UPPSALA UNIVERSITET

Assignment 3

Quiscksort

Yuanjing Yang

June 26, 2025

1 Introduction and Problem Description

The goal of this assignment is to implement a parallel version of the Quicksort algorithm for MPI. We need to build several functions to read unsorted lists of integers and distribute them among different processes, doing *local_sort* function and *global_sort* function in sequences, gathering into the root processes, and printing it out. Here we evaluate different pivot selection strategies to analyze their impact on performance(strong scalability and weak scalability).

2 Algorithm and implementation

Table 1: Parallel Quicksort Algorithm Overview

Step	**Description**
1	The root process (rank 0) uses `read_input` to read the unsorted list of integers. The array is then divided into chunks and distributed to all processes using `MPI_Scatterv`.
2	Each process performs local sorting using the standard C function `qsort`.
3	Pivot selection strategies: • `MEDIAN_ROOT`: Median of root process's local data. • `MEAN_MEDIAN`: Mean of local medians from all processes. • `MEDIAN_MEDIAN`: Median of the medians collected from all processes.
4	Each process splits its data into two parts: values smaller or larger than the pivot.
5	Partner processes exchange relevant data using `MPI_Sendrecv`, ensuring elements less than or equal to the pivot are sent to the lower group, and the rest to the upper group.
6	After recursion finishes, sorted segments are gathered on the root process using `MPI_Gatherv`.

3 Performance Experiment

I ran strong and weak scalability tests on UPPMAX using Rackham.
For strong scalability, I used fixed input size of 1 billion integers(input1000000000.txt), using processes 1, 2, 4, 8, 16 in 3 different strategies.
From the results below, we can see all three pivot strategies exhibit good strong scalability up to 16 processes. Efficiency remains relatively high (above 60%) even with 16 processes. Strategy 2 (MEAN_MEDIAN) and Strategy 3 (MEDIAN_MEDIAN) provide slightly better efficiency and speedup than Strategy

1, especially as the number of processes increases.

Table 2: Strong scalability for Strategy 1 (Median_ROOT)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	237.122	100%	100%
2	121.984	194%	97%
4	67.2208	353%	88%
8	37.1413	634%	79%
16	25.8171	918%	57%

Table 3: Strong scalability for Strategy 2 (MEAN_MEDIAN)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	238.96	100%	100%
2	123.5	193%	97%
4	66.277	361%	90%
8	38.2289	625%	78%
16	24.0662	993%	62%

Table 4: Strong scalability for Strategy 3 (MEDIAN_MEDIAN)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	237.144	100%	100%
2	122.332	194%	97%
4	65.9955	362%	90%
8	37.3793	634%	79%
16	23.8909	993%	62%

