CSE 708: SEMINAR ON PROGRAMMING MASSIVELY PARALLEL SYSTEMS

Implementation of Merge Sort using k-way Merge Sort

Neha Mishra, 50416280

University at Buffalo The State University of New York

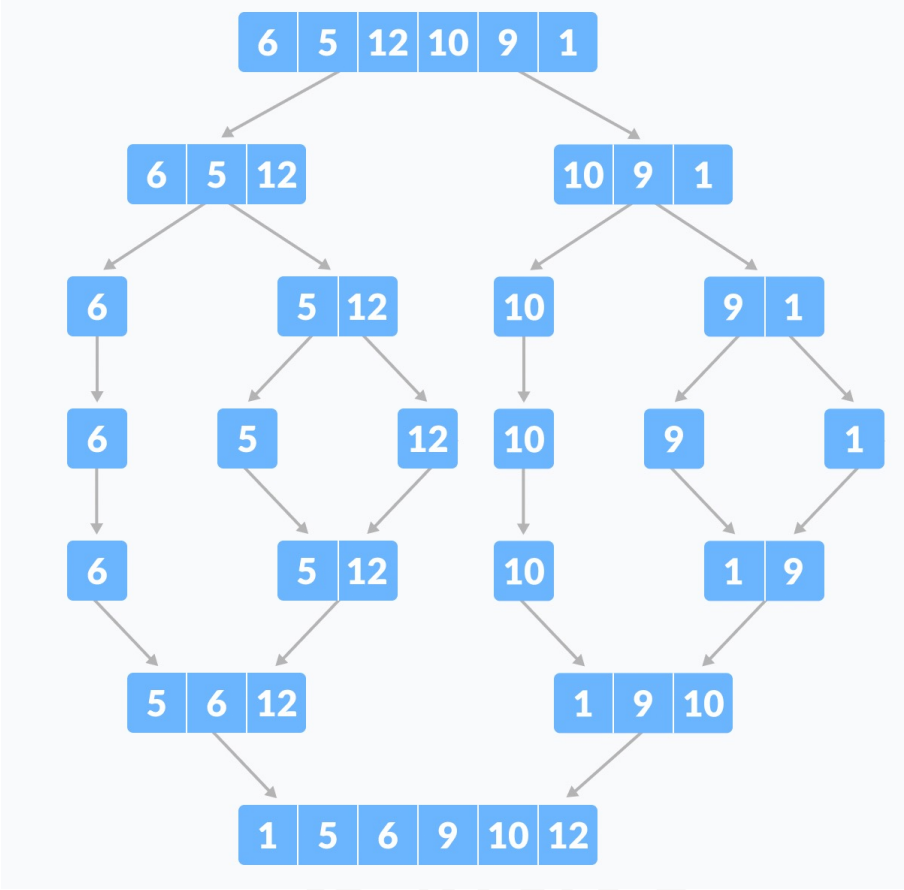
Agenda

- Introduction to Merge Sort
- Sequential Approach
- Parallel Approach- Two-way Merge Sort
- Parallel Approach- Multi-way or k-way Merge Sort
- Challenges and Learnings
- References

Introduction to Merge Sort

Merge sort is a sorting algorithm which is based on the divide and conquer paradigm. The array A of N elements is divided into two almost equal halves. These halves are then sorted recursively and sorted parts are merged into a single sorted sequence.

Time complexity: O(nlogn)



Pseudocode for Merge Sort

```
Function mergeSort(⟨e_1, ..., e_n⟩) : Sequence of Element
    if n = 1 then return ⟨e_1⟩
    else return merge(   mergeSort(⟨e_1, ..., e_⌊n/2⌋⟩),
                         mergeSort(⟨e_⌊n/2⌋+1, ..., e_n⟩))

// merging two sequences represented as lists
Function merge(a, b : Sequence of Element) : Sequence of Element
    c := ⟨⟩
    loop
        invariant a, b, and c are sorted and ∀e ∈ c, e' ∈ a ∪ b : e ≤ e'
        if a.isEmpty          then   c.concat(b); return c
        if b.isEmpty          then   c.concat(a); return c
        if a.first ≤ b.first  then   c.moveToBack(a.PopFront)
        else                         c.moveToBack(b.PopFront)
```

Reference: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev.

