorithm for sorting a small number of elements [CLRS01]. Insertion sort also works efficient with non-random arrays (partially sorted), which are common in practice, even if they have a lot of elements [SW11].

2.1.1.1 Explanation

In our example, presented in Figure 2.1 on the facing page, we want to sort the word `INSERTION` alphabetically. Insertion sort builds the output collection by swapping two elements alternately. Such behavior is less efficient for large data sets, but it works very well with small records.

As we can see, we start with the second element from our input array. In each step of insertion sort we investigate all elements left to currently examined one (named INS), until we find the value, which is lower than INS (it refers to the ASCII sign). If we find such value, we swap the INS element with this element. If not, it means that INS is the smallest element (comparing to elements left to it) and we place INS at the beginning. We can see such behavior at line 3, where the letter E is smaller than all elements left to it. So, we place it at the beginning (cf. line 4). Another possible scenario is when all element left to INS are smaller. We can see such scenario at line 5. In this case, we are doing no swaps and continue with the next element (cf. line 6).

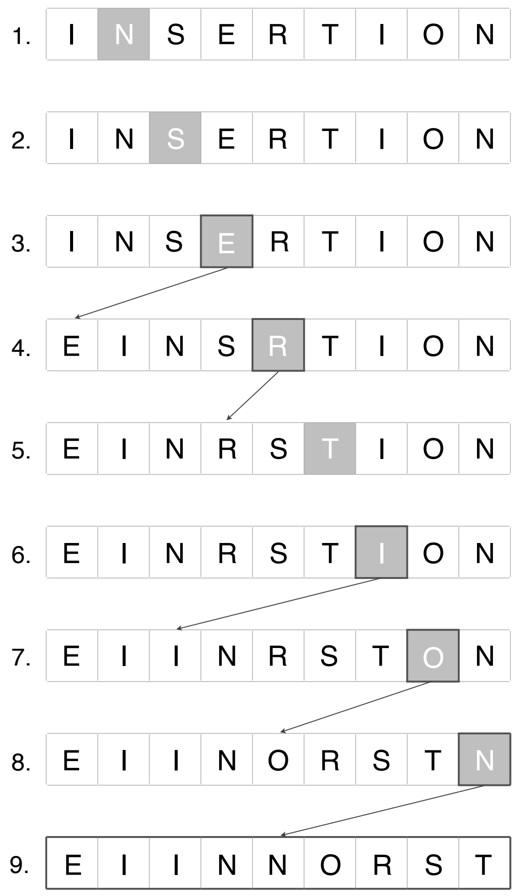

Figure 2.1: Insertion sort workflow

2.1.1.2 Implementation

The insertion sort algorithm has a very simple implementation, comparing to the algorithms we will investigate later. The algorithm itself is shown in Listing 2.1 on the next page.

We start with the for-loop at line 6, where we iterate over all elements in the input array, starting with the second value. We do not need to investigate the first element, as it has no elements left to it. At line 7, we store the value of the currently examined element in a variable *save*. We need it, because we will overwrite this element with the element next to it on the left side, if this element has the higher value.

```
 1  Function: insertionsort(P,n)
 2  Input: unsorted array P
 3  Input: n as number of elements in P
 4  Process: i,j, save
 5
 6  for(i ← 1; i < n; i++)
 7    save ← P[i]
 8
 9    for(j ← i; j ≥ 1 && P[j − 1] > save; j++)
10      P[j] ← P[j − 1]
11    end for
12
13    P[j] = save
14  end for
```

Listing 2.1: insertionsort - adapted from [CLRS01]

In the for-loop at line 9, we are, one by one, replacing the right element with the left one, if the left element is bigger than the element, stored in *save*, according to the second condition in the for-loop at line 9. The actual swap is done at line 10. If the left element is smaller or equal to *save*, or if we have reached the beginning of the array (the first condition in the for-loop at line 9), we make the final swap, placing the *save* element to the position it belongs to (cf. line 13). Then we increment the *i* variable and continue with the next element.

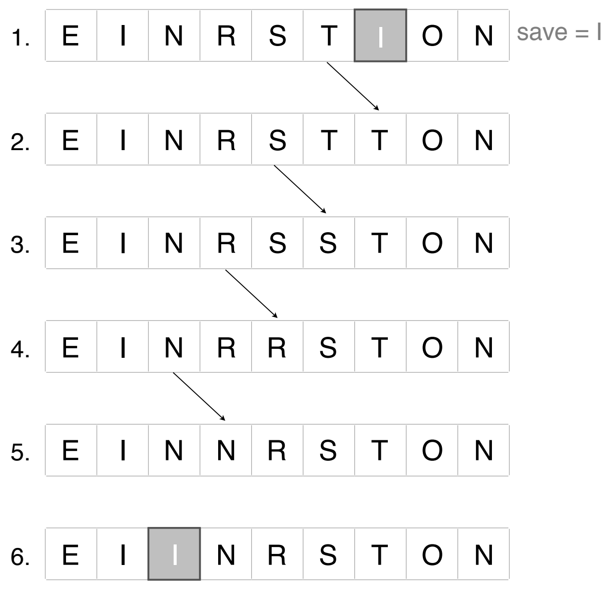

Figure 2.2: Insertion sort - single step workflow

To understand more clearly what happens in the second for-loop, we will have a look on the example, presented in Figure 2.2, which represents the step 6 from the previous example.

Here we want to find the right location for the element I. So, we store this element in *save*. In the next steps, we iteratively replace the element we are currently pointing on (the pointer is the j variable) with the element left to it, until we find the element which is smaller than *save*. In each step, we decrement the j variable, so that pointer shows on the right location. In this example, we need to do it 4 times (cf line 1 - 5) until we finally find the element, which is smaller or equal to *save*. In our case this value is equal to *save*. Thus, we make the final replacement at line 6, and our investigated element is now at the right position.

2.1.1.3 Summary

The qualification of insertion sort is presented in Table 2.1. It is a stable algorithm, as we could see from the previous example(we swap only the elements, which are bigger). It has a time complexity between $O(n^2)$ and $O(n)$. Anyways, its average complexity is also $O(n^2)$. Insertion sort is online, as it needs only one element at the beginning. Insertion sort is also in-place, as it needs only $O(1)$ of additional memory space.

Criteria	Description
Stability	stable
On vs.Off	online
Complexity	$O(n^2)$
Storage	in-place

Table 2.1: Insertion sort qualification

2.1.2 Quicksort

Quicksort is one of the most popular sorting algorithms. There are several reasons for that. Quicksort is easy to implement, it works with different kinds of data and in fact works faster than any other sorting method [SW11]. Quicksort belongs to divide-and-conquer sorting algorithms. It means, we divide by rearranging the original array (in quicksort we call it partitioning) and conquer by recursively sorting the elements [Cor13].

2.1.2.1 Explanation

In the example, presented in Figure 2.3 on the following page, the word ”QUICKSORT” is stepwise sorted in alphabetical order using quicksort. The selected letters (a cell with a stronger border) represents a pivot element, which is needed to rearrange the input array into two other subarrays. There are several possibilities to define a pivot element, e.g. take the first, last or middle element from the input array.

As we can see at line 1, in our example we define the first element a pivot, namely `Q`. In the next step, we put all elements which are smaller than the pivot left to it and

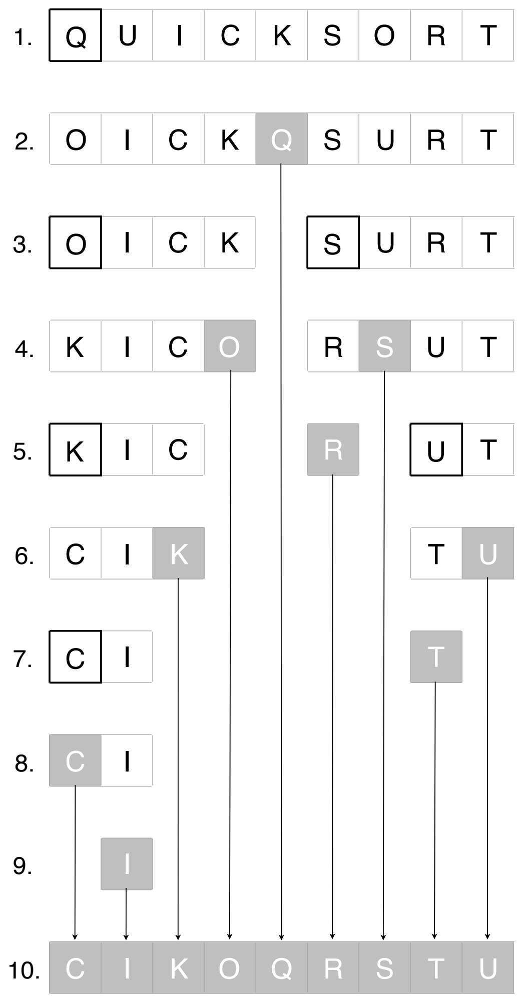

Figure 2.3: Quicksort workflow

those, which are greater, on the right side. We can see the result at line 2. After that, we are recursively doing the same for the two newly created arrays (cf. line 3), divided by pivot element, until the investigated array contains only one entry. As quicksort is done with all recursion steps, our array becomes sorted.

Another interesting point is what actually happens between the first and the second line. All elements, which are greater than pivot element have to be moved to the right side and, all elements which are lower or equal than pivot, respectively, to the left side. The method, which is responsible for that is called *partition*. What it exactly does will be discussed later.

2.1.2.2 Implementation

As mentioned before, Quicksort algorithm is not hard to implement. As you can see in Listing 2.2 on the next page the basic structure has only few lines of code. Though, the partitioning part, which is presented in Listing 2.3 is again a little bit bigger.

```
 1 Function: quicksort(P,l,r)
 2 Input: unsorted array P;
 3 Input: l,r: starting and ending indicies of a subarray of P
 4 Process: q
 5 Returns: sorted array P
 6
 7 if (l < r)
 8   medianOfThree(P,l,r)
 9   q ← partition(P,l,r)
10   quicksort(P, l, q−1)
11   quicksort(P, q+1, r)
12 end if
```

Listing 2.2: quicksort - adapted from [Sed78]

The quicksort-function begins with if-statement (cf. line 7), which ensures that only arrays with more than one element can proceed. It is necessary, because we don't need to sort arrays with one or even without elements. If an array satisfies this condition, we can start with *partition* method (cf. line 9). The $medianOfThree$ method at line 8 is an optional step and will be discussed more detailed in improvements chapter.

```
 1 Function: partition(P,l,r)
 2 Input: unsorted array P;
 3 Input: l,r: starting and ending indicies of a subarray of P
 4 Returns: q as pivot position
 5
 6 pivot ← P[r]
 7 q ← l
 8
 9 while(l < r)
10   if (P[l] ≤ pivot)
11     swap(P[q], P[l])
12     q ← q + 1
13   end if
14   l ← l + 1
15 end while
16
17 swap(P[q], pivot)
18
19 return q
```

Listing 2.3: partition - adapted from [Sed78]

In the *partition* method, the pivot element is represented by the last element in input array, as we can see at line 6. So, we need to sort the array in the way that all elements, which are smaller or equal than the pivot element have to be on the left side and those, which are bigger, on the right side. To understand, how to do that, we will examine the *partition* method in more detail. We will base on the steps one and two from the previous example.

As we can remember, ”Q” was our pivot element. It means, if we want to proceed with *partition* method, we need to swap the first and the last elements with each other (this is all done by $medianOfthree$ method, which we present later). So, at the start of the *partition* method, our input array looks as follows: ”TUICKSORQ”. As mentioned already, at line 6, we define the pivot element, which is the last element in the input array, namely ”Q”. The variable q, declared at line 7 represents the position of pivot and gets the value of l, the start index of a subarray of P. In the while-loop, between the lines 9 and 15, we are iterating over all elements of subarray from the left to the right, excluding the last element, namely our pivot. In each loop, we are checking, if the examined element is smaller or equal than our pivot (cf. line 10). If it is, we swap this element with an element at q and increment q, as we can see at lines 11 and 12. The *swap* method is presented at Listing 2.4. Last final swap for pivot element and element at q. Finally, we can now return the position of the pivot element and continue with the next recursion step.

```
Function: swap(l,r)
Input: l,r: elements to swap
Process: tmp

tmp ← l
l ← r
l ← tmp
```

Listing 2.4: swap

2.1.2.3 Improvements

Median-Of-Three

As we already mentioned, there are several possibilities to determine the pivot element. One possibility is to define the pivot element using the $median - of - three$ method, presented in Listing 2.5 on the facing page. The advantage of this method is that it determines the pivot element using three values, instead of one [Sed78]. The first and the second elements are the first and the last element from the investigated array respectively. The third value is the element in the middle of array and it is calculated as follows: $(l + r)/2$, where l and r are the positions of both values, presented before.

Once we have identified all three values, we can proceed to the main function of the method. Actually, this method is not defining the pivot element as it does not return any value. It swaps the values in the way that the *partition* method selects the best value as pivot element. As *partition* method always defines the last element in the input array as pivot, we need to compare all three presented values with each other, identify the element with the mean value and put it at the end of the input array. As we need three values to figure out the median, we have an additional check at line 6.

```
Function: medianOfThree(P, l, r)
Input: unsorted array P;
Input: l,r as starting and ending indicies of P
Process: m

if ((r − l) > 3)
  m ← (l + r) / 2

  if(a[l]>a[m])
    swap(P[l], P[m])
  end if

  if(a[l]>a[r])
      swap(P[l], P[r])
  else if(a[r]>a[m])
      swap(P[r], P[m])
  end if
end if
```

Listing 2.5: medianOfThree - adapted from [lr04]

2.1.2.4 Summary

The qualification of quicksort is presented in Table 2.2. It is not a stable algorithm, because of pivot elements, whose duplicate appears always on the right side. It has a time complexity between $O(n^2)$ and $O(n * log(n))$. Its average Complexity is also $O(n*log(n))$. Quicksort is offline and still in-place, because it needs $log(n)$ of additional memory space.

Criteria	Description
Stability	not stable
On vs.Off	offline
Complexity	$O(n * log(n))$
Storage	in-place

Table 2.2: Quicksort qualification

2.1.3 Heapsort

The next algorithm we will present is heapsort. The algorithm consists of two phases: heap construction, where we transform an input array into a heap, and sortdown, where we convert the heap back into the array, by removing the elements from the top level of the heap and putting it into the output array [SW11]. After each removal, we reorganize the heap. After all elements have been removed from the heap - the output array will contain all elements in a correct order.

2.1.3.1 Explanation

To explain how heapsort works, we use a small example. In this example, we will first form the heap and then, by removing one element after another from the heap, create a new array with elements in a certain order. To do this, we use letters of the word ”HEAPSORT” as our input array and our goal is to sort it in alphabetical order.

There are two possible implementation variants of heapsort, namely $MinHeap$ and $MaxHeap$. In a $MinHeap$, all children nodes have bigger values than their parents. In a $MaxHeap$, in contrast, parent has the biggest value and the children nodes have smaller values.

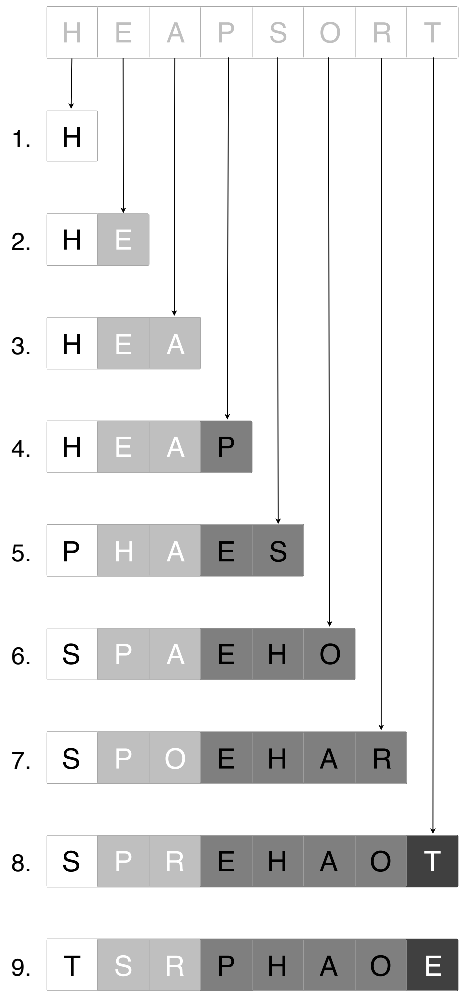

Figure 2.4: Heapsort - heap construction workflow

First, we need to construct the heap. The heap construction phase example is depicted in Figure 2.4. As we want to sort the input array in ascending order, or, in our case, in alphabetical order, as we deal with letters, we need to use the $MaxHeap$. In contrast, if we sort descending, we have to use $MinHeap$. It sounds confusing, because the

MinHeap is the one, which has the minimum values on the top. Thus, we could just take the element from the top and put it into the output array and continue with the next one. Well, actually, in heapsort we have no output array at all. Our output array is the heap itself. We just put the top element at the end of the heap and reduce the size of the heap by one. Thereby, we can continue with the next element, ignoring the elements, which have been processed. Thus, we need to use *MaxHeap* as we always want to have the highest element at the end of the heap after each step.

