Hypercube Quicksort using MPI & Parallel Merge sort using OpenMP

Chikkeri Chinmaya
211IT017
NIT Karnataka,
Surathkal, Mangalore

Umesh
211IT073
NIT Karnataka,
Surathkal, Mangalore

Vishwa Mohan Reddy G
211IT082
NIT Karnataka,
Surathkal, Mangalore

Vismay P
211IT083
NIT Karnataka,
Surathkal, Mangalore

***Abstract*—In the realm of modern computing, the relentless growth of data necessitates the development and optimization of sorting algorithms that can handle vast datasets efficiently. Parallel computing, with its ability to harness the power of multiple processing units concurrently, presents a promising avenue for achieving enhanced sorting performance. This project embarks on the exploration and implementation of parallel sorting algorithms, focusing on Hypercube Quicksort utilizing MPI (Message Passing Interface) and Merge Sort employing OpenMP (Open Multi-Processing).**

I. INTRODUCTION

Sorting, a quintessential operation in computer science, serves as the backbone for various applications ranging from database management to information retrieval. The efficiency of sorting algorithms becomes increasingly crucial as the scale and complexity of datasets continue to escalate. Traditional sorting methods, while effective for smaller datasets, face challenges when confronted with the immense volumes of data prevalent in today's computing landscape.

Parallel computing emerges as a compelling solution to address these challenges by concurrently processing different parts of the dataset, thereby significantly reducing the overall sorting time. This project focuses on the parallelization of two distinct sorting algorithms—Hypercube Quicksort and Merge Sort—leveraging the capabilities of MPI and OpenMP, respectively.

Hypercube Quicksort utilizing MPI and Parallel Merge Sort employing OpenMP are two high-performance parallel sorting algorithms designed to efficiently sort large datasets in a distributed computing environment. These algorithms leverage the power of parallelism to enhance sorting speed, making them suitable for handling substantial volumes of data across multiple processing units.

Hypercube Quicksort, a variant of the Quicksort algorithm, operates on the principle of recursively partitioning the dataset based on a pivot element. With MPI (Message Passing Interface), Hypercube Quicksort exploits a distributed memory model, allowing different processors to handle subsets of the dataset. The hypercube topology, characteristic of MPI, enables efficient communication between processors, each responsible for sorting a portion of the data. By recursively dividing the dataset across the hypercube nodes and performing pivot-based partitioning, this algorithm achieves parallelism and scalability, minimizing communication overhead and improving overall sorting efficiency.

On the other hand, Parallel Merge Sort, utilizing OpenMP, targets shared memory architectures by dividing the sorting task into multiple threads that collaboratively sort segments of the dataset. Merge Sort, a divide-and-conquer algorithm, splits the data into smaller sub-arrays, sorting them individually, and then merging these sorted segments to produce the final sorted array. OpenMP, with its directive-based approach, facilitates thread-level parallelism within a shared memory system. Each thread works on a distinct segment of the dataset, sorting them independently, and subsequently merging these segments in parallel to accomplish the final sorted output.

Both algorithms offer advantages and exhibit distinct characteristics. Hypercube Quicksort with MPI excels in scenarios where a distributed memory environment is available, efficiently managing large datasets across multiple nodes. The algorithm's reliance on pivot-based partitioning and the communication efficiency of MPI ensures scalability for significant data sizes. Conversely, Parallel Merge Sort using OpenMP is ideal for shared memory systems, exploiting multicore processors to sort data effectively within a single node. Its simplicity in implementation and efficient merging technique make it suitable for scenarios with moderate-sized datasets on machines with shared memory architectures.

However, these algorithms also present challenges. Hypercube Quicksort's efficiency might suffer if the dataset is not well-balanced during partitioning across nodes, leading to load imbalances and potentially affecting overall performance. Parallel Merge Sort, although efficient for shared memory systems, might encounter scalability issues when handling extremely large datasets that exceed the memory capacity of a single node.

In conclusion, Hypercube Quicksort with MPI and Parallel Merge Sort using OpenMP are powerful parallel sorting algorithms, each tailored for distinct computing environments. Understanding their strengths, limitations, and applicability to specific hardware configurations is crucial for optimizing sorting performance in various parallel computing scenarios.

II. Serial Sorting Algorithms

A. Quicksort

Quicksort, is one of the most efficient sorting algorithms. At its core lies the strategy of divide and conquer. The algorithm begins by selecting a pivot element from the array. This chosen pivot serves as a reference point, around which the array is partitioned into two sub-arrays—one with elements smaller than the pivot and the other with elements greater than the pivot. The elegance of Quicksort unfolds in its recursive application of the same process to these sub-arrays.

