NC State University

Department of Electrical and Computer Engineering

ECE 506 – Architecture of Parallel Computer

Project 1

By

RAMACHANDRAN SEKANIPURAM SRIKANTHAN

Introduction

The project focusses on parallelizing the radix sort algorithm to provide exponential speedup to conventional radix sort which is implemented serially in sorting very large datasets. The project is being divided into 3 submodules where in each module a different strategy of parallelizing is being implemented. The first module uses OpenMP to parallelize the algorithm which is based on a shared memory model. The MPI modules parallelize the algorithm based on message passing scheme. The final hybrid module parallelizes the algorithm using both of the above-mentioned approaches to exploit the advantages of both the methods. The module is being tested using 5 test bench containing random unsorted graph edges being listed in source -> destination format. The radix sort algorithm sorts the source edges and the map between source and the respective destination is being preserved while sorting.

OpenMP Implementation

The OpenMP paradigm uses a shared memory model. The huge work load is being distributed among various threads to compute parallelly. The threads can communicate with each other using a shared memory. Also, each thread has its own private members as well.

Algorithm Analysis

The algorithm is being analysed using the following approaches to find out the implicit and explicit parallelism in it. The analyses also identify various dependencies in the variables which is being used.

- Iteration Traversal Graph (ITG).
- Loop dependency analysis (LDG).

For Count sorting algorithm:

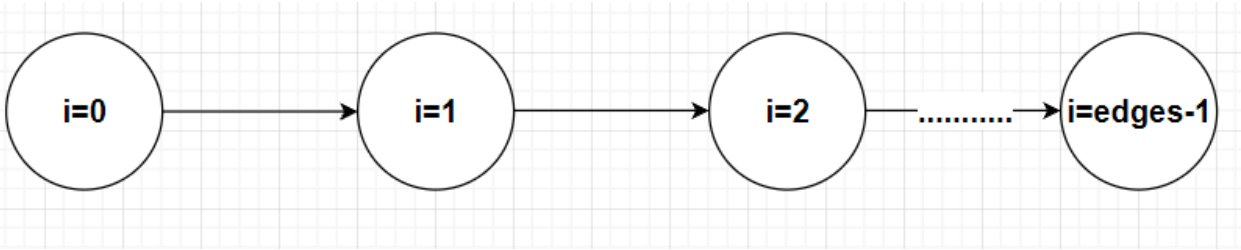
We first initialize the vertex_count array with 0's which can be done parallelly. The C library has a function to allocate a memory block along with initializing it to 0.

Loop 1:

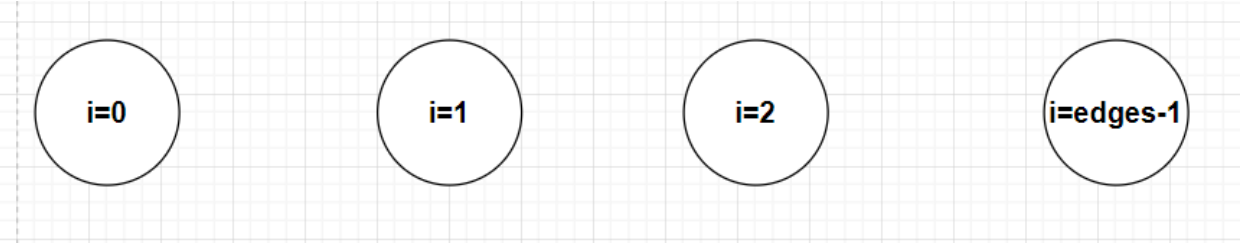
```
for (i = 0; i < graph->num_edges; ++i)
    {
        key = graph->sorted_edges_array[i].src;
        vertex_count[key]++;
    }
```

We first increment the vertex_count array based on the count value of each source element in the input graph.