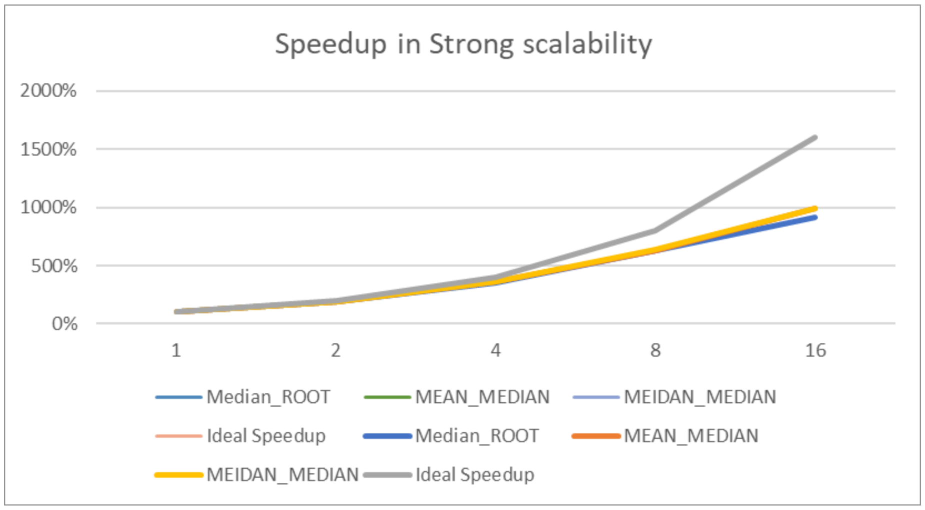

Figure 1: The Speedup of strong scalability

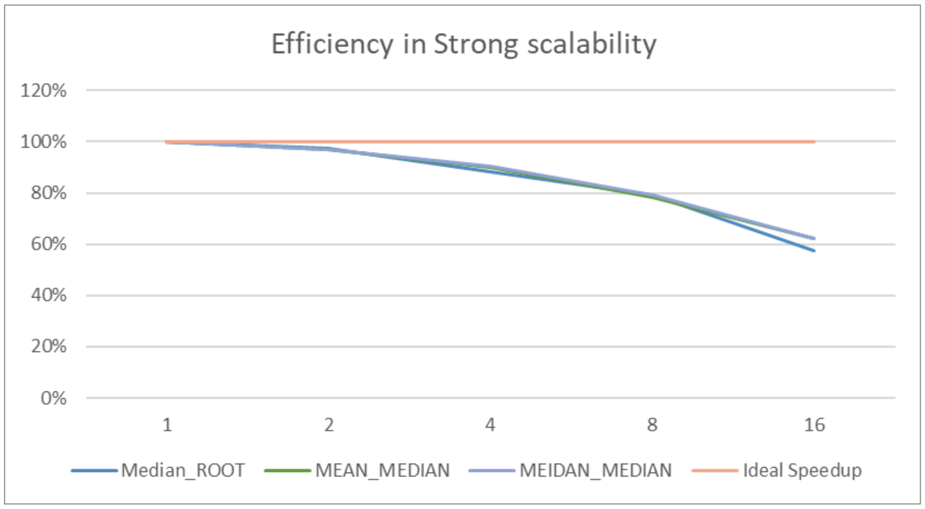

Figure 2: The Efficiency of strong scalability

For weak scalability, I used input scaled with the number of process(125 million integers per process), using processes 1, 2, 4, 8, 16 in 3 different strategies. As the number of processes increases, the efficiency drops significantly. This is expected due to increased communication overhead and imbalance during the recursive splitting and merging phases. All three pivot strategies show similar weak scalability behavior.

Table 5: Weak scalability for Strategy 1 (Median_ROOT)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	46.083	100%	100%
2	67.986	93%	46%
4	110.462	83%	21%
8	198.888	70%	9%
16	400.956	53%	3%

Table 6: Weak scalability for Strategy 2 (MEAN_MEDIAN)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	26.305	100%	100%
2	28.7194	93%	46%
4	31.2929	84%	21%
8	38.131	71%	9%
16	48.7517	54%	3%

Table 7: Weak scalability for Strategy 3 (MEDIAN_MEDIAN)

Processes	**Runtime**	**Speedup**	**Efficiency**
1	26.305	100%	100%
2	28.7194	92%	46%
4	31.2929	84%	21%
8	38.131	69%	9%
16	48.7517	54%	3%