The efficiency of Quicksort, with an average-case time complexity of O(n log n), places it among the fastest sorting algorithms in practical applications. One of its notable features is the potential for in-place sorting, meaning it doesn't demand additional memory for auxiliary data structures.

However, the success of Quicksort is intricately tied to the choice of the pivot. The algorithm's worst-case performance occurs when poorly chosen pivots lead to unbalanced partitioning. Despite this, the versatility and speed of Quicksort make it a formidable choice for various sorting scenarios.

B. Merge Sort

In the realm of sorting algorithms, Merge Sort emerges as a reliable and predictable contender. Merge Sort employs a straightforward divide-and-conquer strategy. The array undergoes recursive division into halves until each sub-array holds only a single element. These single-element sub-arrays are then meticulously merged to construct a fully sorted array.

Merge Sort boasts stability, ensuring that equal elements maintain their relative order in the sorted output. Its consistent O(n log n) time complexity makes it an attractive choice for scenarios involving substantial datasets. However, an aspect to consider is the algorithm's use of additional memory during the merging process, which can be a concern in memory-constrained environments.

III. Scope for Parallelisation

The exploration of parallel algorithms emerges as a compelling avenue in the pursuit of enhancing computational efficiency, particularly when dealing with large datasets. Both Quicksort and Merge Sort, stalwarts of the sorting algorithms, present intriguing opportunities for parallelization, albeit with distinct considerations

A. Quicksort

- **Nature of Divide-and-Conquer:** Quicksort's inherent divide-and-conquer nature provides a natural opening for parallelization.
- **Pivot Selection and Partitioning:** The initial step of selecting a pivot and partitioning the array can be distributed across multiple processors or threads, effectively dividing the workload.
- **Challenges in Load Balancing:** Challenges arise in maintaining load balance, as the choice of pivot significantly influences partition sizes. Careful consideration is required to avoid load imbalances.
- **Recursive Steps:** Parallelization of the recursive steps is possible, with each recursive call potentially executed in parallel. However, managing synchronization overhead is crucial, and hybrid models often integrate shared and distributed memory paradigms.

B. Merge Sort

- **Strength in Merging Phase:** Merge Sort's strength lies in its merging phase, providing a distinct point for parallelization.
- **Concurrent Merging:** Multiple sorted sub-arrays can be concurrently merged, capitalizing on parallel processing capabilities.
- **Challenges in Divide Step:** The divide step is less amenable to parallel execution due to dependencies. Still, parallelism can be beneficial when dealing with large datasets.
- **Communication and Synchronization:** Managing communication and synchronization between concurrently merging threads or processes becomes crucial. Load balancing is essential to ensure equitable distribution of the merging workload and prevent bottlenecks.

In summary, both Quicksort and Merge Sort offer promising avenues for parallelization, leveraging their unique characteristics. The challenge lies not only in designing algorithms that exploit parallel architectures effectively but also in addressing the intricacies of load balancing, communication, and synchronization. The ensuing sections will unravel the parallel implementations of these algorithms, shedding light on the strategies employed to harness the power of parallel computing.

IV. Hypercube Quicksort using MPI

The steps described outline the Hypercube Quicksort algorithm leveraging a distributed environment where each processor performs a portion of the sorting task on its local list. Here's a breakdown of the steps and an algorithmic overview:

Sequential Quicksort on Local Lists:

Each processor initiates by executing the sequential Quicksort algorithm on its locally assigned portion of the dataset. This step ensures initial sorting of the data within the processor's memory space. Choosing a Pivot Close to the Median:

With each processor having sorted its local list, there's an increased chance of choosing a pivot closer to the median within that subset. Median Selection and Broadcast:

One processor (or a designated one) responsible for median selection gathers the median value from its local sorted list. This processor broadcasts the selected median value to all other processors in the system. Division and Swapping:

Based on the received median value, each processor divides its local list into two parts: one with elements less than the median and the other with elements greater than the median. Then, processors swap their elements between the partner

processors based on these divided lists. Merging Received and Remaining Lists:

In Hypercube Quicksort, after the swapping step, each processor combines the remaining half of its local list with the received half from the partner processor. The merging process involves merging the received and remaining halves into a single sorted local list on each processor. Recursion and Final Sorting:

Finally, the algorithm recurses between the upper and lower half processors, repeating the process until the entire dataset is sorted.

