Optimizing QuickSort for Shared and Distributed Memory Systems: A Hybrid Parallel Approach

High Performance Computing 2023
Exercise 2b

Giovanni Coronica
Matricola: IN2000215
31/01/24

1. Introduction

This report addresses the challenge of parallelizing the Quicksort algorithm, a classical and efficient sorting technique, using a hybrid approach that incorporates both MPI and OpenMP. Quicksort, developed by Tony Hoare in 1960, is a divide-and-conquer algorithm renowned for its efficiency in sorting arrays. It operates by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted, a process that continues until the entire array is ordered.

While Quicksort's efficiency is well-established in serial computing, adapting it for parallel computing environments presents unique challenges. These include efficiently distributing data and synchronizing tasks across the different memory paradigms present in multi-core and multi-node architectures. This report explores a hybrid parallelization strategy that leverages MPI for distributed memory management and OpenMP for shared memory systems.

The report outlines the methodology and key design choices involved in this hybrid parallelization of Quicksort. It details how MPI and OpenMP are utilized to optimize the algorithm's performance for sorting large datasets, and how the inherent challenges of parallelization are addressed.

Following the methodology, the report presents a comprehensive analysis of the algorithm's performance. This includes assessing scalability and efficiency through a series of tests conducted in various computational scenarios. The results and discussion sections provide insights into how effectively the parallelized Quicksort algorithm performs, highlighting the advantages and limitations of the hybrid approach.

2. Methodology and design choices

2.1 Software Stack and Compilation

The executions of the hybrid parallel Quicksort algorithm were conducted on a EPYC node within the ORFEO cluster. The software stack utilized OpenMPI library version 4.1.5 compiled with GNU compiler GCC 12.3.1, and OpenMP version 4.5. For local development on a MacBook Air 2022, a different software stack and compilation script were required due to system differences. Detailed instructions for compiling the code are provided in the `README` file in the repository.

2.2 Hybrid Parallel Implementation

2.2.1 Shared Memory Parallelism with OpenMP

The parallelization of Quicksort for shared memory systems employs OpenMP. Key to this approach is the use of OpenMP tasks rather than sections. Tasks provide dynamic scheduling, allowing better load balancing and more efficient utilization of processor resources. This is particularly beneficial for recursive

algorithms like Quicksort, where workloads are not uniform across recursive calls. The `quicksort` function was modified to generate parallel tasks for sorting subarrays, controlled by a threshold (`TASK_SIZE`) to balance task creation overhead with parallel execution benefits.

2.2.2 Distributed Memory Parallelism with MPI

In the MPI-based distributed memory paradigm, the root process initially allocates and populates the dataset. Data is then distributed to various processes using `MPI_Scatterv`, with each process sorting its allocated chunk locally. Once sorted, the chunks are merged using a parallel, tree-structured approach, optimizing the merge phase across processes.

This approach leverages parallel capabilities of distributed systems to lower sorting and merging time, especially with larger data and more processes.

2.2.3 Data Distribution

The choice to have the root process generate and distribute data helps maintain data consistency and simplifies the implementation. This approach is especially effective for uniformly distributed data sets and allows easy scalability across multiple processes.

2.3 Optimization

2.3.1 Pivot Selection

The shift to the median of three pivot selection aims to improve performance by reducing the chances of worst-case scenarios in the partitioning step, especially for certain data distributions.

2.3.2 Merging Complexity

To analyze the time complexity of the `merge_chunks_parallel` operation, let's consider the various steps involved in the operation, given the total size of the array N, the number of processors K, and p being the largest power of 2 less than or equal to K. Each process initially has approximately N/K elements.

1. Adjustment for Non-Power-of-Two Processes: In this step, processes with rank >= p send their data to the first p processes. Each of these processes sends N/K elements and then exits. The first extra processes among the first p processes receive data from one additional process each, merging it with their own data. The time complexity for merging 2 * N/K elements is O(2 * N/K) = O(N/K).
2. Power-of-Two Merging: In this step, the merging is performed in a tree-like fashion among the first p processes. At each level of the tree, pairs of processes merge their data, and the number of active processes is halved. The number of levels in the tree is O(log p). At each level, the size of the data being merged doubles, but the number of merges is halved. The time complexity at each level is O(N/p), and there are log p levels, so the total time complexity for this step is O((N/p) * log p).