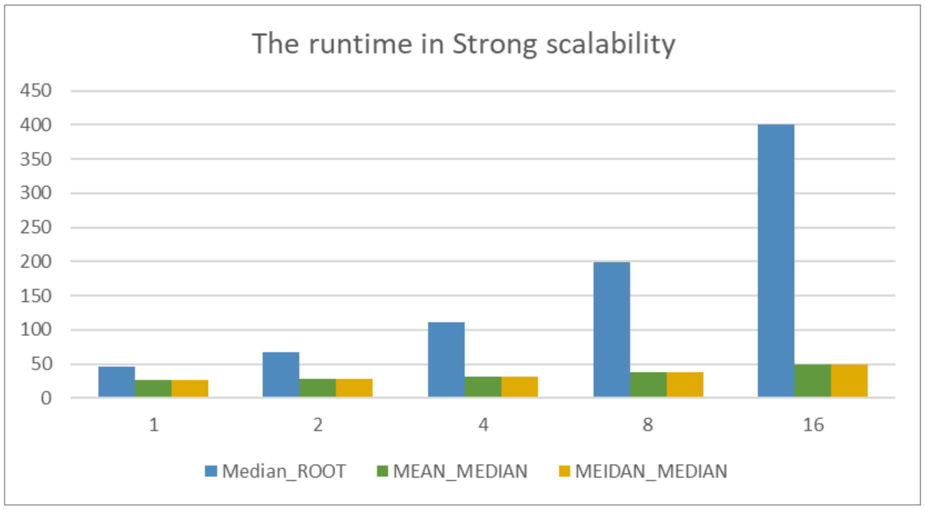

Figure 3: The Runtime of weak scalability

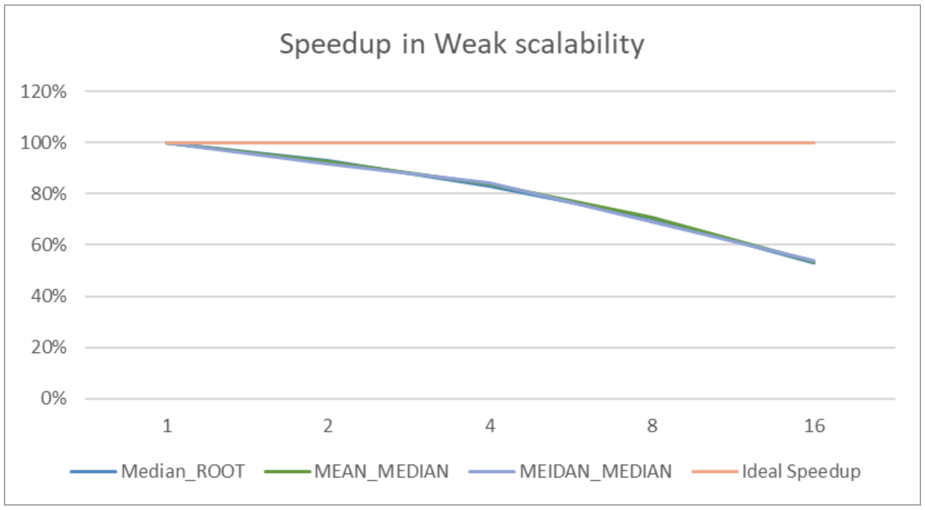

Figure 4: The Speedup of weak scalability

While the speedup values are comparable across strategies, Strategy 2 and 3 maintain better efficiency under strong scalability scenarios. For weak scalability, the difference is minimal. Strategy 1 (Median in Root) is the simplest but results in less balanced partitions and worse scaling at higher process counts.

4 Discussion

The parallel quicksort algorithm implemented in this assignment shows the benifits and drawback of parllelization when appllied to recursive sorting.
For strong scalability, the results indicate good strong scalability all three pivot strategies, we can see the exectution time decrease as we expected. With 8 processes, all strategies achieve more than 6 times, and the efficiency remains high below 8 processes. This result comes from the increasing cimmunication overhead and load imblance during recurive process spliliting, and strategy 1 is worse than other strategies, it can be reasonable due to the it is the simplest and direct strategy, this suggests that balanced pivot selection is important for good scalability.
For weak scalability, as the number of processes and the problem size grow, execution time increases, which also can be attributed to communication overhead and frequent memory allocation and data merging, but the performance of strategy 2 and 3 still better than strategy 1, due to producing more balanced patritions and reducing the depth of recursion.

5 Conclusion

From the result of performance in strong scalability and weak scalability, we can see that a more balanced pivot strategy such as $MEAN\_MEDIAN$ and $MEDIAN\_MEDIAN$ show good strong scalability, and when we tried it on $fixed\_size$ problems(weak scalability), it does not scale efficiently, the main factors are recursive communication overhead, even $MEDIAN\_ROOT$ can lead to minimal communication overhead, but in my experiment, balanced partitioning resulted in fewer recursion levels and more evely distributed workload, the additional communucation overhead was offset by the improved overall parallel efficiency.

