Distributed QuickSort - Hybrid Implementation

1st Dhruvil Gorasiya
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901061@daiict.ac.in

2nd Harshil Kachhadiya
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901411@daiict.ac.in

3rd Akash Kothiya
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901422@daiict.ac.in

4th Denish Hirpara
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901424@daiict.ac.in

5th Krunal Lukhi
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901449@daiict.ac.in

6th Parin Jasoliya
Group in Computational Science-HPC
DA-IICT
Gandhinagar, India
201901451@daiict.ac.in

***Abstract*—An important goal is to optimize the algorithms to work with more performance and utilize the same resources. We are designing and tweaking sorting algorithms such as quicksort. Enables us to design implementations that give a very high performance. In this paper, we have tried to address the problem of sorting an integer array A[] of length N on a multi-core system and distributed memory architecture to take advantage of parallel processing. Quick-sort algorithm is modified in various ways and Benchmarked on an HPC Cluster. The tests and performance trade-offs exist for different implementations and flavors of the Quick-sort algorithm.**

***Index Terms*—Single-pivot, Dual-pivot, Partition**

I. Introduction

Sorting is a very common problem that occurs in many applications. It is a core part of many systems, such as database and search systems. A quote from Donald Knuth - *"Computer manufacturers estimated that more than 25 percent of the running time on their computers was spent on sorting when all their customers were taken into account."*. Hence, coming with efficient Sorting Algorithms which takes maximum advantage of the available constraints is important for High-Performance Computing [6]."

There are many sorting algorithms that perform sorting, one of the most popular and widely used sorting algorithm is Quick sort [3]. It is among the fastest sorting algorithms discovered [9]. Quicksort is a Divide Conquer algorithm. Quicksort is an unstable Sort. The practical applications of sorting include Commercial computing, Search for information, Operations research, Event-driven simulation, Numerical computations, etc. Hence Parallel Algorithms have huge scope in sorting. [14].

Comparison Based Sorting is a very well studied subject. There have been many variants of comparision-based sorts. We will Mainly Focus on Quick Sort and its variants in this paper. Serial Algorithms of Quick Sort are very well studied. Many Variants of quicksort algorithms take advantage of several optimizations.

Parallelization of comparison based sorting algorithm is an extensively researched area. There are several publications that talk about different variants and optimizations on traditional quick sort. quicksort Algorithm also has been extensively studied in the past, and newer, Better Variants like Dual-Pivot QuickSort have emerged. Dual Pivot QuickSort is a very Widespread Algorithm. In contrast to simple quicksort we choose two pivots for partitions and recursivly sort the partitions.

Sample Sort was Traditionally believed to be the best way to take advantage of parallelization. [11] [13] Sample Sort divided the dataset into buckets which were further independently sorted and later combined. The elements in ith bucket were all smaller than every element in $i+1$ th bucket.

In this paper, we discuss our implementation of the paralyzed dual pivot Quicksort using OpenMP and OpenMPI. The algorithm was run for different datasets and the results found are given below.

Here are two major goals of this paper:

1) To find out an easy to implement yet efficient approach to paralyzing quick-sort.
2) To perform an empirical analysis of the performance of sequential and parallel approach to quick sort in terms of CPU time.

1) Hoare's Partition: This is the original partition scheme that Tony Hoare described in the original paper [2]. Here he took two pointers, one pointing to the first element and the other pointing to the last. He moved the pointers toward each other until an inversion(smaller element comes after larger element) was detected. If the elements are in the wrong order, then they are swapped. The algorithm is defined in detail in the later part of the paper with pseudo-code.

2) Dual-Pivot Quicksort: Having more than one pivot is a modification of the original sorting algorithm. Originally the quicksort had two steps. And first, we chose a pivot and recursively called for the partitions created around the pivots. This modification chooses more than one pivot element and recursively calls for the partitions created.

Multi Pivot sorting algorithm takes advantage of the huge improvements that have taken place in modern architectures, like cache improvements. Traditional Hoare's partition is not able to take advantage of these modern systems compared to multi pivot.

In 2009, Vladimir Yaroslavskiy posted in detail how dual-pivot quick-sort outperformed the traditional single-pivot quicksort [15]. The number of swaps was primarily reduced by 20%. This Proved to be so popular that it was included in Java 7 release as the default built-in sorting Algorithm.

II. Hardware Specifications and Data Structure:

We have tested our algorithms on three different machines namely **HPC Cluster**, **Lab - 207 PC** and **Local System**. The hardware specification of these machines are as following.

1) **HPC Cluster:**
   - CPU - 16
   - Socket - 2
   - Cores per Socket - 8
   - Size of L1d cache - 32K
   - Size of L1i cache - 32K
   - Size of L2 cache - 256K
   - Size of L3 cache - 20480K
2) **Lab - 207 PC:**
   - CPU - 4
   - Socket - 1
   - Cores per Socket - 4
   - Size of L1d cache - 32K
   - Size of L1i cache - 32K
   - Size of L2 cache - 256K
   - Size of L3 cache - 6144K
3) **Local System:**
   - CPU - 12
   - Socket - 2
   - Cores per Socket - 6
   - Size of L1d cache - 192K
   - Size of L1i cache - 192K
   - Size of L2 cache - 3MB
   - Size of L3 cache - 8MB

We used array data structure for storing the data. We used uniformly distributed random numbers as a benchmark. For that we used a standard function $default\_random\_engine$ of $C++$. The HPC cluster is able to generate array of size 2^{33}, while Lab - 207 PC and Local system is limited to 2^{29} and 2^{27}.

