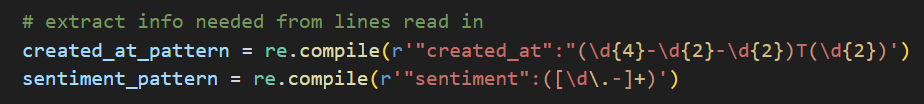
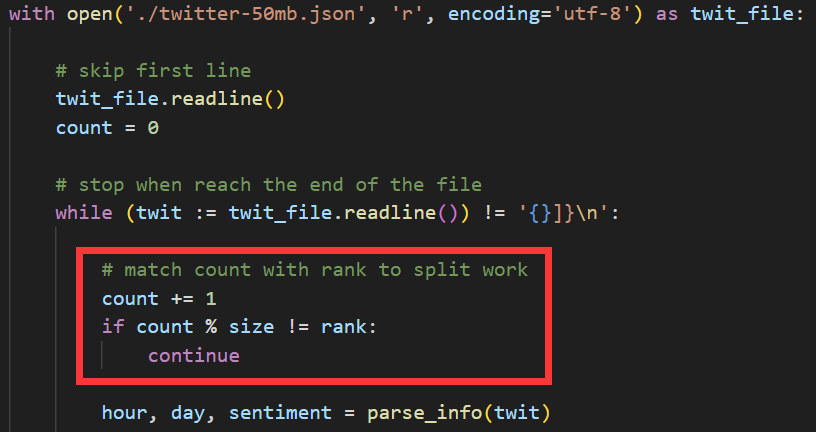
**Implementation approaches**

Our group incorporates a standard structure provided by mpi4py package for initializing MPI communication, and uses built-in comm.gather() method to achieve communication and collection of results at the end, simplifying the implementation of data aggregation across multiple processes. In terms of data processing and storage, we used the Counter collection for statistics related to activity levels, and the Defaultdict collection for statistics related to sentiment to accelerate the processing speed.

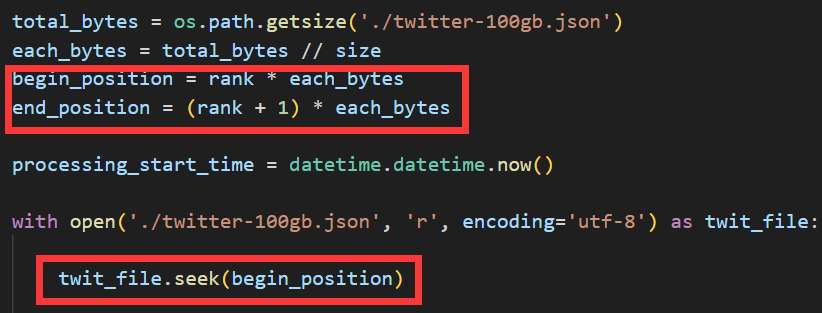
Additionally, although the provided data is in JSON format, considering that the information we require occupies only a small part of each line of JSON string, we finally chose to use regular expression matching as a method to quickly parse the necessary information.



For parallelized part, our group has implemented two approaches. The first straightforward and intuitive approach, involves each core continuously reads lines from the file, assigning a number to each line read in, and then takes turns to process a line of information in a round-robin fashion based on a modulo operation.

