Contents

1. Introduction

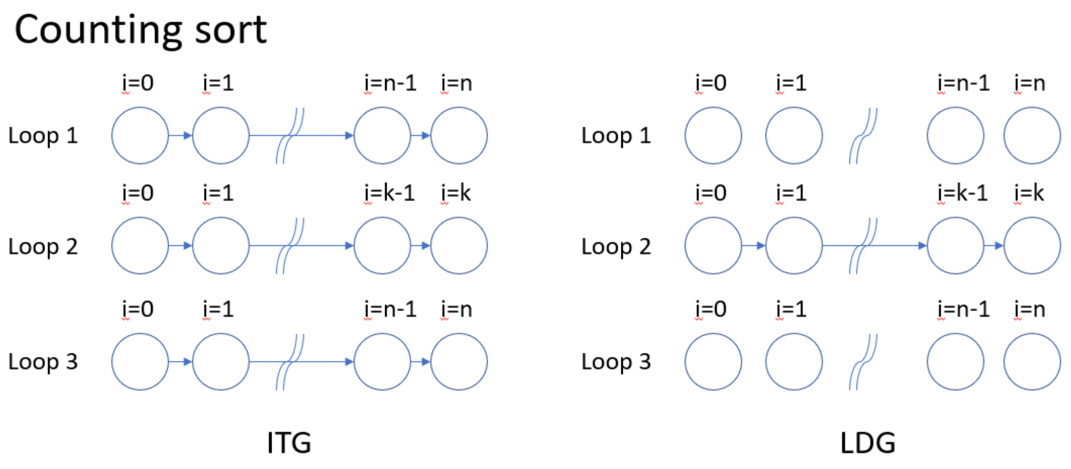
In this project, I utilized three different methods to parallelize the pre-processing stage for building the graph Compressed Sparse Row (CSR), which is predominated by the sorting step. I paralleled count sort with radix sort and OpenMP, MPI, and Hybrid(OpenMP + MPI).

2. Radix sort and Count sort

```
function Counting Sort(input, k)

    count ← array of k + 1 zeros
    output ← array of the same length as input

    loop 1: for i = 0 to length(input) - 1 do
        j = key(input[i])
        count[j] += 1

    loop 2: for i = 1 to k do
        count[i] += count[i - 1]

    loop 3: for i = length(input) - 1 downto 0 do
        j = key(input[i])
        count[j] -= 1
        output[count[j]] = input[i]

    return output
```

Graph 1: pseudocode of count sort

The time complexity of the counting sort is O(n+k). The total space usage of counting sort is O(n+k).