Two Way Merge Sort

- Merge sort's dependency on divide and conquer paradigm makes it an excellent candidate for parallelizing the Merge Sort. So, how we use multi-processors for this.

```
MergeSort(A, n)
1  if n = 1
2     then return A
3  Divide A into two A_left and A_right each of size n/2
4  A'_left ←spawn MERGESORT(A_left, n/2)
5  A'_right ← MERGESORT(A_right, n/2)
6  sync
7  Merge the two halves into A'
8  return A'
```

Reference: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev.

But is it an improvement?

- Biggest hurdle? Even with giving as many processors as needed, the algorithm would need $\Omega(n)$ as the final sequential merge would need comparison of n elements which acts as the biggest blocker.
- Furthermore, the parallel merge this way would be needing O(n) extra space. Our system wouldn't scale well, if the data was to stay on different processors.

Parallel Two-way Merge Sort

- In order to implement a parallel merge sort, we need to parallelize the merging.
- How do we do that?
  - Let's assume our two sorted sequences are: *a and b*
  - Split the two sequences a and b into p pieces: $a_1..a_p$ and $b_1..b_p$
  - This means, merge(a, b)-> merge(a_1, b_1), merge(a_2, b_2)… and so on.
  - The p pieces, with corresponding a and b subarray are running in parallel by having one PE each.
  - In order for this to work, a_i and b_i must not be greater the a_{i+1} and b_{i+1}. This would mean that PE i first checks where does a_i end in a and b_i end b and then it merges ai and bi.

Parallel Merge Sort

- This step where we are finding the cut in the corresponding a and b subarrays, to find the smallest k elements can be done easily in O(k) but our goal is to find it in O(log|a| + log|b|). Here, k can be defined as (n1+n2)/2 where n1 and n2 is the size of the subarrays n1 and n2.
- This is defined as twoSequenceSelect(a, b, k) as defined by Sanders, Mehlorns and Dietzfelbinger. The main idea is to maintain sorted subrange $a[l_a \ldots r_a]$ and $b[l_b \ldots r_b]$ with these properties.
  - The elements $a[1 \ldots l_a]$ and $b[1 \ldots l_b]$ belong to the k smallest element
  - The k smallest elements are contained in $a[1 \ldots r_a]$ and $b[1 \ldots r_b]$.

For k=4;

a	ℓ_a	m_a	r_a	$\bar{a}$	b	ℓ_b	m_b	r_b	$\bar{b}$	$k < m_a + m_b$?	$\bar{a} < \bar{b}$?	action
$[45\bar{\|}68]$	1	2	4	5	$[1\bar{2}\|37]$	1	2	4	2	no	no	$l_b := 3$
$[45\bar{\|}68]$	1	2	4	5	$12[\bar{3}\|7]$	3	3	4	3	yes	no	$r_a := 1$
$[\bar{4}\|]568$	1	1	1	4	$12[\bar{3}\|7]$	3	3	4	3	no	no	$l_b := 4$
$[\bar{4}\|]568$	1	1	1	4	$123[\bar{7}\|]$	4	4	4	7	yes	yes	$r_b := 3$
$[\bar{4}\|]568$	1	1	1	4	$123[\|]7$	4	3	3		finish		$r_a := 1$
$4\|568$			1		$123\|7$			3		done		

Reference: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev.

twoSequenceSelect(a, b, k):

For the ranges, $[l_a \ldots r_a]$ and $[l_b \ldots r_b]$,

```
1   while ra>la and rb>lb:
2       ma= floor((la+ra)/2)
3       mb= floor((lb+rb)/2)
4       a', b'= a[ma], b[mb]
5       if a'<b':
6           if k<ma+mb:
7               rb= mb-1
8           else:
9               la= ma+1
10      else:
11          if k<ma+mb:
12              ra= ma-1
13          else:
14              lb= mb+1
```

Issues with Two-Way Merge Sort

Although we did achieve the goal of keeping the data divided between two processors and not letting one processor overload itself, we still had

Our number of processors will have to fixed to 2 which reduces the whole effort as useless as we wouldn’t be able to scale it.

Multiple Failed Ways to do

- Use of heap: sequential sorting
- Use of tournament sort: doesn't effectively use the power of parallel processing

Parallel Multi-way or k-way Merge Sort

- To implement parallel *p*-way mergesort, we first split the input array *s* into *p* equally sized pieces, possibly trying to allocate PEs on the same NUMA node as the RAM storing that piece of data. Each PE then locally sorts the data allocated to it. This takes time O(*n* log *n*).
- For parallel *p*-way merging, we generalize the splitting idea used in parallel bi- nary (two-way) mergesort. Rather than splitting two sequences into *p* pieces each, we now split *p* sequences into *p* pieces each.