Algorithm for Hypercube Quicksort:

Algorithm 1 Hypercube Quicksort

```
0: procedure HYPERCUBE_QUICKSORT(local_list)
0:     Perform Sequential Quicksort on local_list
0:     median = Select_Median(local_list)
0:     Broadcast median to all processors
0:     low_list, high_list = Divide_Local_List(local_list, median)
0:     Swap_Elements_with_Partner_Processor(low_list, high_list)
0:     merged_list = Merge_Local_Lists(local_list, received_list)
0:     if necessary_criteria_met then
0:         Hypercube_Quicksort(merged_list)
0:     end if
0: end procedure=0
```

V. Parallel Merge Sort

Parallel Merge Sort is a parallelized version of the traditional merge sort algorithm that leverages multiple processors or computing units to expedite the sorting process. The key idea is to divide the dataset into smaller subsets and perform independent merge sort operations on each subset simultaneously. Here's an explanation of the steps along with pseudocode for Parallel Merge Sort:

Parallel Merge Sort Algorithmic Overview:

Divide the Dataset:

The input dataset is divided into smaller subsets, and each subset is assigned to a separate processor. This step is crucial for parallelism, as each processor can independently sort its assigned subset. Parallel Sorting:

Each processor executes a local merge sort on its assigned subset independently and concurrently. The local merge sort ensures that each subset is sorted within the processor's memory space. Merge Phase:

Processors are paired, and each pair collaboratively merges their sorted subsets into a larger sorted subset. This merging step is performed in parallel across all processor pairs. Recursive Merging:

Steps 2 and 3 are repeated recursively until the entire dataset is merged into a single sorted list. The recursion continues until only one processor remains, and it holds the fully sorted dataset. Parallel Merge Sort Pseudocode:

article

Algorithm 2 Parallel Merge Sort

```
0: procedure PARALLEL_MERGE_SORT(local_list)
0:     if (local_list.length ≤ 1) then
0:         return local_list
0:     end if
0:     mid = local_list.length/2
0:     left_half = local_list[0 : mid]
0:     right_half = local_list[mid :]
0:     Parallel_Sort(left_half)
0:     Parallel_Sort(right_half)
0:     local_list = Parallel_Merge(left_half, right_half)
0:     return local_list
0: end procedure
0: function PARALLEL_MERGE(list1, list2)
0:     merged_result = []
0:     i = 0
0:     j = 0
0:     while i < list1.length and j < list2.length do
0:         if list1[i] < list2[j] then
0:             merged_result.append(list1[i])
0:             i = i + 1
0:         else
0:             merged_result.append(list2[j])
0:             j = j + 1
0:         end if
0:     end while
0:     {Add the remaining elements, if any}
0:     merged_result.extend(list1[i:])
0:     merged_result.extend(list2[j:])
0:     return merged_result
0: end function
   =0
```

VI. Results and Discussion

A. Merge Sort

The code compiled successfully and should report error=0 for the following instances:

```
gcc -fopenmp sort_list_openmp.c -o sort -lm
./sort 4 1
./sort 4 2
./sort 4 3
./sort 20 4
./sort 24 8
```

Following are the logs obtained from executing the above-mentioned commands:

List Size	Threads	Error	parallel Time (sec)	serial time (sec)
16	2	0	0.0059	0.0000
16	4	0	0.0063	0.0000
16	8	0	0.0066	0.0000
1048576	16	0	0.0261	0.1715
16777216	256	0	0.5213	3.4749

TABLE I
MERGE SORT RESULTS

B. Hypercube Quicksort

The code compiled successfully and should report error=0 for the following instance:

```
mpic++ qsort_hypercube.cpp -o qsort
mpirun -np 2 qsort 100 -1
```

