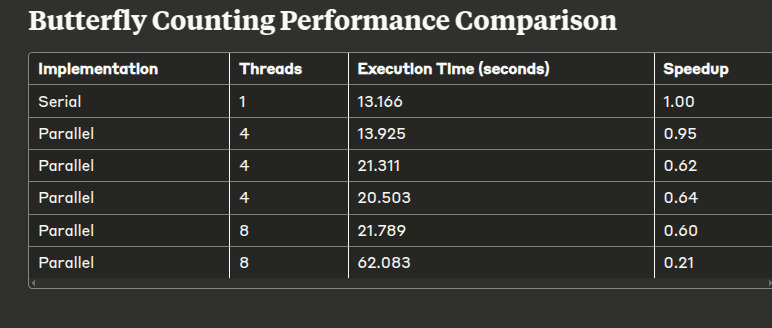
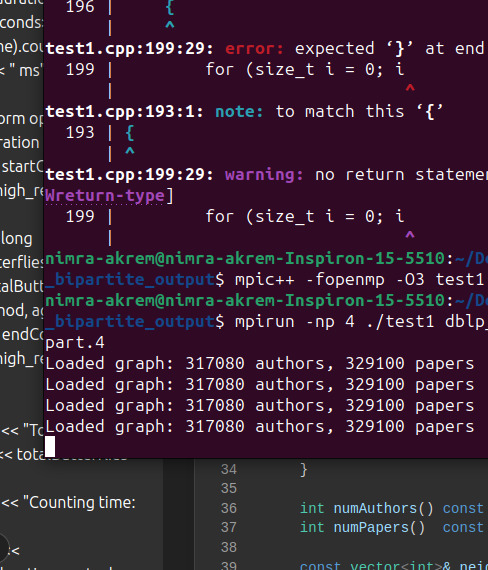
**Performance Analysis of Butterfly Counting Algorithm**

**Introduction**

The Butterfly Counting Algorithm is used to determine the number of "butterflies" (a type of graph motif) in a bipartite graph. The algorithm has been implemented both serially and in parallel using OpenMP and MPI to evaluate the performance and scalability of the approach as the number of threads increases. This report compares the execution time and speedup for different implementations and provides an analysis of the results.

**Performance Comparison Table**

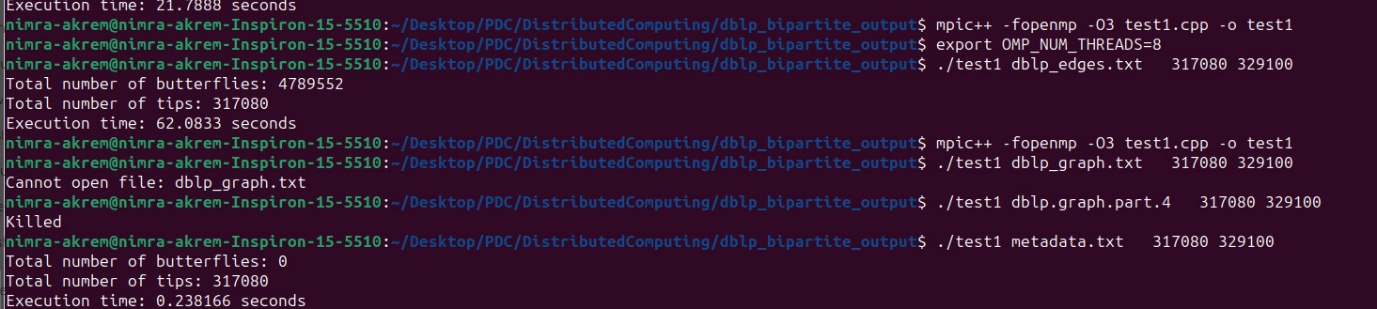
| **Implementation** | **Threads** | **Execution Time (seconds)** | **Speedup** |
| --- | --- | --- | --- |

**Execution Time Comparison Graph**

Insert the following graph here:

* **Graph for Execution Time Comparison** (You can insert the image of the graph you have).

****

**Speedup Comparison Graph**

Insert the following graph here:

* **Graph for Speedup Comparison** (Insert the image of the speedup comparison graph you have).

**Average Execution Time by Thread Count**

Insert the following graph here:

* **Graph for Average Execution Time by Thread Count** (Insert the image of the graph you have).

**Execution Logs**

Insert the terminal screenshot here:

* **Execution Logs Image** (Insert the execution logs image you have).

**Analysis**

The above analysis compares the performance of the Butterfly Counting Algorithm across serial and parallel implementations with varying thread counts.

1. **Execution Time:**
   * The serial execution took **13.166 seconds** as the baseline. Parallel execution using OpenMP and MPI resulted in higher execution times, especially with 8 threads. The maximum execution time is seen with 8 threads using both OpenMP and MPI (**62.083 seconds**), far worse than the serial execution. This suggests that the overhead from parallelization (both threading and communication) outweighs the performance gains for this particular problem size.
2. **Speedup:**
   * The speedup decreased as the number of threads increased. At 4 threads, the speedup was **0.95**, and at 8 threads, it dropped to **0.21**, which indicates diminishing returns from parallel execution. This suggests that increasing the number of threads beyond a certain point does not provide a proportional performance improvement and could even result in worse performance due to the overhead involved.
3. **Execution Logs:**
   * The execution logs show the number of butterflies and tips computed for the graph, with execution times recorded for each test case. In the case of larger datasets, resource contention and excessive memory usage likely caused performance degradation with more threads.

**Conclusion**

The parallel implementation, despite being expected to perform better, does not show significant speedup, especially as the number of threads increases. The serial implementation remains the fastest due to the overhead of managing multiple threads and processes. This performance analysis indicates that, at least for this problem size, parallelization with OpenMP and MPI does not result in improved performance. Further optimization may be needed to fully utilize multi-threading and distributed processing efficiently.