Function *smmSort*(s : *Sequence* **of** *Element*) : *Sequence* **of** *Element*

$sort(s)$; *barrier* // sort locally then synchronize globally

$x := multiSequenceSelect(\langle s@1, \ldots, s@p \rangle, \left\lceil i_{\text{proc}} \frac{\sum_i |s@i|}{p} \right\rceil)$; *barrier* // find splitters

return *multiwayMerge*($\langle$ $s@1[x_1@(i_{\text{proc}}-1)+1..x_1], \ldots,$ // assume

$s@p[x_p@(i_{\text{proc}}-1)+1..x_p]\rangle$) // $x@0 = \langle 0, \ldots, 0 \rangle$

Fig. 5.28. SPMD pseudocode for shared-memory multiway mergesort.

Reference: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev.

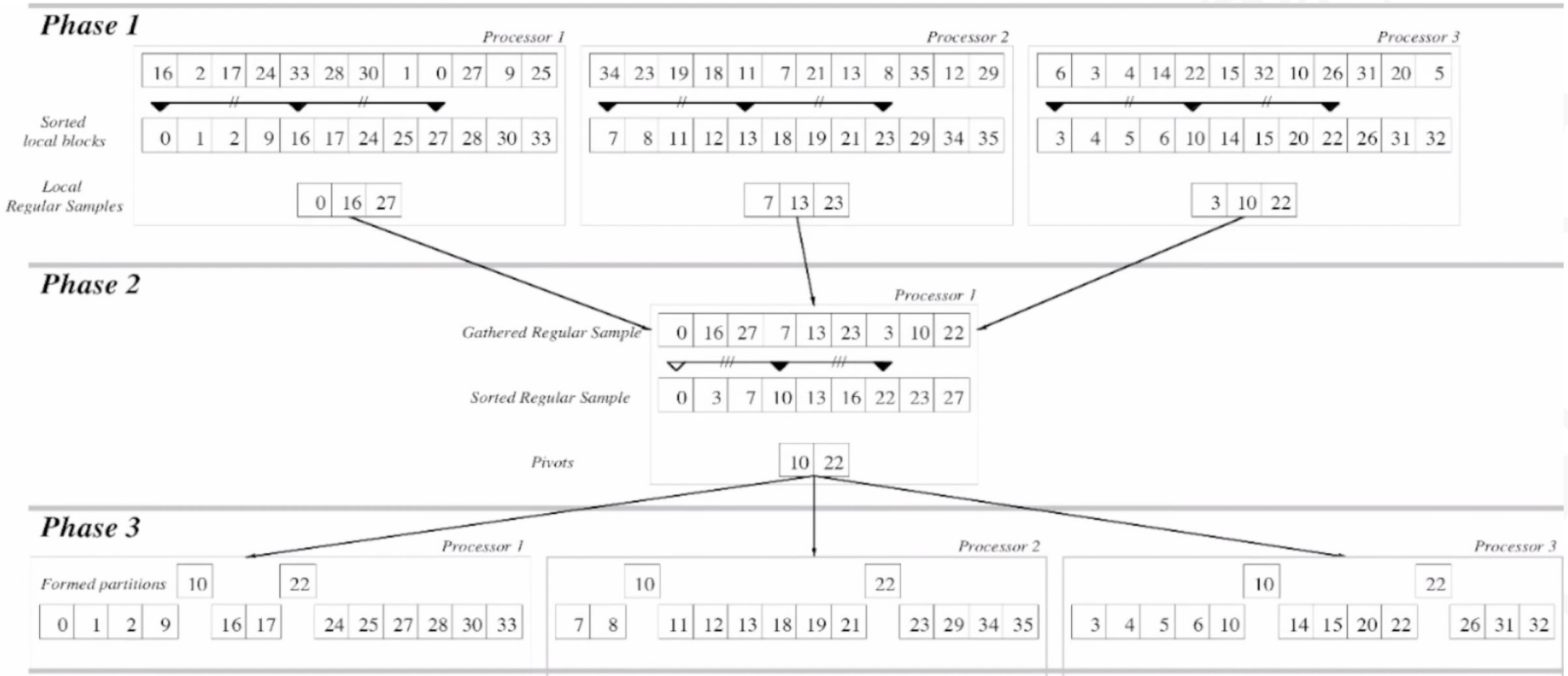
Parallel Multiway Merge Sort

Function *multiSequenceSelect*(S : *Array* **of** *Sequence* **of** *Element*; $k : \mathbb{N}$) : *Array* **of** $\mathbb{N}$
  for $i := 1$ **to** $|S|$ **do** $(\ell_i, r_i) := (0, |S_i|)$