We can see, how each element from the input array is one by one added to the heap. The cells in presented example are filled with different colors. Each of these colors represent different level of the heap. So, for example, at line 4, the parent on the top level H has left child E and right child A. E-Node in turn, has its own child on the left side, namely P, which we just added to the heap. Thus, each next level has two times more space, as every child element can have another two children. Once again, if we take a look at line 9, the result of heap construction for our example, the heap structure for this line is represented in Figure 2.5.

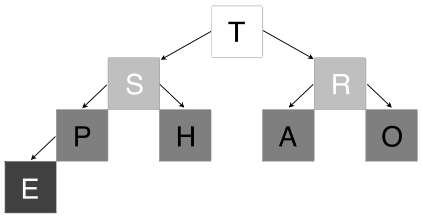

Figure 2.5: Heapsort - heap construction result

As we deal with heap structure, we also need to reorganize the elements in the heap (if needed), so that the parents always have the bigger values than their children. So, for example, in step 4, we need to put ”P” on the top of the heap, because its value is higher than the value of ”E” and ”H”(it refers to the value of ASCII). So we need to swap ”P” and ”E” with each other, and then swap ”P” and ”H”, as shown in Figure 2.6.

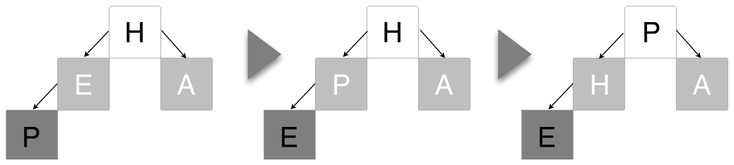

Figure 2.6: Heapsort - heap construction reorganization

The previous example demonstrates an online heap construction algorithm. Well, there are also more efficient solutions for this purpose, namely, to take the whole array at the beginning (means, making it offline), and reorganize only the elements, which have children. We begin with the last element in the heap, which has at least one child, reorganize it and its children, and continue with the next higher child. We also need to reorganize the subsequent levels(the children of children), if they are exists. The offline version of the heap construction phase is presented in Figure 2.7. Thus, we do not need as many reorganization steps as we needed in previously example.

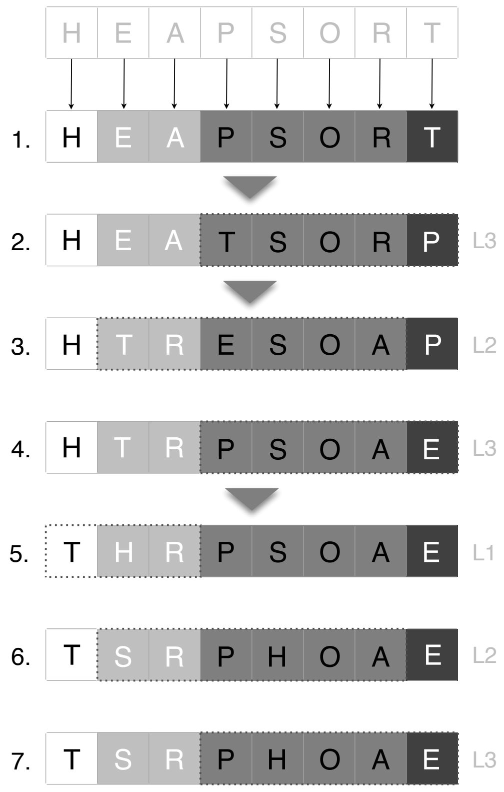

Figure 2.7: Heapsort heap construction offline workflow

In this example, the last element, which has at least one child, is T at level 3. So, we need to reorganize it and its children elements. In our example, we need to swap T and P with each other, as T has a higher value than P. We can see the applied changes at line 2. As the child element(which is now P) has no own children, we can continue with the next higher parent. Thus, we are now have to investigate the element A and element E afterwards. In our example we have combined this both reorganization steps into one(cf. line 3). Parent elements on the second level are now T an R. Now, we need to reorganize those children of T and R on the level 3, which also have at least one child, in our case, only the element P. We can see the final result for the second and the third element at line 4. We are doing the same last time with the top element. As there are no higher element than the top element, we are done and our array now became the $MaxHeap$.

The Second phase of the heapsort is called sortdown. In this phase we want to eliminate the elements from the heap one by one, so that it gets transformed into the output array. As already mentioned before, the $MaxHeap$ structure is known by fact, that the top element is always the highest element in the whole heap. Thus, we need to take the top element, put it to the end of the heap and reorganize the reduced heap, so that the next highest element is placed on the top. Although, removing the top element from the heap is not that easy, because we need to make sure, the heap will not lose its structure (each child node should have a parent). So, we just swap the top element with the last element in the heap(last item on the lowest level). Thereby we make sure that every child node will still have a parent. Before we can repeat this step again, we need to reorganize the heap, so that the next highest value is placed on the top level of the heap. Thus, we are not losing the heap structure, and, step by step, transforming the heap into the array we need.

Returning to our example, we start at the same point we broke up in the first phase of the offline version, namely with a constructed heap, presented at line 1 in Figure 2.8.

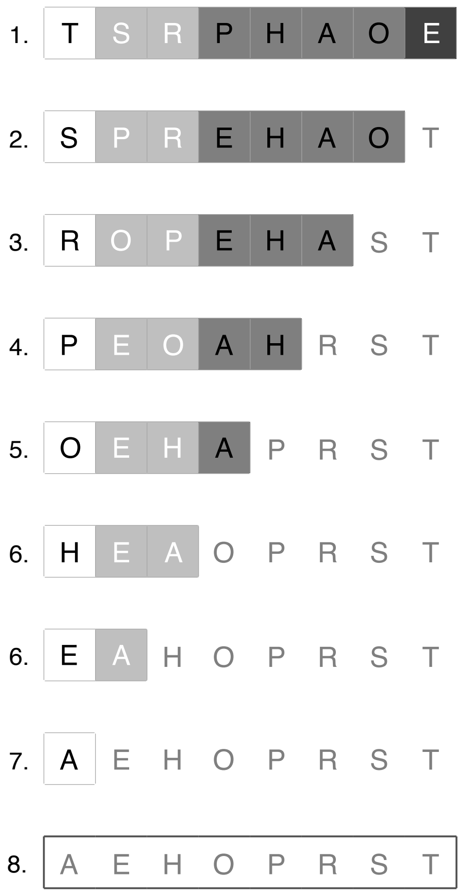

Figure 2.8: Heapsort sortdown workflow

Each step in this example is a combination of several events: swapping of the top element in the heap with the last element in the heap, as described before, and its reorganization. Thus, each subsequent line has the next highest remaining element on the top of the heap. On the right side, we can see, how the output gets gradually created. Same, as in previous graphic, the different cell colors represent different heap levels.

To understand more detailed what happens between the lines, we will have a look on transformation from line 5 to 6, as depicted in Figure 2.9. First, we swap the top element O on the heap with the lowest element A. Second, we reduce the size on the heap by one, thereby eliminating O from the heap. As we ruined the heap structure, we are reorganizing the heap, so that the next highest element, namely H becomes the top element and the next candidate for the output entry.

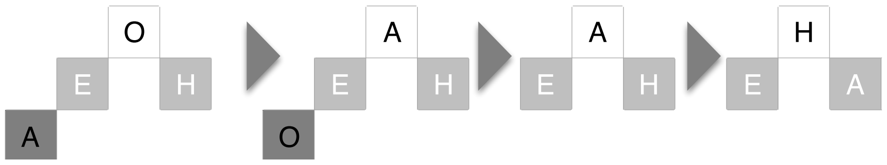

Figure 2.9: Heapsort sortdown in details

In the end we get a completely sorted array and no elements on the heap, as we can see at line 8 in Figure 2.8 on the preceding page.

2.1.3.2 Implementation

The main *heapsort* method is presented in Listing 2.6. It is logically divided in two phases introduced before: heap construction and sortdown. Heap construction, as the name says, is responsible for building the heap and sortdown transforms the heap into a sorted array.

```
Function: heapsort(P, n)
Input: unsorted array P;
Input: n as length of P
Process: i

//heap construction
i ← (n/2) − 1
while (i ≥ 0) do
  doheap(P, n, i)
  i ← i − 1
end while

//sortdown
i ← n − 1
while (i > 0)
  swap(P[i], P[0])
  doheap(P, i, 0)
  i ← i − 1
end while
```

Listing 2.6: heapsort - adapted from [cod12]

As we can remember from previous section, we have introduced two possible solutions for heap construction phase: an offline and an online method. In our implementation we will use the offline variant, as it is more efficient. To start with building the heap, we need first to define the number of elements, which have children. We are doing it as follows: $(n/2) - 1$,where n is the length of the array. The variable i from the code at line 7 gets this value. Thus, in our example we have 4 such elements ((8/2 - 1 = 3) + 1 as zero based)). Next, we need to reorganize the heap for each of these elements. The while-loop between lines 8 and 11 makes sure that we call the *doheap* exactly i - times, thereby reorganizing each of the parents (*doheap* method is responsible for reorganization). After that, our array gets the heap structure.

The second part of the *heapsort* method is responsible for sortdown. First, we redefine our i variable. Now it becomes the number of elements in the heap minus one, means, the position of the last element in the heap(cf. line 14). In our example, i gets a value of 7. The while-loop between the lines 15 and 19 is responsible for transforming the heap into the sorted array. Inside the while-loop, we are first, swap the element at index i with the first element in the heap (cf. line 16). Second, we are reorganizing the heap, without touching the last element in the heap, which we have just added (cf. line 17). Later, we will examine how *doheap* method works, and we will see how it is managed exactly. Last, we decrement the variable i and execute the loop again. Important fact is that while executing the loop for the second time, we are doing nothing with the last element in the heap. We just ignore it. The *doheap* method also reorganizes the heap without taking the last element. As we can see, in contrast to the first while-loop, this time we are not executing the loop when i is equal to zero. We are not doing it, because at the moment the heap has only one element left, this element has already its right location in the output array, so we do not need to do anything with it. Thus, at the moment i becomes zero, we are done and our our array is now sorted.

The *doheap* method, presented in Listing 2.7 on the following page, is a help function, used to reorganize the heap. First, we are storing the element we actually want to sink (cf line 7). The condition of the while loop between the lines 8 and 21 makes sure the sink element has at least one child. The child variable at line 9 gets the position of the left child of the sink element. If the right child of the sink element exists and its value is higher than the value of the left child, we overwrite the child variable with the position of the right child of the sink element(cf. lines 11-13). Now, if the sink element is higher or equal as its biggest child element, we can break the loop, because it means, that the heap is already structured on this and lower levels(cf. lines 15-17). Otherwise(if one of the children is higher than the parent), we are overwriting the parent element with the child element, as shown at line 19. Important is that we are not overwriting the sink variable, just the sink element in the heap. So, at this moment, the biggest child and its parent are equal elements. Next, we replace the position of the parent with the position of its highest child(cf. line 20) and can continue with the next loop, if the current value of i satisfies the while-loop condition. Thus, while loop condition checks if the current value at position of i has at least one child. If the value of i satisfies the condition, we are doing the same again, otherwise, we are overwriting the element at i with the sink element we stored at the beginning and thereby we are done.

```
Function: doheap(P, n, i)
Input: unsorted array P;
Input: n as length of P
Input: i as index of the element to sink
Process: sink, child

sink ← P[i]
while(i < (n / 2))
  child ← i + i + 1

  if(((child + 1) < n) && (P[child] < P[child + 1]))
    child ← child + 1
  end if

  if (sink ≥ P[child])
    break;
  end if

  P[i] ← P[child]
  i ← child
end while

P[i] ← sink
```

Listing 2.7: doheap - adapted from [cod12]

2.1.3.3 Summary

The qualification of heapsort is presented in Table 2.3. Heapsort is not stable. It has always a time complexity of $O(n * log(n))$. It is possible to have the algorithm online, but then it is less efficient. Thereby we define it as offline, because the more efficient version needs the whole array at the beginning. Anyway, heapsort is in-place and needs only $O(1)$ of additional memory space, as it is visible from our array representation.

Criteria	Description
Stability	not stable
On vs.Off	offline
Complexity	$O(n * log(n))$
Storage	in-place

Table 2.3: Heapsort qualification

2.1.4 Introsort

Introsort(for ”introspective sort”) is a hybrid sorting algorithm. It represents the combination of quicksort, heapsort and insertion sort, presented before. In most cases introsort behaves like a quicksort. The difference is that it detects the tendency to the worst case scenario of quicksort, which is $O(n^2)$ and, in this case, switches to the heapsort [Mus97]. The benefit of this switch is that we can still achieve O(n*log(n)) (the worst case time complexity of heapsort is O(n*log(n))).

2.1.4.1 Explanation

As already mentioned, introsort combines multiple sorting algorithms. In order to start with the right algorithm at the right moment, we need to set up some configurable properties for introsort. These properties are *sizethreshold* and *depthlimit*. The *sizethreshold* parameter defines at which moment we have to stop partitioning. It means, we define a parameter, which we check against the sub array size in each recursion step. If the size of the sub array is smaller or equal to the defined parameter, we just break, without creating two more sub arrays from the array we are currently investigating. Another parameter is *depthlimit*. This parameter defines the moment we have to switch to the heapsort algorithm. Similar to the previous one, here we also stop partitioning and let heapsort execute the remaining sorting steps. Thus, in the moment heapsort finishes its work, we can return the array, as it is sorted now. The way we come to a decision for switching to heapsort or not is different than by *sizethreshold*. This time, the configured parameter is decremented after each recursion step. At the moment the *depthlimit* becomes zero, we give the currently investigated sub array over to heapsort. Thus, on the one hand, with *sizethreshold* we are checking against the array length, on another hand, with *depthlimit*, we are checking against the number of recursion steps. An important point is that the *depthlimit* is defined automatically at the beginning of the algorithm, using the following formula: $2 * log_2(r - l)$, where r and l are the starting and ending indicies of the input array. It means, *depthlimit* parameter depends on the size of the input array. The *sizethreshold*, in contrast, has to be defined by the user.

To understand more clearly, how introsort works, we have a small example prepared. In this example, we will sort the letters of the word "INTROSORT" alphabetically. Before starting, we first have to define the two parameters, discussed before. We define *sizethreshold* with 3, means, we stop partitioning, if our sub array contains less than or equal to 3 elements. The *depthlimit* becomes 6 and is calculated as follows: $2 * log_2(8 - 0)$. As we defined both parameters, we can finally start with the actual algorithm. For better understanding of each of the steps, we have prepared a figure, which is depicted in Figure 2.10 on the following page

As our initial array has more than 3 elements and the *depthlimit* is not equal to zero, we can continue with partitioning. As we can see at line 1, we are defining the pivot element, which is in our case, the element "O" in the middle of the word(the selecting of pivot element is done via $medianOfThree$ method, presented in Listing 2.5 on page 11). The grey colored number on the right represents the current value of *depthlimit* (as we can remember, it is decremented with each recursion step). Next, we are reorganizing the array, so that the letters, which are smaller or equal to the pivot, have to be moved to the left side of the pivot element, otherwise, to the right, same, as we did in quicksort.

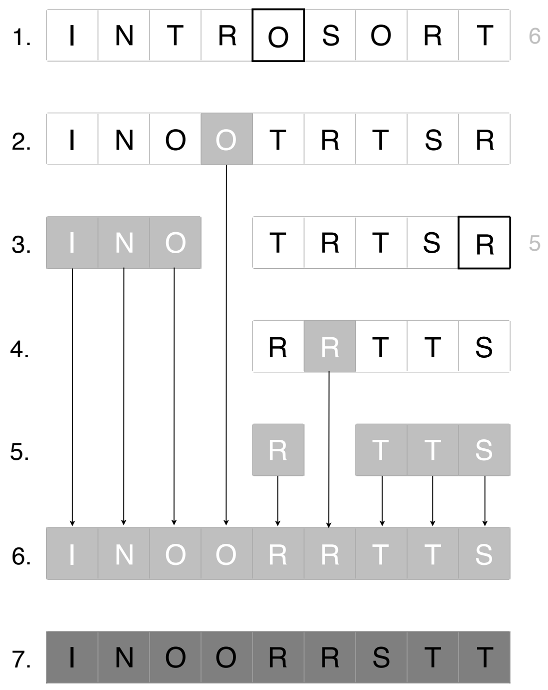

Figure 2.10: Introsort workflow

We can see the result at line 2. Then, we are doing the same again for the right and for the left sub arrays, splitted by the pivot element, excluding the pivot element itself. The left sub array has now the length of 3, which means, we break at the moment, so that the values of this array will be later sorted by insertion sort. The right part, in contrast, has more than 3 elements (there are 5) and the *depthlimit* still not equal to zero (its value is 5). So, we again define the pivot element and split the currently examined array into two sub arrays(cf. line 4). We can see the result at line 5. This time, all our sub arrays have less or equal than 3 element, means we have now to execute one final insertion sort for the complete array(cf. line 6). After the insertion sort, at line 7, all our letters are now alphabetically sorted.