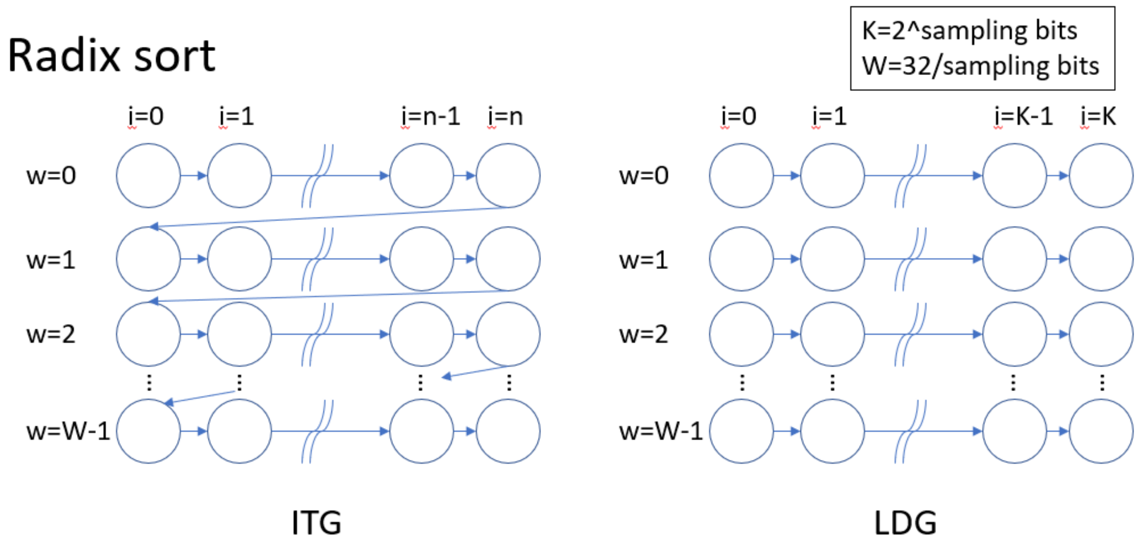


Graph 2: ITG and LDG of counting sort.

```
function Radix Sort(input, w)

    count ← array of k + 1 zeros
    output ← array of the same length as input
    W ← data width divide by sampling bits

    Loop 1: for w = 0 to W do

       Loop 2: for i = 0 to length(input) - 1 do
               j = key(input[i])
               count[j] += 1

       loop 3: for i = 1 to 2^ radix bits do
               count[i] += count[i - 1]

       loop 4: for i = length(input) - 1 downto 0 do
               j = key(input[i])
               count[j] -= 1
               output[count[j]] = input[i]

    return output
```

Graph 3: pseudocode of radix sort

The time complexity of the counting sort is O(Wn). The total space usage of counting sort is O(n+k).



Graph 4: ITG and LDG of radix sort.

In this project, k equals the size of the dataset. When the dataset is large, O(n+k) is larger than O(Wn). Therefore, counting sort will be slower than radix sort. However, if the dataset is very small that O(n+k) is smaller than O(Wn), the counting sort will be faster than the radix sort. Since most of the datasets are extremely large, radix sort is faster than counting sort in most cases.

In addition to speed, memory efficiency is also important. Radix sorting uses fewer keys in the count vector with a bit of increase in array length. For example, the original counting sort has keys from 0 ~ maximum dataset size, which means the length of the count vector is huge. However, radix sort with 8-bit sampling bits has keys from 0~255, which means the count vector's length is 256. If we use shared memory for parallelizing, the length of the count vector will be maximum threads multiple with 256. If we parallelize sorting with 16 threads using shared memory, the length of the count vector is only 16*256=4096, which is still very small.

3. Experiment

I run all the experiments on an AMD rome node. A Rome node has 4 NUMA nodes, one socket, 16 cores per socket, and two threads per core.
I use "mpirun -bind-to numa ./…." to execute all the mpi and hybrid experiments.

3.1 OpenMP version

In this experiment, I parallelized radix sort using OpenMP with different threads and benchmarks.

datasets	gplus	liveJournal	orkut
Count sort(baseline)	27.271876	0.212573	0.750137
Radix sort + OpenMP (16 threads)	4.213924	0.112134	0.384033
Radix sort + OpenMP (8 threads)	5.913761	0.169967	0.561030
Radix sort + OpenMP (4 threads)	12.320224	0.313859	1.056668
Radix sort + OpenMP (2 threads)	21.926819	0.579610	2.089701
Radix sort + OpenMP (1 threads)	27.800297	1.114322	3.856606

Table 1: Comparison between baseline and all OpenMP versions using various datasets.

datasets	gplus	speedup
Count sort(baseline)	27.271876	1
Radix sort + OpenMP (16 threads)	4.213924	6.477

Table 2: Comparison between baseline and the fastest version using gplus dataset.

datasets	orkut	speedup
Count sort(baseline)	0.750137	1
Radix sort + OpenMP (16 threads)	0.384033	1.953

Table 3: Comparison between baseline and the fastest version using orkut dataset.

