Analysis of Word and Character Counting on Parallel Containers

**Complexity analysis of algorithms:**The complexity analysis of the key algorithms used in the program focusing on time complexity with respect to file size and the number of processes is mentioned as follows:

**Character Counting**:

The program uses an array to store the frequency of characters within a specified ASCII range (32-126). Each character from the file chunk is processed in constant time, making the complexity of this phase:

**Time Complexity**: O(n), where ‘n’ is the size of the input chunk processed by a given process.

**Word Counting**:

The words are extracted by splitting the file chunk on non-alphanumeric characters. Inserting or updating word frequencies in an unordered\_map (hash table) takes O(1) time on average for each word.

**Time Complexity**: O(w), where ‘w’ is the number of words in the chunk. Since ‘w’ is proportional to the chunk size, the complexity is O(n) overall.

**Communication (MPI Reduce/Gather)**:

**Character Count Reduction**: After local counting, the program uses MPI\_Reduce to combine the results from all processes. The time complexity for the reduction step is proportional to the number of processes ‘p’.

**Time Complexity**: O(p), where ‘p’ is the number of processes.

**Word Count Aggregation**: For word counting, the word data from all processes are flattened into vectors and gathered at the root process using MPI\_Gatherv. The aggregation and unflattening of the data involve:

**Time Complexity**: O(w\*p), where ‘w’ is the number of words in the largest chunk and ‘p’ is the number of processes.

**Sorting**:

Sorting the top 10 characters and words is done using std::sort, which has a time complexity of O(nlogn).

**Time Complexity**: O(clogc) for characters, and O(wlogw) for words, where ‘c’ is the number of unique characters and ‘w’ is the number of unique words.

**Procedures used in the program:**

The program is split into several distinct steps and functions:

**File Chunking**:

The input file is divided into chunks, each assigned to one process. The last process may handle slightly more data if the file size is not perfectly divisible by the number of processes.

**Character and Word Counting**:

Each process counts the frequency of characters and words in its chunk. Words are identified as consecutive sequences of alphanumeric characters.

Boundary words that are split between chunks are handled by communicating the last word of each chunk to the next process.

**Communication Between Processes**:

The character counts are combined using MPI\_Reduce, which reduces the character frequencies from all processes into the root process.

The word counts are flattened into vectors and sent to the root process using MPI\_Gatherv. These vectors are then unflattened and aggregated to form a global word count.

**Sorting and Displaying Results**:

The root process sorts the characters and words based on their frequencies and outputs the top 10 most frequent characters and words.

**Executions for sample tests:**

These are the output results that matches with the given sample test output files of size 1KB and 10KB.

These are for native processes 1 and 8 for test1.txt and test2.txt.

Outputs for Test1.txt:

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

Outputs for Test2.txt:

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**Average Execution times:**

|  |  |
| --- | --- |
| **Case 1** | **Case 2** |
| **Native Processes** | **Docker Containers** |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Files** | **1** | **2** | **4** | **6** | **8** | **1** | **2** | **4** | **6** | **8** |
| **1MB** | 0.5126 | 0.4899 | 0.4887 | 0.5042 | 0.5028 | 0.5032 | 0.4753 | 0.4568 | 0.4553 | 0.4589 |
| **10MB** | 4.6152 | 4.3666 | 4.2141 | 4.1835 | 4.1479 | 4.7457 | 4.0412 | 3.9854 | 3.9621 | 3.9532 |
| **100MB** | 48.409 | 46.874 | 45.854 | 45.697 | 44.381 | 49.870 | 46.532 | 44.621 | 44.893 | 43.908 |
| **1GB** | 497.41 | 483.13 | 465.73 | 457.83 | 413.79 | 499.23 | 486.78 | 466.12 | 459.32 | 412.09 |

**Observations**:

**Native MPI Processes**: Execution times remain relatively constant as the number of processes increases, indicating that parallelization provides limited benefits for very small file sizes (1MB).

**Docker Containers**: Docker containers slightly outperform native MPI processes across all process counts. This could be attributed to Docker’s isolated environment and efficient resource sharing for small workloads.

**Conclusion**: For small file sizes (1MB), Docker provides a marginal performance advantage. However, the difference is minor due to the minimal computational workload.

For larger files:

**Native MPI Processes**: Execution times improve steadily as the number of processes increases, showing a clear benefit from parallelization as the workload grows. However, the reduction in time plateaus beyond 6 processes.

**Docker Containers**: Docker consistently outperforms native MPI in this case. As the number of containers increases, the performance improves and shows slightly better scalability than native MPI processes.