6 Appendix

```

quicksort.c:
#include "quicksort.h"
#include "pivot.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <mpi.h>

#define NOPRINTING

int check_and_print(int *elements, int n, char *file_name){
    int sort_element=sorted_ascending(elements, n);
```

```c
    if(!sort_element){
        printf("Error:␣the␣elements␣are␣not␣sorted␣in␣ascending␣order.\n");
    }
    FILE *file=fopen(file_name, "w");
    if (!file) return -1;

    for(int i=0; i<n; i++){
      fprintf(file, "%d", elements[i]);
      if(i < n-1) fprintf(file, "␣");
    }
    fprintf(file, "\n");
    fclose(file);
    return 0;
}

int distribute_from_root(int *all_elements, int n, int **local_elements){
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int base_elements=n/size;
    int remainder=n%size;
    int *counts=malloc(size*sizeof(int));
    int *chunk_start=malloc(size*sizeof(int));

    int offset=0;
    for(int i=0; i<size; i++){
        counts[i]=base_elements+(i<remainder? 1:0);
        chunk_start[i]=offset;
        offset+=counts[i];
    }

    int local_n=counts[rank];
    *local_elements=malloc(local_n*sizeof(int));
    MPI_Scatterv(all_elements, counts, chunk_start, MPI_INT, *local_elements, local_n, MPI_INT, 0, MPI_COMM_WORLD);

    free(counts);
    free(chunk_start);

    return local_n;
}

void gather_on_root(int *all_elements, int *local_elements, int local_n) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int *counts = NULL;
    int *displs = NULL;

    if (rank == 0) {
        counts = malloc(size * sizeof(int));
        displs = malloc(size * sizeof(int));
    }

    MPI_Gather(&local_n, 1, MPI_INT, counts, 1, MPI_INT, 0,
       MPI_COMM_WORLD);

    if (rank == 0) {
        displs[0] = 0;
        for (int i = 1; i < size; i++) {
            displs[i] = displs[i-1] + counts[i-1];
        }
    }

    MPI_Gatherv(local_elements, local_n, MPI_INT,
                all_elements, counts, displs, MPI_INT,
                0, MPI_COMM_WORLD);

    if (rank == 0) {
        free(counts);
        free(displs);
    }
}

int global_sort(int **elements, int n, MPI_Comm comm, int
    pivot_strategy) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    if (size == 1) {
        return n;
    }

    // We need to check if size is even before proceeding
    if (size % 2 != 0) {
        if (rank == 0) {
            fprintf(stderr, "Error:␣Number␣of␣processes␣(%d)
               ␣must␣be␣even␣at␣all␣recursion␣levels.\n",
               size);
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