Combining these steps, the overall estimated time complexity of the `merge_chunks_parallel` operation is O(N/K) + O((N/p) * log p). Since p is the largest power of 2 less than or equal to K, and $p <= K$, we can simplify the expression to O(N/K + (N/K) * log K).

3. Results

The performance of the hybrid parallel Quicksort algorithm was evaluated on the ORFEO HPC cluster using an EPYC node configuration. Two setups were tested: one with 24 threads on a single process, and the other with 24 processes each running a single thread. The array sizes tested were 1.000, 10.000, 100.000, 1.000.000, and 10.000.000 elements, with execution flows ranging from 1 to 24. Each configuration was executed 20 times, and the following statistics were gathered for the execution times: mean, median, standard deviation, minimum, and maximum.

3.1 Execution Time Statistics

The data tables contain the calculated mean, median, standard deviation, min, and max execution times (s) for each configuration and array size. These tables provide a detailed view of the performance metrics across the various tested scenarios.

3.1.1 Shared Memory Parallelism with OpenMP

size	nthreads	mean	median	std	min	max
1.000	1	0,000469	0,000466	0,000023	0,000430	0,000523
1.000	2	0,000363	0,000352	0,000053	0,000306	0,000534
1.000	4	0,000440	0,000427	0,000083	0,000311	0,000581
1.000	8	0,000436	0,000412	0,000066	0,000342	0,000567
1.000	12	0,004783	0,000425	0,008137	0,000329	0,024999
1.000	16	0,005466	0,000462	0,007982	0,000357	0,020045
1.000	20	0,021102	0,021939	0,012849	0,000308	0,039446
1.000	24	0,026997	0,027455	0,011403	0,002033	0,043628
10.000	1	0,004990	0,004953	0,000175	0,004800	0,005440
10.000	2	0,003180	0,003167	0,000110	0,002921	0,003342
10.000	4	0,002281	0,002241	0,000303	0,001803	0,002796
10.000	8	0,001669	0,001624	0,000170	0,001477	0,001998
10.000	12	0,004260	0,001785	0,006241	0,001376	0,020571
10.000	16	0,007453	0,001920	0,008458	0,001506	0,021845
10.000	20	0,005104	0,001904	0,006055	0,001394	0,019972
10.000	24	0,021609	0,020935	0,008169	0,001999	0,032313
100.000	1	0,056069	0,055428	0,001447	0,054814	0,059655
100.000	2	0,037764	0,038420	0,002233	0,033317	0,040155
100.000	4	0,024873	0,025687	0,002479	0,019105	0,027722
100.000	8	0,015508	0,015694	0,000934	0,013162	0,017069
100.000	12	0,017185	0,015642	0,005162	0,012161	0,031639
100.000	16	0,015558	0,014015	0,004930	0,011757	0,030653

100.000	20	0,017837	0,013327	0,008418	0,011006	0,037065
100.000	24	0,029418	0,029043	0,010121	0,011142	0,051472
1.000.000	1	0,677407	0,676664	0,012288	0,659296	0,700703
1.000.000	2	0,376782	0,366904	0,018650	0,358715	0,411577
1.000.000	4	0,262463	0,268002	0,028354	0,204515	0,297993
1.000.000	8	0,164809	0,164729	0,013143	0,133281	0,196746
1.000.000	12	0,140566	0,139991	0,013215	0,119447	0,167960
1.000.000	16	0,134955	0,133800	0,016628	0,113270	0,163339
1.000.000	20	0,127383	0,124234	0,013451	0,107949	0,150912
1.000.000	24	0,133515	0,128017	0,022323	0,101356	0,174423
10.000.000	1	7,992876	7,991915	0,078034	7,829550	8,142440
10.000.000	2	4,202275	4,200905	0,051568	4,109120	4,315450
10.000.000	4	2,329521	2,338485	0,058379	2,210220	2,416770
10.000.000	8	1,460068	1,457500	0,032835	1,408530	1,534960
10.000.000	12	1,255944	1,254180	0,049986	1,148410	1,362770
10.000.000	16	1,161716	1,142865	0,056518	1,090730	1,288640
10.000.000	20	1,082664	1,075015	0,037368	1,013780	1,152270
10.000.000	24	1,110637	1,107810	0,036062	1,048030	1,169830