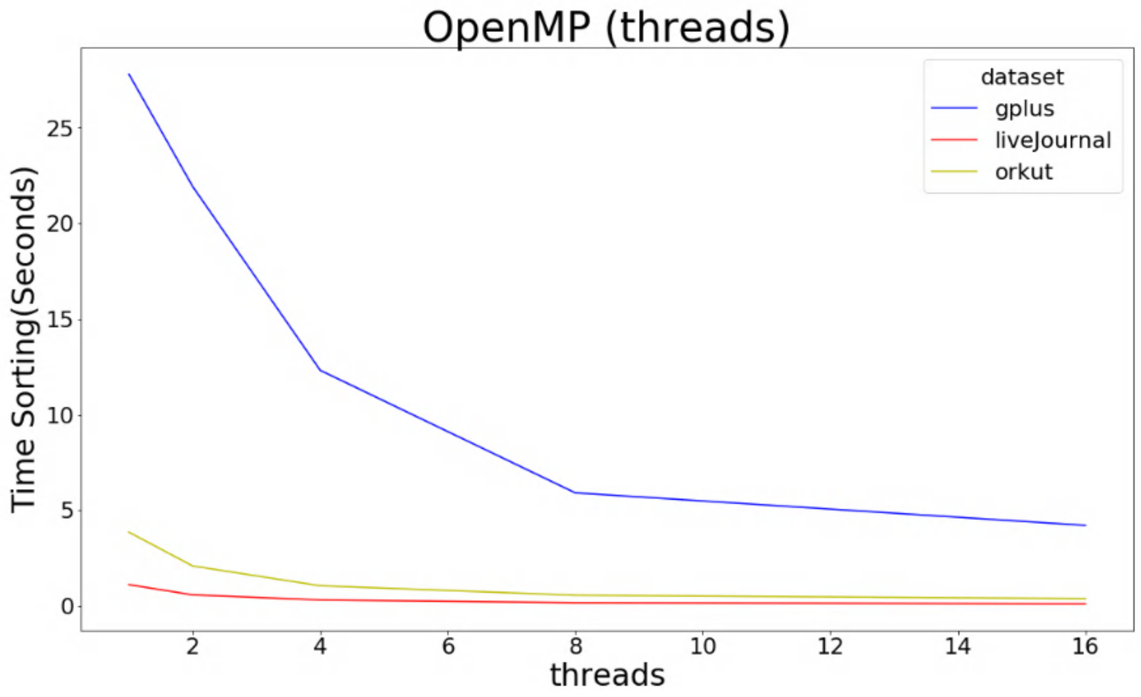
datasets	liveJournal	speedup
Count sort(baseline)	0.212573	1
Radix sort + OpenMP (16 threads)	0.112134	1.895

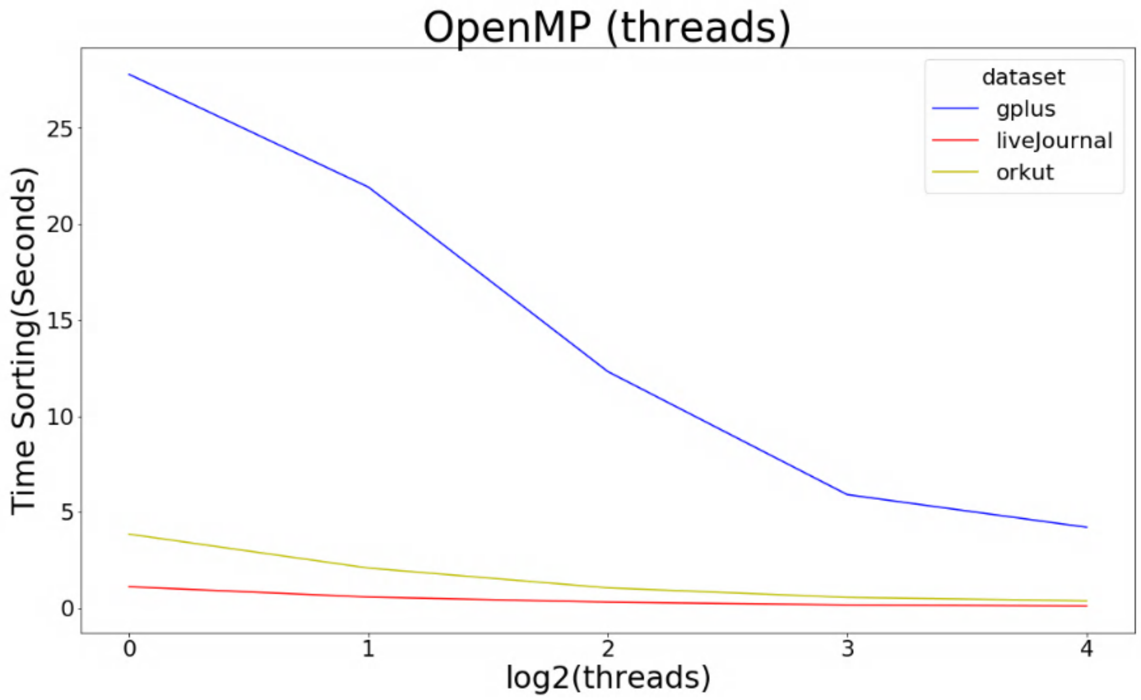
Table 4: Comparison between baseline and the fastest version using liveJournal dataset.