```

```c

     int actual_pivot_value;
    // We need the index of the first element greater than
        pivot_value
    int pivot_split_idx = select_pivot(pivot_strategy, *
        elements, n, comm, &actual_pivot_value);
    int half_size = size / 2;
    int new_color;
    int partner_rank;
    // here I spplit the data into send and keep two groups
    int send_n;
    int keep_n;
    int *send_ptr;
    int *keep_ptr;

    if (rank < half_size) {
        new_color = 0;
        partner_rank = rank + half_size;

        // Here the Elements <= pivot_value are kept,
            elements > pivot_value are sent
        keep_n = pivot_split_idx;
        send_n = n - pivot_split_idx;
        keep_ptr = *elements;
        send_ptr = *elements + pivot_split_idx;

    } else {
        new_color = 1;
        partner_rank = rank - half_size;

        // Elements > pivot_value are kept, elements <=
            pivot_value are sent
        keep_n = n - pivot_split_idx;
        send_n = pivot_split_idx;
        keep_ptr = *elements + pivot_split_idx;
        send_ptr = *elements;
    }

    // Here I exchange sizes first
    int recv_n;
    MPI_Sendrecv(&send_n, 1, MPI_INT, partner_rank, 0,
                 &recv_n, 1, MPI_INT, partner_rank, 0,
                 comm, MPI_STATUS_IGNORE);

    // Allocating receive buffer
    int* received_elements = (int*)malloc((recv_n > 0 ?
        recv_n : 1) * sizeof(int));
    if (recv_n > 0 && !received_elements) {
        fprintf(stderr, "Rank␣%d:␣Malloc␣for␣
            received_elements␣failed\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Here I exchange actual data
    MPI_Sendrecv(send_ptr, send_n, MPI_INT, partner_rank, 1,
                 received_elements, recv_n, MPI_INT,
                     partner_rank, 1,
                 comm, MPI_STATUS_IGNORE);

    // Begin merge
    int new_n = keep_n + recv_n;
    int *merged_elements = (int*)malloc((new_n > 0 ? new_n :
         1) * sizeof(int));
    if (new_n > 0 && !merged_elements) {
        fprintf(stderr, "Rank %d: Malloc for merged_elements
            failed\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    merge_ascending(keep_ptr, keep_n, received_elements,
        recv_n, merged_elements);


    if (received_elements) free(received_elements);

    free(*elements);
    *elements = merged_elements;

    // Spliting communicator
    MPI_Comm new_comm;
    MPI_Comm_split(comm, new_color, rank, &new_comm);
    // Recursive call
    int result_n = global_sort(elements, new_n, new_comm,
        pivot_strategy);
    MPI_Comm_free(&new_comm);
    return result_n;
}
void merge_ascending(int *v1, int n1, int *v2, int n2, int *
    result){
    int i = 0, j = 0, k = 0;
    while (i < n1 && j < n2) {
        if (v1[i] <= v2[j]) {
            result[k++] = v1[i++];
        } else {
            result[k++] = v2[j++];
        }
    }
    while (i < n1) {
        result[k++] = v1[i++];
    }
    while (j < n2) {
        result[k++] = v2[j++];
    }
}

int read_input(char *file_name, int **elements) {
    FILE *file = fopen(file_name, "r");
    if (!file) {
        perror("Couldn't open input file");
        return -1;
    }
    int num_values;
    if (fscanf(file, "%d", &num_values) != 1) {
        perror("Couldn't read element count from input file"
            );
        fclose(file);
        return -1;
    }
    *elements = malloc(num_values * sizeof(int));

    if (!(*elements) && num_values > 0) {
        perror("Memory allocation failed");
        fclose(file);
        return -1;
    }

    for (int i = 0; i < num_values; i++) {
        if (fscanf(file, "%d", &((*elements)[i])) != 1) {
            perror("Couldn't read elements from input file")
                ;
            free(*elements);
            *elements = NULL;
            fclose(file);
            return -1;
        }
    }
    fclose(file);
    return num_values;
}

int sorted_ascending(int *elements, int n) {
    for (int i = 1; i < n; i++) {
        if (elements[i] < elements[i-1]) {
            printf("Error at index %d: %d > %d\n", i - 1,
                elements[i - 1], elements[i]);
            return 0;
        }
    }
    return 1;
}