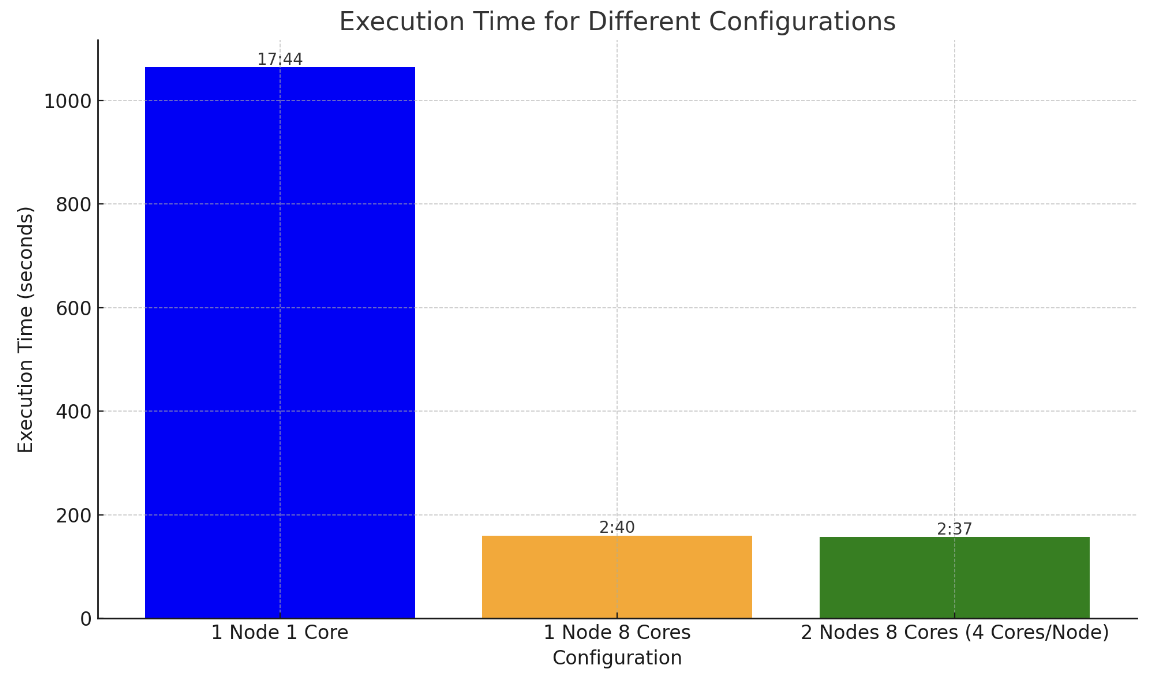


Although this approach is accurate, with each core only needs to parse the lines assigned to it, it still presents bottleneck in I/O part, since each core still needs to read the entire file line by line. Consequently, the improvement from multi-core parallelism was not as significant as expected. Therefore, we subsequently implemented a second approach. In the second approach, our team approximately divided the file into segments with equal byte size for each core and used the file.seek() method with the assigned pointer offset to directly jump to the corresponding section. This allowed each core to only read its allocated portion, reducing unnecessary I/O operations and thereby enhancing the speedup ratio.

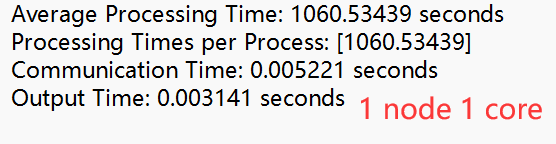
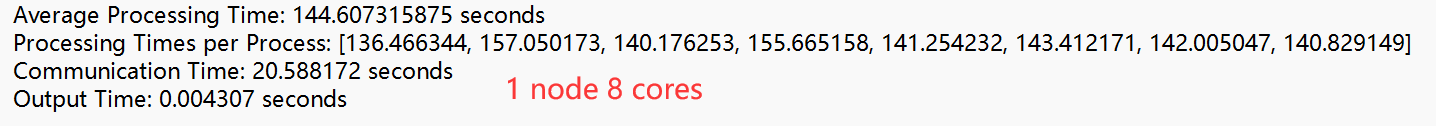


We observed slight difference in the results obtained using the second approach compared to the first one, which we attribute to the imprecise handling of the work division boundaries in our implementation. Specifically, the code used a method of marking the pointer offset with position+len(current\_line) to estimate the start position for each core's next read. However, due to the presence of emoji symbols in the text, the simple use of the len() function in this context can be inaccurate, as it underestimates the actual length, leading each core to process a bit more than its allocated section. This overlap in processing is what slightly altered the results. While we are aware of the file.tell() method as a method to achieve greater accuracy, our testing indicated that the performance overhead of file.tell() was too significant, negating the benefits of parallel I/O. Considering that the goal of this project is not the precision of statistics but rather the exploration of parallelized applications, we finally decided to overlook this issue, which we considered not critical.

**Result**



The chart above displays the results of our project, showing that our team achieved an approximate speedup ratio of 6.5 through parallelization with 8 cores. Additionally, our group added timers to the code to calculate the time consumed by each part.

It can be observed that, if we only consider the data processing part, our code has achieved an approximate speedup ratio of 7.4, which is very close to the theoretical maximum of 8 calculated by Amdahl's Law. Besides, when comparing the speedup ratio achieved on a 50MB file processed locally, we observed a higher speedup ratio as we scaled up the problem size. This observation aligns with Gustafson-Barsis’s Law. However, this parallelization also introduced additional overhead, namely the time consumed in the communication part.

For optimizing the communication time, our group considered that there might be more suitable methods than comm.gather(), such as comm.reduce() or hierarchical merging. However, considering that reduce() method is not compatible with our use of dictionary collections and the performance of merging was not ideal, coupled with the fact that communication only accounts for a small portion of the final result, we ultimately did not change this part of the code.

Within the provided environment, we also found that the configuration of 1 node with 8 cores and 2 nodes with 8 cores had minimal impact on the results. We speculate that the primary reason for this is the SPARTAN platform's excellent load balancing across each node and core, which minimizes the differences between each node and core. Furthermore, since the communication time itself constitutes a small proportion of the total for this problem, the impact of cross-node communication is very minimal.