2.1.4.2 Implementation

The introsort algorithm consists of two parts. The main part, which is calling the recursive part and makes the final insertion sort, is presented in Listing 2.8 on the facing page. The recursive part, which is responsible for the core logic of introsort, is presented in Listing 2.9 on the next page.

While calling the recursive part of the introsort method for the first time, we are also calculating the *depthlimit* with the following formula: $2 * (log_2(r - l))$ as we can see at line 6 in Listing 2.8 on the facing page. The *insertionsort* method at line 7 is called at the end of the algorithm and will be discussed later.

At the beginning of the recursive part we are first, checking, if the size of our array is bigger than the *sizethreshold* constant, which is configurable by user (cf. line 7). If

```
Function: introsort(P,l,r)
Input: unsorted array P
Input: l,r: starting and ending indicies of a subarray of A
Returns: sorted array P

introsortr(P, l, r, 2*(log2(r − 1)))
insertionsort(P, l, r)
```

Listing 2.8: introsort - adapted from [Mus97]

our array satisfies the condition, we can continue, otherwise, we are done with recursive part. The check at line 9, is for the *depthlimit* parameter. If this parameter reaches zero, we are not dividing the currently investigated array in two another parts, we let the *heapsort* manage the sorting of this array for us. This will mean, the *quicksort* tends to the worst case scenario and we have to switch to *heapsort*. At the moment *heapsort* is ready with sorting of the current array, we do not need to continue (array is sorted), means, we can return (cf. line 11).

```
 1  Function: introsortr(P,l,r, depthlimit)
 2  Input: unsorted array P
 3  Input: l,r: starting and ending indicies of a subarray of A
 4  Input: depthLimit
 5  Process: q
 6
 7  while ((r − l) > sizethreshold)
 8
 9    if (depthlimit = 0)
10      heapsort(P,l,r)
11      return
12    end if
13
14    depthlimit ← depthlimit − 1
15    medianOfThree(P,l,r)
16    q ← partition(P,l,r)
17    introsortr(P, q + 1, r, depthlimit)
18    r ← (q − 1)
19
20  end while
```

Listing 2.9: introsortr - adapted from [Mus97]

If *depthlimit* is not equal to zero, we decrement it (cf. line 14) and continue with partitioning. Same as in previously introduced quicksort, we first define the pivot with *medianOfThree* method(cf. line 15), introduced in Listing 2.5 on page 11, and second, call the *partition* method at line 16, which was also introduced previously in Listing 2.3 on page 9. The *q* variable is now holding the position of the pivot element. Now, the interesting part of the algorithm: the right sub array is called recursively(cf. line 17), while we iteratively proceed with the left part. We can see it at line 18, where *r* gets the value of the position of the first element left to pivot and continues at the beginning with the while loop. Both parts, recursive and iterative are called without the pivot element, because pivot element has already its final location and will not change it.

After we have finished with the recursive part, we need to do one final insertion sort, whose is needed for sorting the sub arrays, which size was smaller or equal to $sizethreshold$. Thus, insertion sort has not to do a lot of swaps and thereby, tends to its best case scenario, namely $O(n)$.

2.1.4.3 Summary

The criteria of introsort, as presented in Table 2.4, are mostly the same as by quicksort, except for time complexity. The time complexity of introsort is now equal for best and worst case scenarios and is O(n*log(n)).

Criteria	Description
Stability	not stable
On vs.Off	offline
Complexity	$O(n * log(n))$
Storage	in-place

Table 2.4: Introsort qualification

2.1.5 Radixsort

Radix sort, unlike algorithms presented before, does no comparisons at all, which makes it a non comparison algorithm [Knu73]. Instead of comparing the elements with each other, we will rearrange the order of the individual symbols of each element. More detailed explanation of how it works will be provided later.

There are two basic versions of radix sort. The LSD-Version, which operates by sorting the elements on the rightmost or least significant symbol, and MSD-Version of the algorithm, which is left-to-right, means it investigates the most significant symbols [SZ03].

2.1.5.1 Explanation

Even if it seems to have no big difference between LSD and MSD versions, the way LSD works is completely opposite to the way MSD performs. The interesting fact is that MSD, compared to LSD radix, can be implemented recursively, and thus, after the buckets of MDS radix becomes small enough, we can switch to a simple sorting algorithm, for example insertion sort. The LSD radix sort, in contrast, can only be implemented iteratively. Based on a that, we decided to discuss both, LSD and MSD versions, and present them individually.

LSD-Version

We will have a look on Figure 2.11. In this example we will sort the word ”RADIX-SORT” with LSD version of radix sort. In step 1 we can see each letter of the word ”RADIXSORT” converted in ASCII sign it belongs to. In a step 2 we just split the ASCII sign into individual symbols. So, for example, 82 is splitted into 8(on the top) and 2(on the bottom). Thus, the least significant symbols are on the bottom line. It is done to increase the visibility of what actually happens in the algorithm and thus, make the example more clear. As we investigate the least significant digit algorithm, we have first to sort the symbols, which are rightmost (in our example the symbols at the second row). In step 3 we can see that the symbols in second row are sorted now.

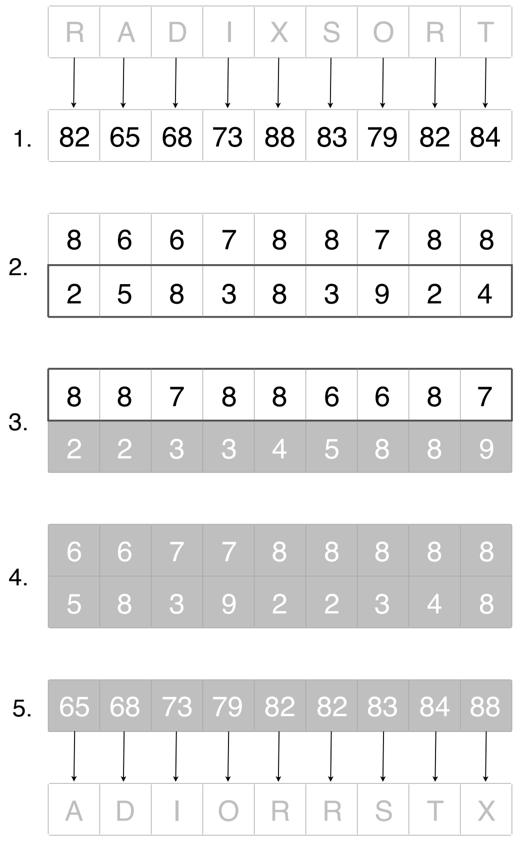

Figure 2.11: LSD radix sort workflow

Next, we are doing the same, but now, with symbols from the top row (as we have only 2 rows in our example). We have to use the resulted array from the previous sorting step, which is now looks as follows: 82, 82, 73, 83, 84, 65, 68, 88, 79. The result of the top row sorting is presented at line 4. Since we have no elements, which consists of 3 or more symbols we are done after finishing sorting the top row symbols. Thus, at line 5, we can see the final result of LSD radix sort and its representation in letters.

MSD-Version

Same as in previous example, here we also want to sort the word "RADIXSORT" but now, using the MSD radix sort algorithm. The first step is the same as we did with LSD radix sorting. The difference begins at step 2. Here, we now start sorting with the top row, instead of the bottom row, like we had in LSD radix sort. As we can remember, top row represents the left-most symbols and thereby the most significant symbols. At line 3, we can see the result of the sorting of the top row symbols.

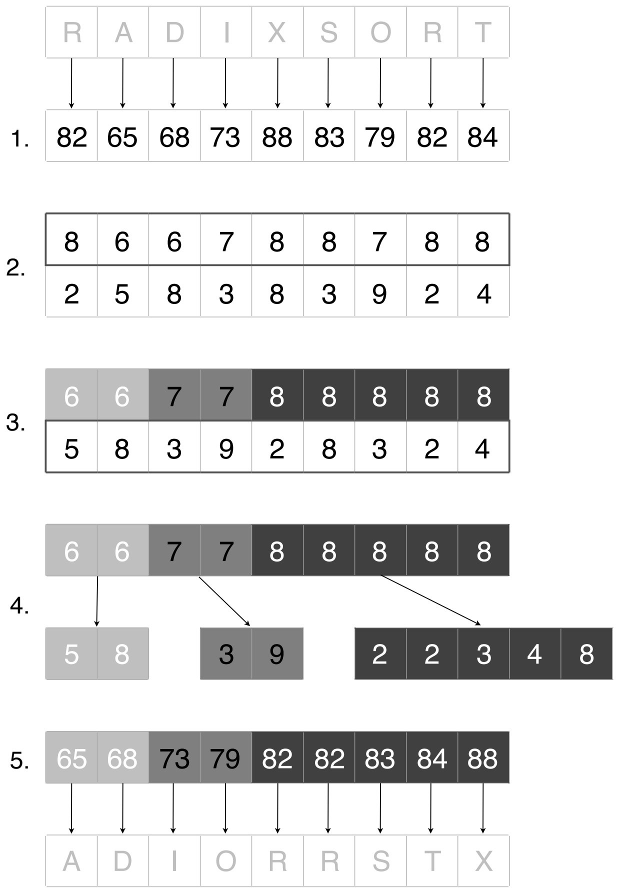

Figure 2.12: MSD radix sort workflow

As we now have sorted the top row symbols, the position of these symbols will never change. Thus, we can sort the lower row symbols recursively, as shown at step 4. We could not do it before at LSD radix sort, because the position of the elements is changing after each step. Since we have only two rows, we are done after this step and can see the result at line 5.

2.1.5.2 Implementation

Next, we will have a look on how the LSD and MSD are implemented, will explain the difference between them and find out when to use which version.

LSD-Version

```
Function: lsdRadix(P,n,w,t)
Input: unsorted array P
Input: n as the number of elements in P
Input: w as length of the highest element
Input: t as histogram length
Process: count, dist, tmp

for(d ← w−1; d ≥ 0; d−−)

  count ← int[t + 1]
  dist ← int[n]

  //create histogram
  for(i ← 0; i < n; i ← i + 1)
    tmp ← P[i].numberAt(d)
    count[tmp + 1] ← count[tmp + 1] + 1;
  end for

  //compute a prefix sum
  for(k ← 0; k < t; k ← k + 1)
    count[k+1] ← count[k+1] + count[k]
  end for

  //perform array changes in dist array
  for(i ← 0; i < n; i ← i + 1)
    tmp ← P[i].numberAt(d)
    dist[count[tmp]] ← P[i]
    count[tmp] ← count[tmp] + 1
  end for

  //copy array changes back to initial array
  for(i ← 0; i < n; i ← i + 1)
    P[i] ← dist[i]
  end for

end for
```

Listing 2.10: LSD radix sort - adapted from [SW11]

The Listing 2.10 represents, how LSD radix sort is implemented. As we can see, it has some extra input parameters, additionally to input array itself (P) and its length (n). The input parameter w represents the length (how many digits it has) of the highest element we have present in our input array. The parameter r stays for the histogram length, or, in other words, it says, how many bins we need to create per sorting level. So, for example, if we want to sort some words according to the english alphabet, the parameter t becomes 26, representing all possible states the symbol (letter) can have. In the example, presented before, we are transforming the letters into the ASCII sign, thus, our histogram length is 10, according to all possible values a single digit can take (from 0 to 9).

The for-loop at line 8 makes sure we iterate over each symbol of our elements. As we investigating the least significant digit radix sort, we are starting with the rightmost digit. That is why at the beginning, the variable d becomes the rightmost element and gets decremented every time the loop ends. Then, we are creating two new auxiliraly arrays. First, the *count* array, which represents the histogram. Although we need to allocate t of memory space for the histogram, we are allocating $t + 1$, because we need an additional memory unit for the second step - building the prefix sum. Second, the *dist* array, which is needed to distribute the elements from the P array and therefore, has the same length as P, namely n.

Next, between the lines 14 and 17, we filling the newly created histogram with values. We are doing it by iterating over all elements in P (cf. line 14), storing the digit of the element on the currently investigated level in variable *tmp* (cf. line 15) and incrementing the histogram value at the position of $tmp+1$ (cf. line 16). Thus, we are counting each of the digits. Later on, we will see, why we are incrementing at the position of $tmp+1$ and not at *tmp*. To understand more clear what is behind this step, we will have a look on how we did it the previously introduced example. The Figure 2.13 represents the third step from the example, where we wanted to sort the second row values. As we can see, after we are done, the values in histogram represents how often each digit occurs.

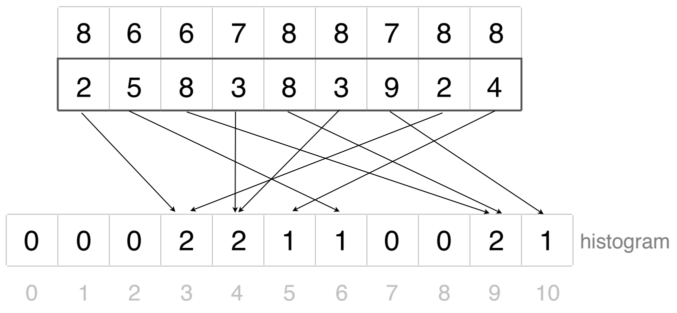

Figure 2.13: Radix sort - Filling the histogram with values

After we have filled the histogram with the right values, we need to calculate the prefix sum, therefore transforming counts to indicies. This is done between the lines 20 and 22. Thus, for each value in the histogram (cf. line 20), we are doing the following: we are summing the subsequent value in the histogram with the current value in the histogram and store the result at the position of the subsequent value (cf. line 21). Again, for better understanding, we will have a look on an example, presented in Figure 2.14 on the next page. We begin with the filled histogram from the Figure 2.13 on the facing page. The arrows comes from the both summands, the current and the previous values, and the result sum of the this both values is stored in fields with grey background, which represent the values of the prefix sum. Thus, at the last line, we can see the newly created prefix sum, based on the histogram values.

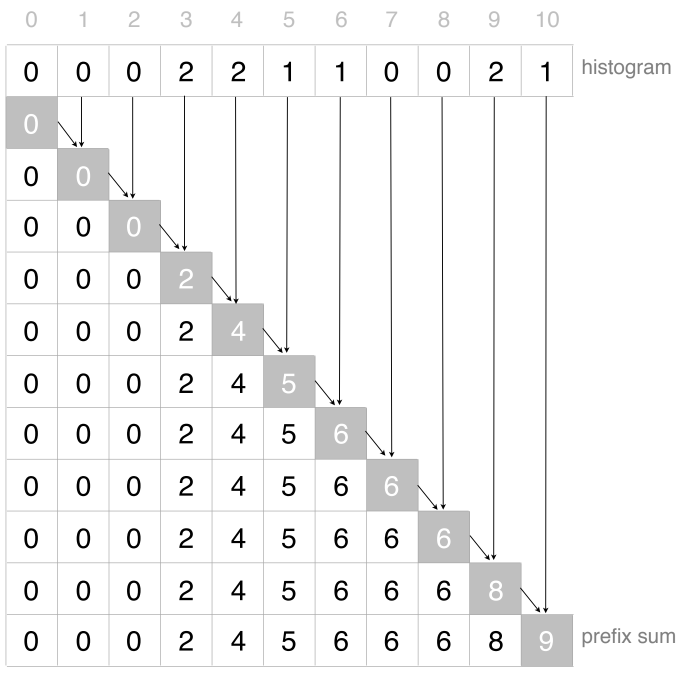

Figure 2.14: Radix sort - Calculating the prefix sum

The third internal for-loop between the lines 25 and 29 fills the previously created *dist* array with the values from the initial array P. Although, the positions of the values are defined in *count* array, which now contains the prefix sum from the previous step. The for-loop at line 25 again, iterates over all elements in array P, same as in the first internal for-loop, storing the digit of the element on the currently investigated level in variable *tmp*. Thus, we obtain the position (with *count*[*tmp*]) where we need to store the currently investigated element in the *dist* array. So, after storing the value at the right position(cf. line 27) we increment the *count* array at this position (cf. line 28), so that the next element with the same digit value will be stored next to this element, and not at the same location. Again, we will see, how it is done in our example (Figure 2.15 on the next page). As we can see the digits are pointing on the position of the digit in the prefix sum (which is *count* array). The value at this position in the prefix sum is, in its turn, the new position of the digit in the *dist* array. As we increment the values in *count* each time we add them to the *dist* array, the numbers near the arrows are showing us the new value in the *count* array after adding. Thus, at the end of this step, the prefix sum array will look as follows: 0, 0, 2, 4, 5, 6, 6, 6, 8, 9, 9.