```

```c


void swap(int *e1, int *e2) {
    int tmp = *e1;
    *e1 = *e2;
    *e2 = tmp;
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc != 4) {
        if (rank == 0)
            printf("Usage: %s <input_file> <output_file> <
                pivot_strategy>\n", argv[0]);
        MPI_Finalize();
        return 1;
    }

    char *input_name = argv[1];
    char *output_name = argv[2];
    int pivot_strategy = atoi(argv[3]);

    int* all_elements = NULL;
    int* local_elements = NULL;
    int total_n = 0;

    double overall_start_time = MPI_Wtime();

    if (rank == 0) {
        total_n = read_input(input_name, &all_elements);
        if (total_n <= 0) {
            if (all_elements) free(all_elements);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Bcast(&total_n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (total_n <= 0) {
        MPI_Finalize();
        return 1;
    }

    double distrubution_start_time = MPI_Wtime();
    int local_n = distribute_from_root(all_elements, total_n
        , &local_elements);
    double distrubution_end_time = MPI_Wtime();
    double current_distr_time = distrubution_end_time -
        distrubution_start_time;
    double local_serial_sort_start_time = MPI_Wtime();
    if (local_n > 1) {
        qsort(local_elements, local_n, sizeof(int), compare)
            ; //        compare
    }
    double local_serial_sort_end_time = MPI_Wtime();
    double current_process_serial_time =
        local_serial_sort_end_time -
        local_serial_sort_start_time;
    MPI_Barrier ( MPI_COMM_WORLD );
    double global_sort_start_time = MPI_Wtime();
    int sorted_n = global_sort(&local_elements, local_n,
        MPI_COMM_WORLD, pivot_strategy);
    double global_sort_end_time = MPI_Wtime();
    double current_process_global_sort_time =
        global_sort_end_time - global_sort_start_time;
    MPI_Barrier ( MPI_COMM_WORLD );
    if (rank == 0) {
        free(all_elements);
    }
    all_elements = NULL;
    if (rank == 0) {
        all_elements = malloc(total_n * sizeof(int));
    }
    double gather_start_time=MPI_Wtime();
    gather_on_root(all_elements, local_elements, sorted_n);
    double gather_end_time=MPI_Wtime();
    double current_process_gather_time = gather_end_time -
        gather_start_time;

    double overall_end_time = MPI_Wtime();
    double current_process_overall_time = overall_end_time -
         overall_start_time;

    if (local_elements) free(local_elements);

    double max_distr_time;
    MPI_Reduce(&current_distr_time, &max_distr_time, 1,
        MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);


    double max_gather_time;
    MPI_Reduce(&current_process_gather_time, &
        max_gather_time, 1, MPI_DOUBLE, MPI_MAX, 0,
        MPI_COMM_WORLD);


    double max_overall_time;
    MPI_Reduce(&current_process_overall_time, &max_overall_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    double max_serial_time;
    MPI_Reduce(&current_process_serial_time, &max_serial_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    double max_global_sort_time;
    MPI_Reduce(&current_process_global_sort_time, &max_global_sort_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);


    if (rank == 0) {
        printf("Initial␣Local␣Serial␣Sort␣(Max):␣%f␣seconds.\n", max_serial_time);
        printf("distribution␣time␣(Max):␣%f␣seconds.\n", max_distr_time);
        printf("Parallel␣Quicksort␣Phase␣(Max):␣%f␣seconds.\n", max_global_sort_time);
        printf("gather␣on␣root␣time␣(Max):␣%f␣seconds.\n", max_gather_time);
        printf("Total␣Execution␣Time␣(Max):␣%f␣seconds.\n", max_overall_time);

        check_and_print(all_elements, total_n, output_name);
        free(all_elements);
    }


    MPI_Finalize();
    return 0;
}
pivot.c:
#include "pivot.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <mpi.h>

int compare(const void* v1, const void* v2){
    return (*(int*)v1-*(int*)v2);
}

int get_median(int* elements, int n) {
    if (n == 0) return 0;
    if (n % 2 == 0) {
        return elements[n / 2 - 1];
    } else {
        return elements[n / 2];
    }
}

int get_larger_index(int *elements, int n, int val) {
    for (int i = 0; i < n; i++) {
        if (elements[i] > val) return i;
    }
    return n;
}

int select_pivot_median_root(int *elements, int n, MPI_Comm
    comm, int *pivot_value) {
    int rank;
    MPI_Comm_rank(comm, &rank);
    int pivot_val = 0;

    if (rank == 0) {
        if (n > 0) {