Overall execution time = paralleled radix sorting + OpenMP overhead

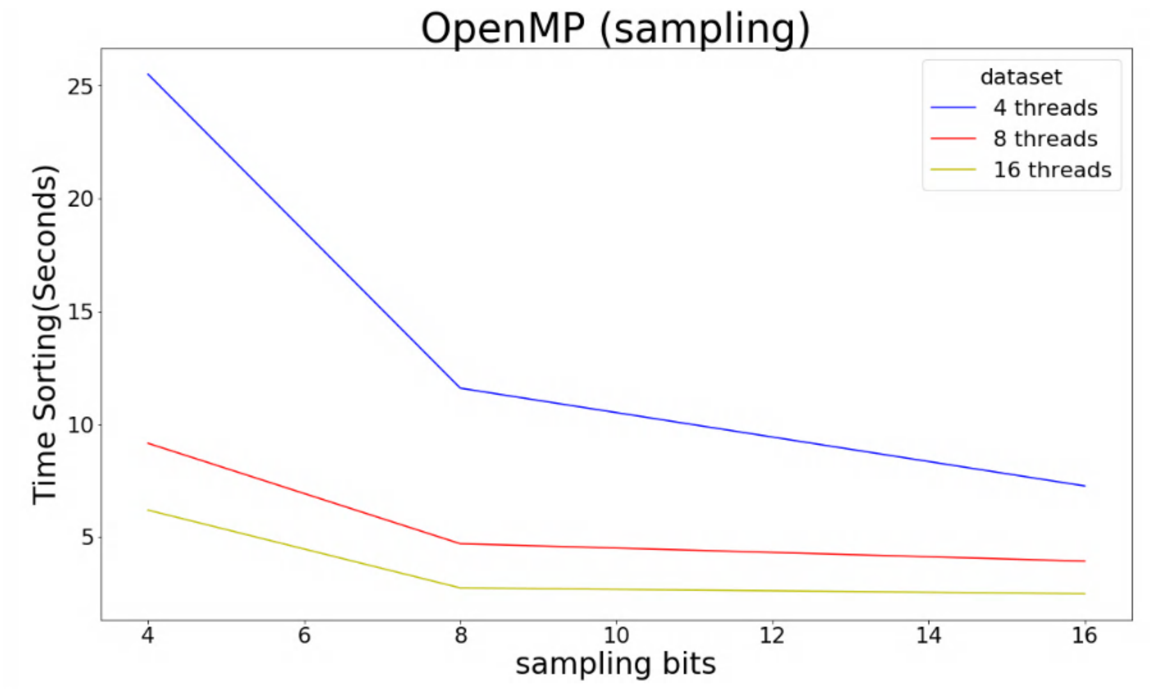
Observing the result of using different datasets (Graph 5), we can find that OpenMP has better performance improvement in the larger datasets and more parallel threads. According to Amdahl's Law, if the proportion of the sorting time is higher, the speedup will be more significant. This phenomenon occurs in larger datasets(such as gplus). Also, if we have more parallel threads, the speedup will be more effective, too. On the other hand, if the proportion of sorting decreases or the ratio of OpenMP overhead increases, the overall execution time increase. This phenomenon occurs when we sort a small dataset or parallel sorting with few threads.

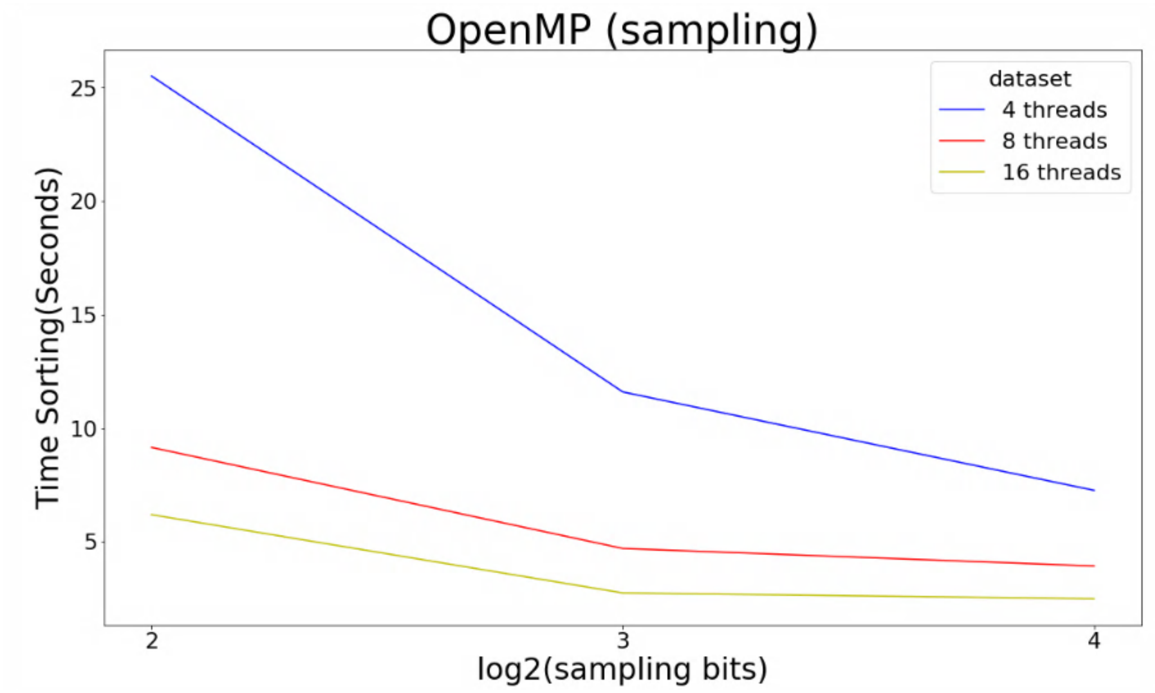
Regarding samplings bits(Graph 6), if we increase sampling bits, the outer for loop decreases iterations. For example, when the sampling bit increases from 2 bits to 4 bits, the iteration times decrease from 16 to 8 iterations, and the overall execution speed increases significantly. However, when we increase the sampling bits from 8 to 16 bits, the overall execution time increases slightly. Since we can speed up the time of sorting, we can not decrease the time of OpenMP overhead.