III. Serial Algorithm

Yaroslasvkiy's dual-partition algorithm uses two pivots instead of 1 [15]. The left pivot is the leftmost element of the array, and the right pivot is the rightmost element of the array. Two-pointers are then declared. The left pointer is declared as the element immediately after the left pivot. The right pointer is declared as the element immediately ahead of the right pivot. Moreover, a third pointer is declared equal to the left pointer. The loop is rotated until this third pointer is less than the right pointer. Now, if an element at the third pointer is smaller than the left pivot, it is stored across the position of the left pointer, and the left pointer increases. And if an element is larger than the right pivot, it is swapped from the right side with the rightmost smallest element less than the right pivot. This process continues until the third pointer crosses the right pointer. At the end of this process, the left pivot and the right pivot will come to a position so that all the elements on the left side of the left pivot are smaller than it, and all the elements on the right side of the right pivot are greater than it. Thus the array is divided into 3 parts with the help of 2 pivots, and then these 3 parts are sorted by the recursion technique. Moreover, if the size of an array is less than 31, then the array is sorted using insertion sort [7]. The flowchart and working example of Yaroslavskiy's algorithm are in the Appendix Section of this paper.

A. *Comparison with $C++$ STL $sort()$:*

We compared our algorithm with $C++$ STL $sort()$. The algorithm has better run time than the $sort()$ as shown in Figure below :

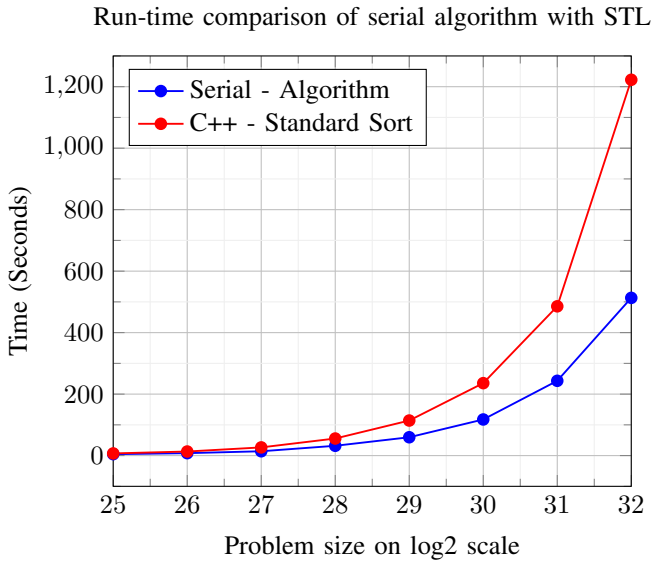

Fig. 1. Run-Time analysis of Yaroslavskiy's dual partition algorithm and $C++$ STL $sort()$

Space Complexity of the Yaroslavskiy's dual partition algorithm is $O(N)$, which is same as simple Quick-sort beacuse it is an inplace sorting algorithm. But the time complexity of the algorithm is $O(Nlog_3N)$ (in worst case $O(N^2)$) which is comparatively better than $C++$ STL $sort()$.

B. *Analysis of Serial Algorithm*

Our test data set contains an array of size 2^7 to 2^{33}. We ran both algorithms on three different machines with our data set. We have found that the Yaroslavskiy's quicksort performs

better each time. Although the number of the swaps and number of bytecode instructions performed by the Yaroslavskiy's algorithm is more than a classical quick sort, Khushagra [7] concluded that it is a lesser number of array scans that makes the Yaroslavskiy's algorithm fast. Also, when the inputs are repetitive, Yaroslavskiy's algorithm has a lesser run time than that of classical quick sort. The significant limitation of the Yaroslavskiy's algorithm is that it requires more memory swap than even classical quicksort.

As we increase the size of the problem size, the size of the data structure also increases, this makes the problem memory bound. The run time of the algorithm increases as a lot of time and memory is used to store each element in an array for a larger data set. And it takes a lot of time to partition an array in such a large data set because every partition has an array scanned, and both pivots return to their original location. And as the data set gets bigger, fewer data will be stored in the cache memory, so the element has to be brought from the outer cache and main memory instead of the internal cache memory, increasing latency, and decreasing bandwidth throughput. So the run time of the algorithm also increases.

The data set is stored in the array. Since Quick Sort is an in-place algorithm, no additional memory is required in the partition. But when a recursive call is made, it is stored in a stack. Therefore, as the size of the data set increases, so does the size of the stack and the stack is likely to overflow. As a result, the memory footprint increases. Therefore we cannot store more than 2^{33} data in an array(for cluster). However, LLC cache misses(1,400,042) are very low against total stores(9,711,391,333) and loads(13,566,406,980) for problem size of 2^{29}.

C. *Profiling Serial Algorithm :*

Function Call	**CPU Time**	**% of CPU Time**
Dual Pivot QuickSort	23.370	34.7%
uniform int distribution	16.289	24.2%
swap	12.790	19.0%
QuickSort	4.241	6.3%

We performed profiling using *vtune* profiler. It has been observed that the recursive calls are taking 34% (which is the highest of among all)of the total CPU time. Effective CPU utilization is 5.3%. Our code cannot fully utilize CPU resources due to load imbalance, thread runtime overhead, and poor threading utilization. This leads us to think about the parallelization of the algorithm.

D. *Performance of Serial Algorithm on different machines:*

As shown in Fig. 2, the Local system has shorter runtime than the other two machines. This is because cache

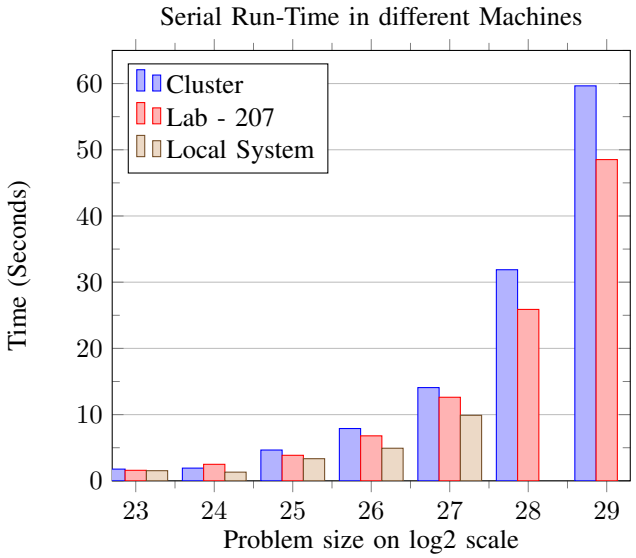

Fig. 2. Rum-Time Analysis in HPC Cluster, Lab - 207 PC and Local System.

sizes in the local system are better than those of cluster and Lab- 207 PC. Remember, our algorithm is memory bound, So cache sizes significantly affect the run-time of the serial algorithm. Parallelization of serial code may decrease memory boundness, and we may expect a better performance of the algorithm on a cluster.