Table 1: Execution Time Statistics for 24 Threads on 1 Process

3.1.2 Distributed Memory Parallelism with MPI

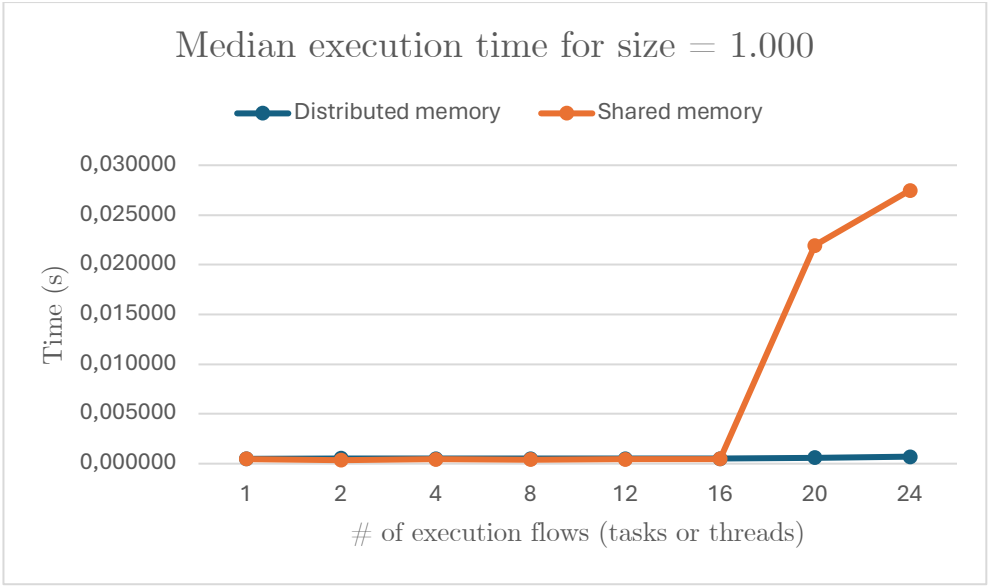
size	ntasks	mean	median	std	min	max
1.000	1	0,000487	0,000487	0,000033	0,000431	0,000554
1.000	2	0,000488	0,000534	0,000105	0,000340	0,000622
1.000	4	0,000517	0,000509	0,000173	0,000273	0,000897
1.000	8	0,000503	0,000503	0,000138	0,000257	0,000730
1.000	12	0,000546	0,000514	0,000210	0,000303	0,001032
1.000	16	0,000608	0,000500	0,000275	0,000327	0,001264
1.000	20	0,000624	0,000609	0,000160	0,000373	0,000965
1.000	24	0,000718	0,000705	0,000178	0,000455	0,001084
10.000	1	0,004991	0,004954	0,000178	0,004715	0,005480
10.000	2	0,002925	0,002934	0,000187	0,002679	0,003321
10.000	4	0,001978	0,001956	0,000181	0,001687	0,002301
10.000	8	0,001400	0,001369	0,000131	0,001232	0,001742
10.000	12	0,001281	0,001273	0,000118	0,001117	0,001580
10.000	16	0,001260	0,001210	0,000122	0,001123	0,001557
10.000	20	0,001315	0,001256	0,000184	0,001101	0,001736