  invariant $\forall i : \ell_i..r_i$ *contains the splitting position of* S_i
  invariant $\forall i, j : \forall a \leq \ell_i, b > r_j : S_i[a] \leq S_j[b]$
  while $\exists i : \ell_i < r_i$ **do**
    $v := pickPivot(S, \ell, r)$
    for $i := 1$ **to** $|S|$ **do** $m_i := binarySearch(v, S_i[\ell_i..r_i])$ // $S_i[m_i] \leq v < S_i[m_i + 1]$
    if $\sum_i m_i \leq k$ **then** $\ell := m$ **else** $r := m$
  return ℓ

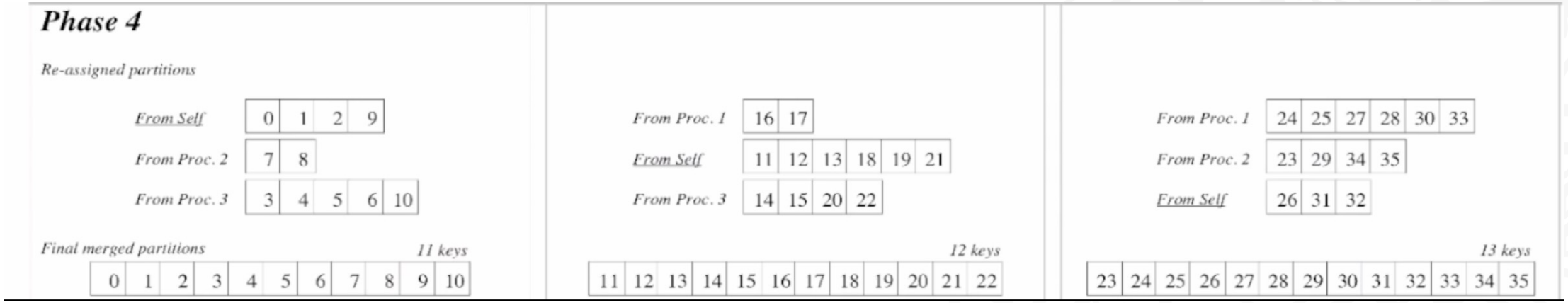
Fig. 5.29. Multisequence selection. Split the sorted input sequences in S such that the sum of the resulting splitting positions is k and such that all elements up to the splitting positions are no larger than the elements to the right of the splitting positions.

Reference: Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger and Roman Dementiev.