IV. Parallel Algorithm using OpenMP

A. *Implemetation approach and challanges*

We use $OpenMP$ directives to parallel quicksort on a shared memory system. Some of the challenges that we faced while parallelizing the algorithm are :

- The serial implementation of the algorithm consists of recursion. So, we had to develop an approach to optimize the code in a manner that provided maximum speedup.
- When we utilized two or more cores to run the parallel implementation, we discovered that every recursion is handled by only 1 Core after a level. So, we did not get the expected speedup.

B. *Optimization Techniques*

We used following $OpenMP$ directives for optimization of serial code.

- We use *#pragma omp single nowait* directive for main function call to execute only once.
- Then we use *#pragma omp task* directive for execute every recursion. Here *task* give new core to every recursion if available.
- We tried scheduling techniques, but we cann't improved runtime further. So, we decided not to go with scheduling.

C. *Pseudo-Code, Flowchart and Working Example*

The pseudo-Code of the parallel algorithm is in Appendix Section. At the same time, Flowchart and Working Example

of the similar algorithm are same as serial wherein parallel each recursive call is executed by available core at that time.

D. Performance Analysis of Parallel Algorithm :

We tested our data set on this parallel algorithm for different cores and we get below results :

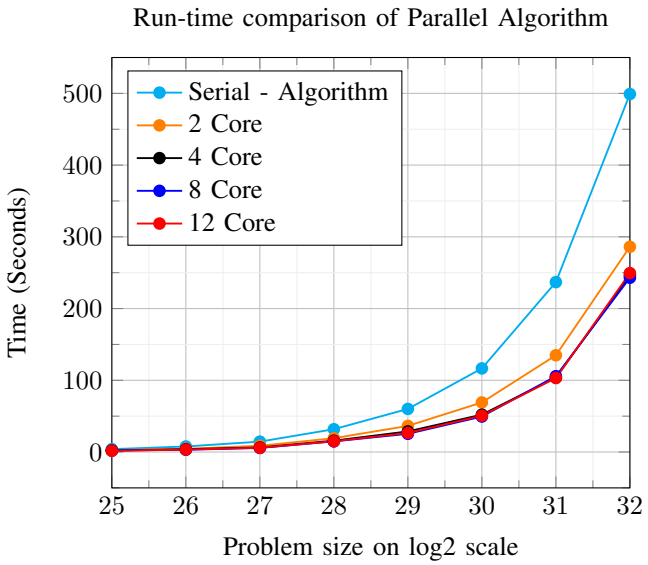

Fig. 3. Run-Time Analysis of parallel algorithm with different cores

From Fig. 3 we get as we increase the cores, the run time of the algorithm decreases. But after rising cores more, we observed that run-time did not decrease more. It remains constant. One possible reason is that our serial algorithm may not be parallelized further.

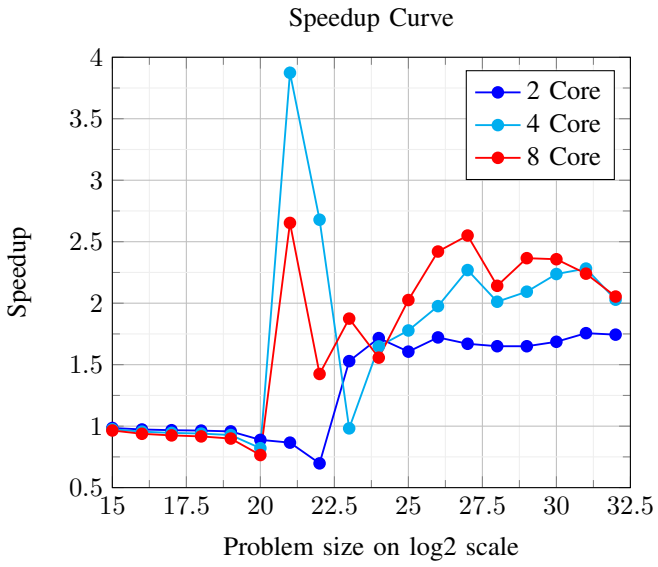

Fig. 4. Speedup on different cores

From Fig. 4 General Trend is as we increase the cores, the speedup increases. For lower values of problem size,

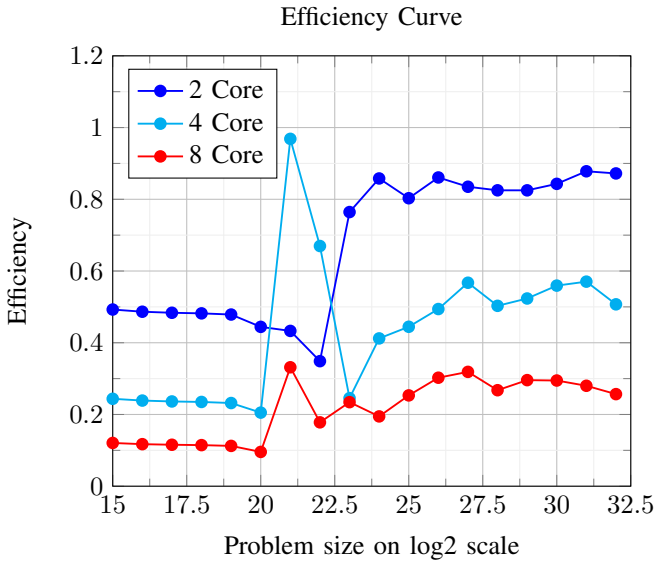

Fig. 5. Efficiency on different cores

speed up is the same. This is because overhead due to core synchronization and communication neglects to speed up achieved by parallelization. When the problem size is bigger, we see a significant difference in speed up for different cores. Also, speed up is constant when the problem size is large. The primary reason for this is that problem is still memory-bound; hence speed achieved in computation is nullified by memory access overhead.