Graph 5: OpenMP version with various datasets and various threads.





Graph 6: OpenMP version with various sampling bits (using gplus benchmark).

3.2 MPI version

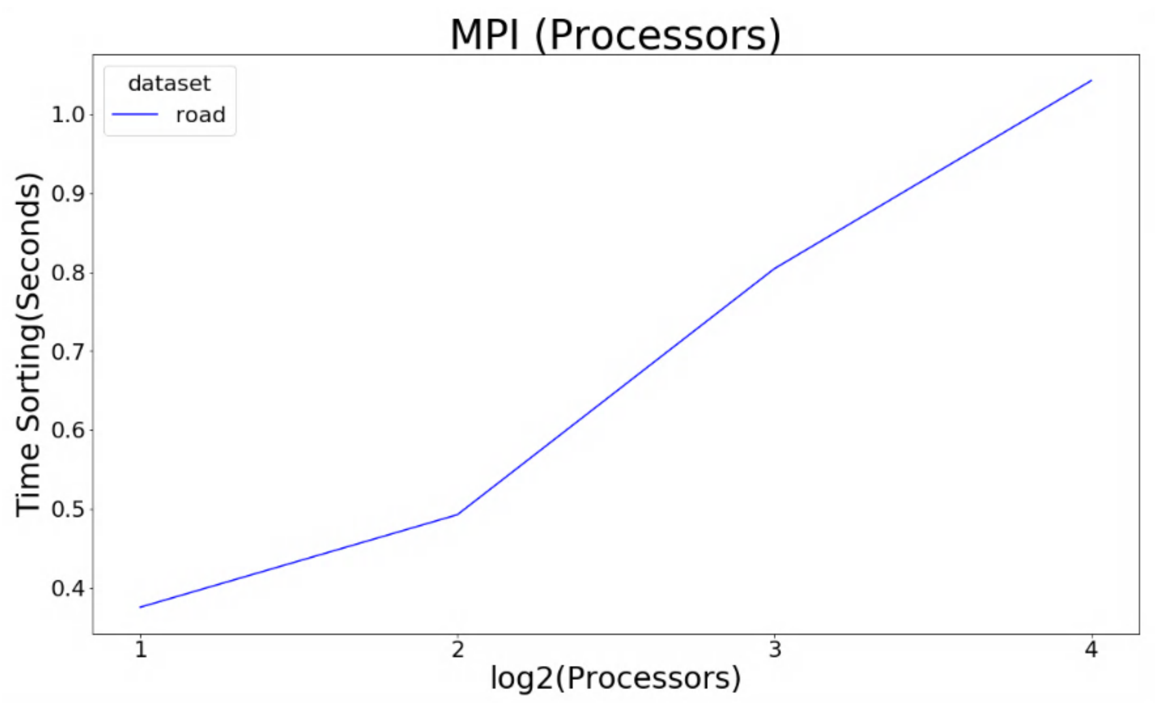
In this experiment, I parallelized radix sort using MPI with different numbers of processors and benchmarks.