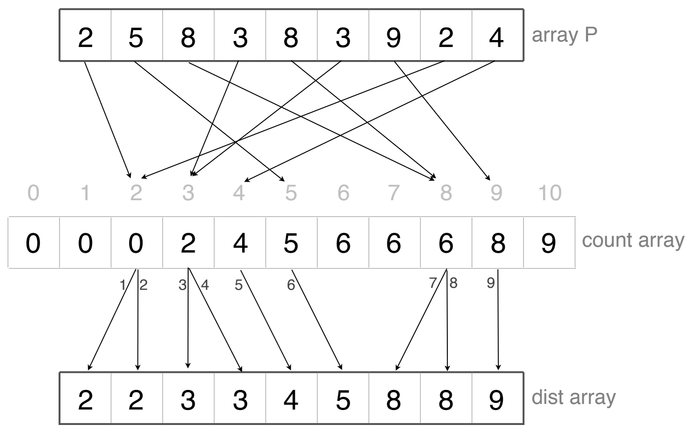

Figure 2.15: Radix sort - Distribution step

The final internal for-loop between the lines 32 and 34 copies the values from the *dist* array back to the P array, so we can continue with the next digit level in the proper way. After repeating the presented steps w times, our array becomes sorted, without doing any comparisons.

MSD-Version

As we can see from the Listing 2.11 on the facing page, the MSD radix sort (at least its recursive version) needs even more input parameters than the LSD version of radix sort. The most of them are more or less the same as in the LSD version. So, for example we have our input array P with its length n, as well as r, which represents the histogram length, which is also 10 for our example with the "RADIXSORT" word. The w parameter from the LSD radix sort is now missing, but the d variable, which was the currently investigated digit level is still there as input parameter. The big difference is that we now start with the leftmost symbol (the most significant) instead of the rightmost from LSD version. Therefore, for MSD algorithm our d becomes zero at the beginning, unlike it was at the beginning of the LSD algorithm $(w - 1)$. We introduce two new input parameters l and r, which represent the left and the right indicies of the sub array of P, as well as the new parameter m, which represents the size threshold for the MSD algorithm. This threshold is needed to stop the recursive calling of the algorithm at the certain moment.

There is another step needed before we can go more deeply into how the presented algorithm works. Namely, how we call this algorithm for the first time. We are doing it as follows: $msdRadixRec(P, n, m, t, 0, n - 1, 0)$. The first 4 paramenters were already discussed. The fifth and sixth parameter are the array indicies (that is why the sixth parameter is $n - 1$ and not just n). The 7th parameter is, as already mentioned, our d, which is, in case of MSD, zero.

```
Function: msdRadixRec(P,n,m,t,l,h,d)
Input: unsorted array P
Input: n as number of elements in P
Input: m as limit of bucket size for insertion switch
Input: t as histogram length
Input: l as left array index
Input: r as right array index
Input: d as recursion depth
Process: count, dist, tmp

//check against the size threshold
if ((r − l + 1) ≤ m)
  InsertionSort(P,l,r,d)
  return
end if

//create histogram
count ← int[t + 1]

//fill histogram
for (int i ← l; i ≤ r; i ← i + 1)
  tmp ← P[i].numberAt(d)
  count[tmp + 1] ← count[tmp + 1] + 1
end for

//compute a prefix sum
for (int k ← 0; k < t; k ← k + 1)
  count[k+1] ← count[k+1] + count[k]
end for

//perform array changes in out array
for (int i ← l; i ≤ r; i ← i+1)
  tmp ← P[i].numberAt(d)
  dist[count[tmp]] ← P[i]
  count[tmp] ← count[tmp] + 1
end for

//copy array changes back to initial array
for (int i ← l; i ≤ r; i ← i + 1)
  P[i] ← dist[i];
end for

//call each bin recursively with the next digit level
for (int k ← 0; k < t; k ← k + 1)
  msdRadixRec(P, n, m, t, l + count[k], l + count[k+1] − 1, d+1);
end for
```

Listing 2.11: MSD radix sort - adapted from [SW11]

Finally, we can start with examining the algorithm itself. The biggest part of the algorithm is the same as by the LSD version, but there are also some differences. So, for example, we are starting with a check at line 12. Here we are checking the length of our array against the size threshold m, which is defined by user. If the check returns true, we are executing the insertion sort for the current sub array and returning after. If not, we can step over to the actual radix sort structure.

Interesting thing is that we can skip explaining the next 4 for-loops, because it is doing exactly the same we did in LSD version. Thereby, we can continue at the lines 44 to 46, where we are recursively calling the algorithm again. The for-each loop at line 44 makes sure, we are iterating over all elements in the count array. At the line 45, we have the recursive call. The first 4 parameters are the same. The last parameter d is now increased by one, as we have to sort the next digit level at the next recursion step. The left and the right indicies are calculated based on the entries in the count array.

2.1.5.3 Summary

The criteria of MSD radix sort are presented in Table 2.5. The MSD radix sort is a stable algorithm and is offline. The time complexity is $O(n)$ for best and $O(n * r)$ for the worst case scenario, where r is the average element size. Its average complexity is also $O(n*r)$. The presented version is not in-place as it needs to hold the *dist* array and the histogram array in memory .There is also an in-place version of radix sort possible, but in this case the algorithm becomes not stable. The criteria for LSD version are the same, except for time complexity, which is always $O(n * r)$.

Criteria	Description
Stability	stable
On vs.Off	offline
Complexity	$O(n * r)$
Storage	not in-place

Table 2.5: Radixsort qualification

2.2 Sort-Merge Join

On the one hand, sort-merge join algorithm can be described as efficient join algorithm if we compare it to nested loop join algorithm. On the other hand, there are also other join algorithms, which perform better, for example a hash join. Sort-merge join can be logically divided into two parts - sorting and merging. Sorting operation is responsible for sorting of the two relations which have to be joined, while in merging operation the relations are merged together [JC13].

2.2.1 Explanation

As we can see in Figure 2.16, here we want to merge two words, `SORTE` and `MERGE` together, or, to be more precisely, its letters. There is no mistake in the wort `SORT`. We are using the additional `E` at the end, because we also want to explain duplicate handling. As we want to use sort-merge join for merging our words together, we need to sort these words before. Thus, we are sorting the letters in both words in alphabetical order and can see the result at line 1. The numbers above the letters represent the TID concept (TID for tuple identifier), which is needed to store the original location of the number, thus, remembering the position of the elements before sorting [SHS05].

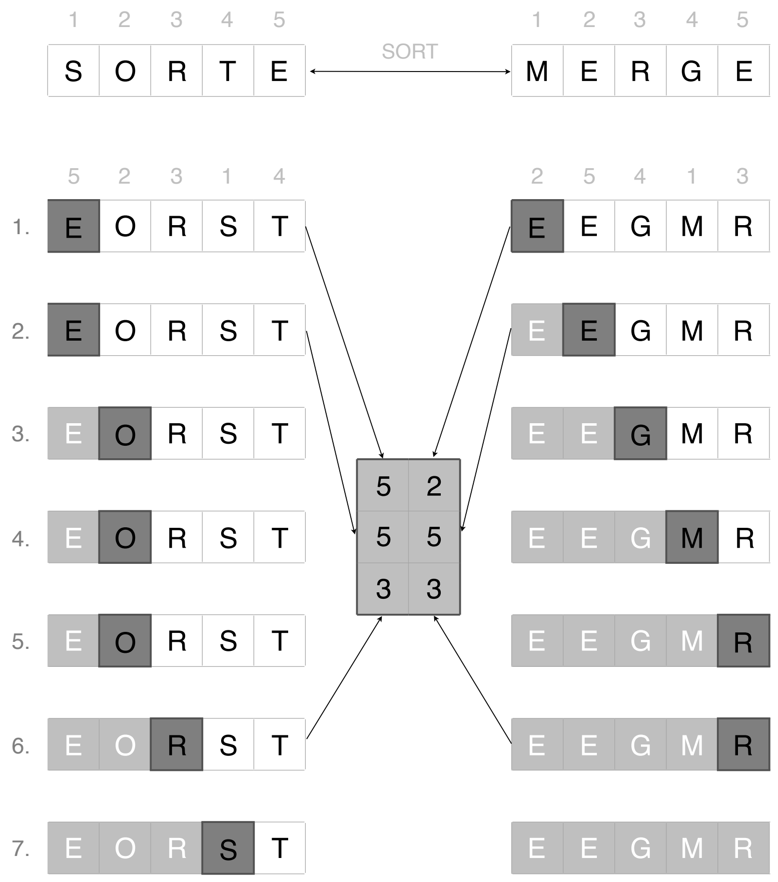

Figure 2.16: Sort-merge join workflow

Now, we can finally step over to merging step. We begin with investigating the first elements of both relations. The currently examined elements have the dark background. The elements with the lightly grey background were already investigated and the elements with the white one are not. So, the merging step of sort-merge join can be introduced in 3 logical steps:

1. If the right value is greater than the left value, then go to the next element on the left
2. If the left value is greater than the right value, then go to the next element on the right
3. If the elements are equal, then create a result tuple and check if there are some other matches next to the matched values on both sides. Then, go to the next element in both relations (which is not equal to the matched element).

As we can see, the first elements from both relations are equal, thereby we have the first match. When we find a match, we are adding the TID's (as a tuple) into the output relation (the output relation is presented in the middle of the figure). Such addition is shown with the arrows in our example. Thus, after second step we already have two tuples in the output relation: the `E`-letter in the first word merged with two initial `E`-letters from the second word. Different to the subsequent steps, if we find a match, in the final step, we need to go to the next element in both relations, as already mentioned. As we can see, this is done in step 3.

In the lines 3, 4, and 5, we have no matches. So, we just compare the values with each other and go to the next element on the side, which has the lower element. In other words, while investigating the element on the left side, we have to iterate over the elements on the right side until we find the element greater then the investigated one. As soon as we have found such element we are switching the side. Now, the element we found becomes investigated and we have to iterate over the elements on the left side. Important is that we remember the location we broke up last time, to continue with element at this location and not starting the loop from the beginning. It makes the sort-merge algorithm so efficient with sorted values.

Back to our example. At line 6 we have another match. This time with `R`-letter. Thus, same as in line 1 and 2, we put the tuple with the TID's of the `R`-letters to the output relation(thereby increasing the number of outputs to 3). Then, we are checking in both partitions if there are some more `R`-letters right to the currently found ones. As we find none, we just go to the next element in both relations, thereby reaching end of relation in the right relation.

Reaching the end of relation is a termination criterion for both relations, means we have finished merging.

2.2.2 Implementation

The first thing we do in the implementation of the sort merge algorithm is the sorting of the elements. As we can see from Listing 2.12 on the next page, the sorting step is represented at lines 8 and 9.

From line 12 on, we are starting with the merging phase. At the beginning of the merge phase we first define the help variables l with first tuple from the left relation(named `L`) and r with first tuple from the right relation(named R) at line 12 and 13. The while-loop at line 16 makes sure that we continue with the inner part until one of the relations reaches its end.

```
 1 Function: sortmerge(R,L)
 2 Input: relation R with $r_1$..$r_n$ tuples with (tid,value)
 3 Input: relation L with $l_1$..$l_m$ tuples with (tid,value)
 4 Process: r, $r_x$, l, $l_x$ as tuples
 5 Returns: output relation Q with (tid, tid)
 6
 7 //sorting phase
 8 sort R on R.value
 9 sort L on L.value
10
11 //merging phase
12 r $\leftarrow r_1$
13 l $\leftarrow l_1$
14
15 //EOR stands for "end of relation"
16 while ((r $\neq$ EOR) and (l $\neq$ EOR)) do
17   if (r.value > l.value)
18     //next(x) returns the tuple next to x in a relation
19     l $\leftarrow$ L.next(l)
20   else if (r.value < l.value)
21     r $\leftarrow$ R.next(r)
22   else
23     //put(tid, tid) adds a tuple to a relation
24     Q.put(r.tid, l.tid)
25
26     //tuples that match with r
27     $l_x$ $\leftarrow$ L.next(l)
28     while (($l_x$ $\neq$ EOR) and (r.value = $l_x$.value))
29       Q.put(r.tid,l_x.tid)
30       $l_x$ $\leftarrow$ L.next($l_x$)
31     end while
32
33     //tuples that match with l
34     $r_x$ $\leftarrow$ R.next(r)
35     while (($r_x$ $\neq$ EOR) and ($r_x$.value = l.value))
36       Q.put($r_x$.tid,l.tid)
37       $r_x$ $\leftarrow$ R.next($r_x$)
38     end while
39
40     r $\leftarrow$ R.next()
41     l $\leftarrow$ L.next()
42   end if
43 end while
44
45 return Q
```

Listing 2.12: Sort-Merge Join - adapted from [SHS05]

There are three different situations, which are defined with if-statements, possible inside of the while-loop:

1. The value of the element from the R-Relation is greater than the value of the element from the L-Relation(lines 17-19). In this case we are taking the next element from the L-Relation and store it in variable l. After l gets a new value, we start from the beginning.

2. The value of the element from the R-Relation is less than the value of the element from the L-Relation(lines 20-22). In such case, we are doing the opposite. We are taking the next element from the R-Relation, store it in variable r and start from the beginning.

3. The value of the element from the R-Relation is equal to the value of the element from the L-Relation(lines 23-42). This is the most interesting case. First, we have found a match, so we create a new tuple with TID of l and TID of r and put it into the output relation Q, as shown at line 26. Second, between the lines 28 and 39 we are checking if there are more elements with the same value from both sides. If we find such elements we also put the matching tuple in output relation Q. Last, after we found out all possible matches, l and r gets the next tuples from `L` and `R`-Partitions respectively and we can start from the beginning.

3. MPSM

Massively parallel sort-merge gives us a new look on the familar sort-merge join, which is used to process massively parallel multi-core data. MPSM is based on partial partition-based sorting and, in contrast to typical sort-merge joins, it is not using the difficult to parallelize merging step to sort it completely. Instead, it runs independently in parallel [AKN12a].

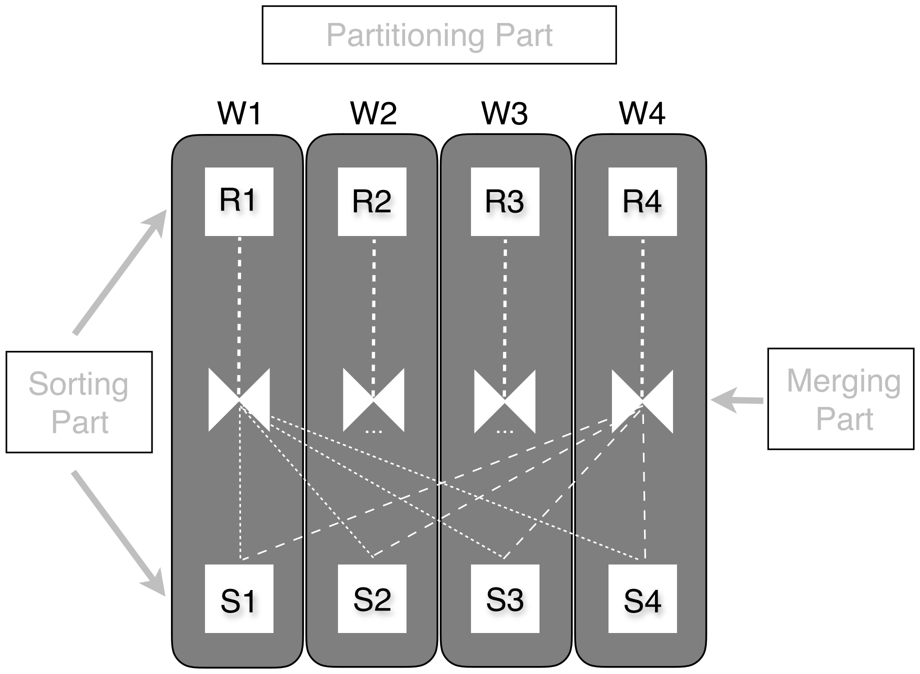

Figure 3.1: MPSM join with 4 Workers, adapted from [AKN12a]

In this work, we are concentrating on a basic form of MPSM, so called B-MPSM. There are also two other versions of MPSM, which extend from B-MPSM, namely P-MPSM and D-MPSM, presented in [AKN12a]. P-MPSM additionally ranges the input data partitionally and D-MPSM is a RAM-constrained implementation of MPSM. At the end of the explanation section, we will have a look on the P-MPSM, as it is more efficient implementation of MPSM, to see where the difference between them both is(B-MPSM and P-MPSM).

The main structure for all MPSM algorithms is presented in Figure 3.1 on the preceding page. All MPSM structures have the same steps: partitioning, sorting and merging. Well, the steps themself are implemented differently in most of the cases. All of these steps will be discussed separately for B-MPSM as well as the difference with P-MPSM.

3.1 Explanation

At the beginning of the B-MPSM (Basic-MPSM) we get two data inputs: S and R, which we need to join. We call R data input the private part and S data input the public part.

We start with a partitioning part, where each data input is equally divided in chunks among the workers. The next step is sorting, where each worker sorts the private as well as public part. After the sorting step, each private part chunk is merged with all public part chunks.