Parallel Multiway Merge Sort- Pictorial Depiction

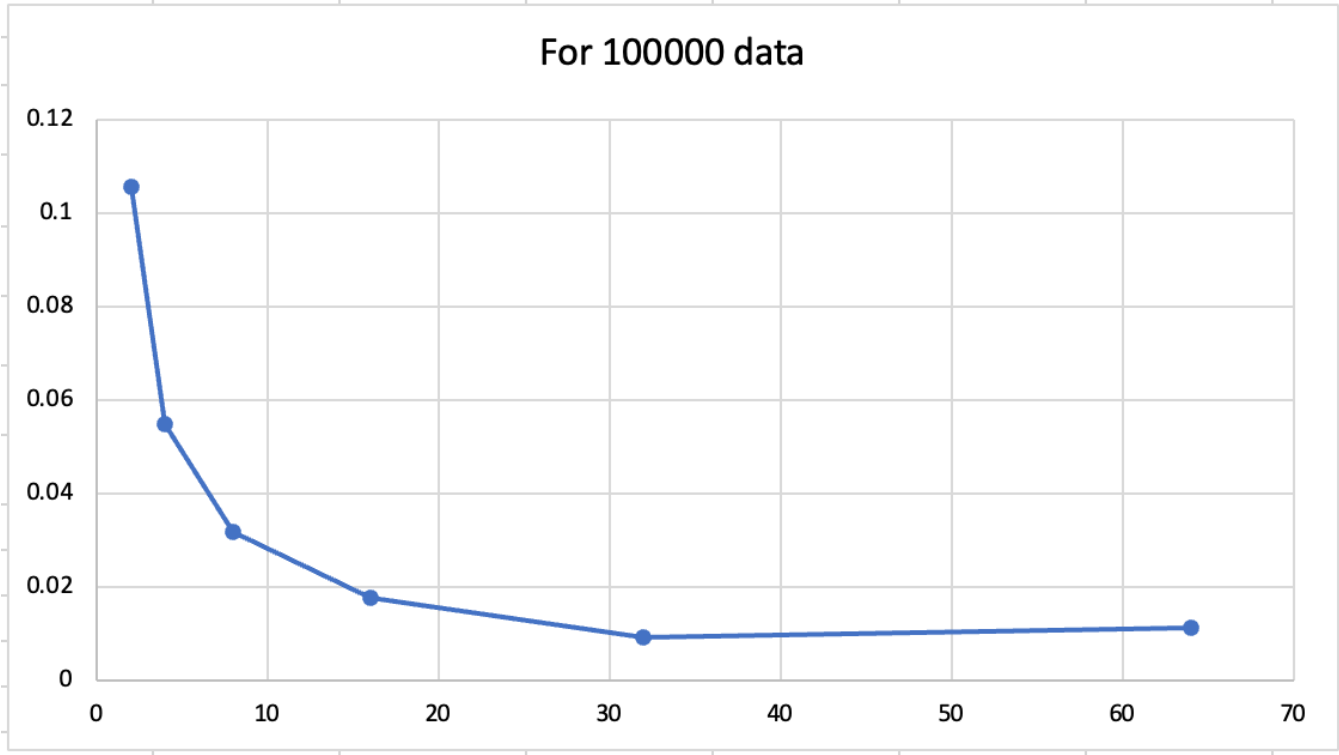


Parallel Multiway Merge Sort- Pictorial Depiction- continued



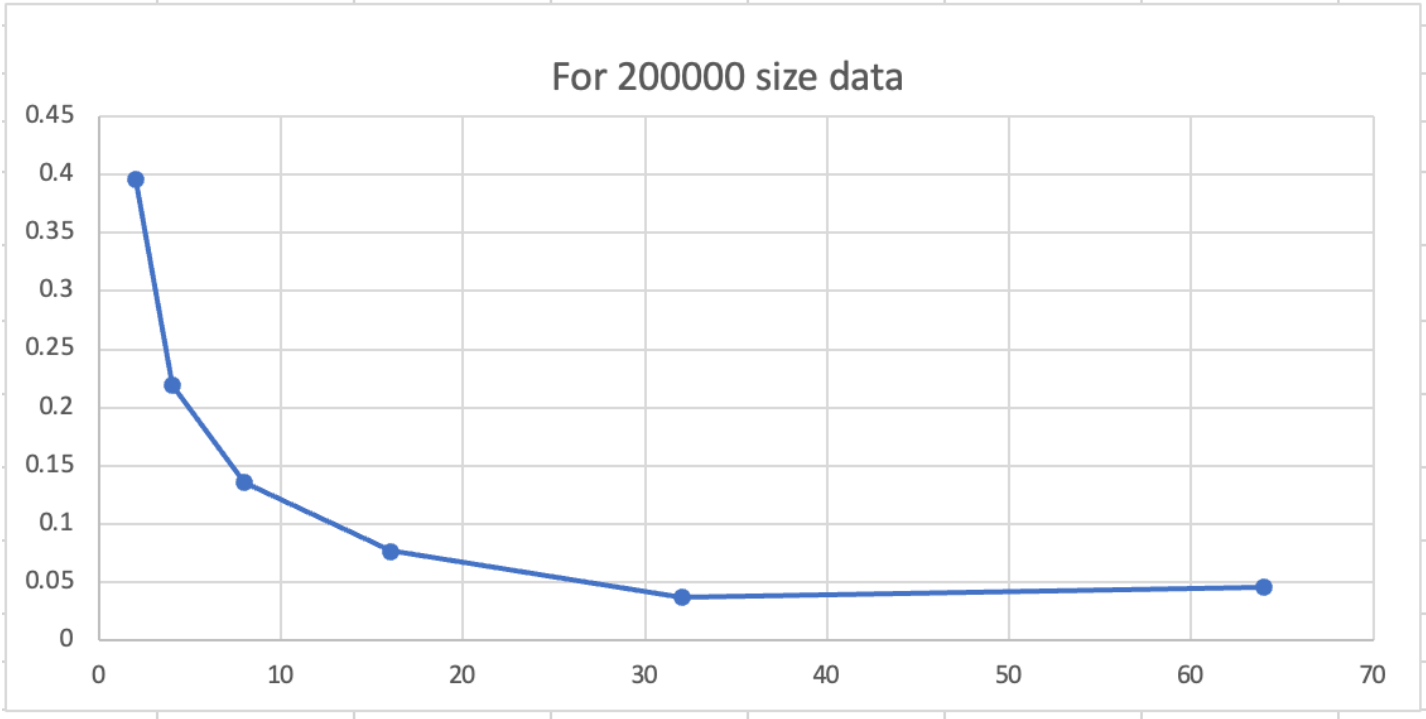
Trend for 100,000 data

No. of Processors	Time (in s)
2	0.1057
4	0.0548
8	0.0318
16	0.0176
32	0.0092
64	0.0112