Sampling bits	2	4	8	16
Time sorting (seconds)	169.7	84.5	42.7	22.9

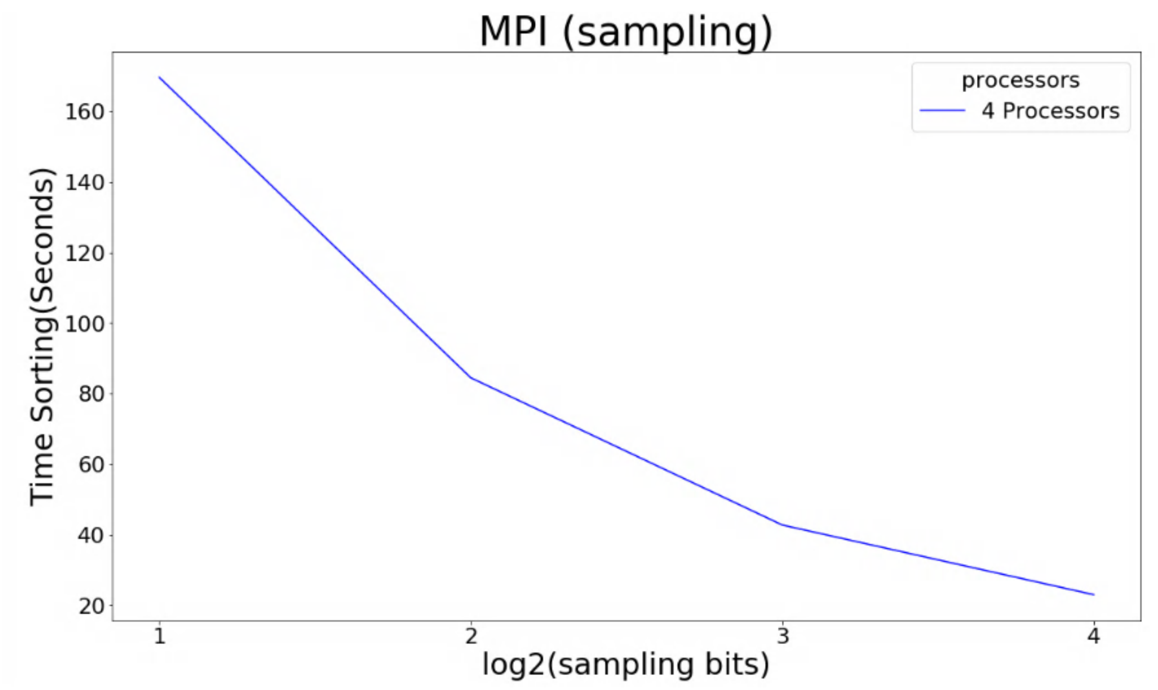
Table 5: sorting time of different sampling bits

Due to the overhead of the MPI command, the overall execution time increase instead of decrease. Also, when I increase the number of processors, the speed slows down significantly. I think this is because the communication overhead between processors is very high. I use the Allreduce MPI command after the sorting. Since the size of the sorted edge array is enormous, it takes a long time to reduce the whole array from each processor and then broadcast the array back to them.

Regarding the sampling bits(Graph 8), the sorting executes faster when I use more sampling bits because the outer loop executes fewer iterations. Refer to Table 5, when the sampling bits are 8, radix sort does 4 times outer for loop and takes almost 43 seconds to finish sorting. When the sampling bits are 16, radix sort does 2 times outer loop and takes roughly 23 seconds to complete sorting. The radix sort speedup almost 2 times. However, there are some diminishing returns. The 16 sampling bits radix sort only speeds up 1.86 times to 8 sampling bits radix sort. Since the maximum value of keys increases from 256 to 65536. The MPI sends and receives data more times between processors while accumulating the count vector. This causes some overhead and lets 16 sampling bits radix sort slower than expected.



