**PDC Sorting**

**1. Bubble Sort**

- **Concept**:
    - Repeatedly compares adjacent elements in the array and swaps them if they are in the wrong order.
    - This process is repeated until no more swaps are needed.
- **Parallelization with MPI**:
    - Divide the array into smaller chunks for different processes.
    - Each process performs bubble sort on its chunk.
    - After sorting, the results can be merged by comparing the smallest (or largest) elements from each process.

**2. Insertion Sort**

- **Concept**:
    - Starts with an empty sorted portion and picks each element from the unsorted portion, inserting it into the correct position in the sorted portion.
    - Repeats this until all elements are sorted.
- **Parallelization with MPI**:
    - Each process works on a different chunk of the array, sorting it independently using insertion sort.
    - After sorting, results are gathered and merged. MPI can handle this using MPI_Gatherv and then a final merge step.

**3. Merge Sort**

- **Concept**:
    - A divide-and-conquer algorithm that splits the array into two halves, recursively sorts each half, and then merges them back together in sorted order.
- **Parallelization with MPI**:

- Divide the array into halves and distribute them to different processes.
- Each process sorts its subarray and then uses MPI to merge the results. The merging process can also be parallelized.

**4. Quick Sort**

- **Concept**:
  - Selects a pivot element and partitions the array such that elements smaller than the pivot go to one side and elements greater than the pivot go to the other side.
  - This is done recursively for each partition until the entire array is sorted.
- **Parallelization with MPI**:
  - The array is divided into subarrays, and each subarray is handled by a different process.
  - Each process performs quick sort on its subarray.
  - The results are combined using MPI's MPI_Gather or MPI_Reduce, and the final sorted array is obtained.

**5. Shell Sort**

- **Concept**:
  - An extension of insertion sort that sorts elements far apart first, gradually reducing the gap between elements.
  - This reduces the number of shifts needed compared to regular insertion sort.
- **Parallelization with MPI**:
  - The array can be divided into subarrays, and each process sorts its chunk of the array using shell sort.
  - After sorting, the results are gathered, and merging is done in parallel.

**MPI Parallelization Concepts:**

- **Data Distribution**:
  - In MPI parallelization, the array is divided into smaller chunks, and each chunk is sent to a different process using MPI_Scatter or MPI_Scatterv.

- **Local Sorting**:
    - Each process independently sorts its chunk using the chosen sorting algorithm.
- **Gathering Results**:
    - After sorting, the results from all processes are gathered back together into a single array using MPI_Gather or MPI_Gatherv.
- **Final Merging**:
    - In algorithms like merge sort or quick sort, the sorted subarrays are merged in parallel to achieve the final sorted array.

## MPI commands

**1. MPI_Init**

- **Purpose**: Initializes the MPI environment.
- **Usage**: Must be called at the beginning of any MPI program. It sets up the necessary resources for communication.
- **Syntax**: MPI_Init(&argc, &argv);

**2. MPI_Comm_rank**

- **Purpose**: Determines the rank (ID) of the calling process within a communicator.
- **Usage**: Each process can determine its unique identifier in the group of processes.
- **Syntax**: MPI_Comm_rank(MPI_COMM_WORLD, &rank);

**3. MPI_Comm_size**

- **Purpose**: Determines the total number of processes in a communicator.
- **Usage**: Helps in identifying how many processes are involved in the parallel execution.
- **Syntax**: MPI_Comm_size(MPI_COMM_WORLD, &size);

**4. MPI_Send**

- **Purpose**: Sends data from one process to another.
- **Usage**: Used to send a message (data) from one process to another, with a specified rank and communicator.
- **Syntax**: MPI_Send(buffer, count, datatype, dest, tag, MPI_COMM_WORLD);

**5. MPI_Recv**

- **Purpose**: Receives data sent by another process.
- **Usage**: Used to receive data from a specific process within a communicator.
- **Syntax**: MPI_Recv(buffer, count, datatype, source, tag, MPI_COMM_WORLD, &status);

**6. MPI_Bcast**

- **Purpose**: Broadcasts a message from one process to all other processes.
- **Usage**: Typically used for distributing data to all processes in a communicator.
- **Syntax**: MPI_Bcast(buffer, count, datatype, root, MPI_COMM_WORLD);

**7. MPI_Scatter**

- **Purpose**: Distributes chunks of an array from one process to all other processes.
- **Usage**: One process (the root) distributes different portions of data to all processes in the communicator.
- **Syntax**: MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, MPI_COMM_WORLD);

**8. MPI_Gather**

- **Purpose**: Gathers data from all processes to one process.
- **Usage**: Collects data from all processes and stores it in the root process.
- **Syntax**: MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, MPI_COMM_WORLD);

**9. MPI_Scatterv**

- **Purpose**: Distributes variable-sized chunks of data from one process to others.

- **Usage**: A more flexible version of MPI_Scatter that allows sending data with different counts to each process.
- **Syntax**: MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, MPI_COMM_WORLD);

**10. MPI_Gatherv**

- **Purpose**: Gathers variable-sized chunks of data from multiple processes to one process.
- **Usage**: A more flexible version of MPI_Gather that allows receiving data with different counts from each process.
- **Syntax**: MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, MPI_COMM_WORLD);

**11. MPI_Barrier**

- **Purpose**: Synchronizes all processes in a communicator.
- **Usage**: Ensures that all processes reach this point before proceeding.
- **Syntax**: MPI_Barrier(MPI_COMM_WORLD);

**12. MPI_Finalize**

- **Purpose**: Terminates the MPI environment.
- **Usage**: Should be called at the end of an MPI program to clean up resources.
- **Syntax**: MPI_Finalize();

**13. MPI_Reduce**

- **Purpose**: Performs a reduction operation (e.g., sum, max, min) across all processes and sends the result to one process.
- **Usage**: Typically used for performing operations like summing or finding the max across all processes.
- **Syntax**: MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, MPI_COMM_WORLD);

**14. MPI_Allreduce**

- **Purpose**: Similar to MPI_Reduce, but the result is returned to all processes, not just the root.

- **Usage**: Performs a reduction across all processes and distributes the result to every process.
- **Syntax**: MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, MPI_COMM_WORLD);

**15. MPI_Wait**

- **Purpose**: Waits for a non-blocking operation to complete.
- **Usage**: Used when non-blocking communication (e.g., MPI_Isend) is used to synchronize the process after sending/receiving.
- **Syntax**: MPI_Wait(&request, &status);