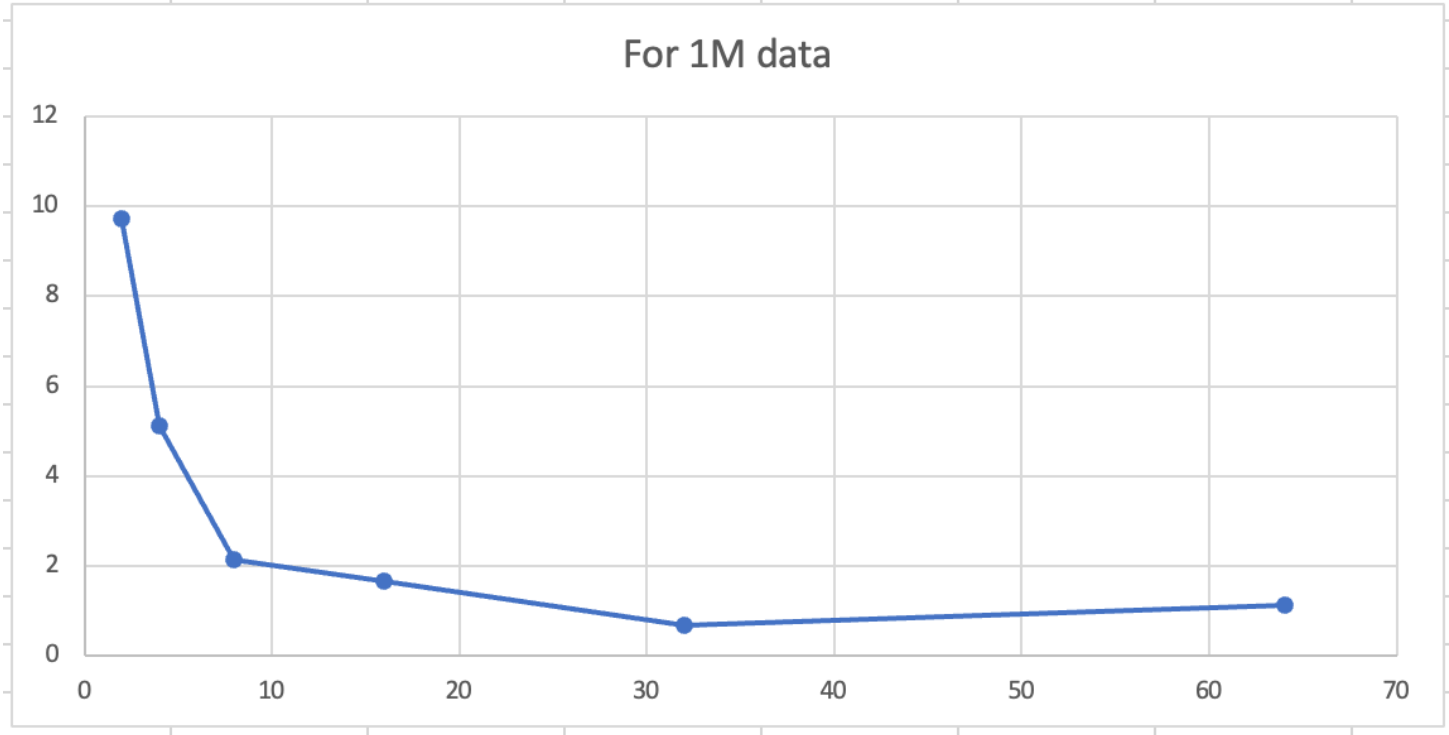
Trend for 200,000 data

No. of Processors	Time
2	0.3963
4	0.2194
8	0.1355
16	0.0767
32	0.0372
64	0.0454



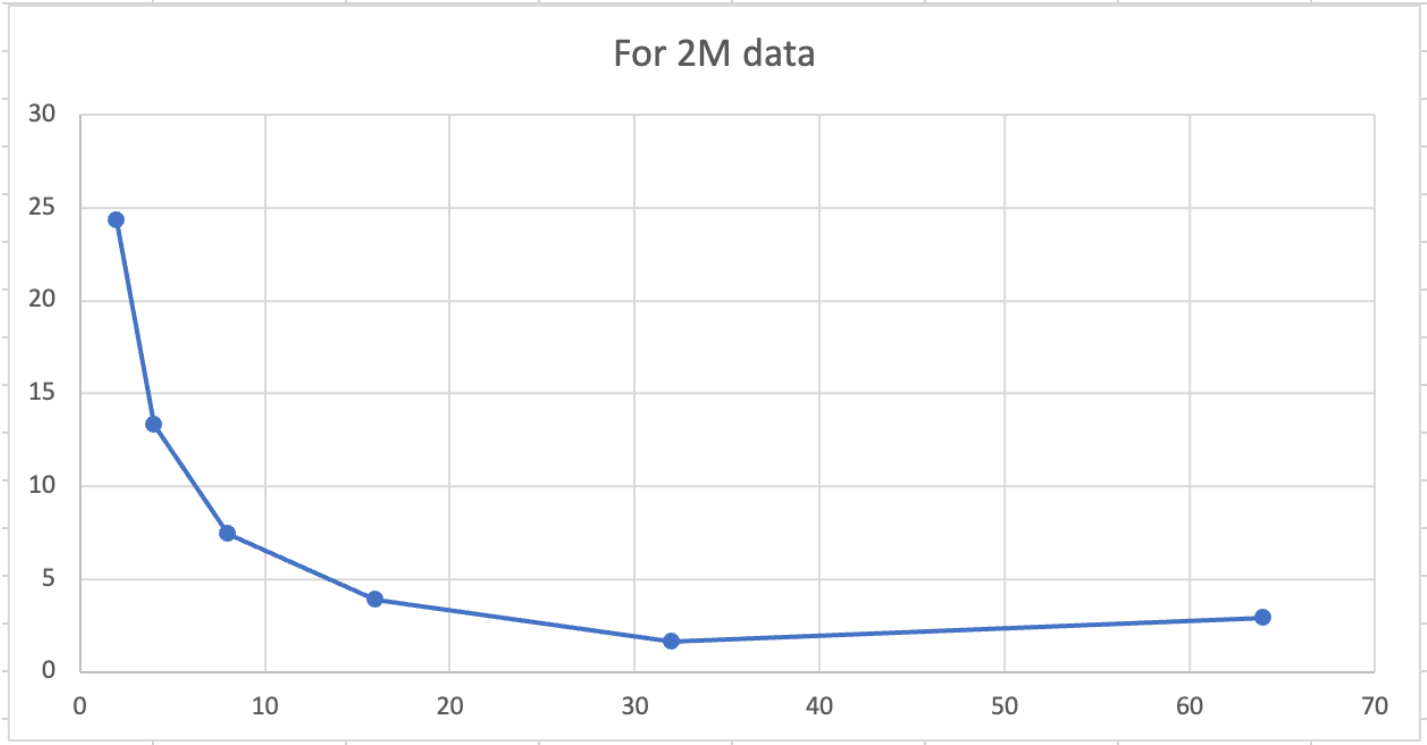
Trend for 1,000,000 data

No. of Processors	Time
2	9.7190
4	5.1225
8	2.1299
16	1.6731
32	0.6783
64	1.1193



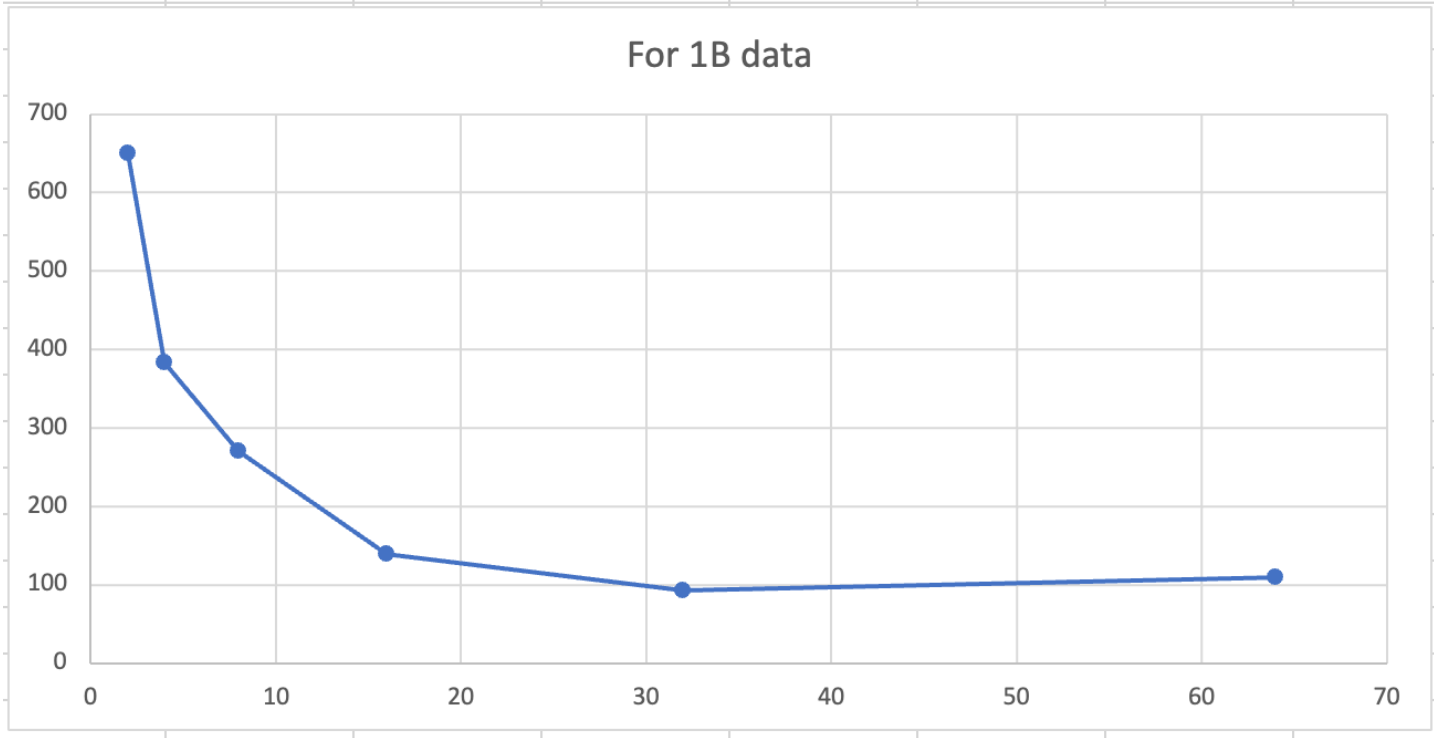
Trend for 2,000,000 data

No. of Processors	Time
2	24.3587
4	13.3234
8	7.4507
16	3.9068
32	1.6622
64	2.9267



Trend for 1,000,000,000 data

No. of Processors	Time
2	650.1127
4	383.4366
8	270.4521
16	139.1237
32	92.85
64	108.9378



Challenges and Learnings

- Learnings: MPI, CCR, SLURM, SRUN, Multiple kinds of sorting, Distributed Sorting, etc.
- Understanding of how the decline of performance with the increase of processors.
- Increase in data up and beyond 1B resulted in slow run time algorithm.

Things that improved test results

- Initially, the code randomly generated random numbers which was then used by the processors to further sort. The change was made in two parts. One code generated random numbers and saved it in a file. Our main code of parallel k way merge sort picked these number for solving our sorting problem.
- Removal of unnecessary MPI Scatter and MPI Gather which wasn’t relevant.

References

- *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox by Roman Dementiev, Martin Dietzfelbinger, Peter Sanders, Kurt Mehlhorn*
- *https://www.cse.wustl.edu/~angelee/archive/cse341/fall14/handouts/lecture06.pdf*
- *Image referenced from: Zaric Zola Parallel and Distributed Processing Class Presentation.*