Graph 7: MPI version with various processor numbers.(using road benchmark)



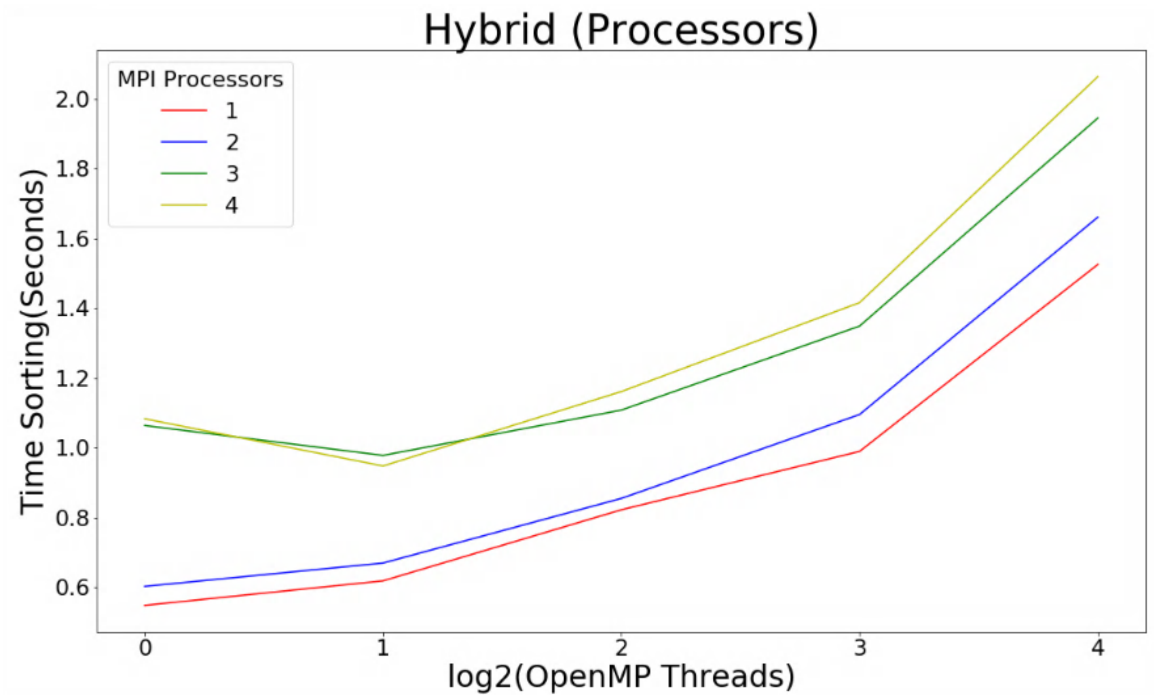
Graph 8: MPI version with various sampling bits. (using gplus benchmarks)

3.3 Hybrid OpenMP/MPI

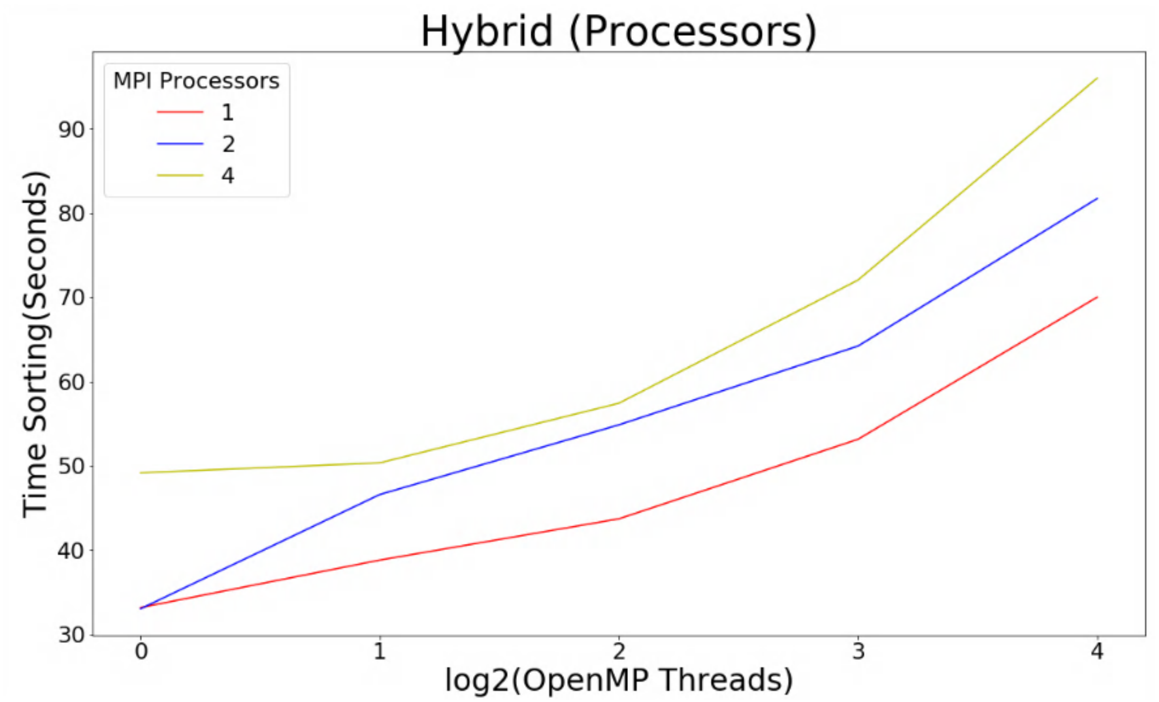
In this experiment, I parallelized radix sort using MPI+OpenMP with different numbers of processors and benchmarks.