And from Fig. 5 we can see that as the core size increases, the algorithm's efficiency decreases. This shows us that our serial algorithm can be parallelized up to 50%.

E. Performance on Different Hardware

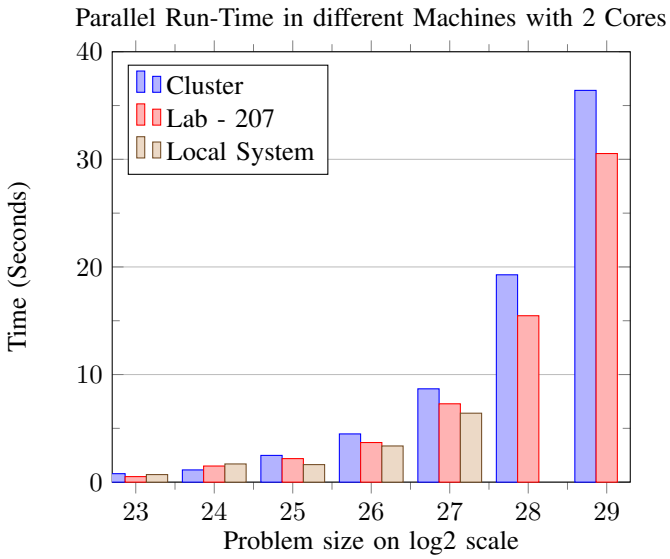

Fig. 6. Run-Time Analysis of parallel algorithm on different machines

We can see in Fig. 7 that the local system performed better on the 2-core parallel algorithm. Efficiency analysis in the previous section showed that the parallel algorithm is most efficient when executed on 2 cores. All three machines can allocate 2-cores, the performance should depend on the memory access pattern, and we know that the local system has the best memory hardware. Hence, we get the best performance on that machine.

F. *Speedup Curve on Different Architecture*

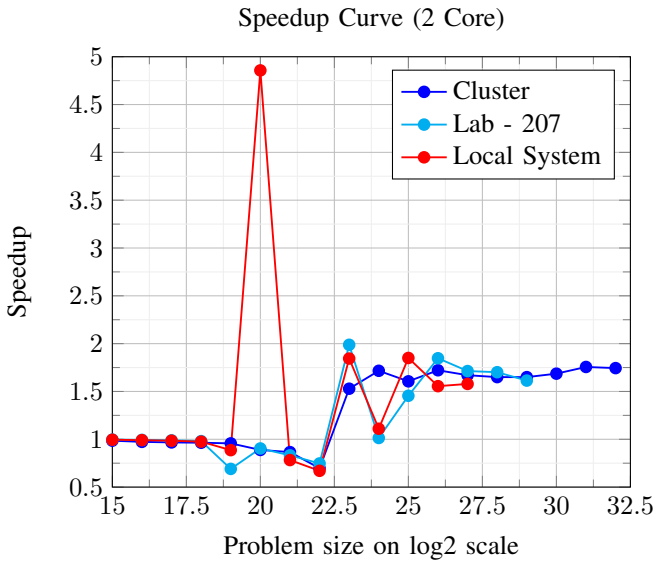

Fig. 7. Speedup on different machines

The above figure shows speedup on three different hardware Cluster, Lab - 207, and Local System. The figures show that speedup is nearly the same in all three architectures. So the algorithm is portable.

G. *Profiling of Parallel Algorithm :*

Profiling of parallel algorithm showed us that Effective CPU utilization had been quadrupled. But, the memory-bound matrix is the same. This implies that the algorithm is still memory-bound, but we can get better performance in terms of speed up if we use more cores.

Also, we observed that the partition function takes the most CPU time out of total time. We tried to parallelize the partition function, but we failed. We are getting speed-up right now is only because of parallelization in a recursive function. Speed can be improved if we parallelize the partition algorithm as well.

There are several OpenMP-based implementations of the single-pivot quicksort algorithm, but there is no OpenMP-based implementation of the Dual-pivot Quicksort algorithm as per our knowledge. OpenMP-based single-pivot quick sort performs worse as compared to our OpenMP-based performance. However, there is a thread-based implementation of Yaroslavisky's Dual-Partition algorithm, but it takes significantly more time to sort the data than our algorithm.

V. Hybrid Implementation (OpenMP + MPI)

This section discusses the hybrid (OpenMP+MPI) implementation of the dual-pivot algorithm. In addition to it, we will compare the performance of this implementation with the version of hypercube quicksort [12].

A. *Implementation Approach*

We scatter the entire array on the p processor and sort each chunk in parallel using an OpenMP-based dual-pivot algorithm. Next, we merge these sorted chunks using the merge function. Here, we use a tree-based merge algorithm to best use MPI functionalities. Each core sends its sorted subsequence to its neighbor, and a merge operation is performed at each step. This reduces the time complexity from $\mathcal{O}(n_1+n_2)$ to $\mathcal{O}((logp)(n_1+n_2))$. Pseudocode is present in the appendix section.

B. *Peformance Analysis of MPI Implementation*

The array of size n is divided into p chunks of size n/p and each chunk gets sorted in parallel on p core. It takes $\mathcal{O}((n/p)log(n/p))$ time. Then we merge the arrays using tree based merge function. It takes $\mathcal{O}(nlogp)$ time. So, overall time complexity turns out to be $\mathcal{O}((n/p)log(n/p)+nlogp)$.