ITG



LDG



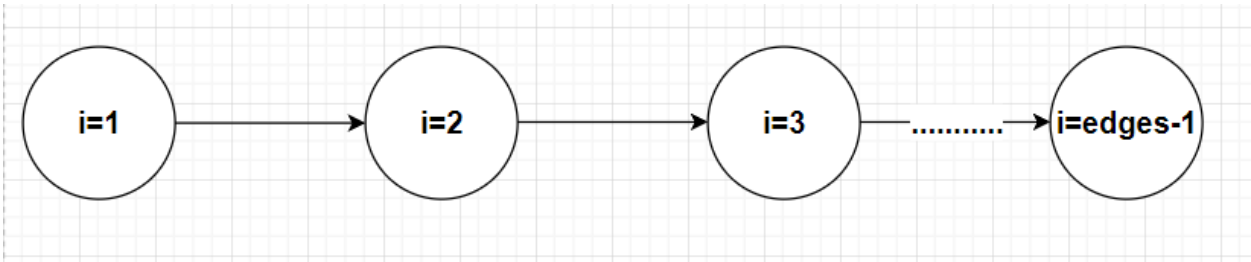
Here we can observe that the iteration is performed sequentially from start of the graph edge list till total number of edges. Also, from LDG we can see that there is no dependency between various iterations of the loop and so its safe to parallelize this loop.

Loop 2:

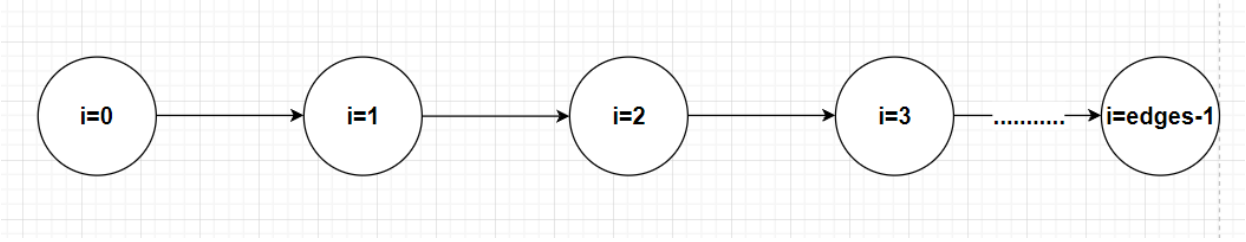
```
for (i = 1; i < graph-> num_edges; ++i)
    {
        vertex_count[i] += vertex_count[i - 1];
    }
```

We now perform the cumulative sum of the elements present in the vertex_count array.

ITG



LDG



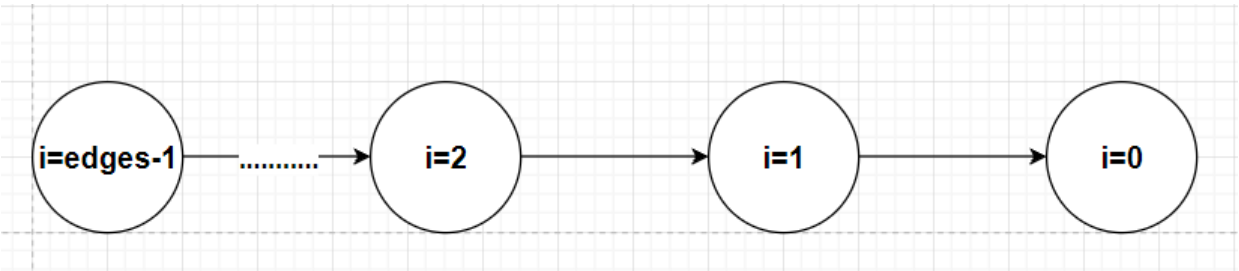
We can see from the LDG, that there is an underlying true dependency between previous iteration's values. Each element in the vertex_count array is being summed with the previous element of the array to find out the cumulative sum.