3.1.1 Partitioning Step

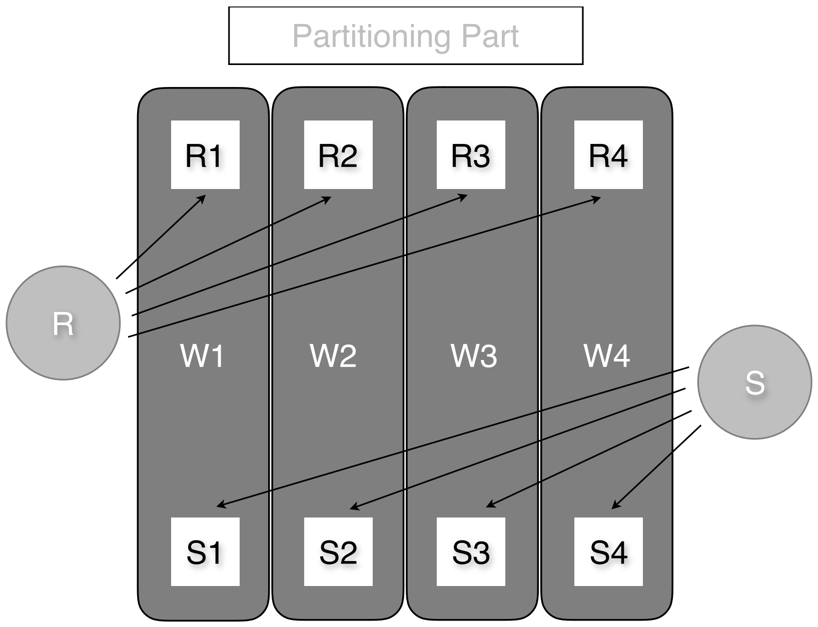

Figure 3.2: B-MPSM join with 4 Workers - Partitioning Part, adapted from [AKN12a]

As we can see from Figure 3.2 on the preceding page, in an example with 4 workers, the data input R is divided into 4 equally sized parts($R_1..R_4$) and assigned to the associated worker(R_1 to W_1 and so on). The same is done for data input S. It is also equally divided in 4 parts($S_1..S_4$) and assigned to the workers. The number of workers complies the number of cores on the machine the algorithm is executed on.

As it is not always possible to divide the data equally, the last worker gets the rest elements after dividing. It means, additionally it gets N mod W elements, where N is the size of the initial part (R or S) and W is the numbers of workers we have. So, for our example with 4 workers, if we have 103 elements in the initial part, the first 3 workers gets 25 elements each (103 / 4). The last worker, in its turn, gets 28 elements ((103 / 4) + (103 mod 4)). As MPSM works with a huge amount of data, we are not dividing the rest elements between the workers, because exact division would complicate the splitting process while it brings only a small benefit.

3.1.2 Sorting Step

In the sorting step, each worker sorts its own private, as well as public part. There is a multi-phase sorting algorithm behind the sorting step. The reason it is developed so complicated is that the MPSM is developed to join large join keys.

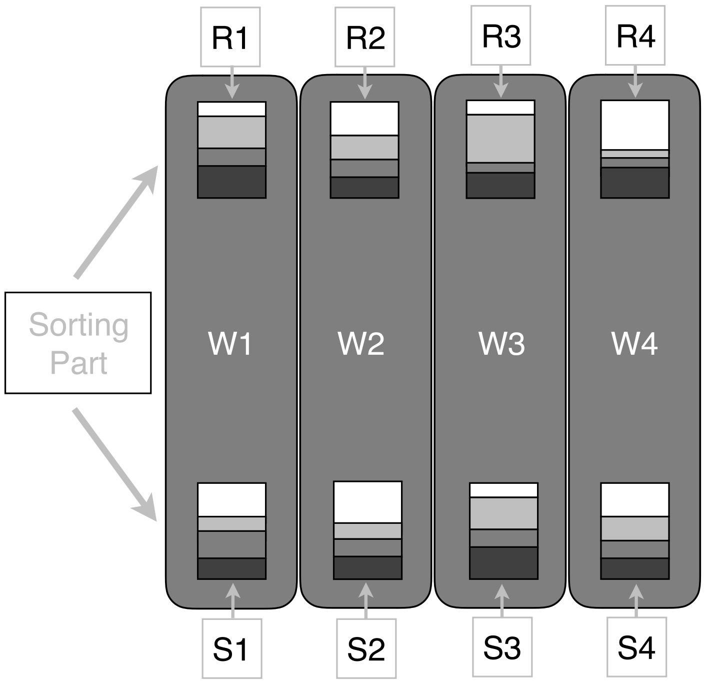

Figure 3.3: B-MPSM join with 4 Workers - Sorting Part, adapted from [AKN12a]

Well, before we can start examining what happens in the sorting step, we need first to define some parameters for it. First of all, the `NUMBER_OF_BITS`(named B) parameter . It represents the number of bits we are investigating in each recursion step of the radix sort algorithm. According to the value of B, we can calculate how many partitions will be created in each recursion step. We calculate it as follows: 2^B. So, for example, if we need to investigate 8 bits per recursion, the radix algorithm will create 256 partitions, according to all possible combinations for 8 bits.

The second parameter is the `INTROSORT_SWITCH` (named T). It is a size threshold for radix sort. It means, if one of the partitions, created by radix algorithm contains less elements than defined in T, we need to switch to introsort algorithm.

The last parameter we need is `INSERTIONSORT_SWITCH` (named I), which is also the size threshold, but this time for introsort. Same to the previous one, we switch to insertion sort if introsort partition contains less elements than defined in I.

Now, we will investigate each sorting step phase individually [AKN12a, AKN12b]:

1. recursive in-place Radix sort, which creates 2^B partitions, based on B most significant bits, in each recursion step. Partitions are created by calculating the 2^B bucket histogram for figuring out the borders for each partition. Next, the data elements are moved into the partition to which they belong.

2. Introsort, which is executed on partitions with less than T elements, resulted from the first step. The introsort algorithm works as follows:

   (a) $2 * log_2(N)$ quicksort recursion steps, where N is the length of the data input at the beginning. If it is not enough (data input is still not sorted), switch to heapsort.

   (b) if quicksort partition contains less than I elements, switch to the final insertion sort.

The default configuration for MPSM, provided by [AKN12a] is the following:

1. `NUMBER_OF_BITS`$(B) = 8$

2. `INTROSORT_SWITCH`$(T) = 100$

3. `INSERTIONSORT_SWITCH`$(I) = 16$

We can see the result of sorting in Figure 3.3 on the preceding page. In this example, the white fields of the parts represent the lowest values. Respectively, the darker the color the higher the value. Thus, we can see that all parts in all workers are sorted now.

3.1.3 Merging Step

The merging step is done by sort-merge join presented before. We can use it, because at this moment all chunks are already sorted. Every private input chunk needs to be merged with all public input chunks to make sure all elements have been processed.

First, we merge each private part with each public part from the same worker together. As we want to merge every private part with all public parts, not only with the one in the current worker, we need to invoke the public parts from other workers, as shown in Figure 3.4. So, since every worker is done with merging its own private and public parts together, we alternately invoke the public parts from another workers. Thus, in total, we are merging 16 times in our example with 4 workers (each private with each public). After we finished merging, both our initial parts are now merged together and thus, we are done.

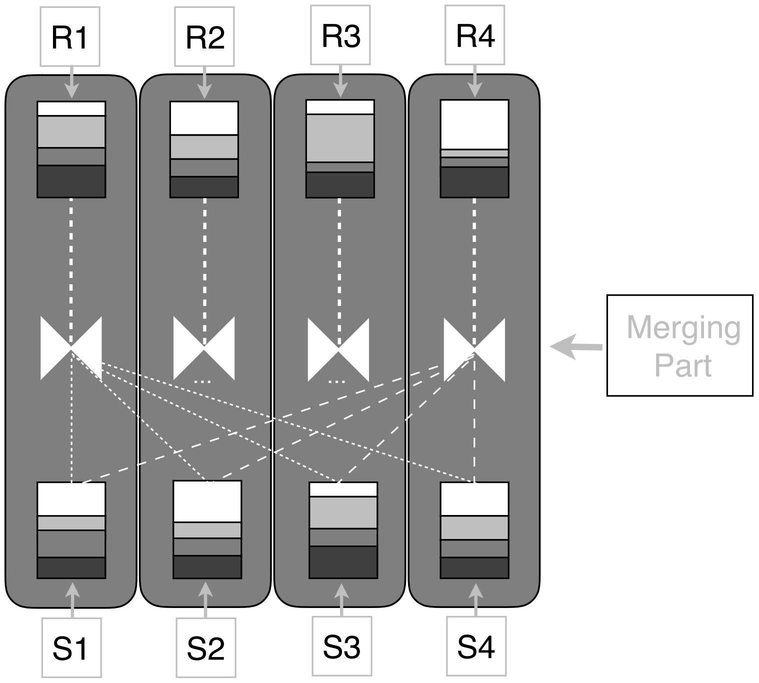

Figure 3.4: B-MPSM join with 4 Workers - Merging Part, adapted from [AKN12a]

3.1.4 P-MPSM

The range partitioned MPSM (P-MPSM) represents an improved version of B-MPSM. Additionally to B-MPSM, the P-MPSM join has an extra phase, namely range partition phase, as presented in Figure 3.5 on the following page. In this phase, the private data input is divided between the workers in the way that after this phase, all parts ($R_1..R_4$) are range partitioned. This is done using a histogram-based technique to make sure the chunks in each worker are balanced [AKN12a].

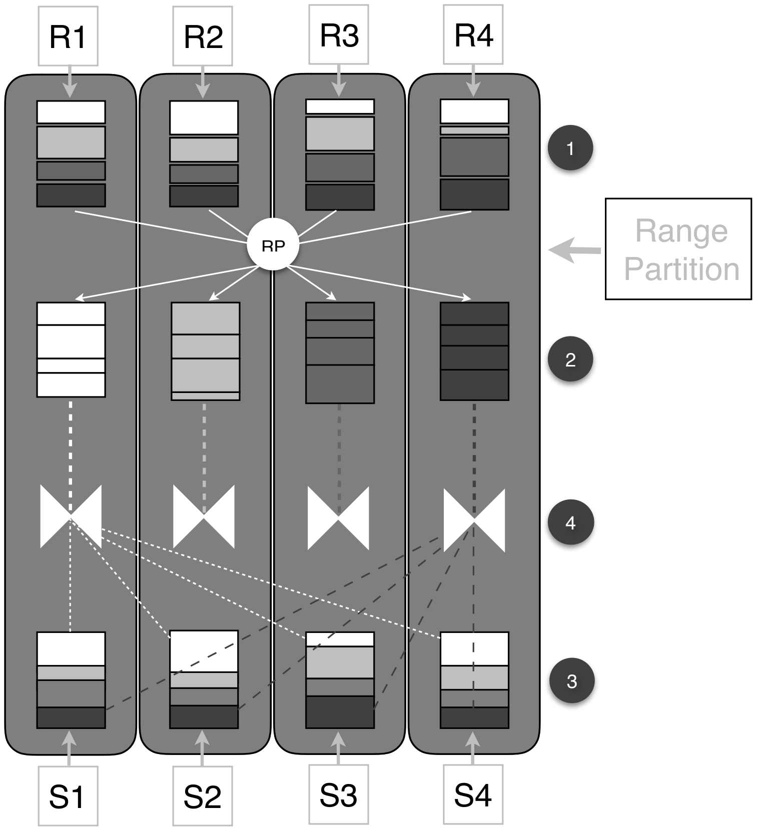

Figure 3.5: P-MPSM join with 4 Workers, adapted from [AKN12a]

In step 1 (steps are shown on the right) each worker gets its own private and public part, same as in B-MPSM. Then, each private part value in each worker is range partitioned. The ranges for all workers are the same. So, after this step, we know exactly which value belongs to which range partition. Important, is that we are not sorting the values in step 1. The different shades (from white over light grey and dark grey to black) are showing in this example the range partitions we have, and not that the values are sorted, like we had in previous examples. Thus, after the step 1 in each worker we have clustered the private part in 4 partitions (as we use 4 workers in our example), but the values within these partitions are still not sorted.

In step 2, we assign partitions with the same range to the associated worker. Thus, for example, worker 1 gets all first partitions, which have the lowest values (white shade) from all workers. Worker 2 gets the second level partitions (light grey shade) and so on. Now, each worker contains the values in a specific range. Additionally, we are now sorting the values. So, in the end of step 2, the private part values in each worker are not only sorted, they are additionally range partitioned. In contrast, the public part values are just sorted, same, as we had in B-MPSM implementation (step 3).

The profit of the range partition phase can be seen in merge phase (step 4). Same as before, we are merging all private parts with each public part. The difference is that we now searching for the first element from the private part in the public part using interpolation search [Sed90], instead of iterating over all values from the beginning of the public part. Thus, we can start at the right position in public part, iterate over both parts, put the matched tuples in output relation, if these were found, and break at the moment the next value in public part is higher than the last value in private part.

As we can see, for P-MPSM we need more time for sorting the private parts in each worker. The benefit is that we need less time for merging step.

3.2 Implementation

In this chapter we will introduce the implementation of B-MPSM algorithm. We have splitted each single B-MPSM step (partitioning, sorting, merging) in associated engines, which are responsible for this steps. Thus, we are going to introduce the implementation of 3 engines, which are responsible for B-MPSM logic. Additionally, in the last section, we will discuss how the parallelization works in B-MPSM.

3.2.1 Splitting Engine

At the beginning of MPSM we need to divide data equally between workers, so that every worker gets the same number of tuples. A tuple consists of a TID and a value, which is the actual value from the input array. The TID is needed to identify the elements of the input array, otherwise we cannot locate the original position of the element after the array becomes sorted [SHS05]. The original location is needed to connect resulted tuples with other elements in the table it belongs to, as we want to sort the full table and not only one column. Thus, we are faced with several tasks, before we can let workers do their job, namely:

1. We need to transform all elements from the initial array into tuples with following form: <TID, Value>.
2. Created tuples must be equally divided between workers, i.e. we need to split initial data set into a certain number of sets, depending on the number of workers.
3. Since we deal with C++, we also need to store a length of the sets

We are introducing some new structures, which will help us, on one hand, to solve these tasks, on other hand, to organise the code. The structure we will start with is TID-struct, presented in Listing 3.1 on the next page. The struct is needed to use tuples instead of simple values and represents the TID concept mentioned before.

Next struct in our list is Array, presented in Listing 3.2 on the following page. As the name says, the struct will contain an array with tuples as well as length of the array.

```
template<typename T>
struct TID {
  long tid;
  T value;
};
```

Listing 3.1: TID struct

```
template< typename T>
struct Array {
  TID<T> *ary;
  size_t size;
};
```

Listing 3.2: Array struct

A *Part* struct is what every worker will receive as an input. It contains two Array structs, one for the private and one for the public input. We are also storing the length of the longest of both Array structs. Storing it brings a great benefit for the sorting phase, but we shall return to it later. Of course, the Array's contain only a part of the input data set, and not a full one, therefore the Part structs are created only after the splitting of the input array. The *Part* struct itself is shown in Listing 3.3.

```
template< typename T >
struct Part {
  Array<T> privatePart;
  Array<T> publicPart;
  size_t maxLength;
};
```

Listing 3.3: Part struct

Finally, we created a new class Structure, which contains all *Part* struct elements, as shown in Listing 3.4 on the facing page. The class has a constructor, which receives both initial arrays as a parameter. The lengths of the arrays are also transmitted.

Once we implemented the structs, we now have basic structures for MPSM, but the converting step is still missing. So, we need to introduce some new functions to fill the newly introduced structs.

In Listing 3.5 on the next page, we are presenting the content of the constructor of previously introduced *Structure* class. We will have a look on the content more detailed. At line 5 we are allocating an array with part structs. At lines 8 and 9 we are deciding how many elements every worker will contain. It is done by dividing the total number of elements in array by the number of workers we have. Note, the last worker can get (w - 1) more elements (where w is the number of workers), than all other workers. We handled the separation this way, because an almost exact division would complicate the

```
template< class T >
class Structure {
public:
  Parts<T> *parts;
  size_t maxLength;
  Structure(T*, T*, size_t, size_t);
};
```

Listing 3.4: Structure class

```
 1 template< class T >
 2 Structure<T>::Structure(T *privatePart, T *publicPart,
 3   size_t privatePart_size, size_t publicPart_size) {
 4
 5   parts = new Parts<T>[NUMBER_OF_WORKER_THREADS];
 6   maxLength = 0;
 7
 8   size_t privateDivider = privatePart_size / NUMBER_OF_WORKER_THREADS;
 9   size_t publicDivider = publicPart_size / NUMBER_OF_WORKER_THREADS;
10
11   for (size_t i = 0; i < NUMBER_OF_WORKER_THREADS; i++) {
12
13     parts[i] = *setParts(privatePart, publicPart,
14       privateDivider * i,
15       privateDivider * (i + 1) + modulo(privatePart_size, i),
16       publicDivider * i,
17       publicDivider * (i + 1) + modulo(publicPart_size, i));
18
19     maxLength = getMax(maxLength, parts[i].maxLength);
20   }
21 }
```

Listing 3.5: Structure constructor

splitting process while it brings only a small benefit for the rest of the algorithm. Even if we will process on a 64 core machine, the last worker will contain 63 elements more in worst case. Taking into account, that MPSM is mainly designed for a huge amount of data, will mean, that such a small amount of extra elements will have only a minimal influence on algorithm performance.