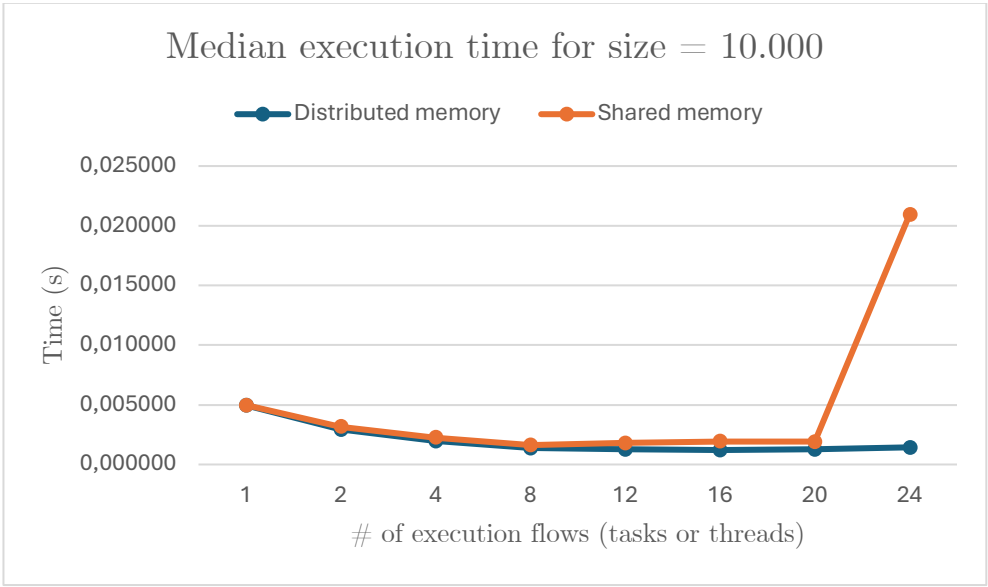
10.000	24	0,001505	0,001420	0,000246	0,001138	0,002038
100.000	1	0,055565	0,055228	0,000882	0,054479	0,057759
100.000	2	0,030663	0,030420	0,000974	0,029558	0,033452
100.000	4	0,018291	0,018206	0,000416	0,017630	0,019231
100.000	8	0,012480	0,012387	0,000338	0,012026	0,013223
100.000	12	0,010835	0,010766	0,000360	0,010380	0,012031
100.000	16	0,010195	0,010117	0,000341	0,009610	0,010828
100.000	20	0,010369	0,010168	0,000710	0,009486	0,011925
100.000	24	0,010607	0,010639	0,000848	0,009559	0,012550
1.000.000	1	0,681115	0,681865	0,018550	0,656932	0,712998
1.000.000	2	0,363175	0,363423	0,010502	0,348769	0,381451
1.000.000	4	0,209077	0,207534	0,005949	0,202457	0,225780
1.000.000	8	0,135031	0,134767	0,003569	0,129271	0,141447
1.000.000	12	0,118097	0,116890	0,003235	0,114545	0,124984
1.000.000	16	0,100694	0,099666	0,002607	0,097684	0,105844
1.000.000	20	0,104785	0,103012	0,006776	0,099787	0,128699
1.000.000	24	0,130016	0,138230	0,020717	0,096759	0,155661
10.000.000	1	8,026613	8,032850	0,049645	7,943740	8,136380
10.000.000	2	4,208000	4,202795	0,037125	4,145040	4,264400
10.000.000	4	2,426833	2,424670	0,023676	2,383150	2,492600
10.000.000	8	1,587851	1,584640	0,017853	1,559660	1,631320
10.000.000	12	1,371314	1,376265	0,026317	1,334340	1,432010
10.000.000	16	1,169530	1,169850	0,012097	1,149980	1,193890
10.000.000	20	1,161482	1,162065	0,014152	1,131620	1,183790
10.000.000	24	1,143038	1,136270	0,033574	1,099570	1,205980

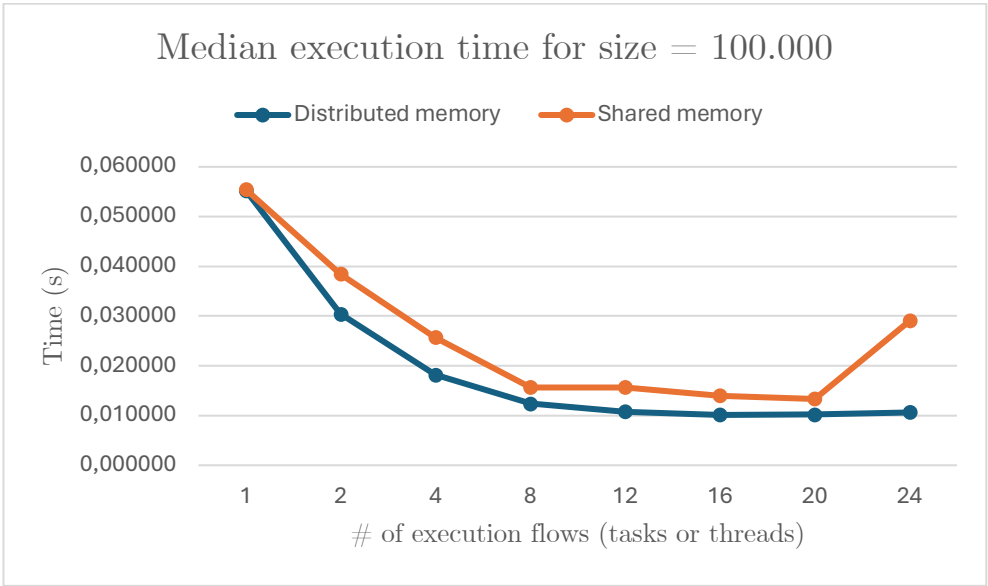
Table 2: Execution Time Statistics for 1 Thread on 24 Processes