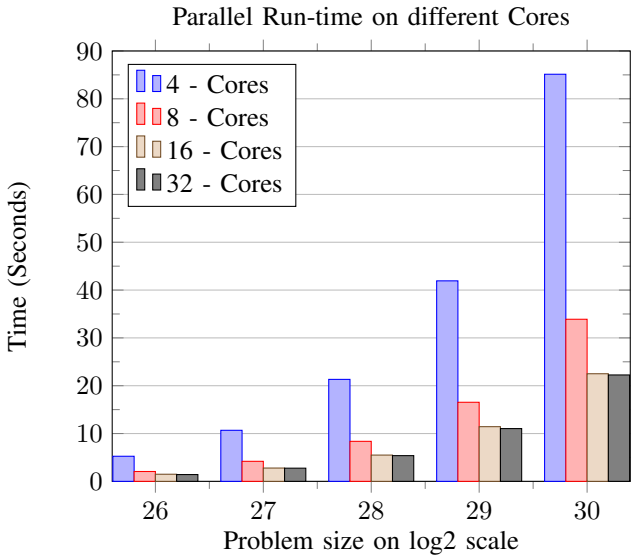

Fig. 8. Run-Time Analysis of parallel algorithm with different core

We see an expected trend in the run time curve as we increase the core and run time decreases. For smaller sizes, the difference in run time is not that significant because the communication overhead of MPI affects the run time. The difference is noticeable when the problem size is larger, as performance gain by MPI parallelization dominates run time over communication overhead.

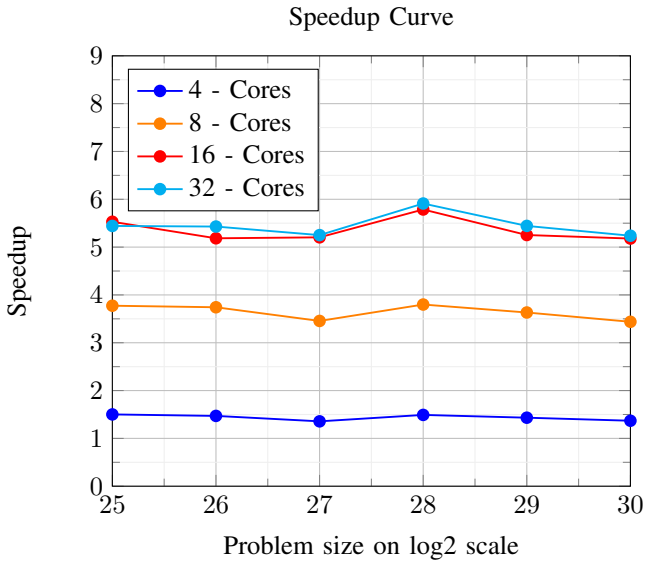

Fig. 9. Speed-up curve

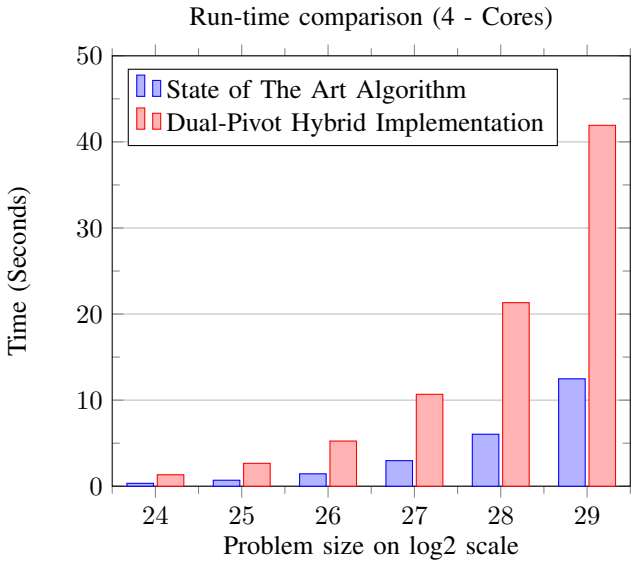

Fig. 10. Run time on 4 core

Speedup curve does not look as we expected. Remember time complexity function, it involves p and $1/p$ term. Therefore, when we increase number of cores after some point, we might not get desired results. For higher number of cores, time complexity of merge function will dominate over reduction of run time achieved by parallel sorting. This affects the speed up as well. And from the figure we can see that as the problem size increases the speed up remains nearly same.

C. *Comparison with State of The Art Algorithm*

In this section, we present a run time comparison of our MPI algorithm with the hypercube quicksort [12] algorithm. Following are the specification of the architecture of the machine we used to run the algorithms.

Hardware details:

- Base speed - 2.5GHz
- CPU - 4
- Socket - 1
- Cores per Socket - 4
- Size of L1 cache - 256K
- Size of L2 cache - 1MB
- Size of L3 cache - 8MB

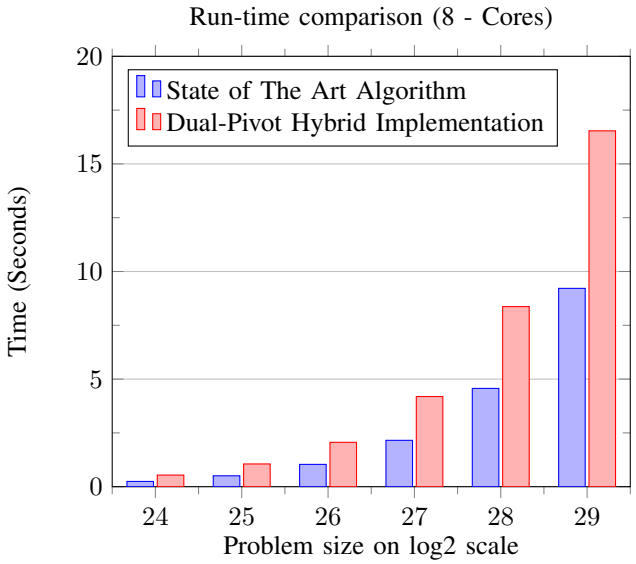

Fig. 11. Run time on 8 core

We observe that hypercube quicksort is performing better every time. When the problem size is small, the difference in run time is also slight, but it is clear that hypercube quicksort is highly scalable. We could scale the problem size up to 2^{29}, the maximum possible size of the array on the given architecture. The reason for the difference in run time is the difference in approach. You can refer to the original paper [12] for more details.