Between the lines 11 and 20 we are initiating the Part structs of our parts array by calling the `setParts` method, which is described in Listing 3.6 on the following page. As parameters it gets the input array with boundaries for private as well as for public data input. The `setParts` method, in its turn, is creating each part individually by calling `setPart` method (with one s), which finally transforms the input array elements into tuples. The `setPart` method is presented in Listing 3.7 on the next page. The initialisation of the variable `maxLength` from `Part` struct occurs in `setParts` method.

```
template< typename T >
Parts<T>* setParts(T *privatePart, T *publicPart,
  size_t privateBegin, size_t privateEnd,
  size_t publicBegin, size_t publicEnd) {

  Parts<T> *parts = new Parts<T>;
  parts->privatePart = *setPart(privatePart, privateBegin, privateEnd);
  parts->publicPart = *setPart(publicPart, publicBegin, publicEnd);
  parts->maxLength = getMax(privateEnd - privateBegin,
                             publicEnd - publicBegin);

  return parts;
}
```

Listing 3.6: setParts method

```
template< typename T >
Array<T>* setPart(T *part, size_t first, size_t last) {

  Array<T> *collection = new Array<T>;
  collection->size = last - first;
  collection->ary = new TID<T>[last - first];

  for (size_t i = 0; i < last - first; i++) {
    TID<T> *element = new TID<T>;
    element->tid = i + first;
    element->value = part[i + first];
    collection->ary[i] = *element;
  }
  return collection;
}
```

Listing 3.7: setPart method

In summary, our splitting engine handles the following problems:

1. We transformed input array elements into tuples, so now, we can identify the elements after the sorting step (*setPart* method).
2. We divided the input data among the workers, so that every worker, except the last one, gets an equal amount of elements to work with.
3. We are not only maintain the size of the resulted tuple sets, we also storing the length of the longest part, what improves the sorting performance.

3.2.2 Sorting Engine

In previous sections we had a look on algorithms, which are directly or indirectly used in MPSM. We also explored an algorithm, which combines several other algorithms (a so-called hybrid sorting algorithm), namely introsort. The sorting phase of MPSM also consists of two sorting algorithms: radix sort and introsort.

Before we move on to a more detailed description of each of the algorithms used, we will discuss some preparation steps in sort method, which is presented in Listing 3.8. The sort method gets as parameter an *Array* struct, which contains the information about tuple array and its size. If the size of the array is smaller than 2, means there is only one tuple or none, we do not need to do anything.

```
 1  template< class T >
 2  void sort(Array<T> data_array) {
 3
 4    size_t data_size = data_array.size;
 5    TID<T> *data = data_array.ary;
 6
 7    if (data_size < 2)
 8      return;
 9
10    if (data_size >= INTROSORT_SWITCH) {
11
12      //create shifter and mask based on data in configs
13      unsigned long shifter = sizeof(T) * 8 - LOG2_OF_NUMBER_OF_BINS;
14      T mask = (T)(((T)(NUMBER_OF_BINS - 1)) << shifter);
15
16      radixSort<NUMBER_OF_BINS, LOG2_OF_NUMBER_OF_BINS, INTROSORT_SWITCH>
17                (data, data_size - 1, mask, shifter);
18    }
19    else {
20      introsort(data, data_size);
21    }
22  }
```

Listing 3.8: Preparation for sorting method

The `INTROSORT_SWITCH` constant, defined in configuration file, tells us, when do we need to switch from radix sort to introsort. The default value for this constant is 100 [AKN12a]. It means, if the partition contains less than 100 elements, we have to sort it with introsort (cf. line 10). As it is not hard coded, we can configure this constant in configuration file, which is presented in Listing 3.9 on the next page.

There are another two interesting constants, presented in configuration file, namely `NUMBER_OF_BINS` and `LOG2_OF_NUMBER_OF_BINS`. The `NUMBER_OF_BINS` constant represents the number of bins radix sort creates in each recursion step. The `LOG2_OF_NUMBER_OF_BINS` is simply a binary logarithm of bins number. It is also stored in configuration file and not calculated directly in algorithm for performance reasons. The default value for

```

const int NUMBER_OF_WORKER_THREADS = 8;

const unsigned long NUMBER_OF_BINS = 256;
const unsigned long LOG2_OF_NUMBER_OF_BINS = 8;

const unsigned long INTROSORT_SWITCH = 100;
```

Listing 3.9: Configuration

the NUMBER_OF_BINS constant is 256 [AKN12a]. Therefore, the default value for the LOG2_OF_NUMBER_OF_BINS constant is 8.

Returning to our algorithm, we can conclude, that if our array is smaller than the value of INTROSORT_SWITCH constant, we are executing the final introsort, if not, we can proceed with radix sort. There are 2 additional steps needed to proceed radix sort.

First, we need to calculate the shifter, a variable, which says, how many bits we need to shift after each recursion step of radix sort. The shifter is calculated as follows: we are taking the datatype size, which is presented in byte, and multiply it by 8, to convert it into bit value. The resulted value has to be reduced by LOG2_OF_NUMBER_OF_BINS constant, so we find out how many bits we need to shift. We will introduce a quick example: We have an integer as datatype for our input array, which has a size of 4 byte, means 32 bits. We also keep the default number of bins, namely 256. Thus, we have to subtract 8 from 32 to receive the shifter value. Therefore, in the first recursion step of radix sort, we will shift 24 bits to the right, thus, investigating 8 most significant bits of the input elements.

Finally, we can start with radix sort. We already presented an abstract version of MSD radix sort in Listing 2.11 on page 29. The implementation of MSD radix sort, used in MPSM, is nearly the same, except that we are using the in-place version of it [CBB+15].

3.2.3 Merging Engine

After we completed the data sorting, we can move on to merging step. Comparing to the other engines, the structure of the merging engine is very simple. The only thing we are using here is sort-merge, or to be more precise, the second phase of sort-merge algorithm, as the first phase is done by sorting engine. Thus, we are merging each private part with each public part. The number of merges we need can be calculated with the following formula: w^2, where w is the number of workers we use, which in its turn, represented by the number of cores we have on our machine. So, for our example with 4 workers, in the merging engine we are calling the merge join 16 times in total. After the merging engine is done with its work, our private and public data inputs are merged together.

3.2.4 Multi Threading

Now, let us combine all introduced engines into one and see how the parallelisation of processes is implemented. At Listing 3.10 we can see the core method of MPSM algorithm. We start with a splitting engine, which returns the *Structure* class. Then, we are creating the certain number of threads, depending on the number of cores we have or the number of `NUMBER_OF_WORKER_THREADS` constant we entered in configuration file. Each of these threads is sorting its own part of elements, defined in *Structure* class. This is what sorting engine is managing. Finally, after all workers have completed their sorting work, we are merging each private part with all public parts, again, in parallel. Since each worker has finished merging, we are done. As we can see, we are adding a new thread each time, instead of creating the thread pool. The reason for this is that the time the worker needs to execute its work is huge, comparing to the small overhead of the thread creation.

```

void MPSM(T *privatePart, T *publicPart,
  size_t privatePart_size, size_t publicPart_size) {

  Structure<T> *structure = new Structure<T>(privatePart, publicPart,
                                privatePart_size, publicPart_size);

  Parts<T> *parts = structure->parts;

  boost::thread_group threads;

  //sorting phase
  for (int k = 0; k < NUMBER_OF_WORKER_THREADS; k++) {
    Parts<T> part = parts[k];

    threads.add_thread(new boost::thread(
                             boost::bind(&threadSortCall<T>, part)));
  }
  threads.join_all();

  //merge phase
  for (int k = 0; k < NUMBER_OF_WORKER_THREADS; k++) {
    Array<T> publicPart = parts[k].publicPart;

    for (int l = 0; l < NUMBER_OF_WORKER_THREADS; l++) {
      threads.add_thread(new boost::thread(
                          boost::bind(&threadMergeCall<T>,
                          publicPart, parts[l].privatePart)));
    }
    threads.join_all();
  }
}
```

Listing 3.10: MPSM main

3.3 Improvements

In this section we will present the optimizations we did for the MPSM algorithm. As we were concentrating on the B-MPSM version, we were trying to improve the algorithm in the way, that our improvements are also applicable for the P-MPSM join, an improved version of B-MPSM. We will present 2 improvements, namely the `SMART_SHIFTER` and `INT2INS`. We will investigate both improvements to understand which benefit they bring.

3.3.1 Smart Shifter

We start with the smart shifter, which was developed to increase the performance of radix sort and therefore increasing the performance of the MPSM algorithm. To understand how smart shifter works, we need first to remember how radix sort algorithm performs. It gets an array of elements as an input, represents these elements in bits and investigates a specific number of bits in each recursion level. We have defined the number of bits, examined by radix sort in each recursion level in `LOG2_OF_NUMBER_OF_BINS` parameter in our configuration file, presented in Listing 3.9 on page 46. For MPSM algorithm this parameter is equal to 8.

So, if our input array has an unsigned integer as a data type, the radix sort algorithm will have 4 recursion levels, investigating 8 bits in each recursion level. Thus, if our input array, for example, will contain only the values from 0 to 255 (means only the 8 least significant bits are set), the radix sort will still create 256 partitions in each recursion level. However, in first 3 recursion levels, all elements from our input array will be placed in the first bin, where all bits are set to zero. Only in the fourth recursion level, which investigates the last 8 bits of integer, the values of the input array gets partitioned in different bins (of course, only if all of these values are not equal). Thus, we can actually skip the first 3 recursion levels of the radix sort and start directly with the fourth level, because it is the only level, where the actual partitioning happens. This is exactly what the smart shifter does.

The smart shifter needs an additional parameter, which says, which range the elements in our input array have. Therefore we have introduced an additional parameter in our configuration file, namely `VALUES_RANGE`. We have upgraded the *sort* method, presented in Listing 3.8 on page 45, to make the smart shifter work. The new implementation of the *sort* method is presented in Listing 3.11 on the facing page.

As we can see, at line 12 we define a variable *bitsnumber* which represents the number of bits for the current data type T. The implementation of smart shifter is between the lines 17 and 22. The variable *limit* gets the value range of our input array. Then, the while loop at line 19 determines the first bit, which is not set to zero, starting with the most significant bit. We are storing this bit in r. As we are always shifting by the value, defined in `LOG2_OF_NUMBER_OF_BINS`, we can not simply assign r to the shifter. We need to transform r in the way that r mod `LOG2_OF_NUMBER_OF_BINS` returns zero, or, if it already returns zero - additionally subtract the `LOG2_OF_NUMBER_OF_BINS` from r. This is done at line 20 and the result is assigned to the variable $shifter$.

```
template< class T >
void sort(Array<T> data_array) {

  size_t data_size = data_array.size;
  TID<T> *data = data_array.ary;

  if (data_size < 2)
    return;

if (data_size >= INTROSORT_SWITCH) {

    unsigned long bitsnumber = sizeof(T) * 8;
    unsigned long shifter;

#ifdef SMART_SHIFTER

    unsigned long r = 1;
    unsigned long limit = VALUES_RANGE;
    while (limit >>= 1) r++;
    shifter = (r \% LOG2_OF_NUMBER_OF_BINS == 0) ?
            r - LOG2_OF_NUMBER_OF_BINS :
            (r / LOG2_OF_NUMBER_OF_BINS) * LOG2_OF_NUMBER_OF_BINS;

#else

    shifter = bitsnumber - LOG2_OF_NUMBER_OF_BINS;

#endif // SMART_SHIFTER

    //create shifter and mask based on data in configs
    unsigned long shifter = sizeof(T) * 8 - LOG2_OF_NUMBER_OF_BINS;
    T mask = (T)(((T)(NUMBER_OF_BINS - 1)) << shifter);

    radixSort<NUMBER_OF_BINS, LOG2_OF_NUMBER_OF_BINS, INTROSORT_SWITCH>
              (data, data_size - 1, mask, shifter);

  }
  else {
    introsort(&data[0], data_size);
  }
}
```

Listing 3.11: New preparation for sorting method

We will have a look on how it works, based on the previous example. As we have the values between 0 and 255, the `VALUES_RANGE` parameter becomes 255. The *bitnumber* becomes 32, as the unsigned integer data type has 32 bits. The variable r after the while loop at line 19 becomes 8, according to the first bit, which is not equal to zero, starting with the most significant bit. As r is divisible by `LOG2_OF_NUMBER_OF_BINS` without remainder, we additionally subtract `LOG2_OF_NUMBER_OF_BINS` from r. So, in our case, we are not shifting at all,instead of shifting for the 24 bits (cf. line 26), like in regular radix sort algorithm, thereby investigating the 8 least significant bits directly at the beginning of the radix sort.

3.3.2 Int2Ins

As we know, if any radix sort partition contains less elements than defined in `INTROSORT_SWITCH` we switch to introsort. Well, with our configuration it means that we switch to introsort, if the result partition contains less than 100 elements. The introsort, in its turn, calculates the switch to heapsort with the following formula: $2 * log_2(n)$, where n is the number of elements, and the calculation of log_2 is a very expensive operation in terms of performance. Additionally to the calculation of log_2, the introsort is using the *partition* method, which iterates over all elements of the sub array in each recursion level of introsort and swapping them if needed. It also uses the $medianOfThree$ method for determining the pivot element, which potentionally can have up to 3 swaps per call. Under certain conditions, we also switch to heapsort for the sorting. Finally we make the final insertion sort in any case.

```
template< class T >
void introsort(TID<T>* data, size_t n)
{
#ifdef INT_TO_INS

  insertion(&data[0], n);

#else

  size_t depth = log(n, 2) * 2;
  introsort_r(&data[0], 0, n - 1, depth);
  insertion(&data[0], n);

#endif // INT_TO_INS
}
```

Listing 3.12: New main Introsort Method

Due to a lot of overhead, created by the introsort (points, presented before), we believe that for the arrays, which contains less than 100 elements, the simple insertion sort will perform better than the introsort. Therefore, if any radix sort partition contains less than 100 elements, we are using the insertion sort to sort it, instead of introsort.

The implementation of how we manage it can be seen in Listing 3.12 on the preceding page. We have just changed the main *introsort* method, where the recursion method of introsort is called, in the way that it just calls the insertion sort.

4. Evaluation

In this chapter, we present results of our tests we have made for the B-MPSM algorithm to evaluate the performance of the improvements we have presented in Section 3.3 on page 48. We will start with presenting the test environment, meaning the hardware of our test machine, continue with the tests for B-MPSM algorithm itself, without any improvements, so we have test data we can compare with, and finally, we will evaluate the B-MPSM algorithm with each of the improvements we made. For every test, that we present, we took 50 measurements to achieve the stable results.

4.1 Test Environment

We execute our performance evaluation on a machine with an Intel(R) i7-4790 clocked at 3.60GHz as a processing device. This device has 4 physical cores(and due to hyper-threading 8 hardware contexts) and has 32KB L1-Cache and 256KB L2-Cache per core as well as 8MB L3-Cache. It has 16 GB of main memory. The operating system we have used is 64-bit version of Windows 10.0.10240.

4.2 Standard B-MPSM

For our tests we have used 2 data sets as input array filled with random values in different length: 5Mio and 10Mio tuples. For each of the arrays we did the tests in 5 different value ranges: 10000, 100000, 1Mio, 10Mio and 100Mio. As the configuration for the tests we took the default configuration for MPSM algorithm, namely:

1. `NUMBER_OF_BITS` = 8
2. `INTROSORT_SWITCH` = 100
3. `INSERTIONSORT_SWITCH` = 16

Thereby we provided 10 tests with 50 measurements each, resulting in 500 measurements in total. We can see the result for the array with 5Mio entries in Figure 4.1 and the test result for the array with 10Mio entries in Figure 4.2. The results represent an average value of the measurements we did.