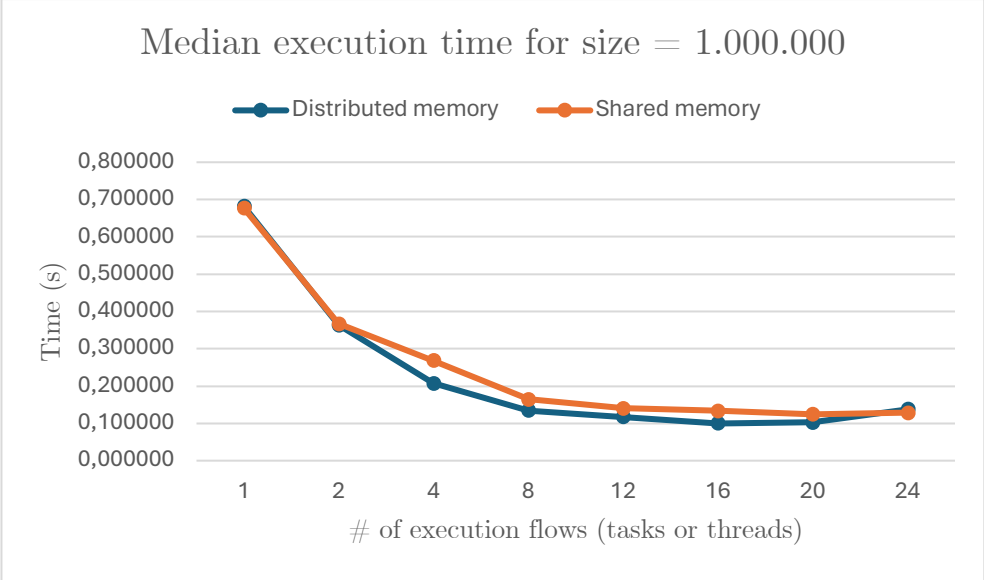
3.2 Comparative Analysis of Execution Times in Shared vs. Distributed Memory Environments

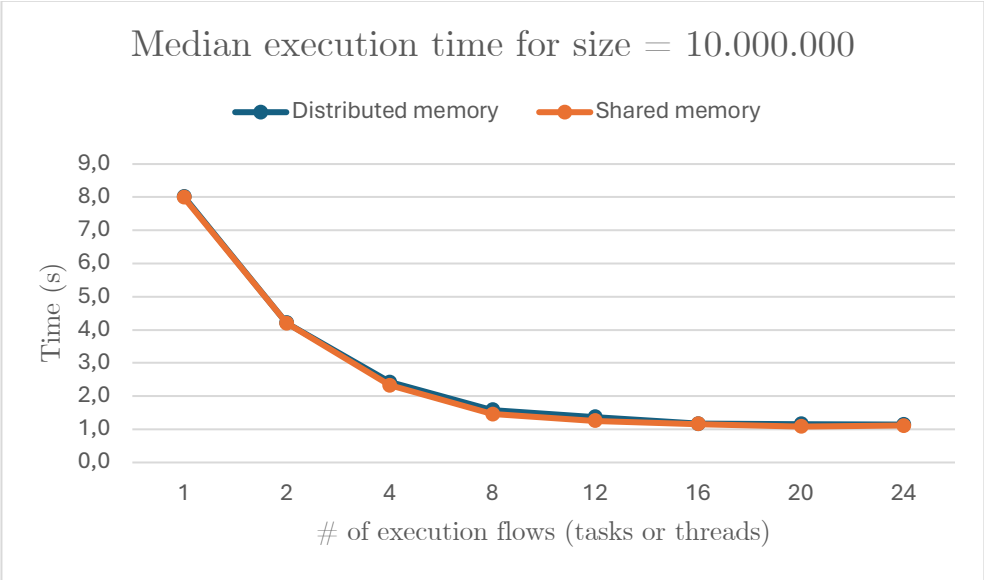
In this subsection, we present a comparative analysis of the median execution times for the Quicksort algorithm implemented in shared and distributed memory environments. The five graphs illustrate the performance of the hybrid parallel Quicksort algorithm by comparing the median execution times achieved using OpenMP and MPI for the specified data sizes. These visual representations provide a clear comparison of how each approach scales with increasing amounts of data and parallelism, allowing us to draw insights into the relative efficiencies and identify potential bottlenecks in the parallel sorting process.