VI. Conclusion

Dual pivot quick-sort is the better-optimized version of partitioned-based algorithms. We observed that its serial implementation outperformed STL quicksort. We parallelized dual-pivot quicksort using two parallel interfaces, namely 1) OpenMP and 2) OpenMPI. Both gave a significant performance improvement. Finally, we compared our algorithms with the state-of-the-art algorithm. We observed that state of art algorithm is more scalable than our algorithms. In OpenMP implementation, there is a scope for improvement through parallelization of the partition algorithm. In MPI implementation, one can explore the approach in which one can broadcast pivots across all processes and then use them as pivots in each process and then gather sorted the arrays after specific operations. For more insight, you can refer to the paper by Prasad [10].

References

[1] Alaa Ismail Elnashar. Parallel performance of mpi sorting algorithms on dual-core processor windows-based systems. *arXiv preprint arXiv:1105.6040*, 2011.

[2] Charles AR Hoare. Quicksort. *The computer journal*, 5(1):10–16, 1962.

[3] Piotr Indyk. Introduction to algorithms. 2004.

[4] Kil Jae Kim, Seong Jin Cho, and Jae-Wook Jeon. Parallel quick sort algorithms analysis using openmp 3.0 in embedded system. In *2011 11th International Conference on Control, Automation and Systems*, pages 757–761. IEEE, 2011.

[5] Mohammad FJ Klaib, Mutaz Rasmi Abu Sara, and Masud Hasan. A parallel implementation of dual-pivot quick sort for computers with small number of processors. *Indonesia Journal on Computing (Indo-JC)*, 5(2):81–90, 2020.

[6] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[7] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J Ian Munro. Multi-pivot quicksort: Theory and experiments. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 47–60. SIAM, 2014.

[8] Jie Liu, Clinton Knowles, and Adam Brian Davis. A cost optimal parallel quicksorting and its implementation on a shared memory parallel computer. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 491–502. Springer, 2005.

[9] Marian Mureşan. Sorting algorithms. In *Introduction to Mathematica® with Applications*, pages 35–53. Springer, 2017.

[10] Prasad Perera. Parallel quicksort using mpi & performance analysis. *Retrieved on March*, 1, 2011.

[11] Andrew Sohn and Yuetsu Kodama. Load balanced parallel radix sort. In *Proceedings of the 12th international conference on Supercomputing*, pages 305–312, 1998.

[12] Hari Sundar, Dhairya Malhotra, and George Biros. Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th international ACM conference on international conference on supercomputing*, pages 293–302, 2013.

[13] Hari Sundar, Rahul S Sampath, and George Biros. Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

[14] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pages 372–381. IEEE, 2003.

[15] Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

APPENDIX

A. Pseudocodes

Algorithm 1 Quick-Sort Algorithm using OpenMP

```
QUICKSORT_OPENMP(A, size)
#pragma omp parallel
    #pragma omp single nowait
        DUALPIVOTQUICKSORT(A,A+size)
```

Algorithm 2 Small-Sort Algorithm

```
SMALLSORT(A, left, right)
for i ← left → right − 1 do
    for j ← i → left and (A[j + 1] < A[j]) do
        Swap A[j + 1] and A[j]
    end for
end for
```

Algorithm 3 Yaroslavskiy's Dual-Partition Algorithm

```
DUALPIVOTQUICKSORT(start,end)
left ← 0
right ← end − start + 1
if right ≤ left then
    return
end if
if right − left ≤ 31 then
    SMALLSORT(A, left, right)
    return
end if
idx1 ← left + (right − left)/3
idx2 ← right − (right − left)/3
Swap A[idx1] and A[left]
Swap A[idx2] and A[right]
if A[right] < A[left] then
    Swap A[left] and A[right]
end if
pivot1 ← A[left]
pivot2 ← A[right]
if pivot1 < pivot2 then
    PARTITION1(start, end, left, right, pivot1, pivot2)
else
    PARTITION2(start, end, left, right, pivot1, pivot2)
end if
```

Algorithm 4 Partition

```
PARTITION1(start, end, left, right, pivot1, pivot2)
less ← left + 1
greater ← right − 1
while !(A[greater] < pivot2) do
    greater ← greater − 1
end while
while !(pivot1 < A[less]) do
    less ← less + 1
end while
k ← less
while k ≤ greater do
    if !(pivot1 < A[k]) then
        Swap A[k] and A[less]
        less ← less + 1
    else if !(A[k] < pivot2) then
        Swap A[k] and A[greater]
        greater ← greater − 1
        while !(A[greater] < pivot2) do
            greater ← greater − 1
        end while
        if !(pivot1 < A[k) then
            Swap A[k] and A[less]
            less ← less + 1
        end if
    end if
    k ← k + 1
end while
Swap A[less − 1] and A[left]
Swap A[greater + 1] and A[right]
#pragma omp task
    DUALPIVOTQUICKSORT(start, start + less − 1)
#pragma omp task
    DUALPIVOTQUICKSORT(start + greater + 2, end)
#pragma omp task
    DUALPIVOTQUICKSORT(start + less,
                                start + greater + 1)
```

Algorithm 5 Partition

PARTITION2($start$, end, $left$, $right$, $pivot1$, $pivot2$)