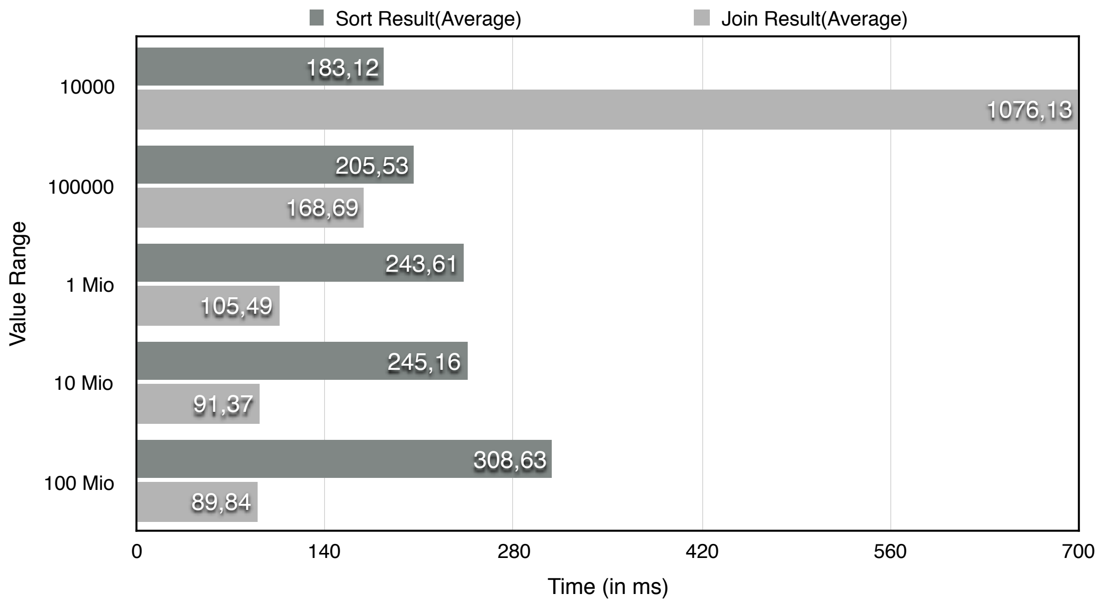

Figure 4.1: Standard B-MPSM tests for 5Mio tuples

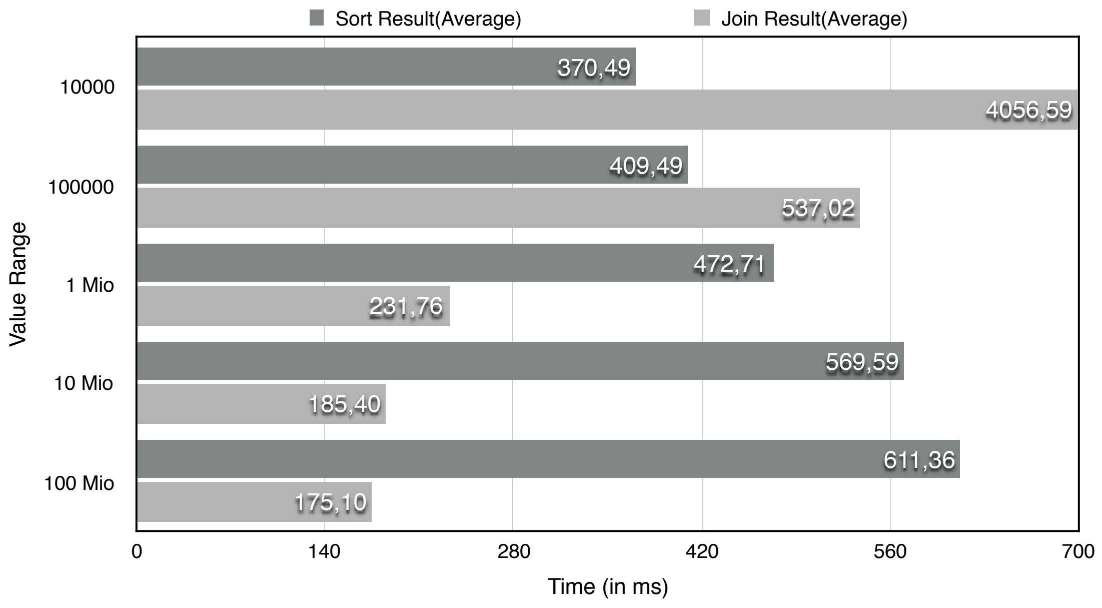

Figure 4.2: Standard B-MPSM tests for 10Mio tuples

As we can see, the sorting engine as well as the merging engine depends on the value range and the number of elements in the input array. The higher the value range, the longer the sorting engine needs to sort this data. The reason for this fact is that the radix sort needs more time to sort data with higher value range, because then it needs to create partitions on the high recursion levels (while examining the most significant bits). In contrast, the merging engine needs less time with the high value range, because the probability of having the same values in private and public inputs is lower and therefore the merging engine needs to do less work. We can see, that the behavior is the same for 5Mio entries array and 10Mio entries array.

Thus, we can conclude, that the B-MPSM algorithm depends on the number of elements in the array. The sorting step needs more time to sort the values with the high value range and less time for sorting the values with the low value range. The merging step behaves opposite to the sorting step. It needs less time to sort the values with the high value range and more time to sort the values with a low value range.

4.3 Comparison with Smart Shifter

In this section we are providing the comparison between Standard B-MPSM algorithm with this algorithm additionally using the smart shifter improvement, presented in Section 3.3.1 on page 48. First, we are doing the same tests we did in the previous section with activated smart shifter. Second, we will compare the both tests with each other and finally, evaluate the results.

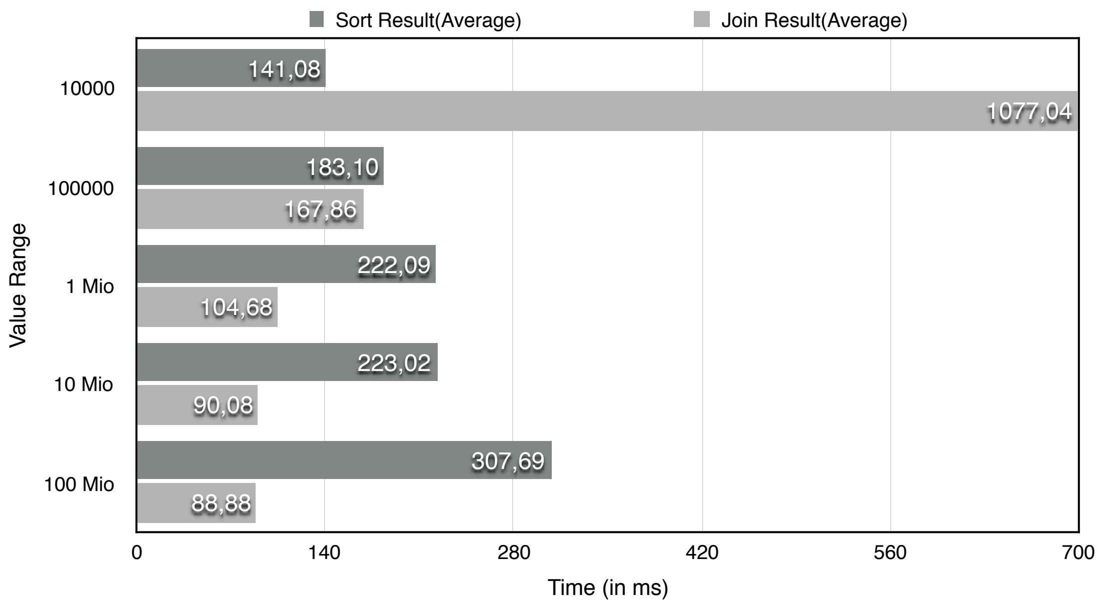

Figure 4.3: B-MPSM tests with smart shifter for 5Mio tuples

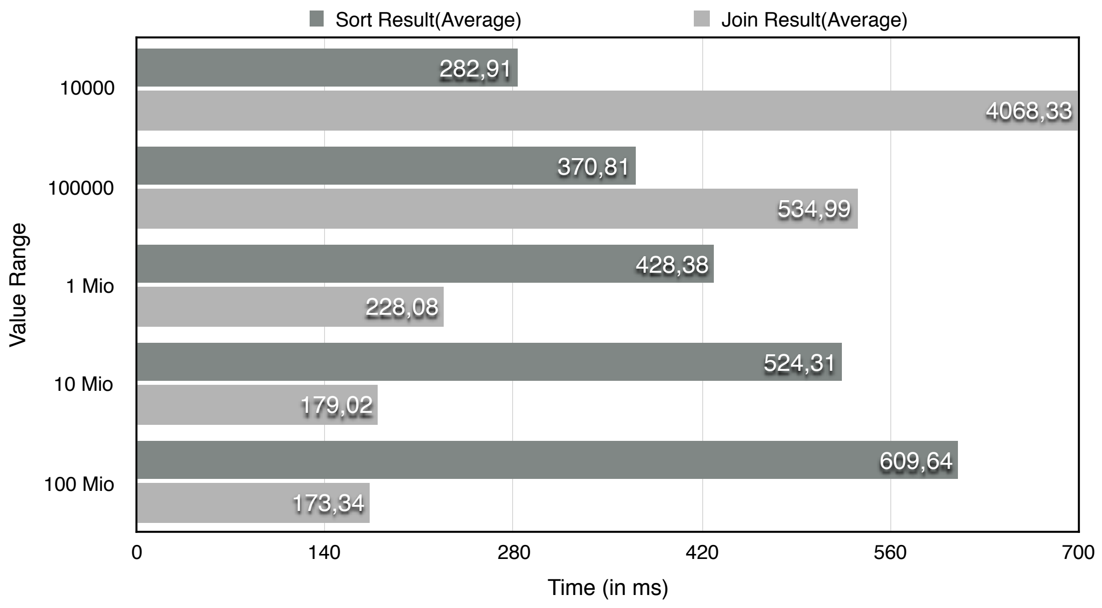

Figure 4.4: B-MPSM tests with smart shifter for 10Mio tuples

The test results using the smart shifter for 5Mio array entries are presented in Figure 4.3 on the preceding page and for the 10Mio array entries in Figure 4.4. As we can see, the behavior of the B-MPSM with smart shifter is almost the same as we had in standard B-MPSM version. The performance depends on the number of elements in the array. Same as in the tests for the standard version, the sorting engine performs better with low range values and merging engine with higher ones.

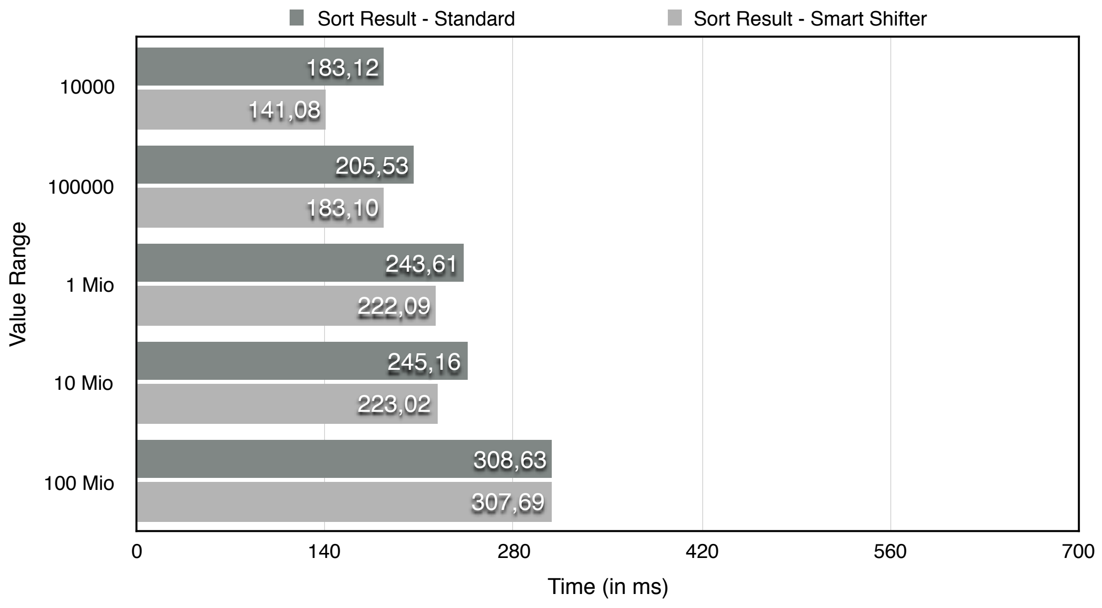

Figure 4.5: Comparing sorting phase with smart shifter for 5Mio tuples

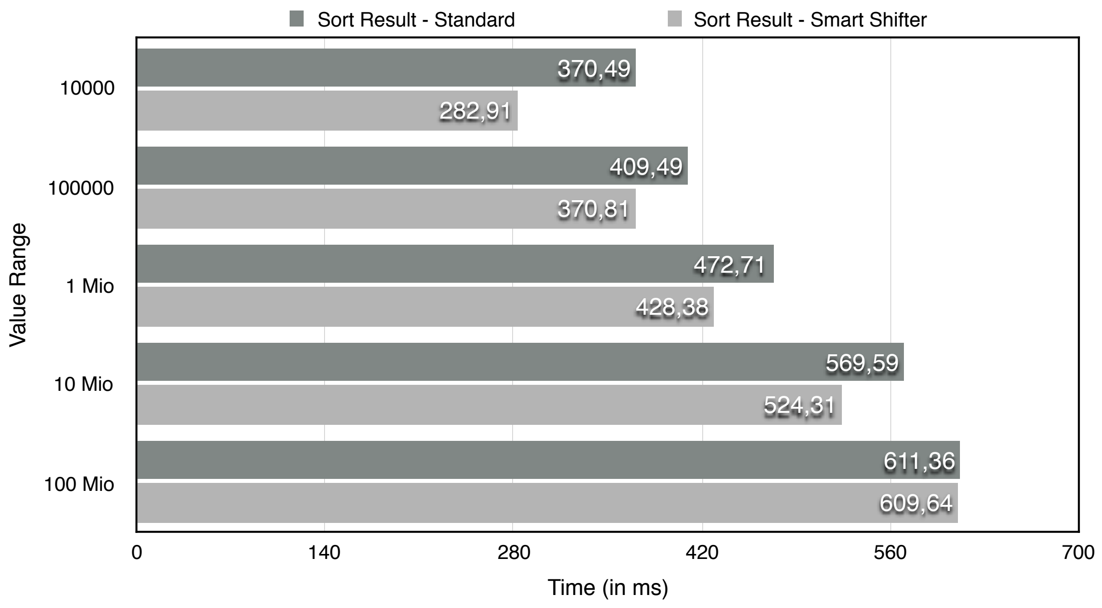

Figure 4.6: Comparing sorting phase with smart shifter for 10Mio tuples

Now, we will compare the sorting results from the standard B-MPSM with the sorting results of the improved B-MPSM by smart shifter, as we can see in Figure 4.5 on the facing page for 5Mio array entries and Figure 4.6 for 10Mio array entries.

As we can see, the algorithm with smart shifter performs much better in sorting with the low value range. However, we can see that the difference becomes smaller when the value range increases, and it becomes almost equal at the value range of 100Mio tupel. That is because the higher the value range the less recursion steps we can save using smart shifter. So, for example, with 10000 value range we are shifting only 2 times using smart shifter, instead of 4 times using standard B-MPSM algorithm. In contrast, with 100Mio as value range we need to shift 4 times, same as in the standard version, as the most significant bit is equal to 27 and thereby between 25 and 32.

We do not need to compare the results from merging engine, because this improvement has no influence on the merging step. Therefore, the join results are the same as in the standard version of B-MPSM.

Depending on the performed tests, we can conclude, that the smart shifter has no negative effect on the performance of B-MPSM. Under certain conditions (with the low value range) it can significantly increase the performance of the whole algorithm.

4.4 Comparison with Int2Ins

In this section we will compare the previously introduced Int2Ins improvement for B-MPSM algorithm with the standard version of B-MPSM. Same as in previous section we will begin with tests for the algorithm with Int2Ins improvement. Then we will compare it with the standard version of B-MPSM and evaluate the results after it.