Following are the logs obtained from executing the above-mentioned command:

```
[Proc: 0] 200 199 198 197 196 195 194 193
[Proc: 0] 192 191 190 189 188 187 186 185
[Proc: 0] 184 183 182 181 180 179 178 177
[Proc: 0] 176 175 174 173 172 171 170 169
[Proc: 0] 168 167 166 165 164 163 162 161
[Proc: 0] 160 159 158 157 156 155 154 153
[Proc: 0] 152 151 150 149 148 147 146 145
[Proc: 0] 144 143 142 141 140 139 138 137
[Proc: 0] 136 135 134 133 132 131 130 129
[Proc: 0] 128 127 126 125 124 123 122 121
[Proc: 0] 120 119 118 117 116 115 114 113
[Proc: 0] 112 111 110 109 108 107 106 105
[Proc: 0] 104 103 102 101
[Proc: 0] number of processes = 2,
initial local list size = 100,
hypercube quicksort time = 0.000124
[Proc: 0] check_list: local_error = 0
[Proc: 0] Congratulations.
The list has been sorted correctly.
[Proc: 0] 1 2 3 4 5 6 7 8
[Proc: 0] 9 10 11 12 13 14 15 16
[Proc: 0] 17 18 19 20 21 22 23 24
[Proc: 0] 25 26 27 28 29 30 31 32
[Proc: 0] 33 34 35 36 37 38 39 40
[Proc: 0] 41 42 43 44 45 46 47 48
[Proc: 0] 49 50 51 52 53 54 55 56
[Proc: 0] 57 58 59 60 61 62 63 64
[Proc: 0] 65 66 67 68 69 70 71 72
[Proc: 0] 73 74 75 76 77 78 79 80
[Proc: 0] 81 82 83 84 85 86 87 88
[Proc: 0] 89 90 91 92 93 94 95 96
[Proc: 0] 97 98 99 100 101
[Proc: 1] 100 99 98 97 96 95 94 93
[Proc: 1] 92 91 90 89 88 87 86 85
[Proc: 1] 84 83 82 81 80 79 78 77
[Proc: 1] 76 75 74 73 72 71 70 69
[Proc: 1] 68 67 66 65 64 63 62 61
[Proc: 1] 60 59 58 57 56 55 54 53
[Proc: 1] 52 51 50 49 48 47 46 45
[Proc: 1] 44 43 42 41 40 39 38 37
[Proc: 1] 36 35 34 33 32 31 30 29
[Proc: 1] 28 27 26 25 24 23 22 21
[Proc: 1] 20 19 18 17 16 15 14 13
[Proc: 1] 12 11 10 9 8 7 6 5
[Proc: 1] 4 3 2 1
[Proc: 1] check_list: local_error = 0
[Proc: 1] 102 103 104 105 106 107 108 109
[Proc: 1] 110 111 112 113 114 115 116 117
[Proc: 1] 118 119 120 121 122 123 124 125
[Proc: 1] 126 127 128 129 130 131 132 133
[Proc: 1] 134 135 136 137 138 139 140 141
[Proc: 1] 142 143 144 145 146 147 148 149
[Proc: 1] 150 151 152 153 154 155 156 157
[Proc: 1] 158 159 160 161 162 163 164 165
[Proc: 1] 166 167 168 169 170 171 172 173
[Proc: 1] 174 175 176 177 178 179 180 181
[Proc: 1] 182 183 184 185 186 187 188 189
[Proc: 1] 190 191 192 193 194 195 196 197
[Proc: 1] 198 199 200
```

C. Analysis

In our comparative analysis, both Hypercube Quicksort and Parallel Merge Sort have demonstrated distinct strengths and limitations. Hypercube Quicksort, designed for distributed memory systems, showcases remarkable scalability and parallelism. However, challenges in load balancing and potential communication overhead in larger systems warrant further optimization.

On the other hand, Parallel Merge Sort, tailored for shared memory architectures, exhibits efficient multicore utilization for moderate-sized datasets. Yet, scalability issues become apparent with significantly larger datasets, particularly during the sorting of smaller segments by individual threads.

These algorithmic insights highlight the need for targeted optimizations to enhance their respective performances in diverse computing environments and dataset scales. Future work may focus on addressing the identified limitations to broaden the applicability and efficiency of these parallel sorting algorithms.

VII. Conclusion

The experimental results demonstrate that both Hypercube Quicksort and Parallel Merge Sort offer substantial performance improvements over their sequential counterparts, when the dataset becomes larger and larger. The algorithms effectively utilize the parallel capabilities of MPI and OpenMP to handle large datasets efficiently. These parallel sorting algorithms can be valuable tools for high-performance computing applications that require rapid data organization and analysis.

References

[1] A. Geist, A. Beguelin, J. Dongarra, V. Naik, V. Bhaskar, W. Chen, P. Frederickson, and D. Satterfield, MPI, The Complete Reference: High-Performance Message-Passing Communication, Volume 1. MIT Press, 1994.

[2] D. Chapman, G. Jost, and R. Van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

[3] M. J. Flynn, "Some computer organizations for efficient scientific computation," Computers, vol. 23, no. 8, pp. 46-51, 1990.

[4] Quinn, M. J. (2002). Parallel programming in C with MPI and OpenMP (2nd ed.). McGraw-Hill, Inc.

[5] Grama, A., Gupta, A., Karypis, G., Kumar, V. (2003). Introduction to parallel computing (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.