$less \leftarrow left + 1$
$greater \leftarrow right - 1$
while $pivot1 <$ A[$greater$] **do**
    $greater \leftarrow greater - 1$
end while
while A[$less$] $< pivot1$ **do**
    $less \leftarrow less + 1$
end while
$k \leftarrow less$
while $k \leq greater$ **do**
    if A[k] $< pivot1$ **then**
        Swap A[k] and A[$less$]
        $less \leftarrow less + 1$
    else if $pivot1 <$ A[k] **then**
        Swap A[k] and A[$greater$]
        $greater \leftarrow greater - 1$
        while $pivot1 <$ A[$greater$] **do**
            $greater \leftarrow greater - 1$
        end while
        if A[k] $< pivot1$ **then**
            Swap A[k] and A[$less$]
            $less \leftarrow less + 1$
        end if
    end if
    $k \leftarrow k + 1$
end while
Swap A[$less - 1$] and A[$left$]
Swap A[$greater + 1$] and A[$right$]
#pragma omp task
    DUALPIVOTQUICKSORT($start$, $start + less - 1$)
#pragma omp task
    DUALPIVOTQUICKSORT($start + greater + 2$, end)

Algorithm 6 Quick-Sort Algorithm using OpenMP + MPI

QUICKSORT_MPI(argc, argv)

omp_set_num_threads(num_threads)
MPI_Init($argc$**,** $argv$**)**
MPI_Comm_size(MPI_COMM_WORLD, p)
MPI_Comm_rank(MP_COMM_WORLD, id)
if $id == 0$ **then**
    Generate an Array
end if
MPI_Barrier(MPI_COMM_WORLD)
MPI_Bcast (n, 1, MPI_LONG_LONG_INT, 0,
MPI_COMM_WORLD)
if mod(n,p)==0 **then**
    $c \leftarrow \frac{n}{p}$
else
    $c \leftarrow \frac{n}{p+1}$
end if
$chunk \leftarrow size\_of$(c)
MPI_Scatter (arr, c, MPI_DOUBLE, chunk, c,
MPI_DOUBLE, 0, MPI_COMM_WORLD)
if $n \geq c \cdot (id + 1)$ **then**
    $s \leftarrow c$
else
    $s \leftarrow$n - c·id
end if
#pragma omp parallel
    #pragma omp nowait
        DUALPIVOTQUICKSORT($chunk$, $chunk + s$)
for $i \leftarrow 1 \rightarrow p$ **do**
    if $mod(id, 2 * step)! = 0$ **then**
        MPI_Send (chunk, s, MPI_DOUBLE, id - step,
        0, MPI_COMM_WORLD)
        break
    end if
    if $id + step < p$ **then**
        if $n \geq c \cdot (id + 2 \cdot step)$ **then**
            $o \leftarrow c \cdot step$
        else
            $o \leftarrow$n - c·id + step
        end if
        other$\leftarrow size_of(o)$
        MPI_Recv (other, o, MPI_DOUBLE, id+step, 0,
        MPI_COMM_WORLD, status)
        $arr \leftarrow$ MERGE($chunk$, s, $other$, o)
        $chunk \leftarrow arr$
        s$\leftarrow$ s + o
    end if
end for
MPI_Finalize()