The effect of the increasing number of MPI processors is similar to the MPI experiment. When I use more MPI processors, the sorting becomes slower. I think this is because the overhead of communication between processors is too high since the Hybrid version needs to synchronize sorted edges array between processors, which is the same problem in the MPI version.

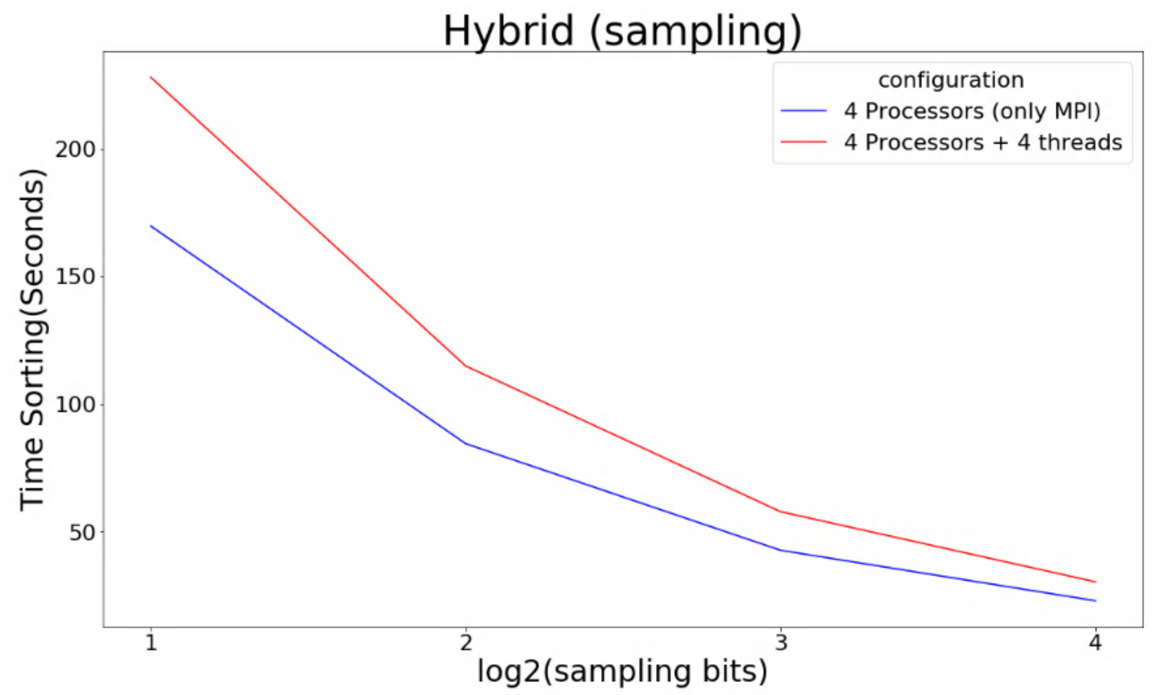
The effect of increasing OpenMP threads slow down the sorting in most of the case. However, in some particular cases, the performance of two threads is the same or even better than one thread. In theory, the Rome processor has four NUMA nodes, and each NUMA node has four cores that can execute eight threads. The best utilization should be four MPI nodes with eight threads. I think the phenomenon of slow down is due to memory locality problems or improper initialization of shared data.



Graph 9: Hybrid version with various OpenMP threads and MPI processor numbers (using road benchmark).



Graph 10: Hybrid version with various OpenMP threads and MPI processor numbers (using gplus benchmark).



Graph 11: Hybrid version with various sampling bits, compared with MPI version (using gplus benchmark).

4. Summary

Compare the three versions of parallel sorting. OpenMP version is the only version that faster than baseline. Regarding the size of datasets, OpenMP might need more threads to speed up faster than baseline. MPI has huge overheads to communication between processors. Thus MPI version is much slower than the baseline. Finally, the Hybrid version is the slowest due to the overhead of MPI and OpenMP.