Median execution time for size = 1.000
Distributed memory
Shared memory
0,030000
0,025000
0,020000
0,015000
0,010000
0,005000
0,000000
Time (s)
1
2
4
8
12
16
20
24
# of execution flows (tasks or threads)


Median execution time for size = 10.000
Distributed memory
Shared memory
0,025000
0,020000
0,015000
0,010000
0,005000
0,000000
Time (s)
1
2
4
8
12
16
20
24
# of execution flows (tasks or threads)


Median execution time for size = 100.000
Distributed memory
Shared memory
0,060000
0,050000
0,040000
0,030000
0,020000
0,010000
0,000000
Time (s)
1
2
4
8
12
16
20
24
# of execution flows (tasks or threads)





4. Discussion

The analysis of the Quicksort algorithm's performance across shared and distributed memory paradigms uncovers specific trends that illustrate the impact of execution flow count and data size on algorithm efficiency. By examining the provided execution time graphs for various data sizes, we can infer the relative strengths of each approach and how they scale with increasing data sizes and execution flows.

4.1 Performance on Smaller Data Sizes (up to 100,000 Elements)

For smaller data sizes, the distributed memory implementation consistently outperforms the shared memory version. This is due to the relatively low overhead associated with parallel merging operations in the MPI implementation, which does not significantly affect the total computation time. In contrast, the

shared memory implementation using OpenMP suffers from increased overhead as the number of threads grows. Thread management and potential contention for shared resources likely cause this overhead, which leads to diminishing returns on performance with more threads.

4.2 Performance on Larger Data Sizes (1,000,000 Elements)

At 1,000,000 elements, the shared memory implementation shows improved performance over distributed memory when using the maximum number of execution flows (24). This change indicates that the computational complexity of parallel merging in distributed memory starts to influence the overall execution time significantly. As the number of execution flows increases, the cost of merging chunks in MPI becomes more pronounced. Meanwhile, the overhead of managing many threads in OpenMP does not outweigh the benefits of parallelism, leading to more efficient performance compared to the distributed approach.

4.3 Performance on Very Large Data Sizes (10,000,000 Elements)

For the very large dataset of 10,000,000 elements, the results are more straightforward: the shared memory implementation exhibits better performance across nearly all execution flow counts greater than one. It seems that the overhead of distributing and merging large datasets in distributed memory systems becomes a dominant factor in the total computation time, which is not offset by the parallel computation benefits. In contrast, the shared memory approach is able to leverage the available threads more effectively, possibly due to more efficient data locality and less communication overhead compared to distributed memory operations.

4.4 Scalability Considerations

The strong scalability of the distributed memory approach is evident for smaller data sizes, where increasing the number of execution flows results in a proportional decrease in execution time. However, as data size grows, the scalability of MPI diminishes, especially when the computational cost of merging becomes significant.

In contrast, the shared memory approach initially shows poorer scalability due to overheads at smaller data sizes. Yet, as the data size becomes larger, OpenMP's scalability improves, particularly at the highest data size tested (10,000,000 elements). This suggests that for very large-scale problems, shared memory parallelism may be a more effective strategy, particularly when the number of threads is high.

5. Conclusions

The performance analysis of the hybrid parallel Quicksort algorithm reveals that distributed memory parallelism with MPI is optimal for smaller data sizes due to low overhead, while shared memory parallelism using OpenMP becomes more effective as data sizes increase, particularly when the cost of parallel merging in MPI becomes significant. These results suggest that the choice between shared and distributed memory implementations should be guided by the size of the dataset, with shared memory approaches preferred for very large data sizes to leverage thread-level parallelism and minimize communication overhead.