B. Flow Chart

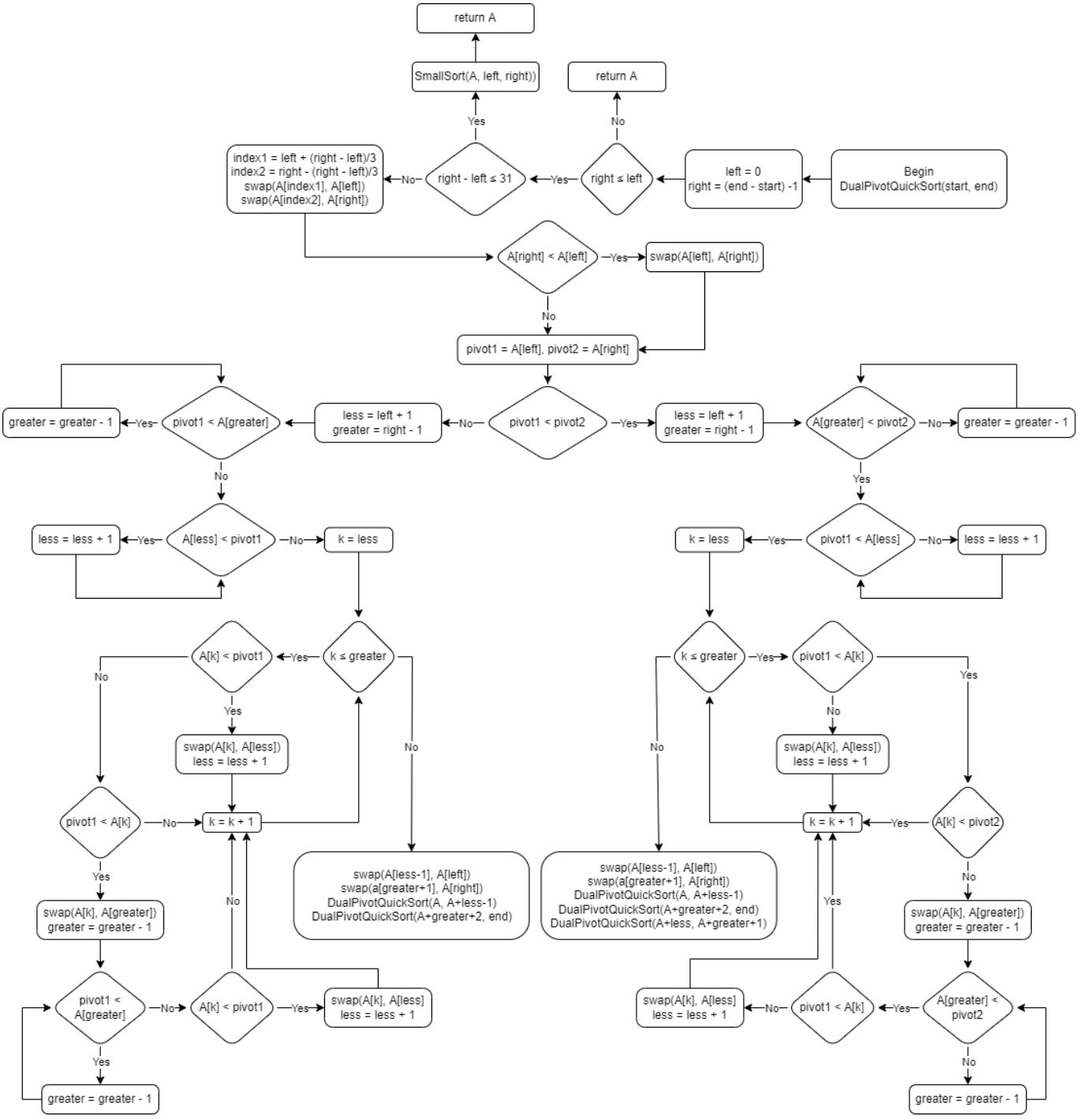

Fig. 12. Flow Chart of Algorithm

C. Working Example

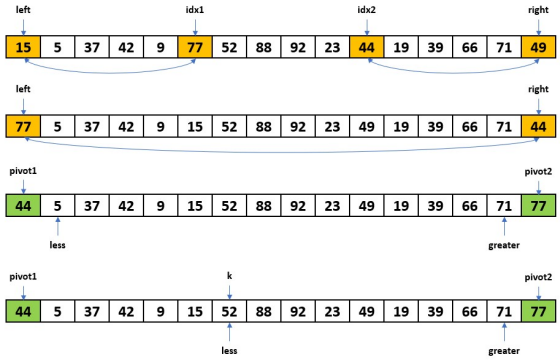

Fig. 13. Working Example 1

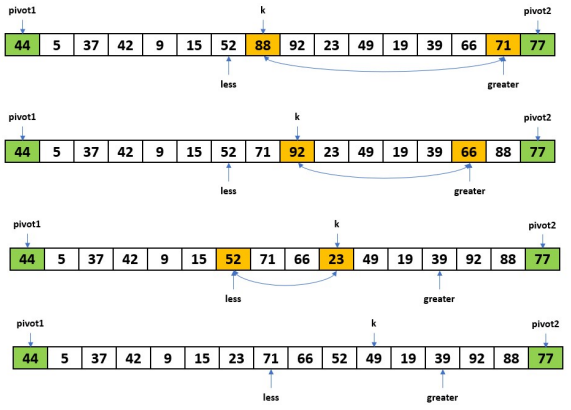

Fig. 14. Working Example 2

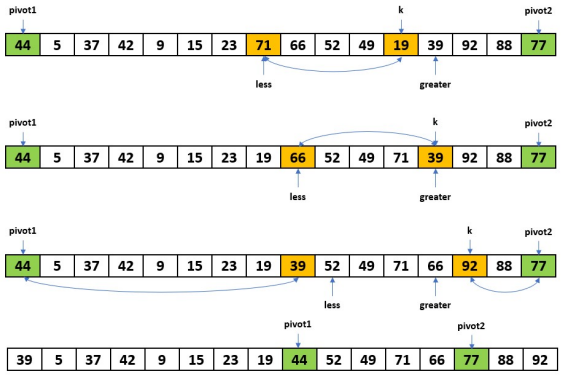

Fig. 15. Working Example 3

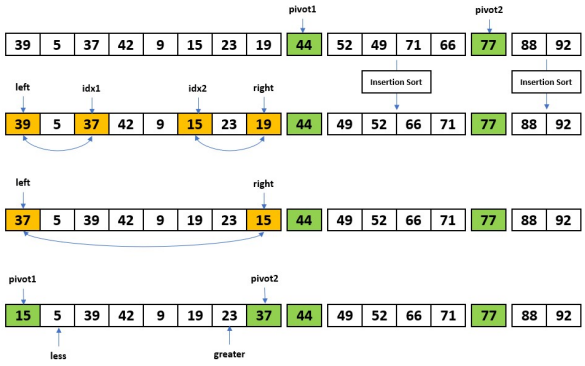

Fig. 16. Working Example 4

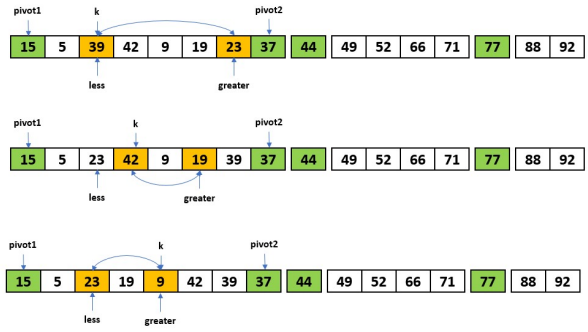

Fig. 17. Working Example 5

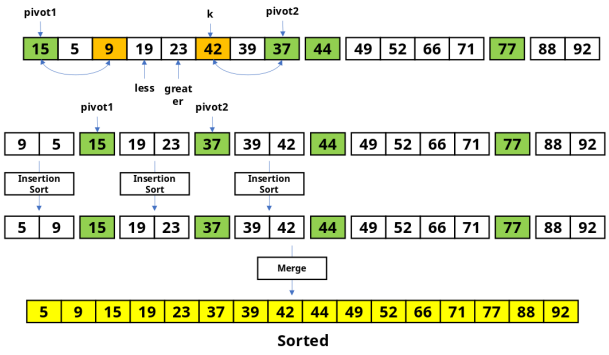

Fig. 18. Working Example 6