            pivot_val = get_median(elements, n);
        }
    }
    MPI_Bcast(&pivot_val, 1, MPI_INT, 0, comm);

    *pivot_value = pivot_val;
    return get_larger_index(elements, n, pivot_val);
}

int select_pivot_mean_median(int *elements, int n, MPI_Comm
    comm, int *pivot_value) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    int local_median = 0;
    int has_elements = (n > 0) ? 1 : 0;
    if (n > 0) {
        local_median = get_median(elements, n);
    }

    int* all_medians = NULL;
    int* has_elements_array = NULL;
    if (rank == 0) {
        all_medians = (int*)malloc(size * sizeof(int));
        has_elements_array = (int*)malloc(size * sizeof(int)
            );
        if (!all_medians || !has_elements_array) {
```

```c
            perror("Rank 0: Malloc failed in select_pivot_mean_median");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Gather(&local_median, 1, MPI_INT, all_medians, 1, MPI_INT, 0, comm);
    MPI_Gather(&has_elements, 1, MPI_INT, has_elements_array, 1, MPI_INT, 0, comm);

    int pivot_val = 0;
    if (rank == 0) {
        long long sum = 0;
        int count = 0;
        for (int i = 0; i < size; i++) {
            if (has_elements_array[i]) {
                sum += all_medians[i];
                count++;
            }
        }
        pivot_val = (count > 0) ? (int)(sum / count) : 0;
        free(all_medians);
        free(has_elements_array);
    }
    MPI_Bcast(&pivot_val, 1, MPI_INT, 0, comm);

    *pivot_value = pivot_val;
    return get_larger_index(elements, n, pivot_val);
}

int select_pivot_median_median(int *elements, int n, MPI_Comm comm, int *pivot_value) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    int local_median = 0;
    int has_elements = (n > 0) ? 1 : 0;
    if (n > 0) {
        local_median = get_median(elements, n);
    }

    int* all_medians = NULL;
    int* has_elements_array = NULL;
    if (rank == 0) {
        all_medians = (int*)malloc(size * sizeof(int));
        has_elements_array = (int*)malloc(size * sizeof(int));
        if (!all_medians || !has_elements_array) {
            perror("Rank 0: Malloc failed in select_pivot_median_median");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Gather(&local_median, 1, MPI_INT, all_medians, 1, MPI_INT, 0, comm);
    MPI_Gather(&has_elements, 1, MPI_INT, has_elements_array, 1, MPI_INT, 0, comm);

    int pivot_val = 0;
    if (rank == 0) {
        int* valid_medians = (int*)malloc(size * sizeof(int));
        if (!valid_medians) {
            perror("Rank 0: Malloc failed for valid_medians");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        int valid_count = 0;

        for (int i = 0; i < size; i++) {
            if (has_elements_array[i]) {
                valid_medians[valid_count++] = all_medians[i];
            }
        }

        if (valid_count > 0) {
            qsort(valid_medians, valid_count, sizeof(int), compare);
            pivot_val = get_median(valid_medians, valid_count);
        }

        free(all_medians);
        free(has_elements_array);
        free(valid_medians);
    }
    MPI_Bcast(&pivot_val, 1, MPI_INT, 0, comm);

    *pivot_value = pivot_val;
    return get_larger_index(elements, n, pivot_val);
}

int select_pivot_smallest_root(int *elements, int n, MPI_Comm comm, int *pivot_value) {
    int rank;
    MPI_Comm_rank(comm, &rank);
    int pivot_val = 0;

    if (rank == 0 && n > 0) {
        pivot_val = elements[0];
    }
    MPI_Bcast(&pivot_val, 1, MPI_INT, 0, comm);

    *pivot_value = pivot_val;
    return get_larger_index(elements, n, pivot_val);
}

int select_pivot(int pivot_strategy, int *elements, int n,
    MPI_Comm communicator, int *pivot_value) {
    int pivot_index_result = 0;

    switch (pivot_strategy) {
        case MEDIAN_ROOT:
            pivot_index_result = select_pivot_median_root(
                elements, n, communicator, pivot_value);
            break;
        case MEAN_MEDIAN:
            pivot_index_result = select_pivot_mean_median(
                elements, n, communicator, pivot_value);
            break;
        case MEDIAN_MEDIAN:
            pivot_index_result = select_pivot_median_median(
                elements, n, communicator, pivot_value);
            break;
        default: // SMALL_ROOT
            pivot_index_result = select_pivot_smallest_root(
                elements, n, communicator, pivot_value);
            break;
    }
    return pivot_index_result;
}
```