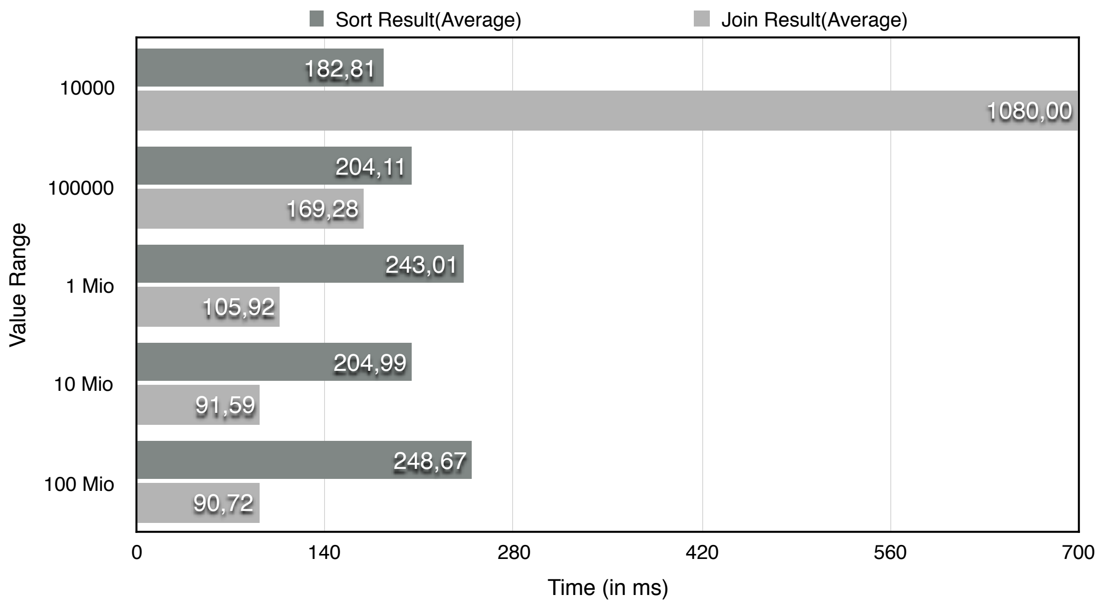

Figure 4.7: B-MPSM tests with int2ins for 5Mio tuples

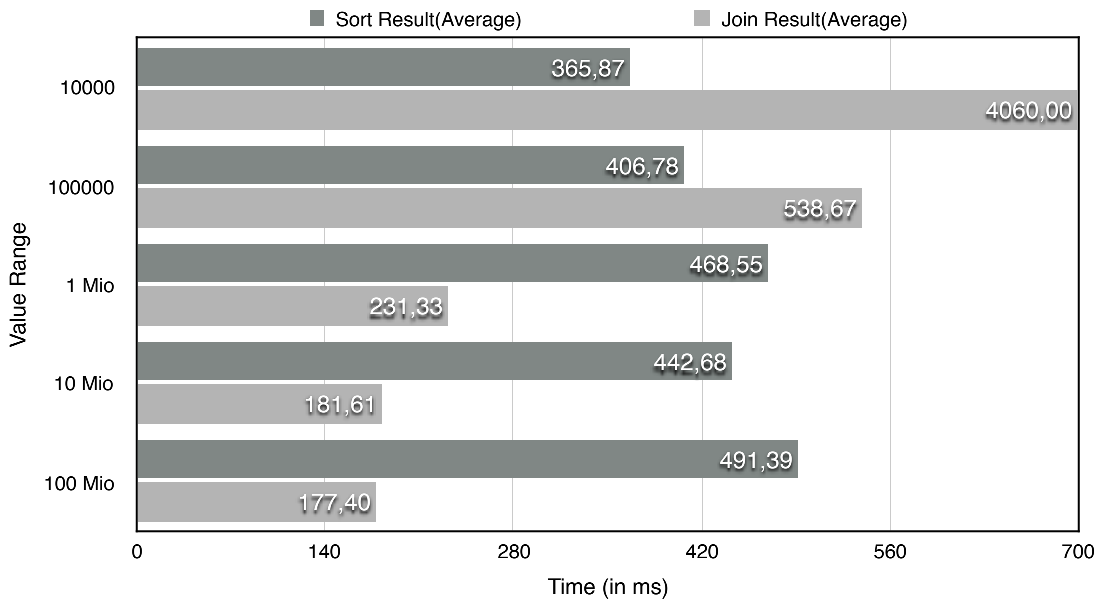

Figure 4.8: B-MPSM tests with int2ins for 10Mio tuples

The tests for the Int2Ins improvement for both, 5Mio and 10Mio data entries are presented in Figure 4.7 and Figure 4.8 respectively. This time the result values for the sorting phase are totally different comparing to the values resulted in previous tests. The result for the value range of 10 Mio is now not even a little smaller than the result with value range of 1 Mio. It is now significantly lower and is almost the same as the result for value range of 100000. We can see the same behavior for the array with 10Mio entries. This time the result for 10 Mio value range it is not equal to the result for the value range of 100000, but it still much lower than the previous one. The behavior of the merging engine, in contrast, is similar to the previosly executed tests. It still has the high result values if the value range is low, and the lower results the higher the value range is.

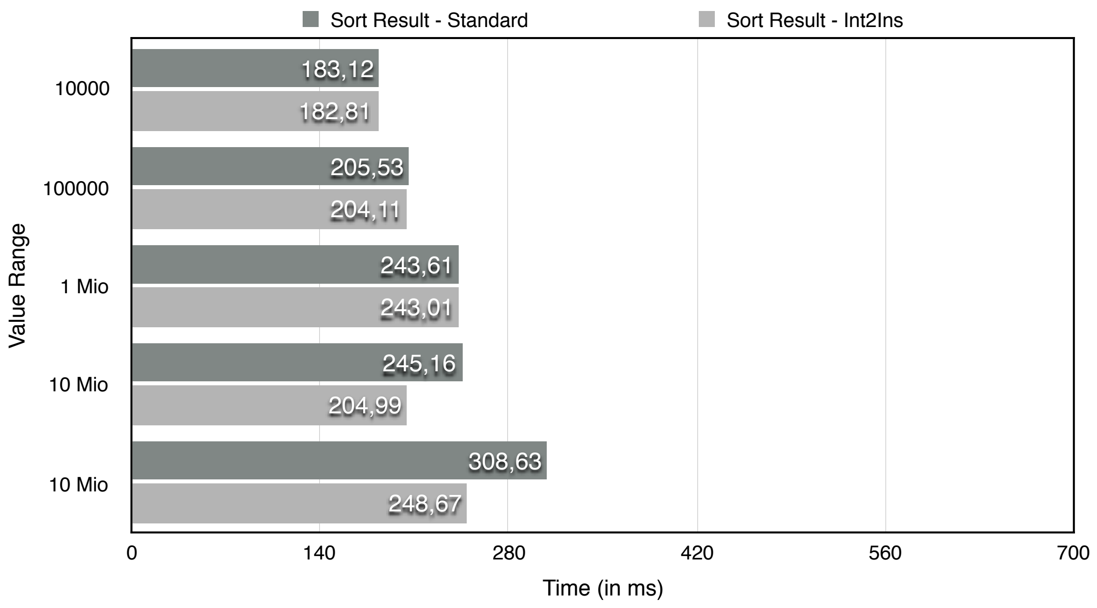

Figure 4.9: Comparing sorting phase with int2ins for 5Mio tuples

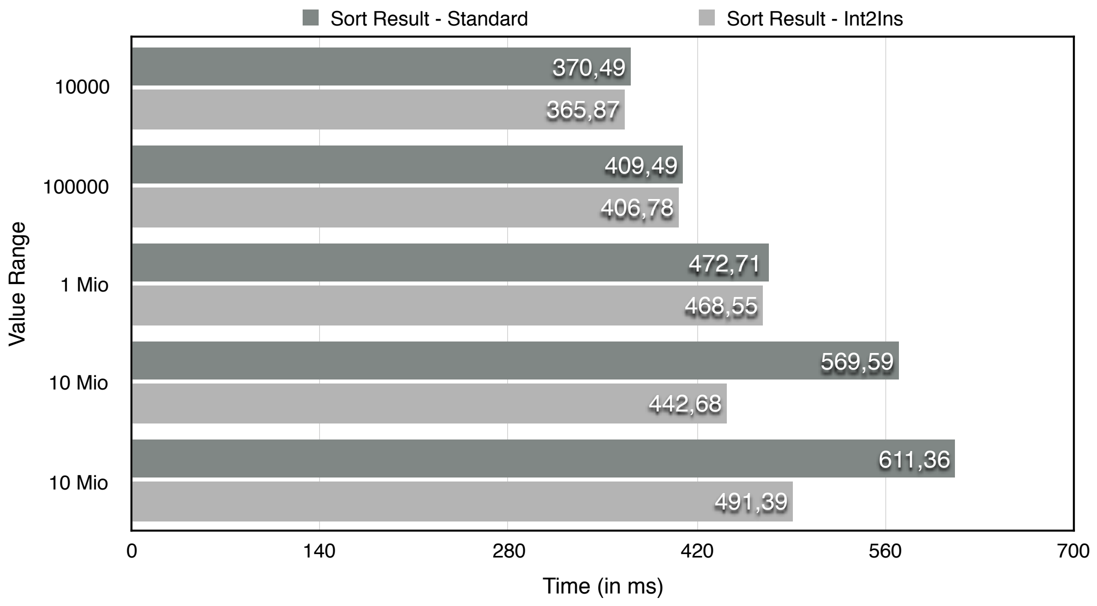

Figure 4.10: Comparing sorting phase with int2ins for 10Mio tuples

Now we will compare the Int2Ins improvement with the results for the standard B-MPSM, which are presented in Figure 4.9 on the preceding page and Figure 4.10 on the previous page for the sorting phase. As we can see, in the first 3 value ranges in the sorting results, there are almost no profit using the Int2Ins. In the following two value ranges (10 Mio and 100 Mio), in contrast, Int2Ins performs about 20 percent faster. As already mentioned before, the Int2Ins improvement has no influence on the merging engine, therefore, for the join phase the results are almost the same.

One cannot deny that it is still not clear in which cases the insertion sort performs better than the introsort. We have seen, that in 3 of 5 test cases the performance of insertion sort was almost the same as by introsort, and only in 2 cases it was better. The reason for that could be that on the low value ranges, the radix sort does the whole work by itself, without switching to the introsort. It is also possible, that the most partitions contains less than 16 values, then, it means int2ins behaves like introsort. To make it clear, we introduce additional tests, where we directly evaluate the performance of both, insertion sort and introsort, algorithms.

As we can see in Figure 4.11, we did the tests with 5 different arrays. Each of these arrays has 100 values, the maximal possible number of elements, according to the default configuration of MPSM. This time we did 100 measurements for each test case and present the results in nanoseconds. Due to the high standard deviation of the test results, we additionally use the interquartile mean [DRSC06], a type of truncated mean. To be more precise, we are eliminating 25 percent of the best and the worst test results to reduce the standard deviation.

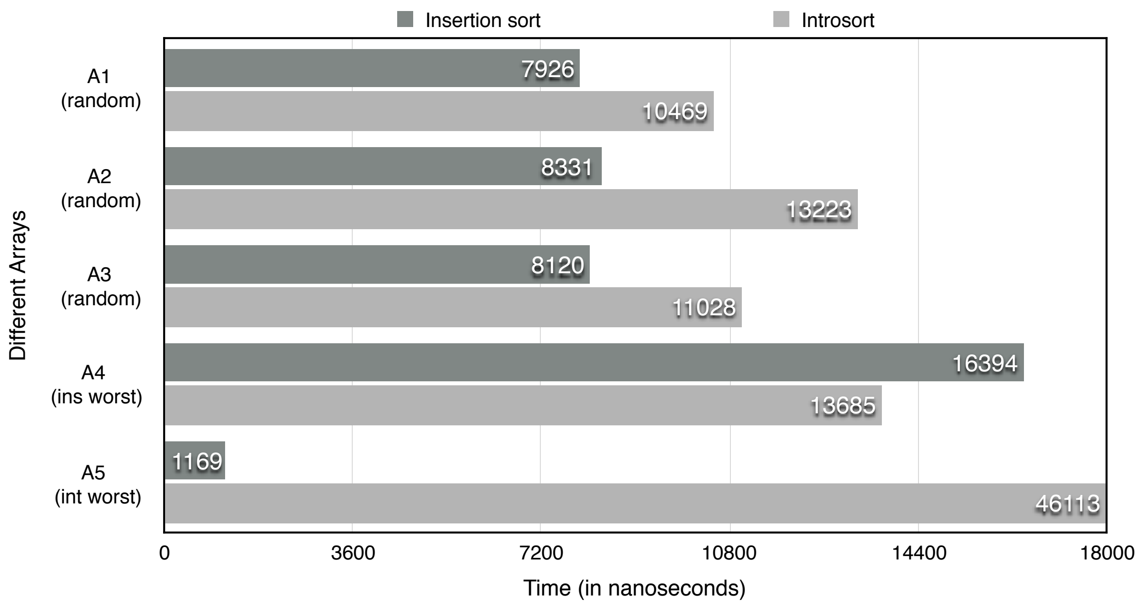

Figure 4.11: Comparing insertion sort with introsort

First, we introduce tests with three different arrays filled with random values. The insertion sort performs better in each of these cases(from 32 up to 58 percent). Next, we did the test with absolute worst case scenario for insertion sort: with the values, sorted in descending order. This time the introsort performs 20 percent better than the insertion sort. In the last test case, we filled the array with equal values, so we created the worst case scenario for the introsort. This time, insertion sort performs significantly better than the introsort. Another important fact is that the less elements in the array the better insertion sort will perform.

We can conclude, that the Int2Ins has almost no negative influence on the performance of B-MPSM algorithm. Conversely, in most cases insertion sort performs significantly better than the introsort.

5. Conclusion

In this work, we have been concentrated on the performance improvements for B-MPSM algorithm. Very important for us was to find such improvements, which are also applicable for the P-MPSM join. To conclude our results, we summarize main points of our 3 contributions to this work. These contributions are: (1) Investigate and review each single algorithm of the MPSM and find possible improvements for these algorithms, (2) Investigate and review the MPSM itself and look for possible improvements for it, (3) evaluate the performance of MPSM algorithm under different improvements we found.

The single algorithms on MPSM

We reviewed all single algorithms, which are used in MPSM. We also presented the implementation of these algorithms. We could find out an improvement for radix sort, using the smart shifter, which can significantly improve the performance of radix sort under certain conditions. This improvement also has no negative influence on the MPSM algorithm, in case of performance, means, there is no possibility MPSM will perform worse, using the smart shifter.

The MPSM algorithm

We have reviewed the MPSM algorithm itself and were looking for the possible improvements for that. So, we have introduced an Int2Ins improvement, which significantly increases the performance on the high value range. In our test cases we could find the scenario, where Int2Ins perform worse than the standard version of MPSM algorithm, but this scenario is very rare (with around 100 values, sorted in descending order). In most cases the insertion sort outperforms the introsort, especially in the worst case scenario for introsort with the array filled with equal values.

Evaluating the performance of improvements

Using both improvements we could achieve the significant performance increase for the MPSM algorithm. Both improvements we have presented are not conflicting with each other. Thus, we could achieve the performance increase up to 25 percent using both improvements for every value range, as Int2Ins works better for high value range and smart shifter with the lower ones.

6. Future Work

Since we were concentrating on the B-MPSM version of the algorithm, we can upgrade the B-MPSM to the P-MPSM version. The P-MPSM version provides a different way of sorting and thereby the algorithm is more efficient by joining. The improvements we have presented are working on both versions of the algorithm, as the P-MPSM version improves the way to sort data and not the sorting algorithms themself.

The concurrency management could be also improved by using a thread pool instead of creating the single threads. It should bring only an insignificant benefit, because the workers need much more time to complete their tasks comparing to the overhead for creating threads.

Bibliography

[AKN12a] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(10):1064–1075, 2012. (cited on Page vii, 1, 35, 36, 37, 38, 39, 40, 45, and 46)

[AKN12b] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Computing Research Repository (CoRR)*, abs/1207.0145, 2012. (cited on Page 38)

[Bac94] Paul Bachmann. *Die Analytische Zahlentheorie.* 1894. (cited on Page 4)

[CBB$^+$15] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 8(12):1518–1529, 2015. (cited on Page 46)

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001. (cited on Page xi, 4, and 6)

[cod12] codecodex.com. Heapsort. Website, February 2012. Available online at http://www.codecodex.com/wiki/Heapsort/; visited on April 1st, 2016. (cited on Page xi, 16, and 18)

[Cor13] Thomas H. Cormen. *Algorithms Unlocked.* MIT Press, 2013. (cited on Page 7)

[DRSC06] Thomas F. Droege, Michael W. Richmond, Michael P. Sallman, and Robert P. Creager. Tass mark iv photometric survey of the northern sky. *Publications of the Astronomical Society of the Pacific (PASP)*, 118(850):1666, 2006. (cited on Page 60)

[ECW92] Vladimir Estivill-Castro and Derick Wood. Skip sort—an adaptive randomized algorithm or expected time adaptivity is best. In *Computer science*, pages 179–187. Springer, 1992. (cited on Page 4)

[JC13] J.Jayashree and C.Ranichandra. Join Algorithm for Efficient Query Processing for Large Datasets. *Asian Journal of Computer Science and Information Technology (AJCSIT)*, 2(3), 2013. (cited on Page 30)

[Kar92] Richard M Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *Proceedings of the International Federation for Information Processing (IFIP)*, volume 12, pages 416–429, 1992. (cited on Page 4)

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. (cited on Page 22)

[lr04] linux related.de. Quicksort. Website, 2004. Available online at http://www.linux-related.de/index.html?/coding/sort/sort_quick.htm; visited on April 30st, 2016. (cited on Page xi and 11)

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software - Practice and Experience (SPE)*, 27(8):983–993, 1997. (cited on Page xi, 19, and 21)

[Sed78] Robert Sedgewick. Implementing quicksort programs. *Communications of the Association for Computing Machinery (CACM)*, 21(10):847–857, 1978. (cited on Page xi, 9, and 10)

[Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990. (cited on Page 41)

[SHS05] Gunter Saake, Andreas Heuer, and Kai-Uwe Sattler. *Datenbanken - Implementierungstechniken (2. Aufl.)*. MITP, 2005. (cited on Page xi, 31, 33, and 41)

[SS06] Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen - eine Einführung mit Java (3. Aufl.)*. dpunkt.verlag, 2006. (cited on Page 3)

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011. (cited on Page xi, 4, 7, 11, 25, and 29)

[SZ03] Ranjan Sinha and Justin Zobel. Efficient trie-based sorting of large sets of strings. In *Proceedings of the Twenty-Sixth Australasian Computer Science Conference (ACSC2003)*, pages 11–18. Australian Computer Society, Inc., 2003. (cited on Page 22)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 29.05.2016