Loop 3

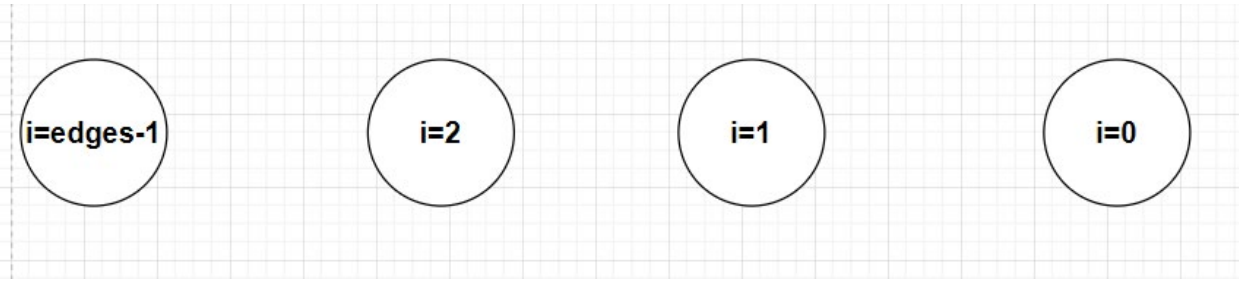
```
// fill-in the sorted array of edges
    for(i = graph->num_edges - 1; i >= 0; --i)
    {
        key = graph->sorted_edges_array[i].src;
        pos = vertex_count[key] - 1;
        sorted_edges_array[pos] = graph->sorted_edges_array[i];
        vertex_count[key]--;
    }
```

Finally, we out the sorted position of the element from the count values in the vertex_count array and place it in the correct position in the original array.

ITG



LDG

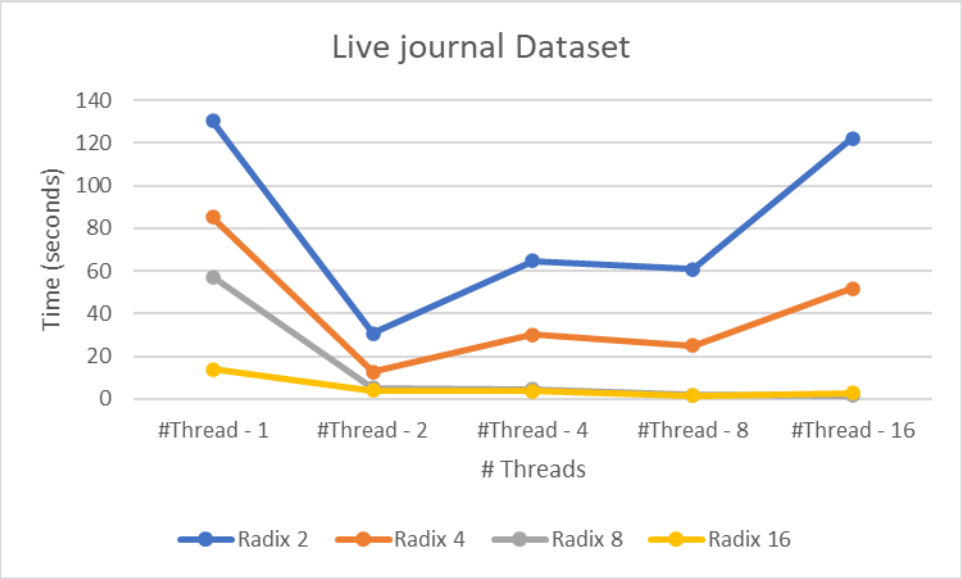


We can conclude that there is no dependency in the final loop and so it's safe to parallelize.

Speed up Analysis on increasing the number of processors and radix

This performance analysis captures the time taken for the parallel radix sorting algorithm, by varying the number of processors (i.e.) number of threads which is being used in this case for each value of the radix. The radix value is changed from 2,4,8 to 16 since we have only 32-bit numbers in our unsorted inputs. For each radix value the number of threads is being increased from 2,4,8 to 16. Each value in the below tabulation indicates time in seconds for the algorithm to complete sorting.

Plot of Time(seconds) Vs #Threads for each radix in live journal dataset



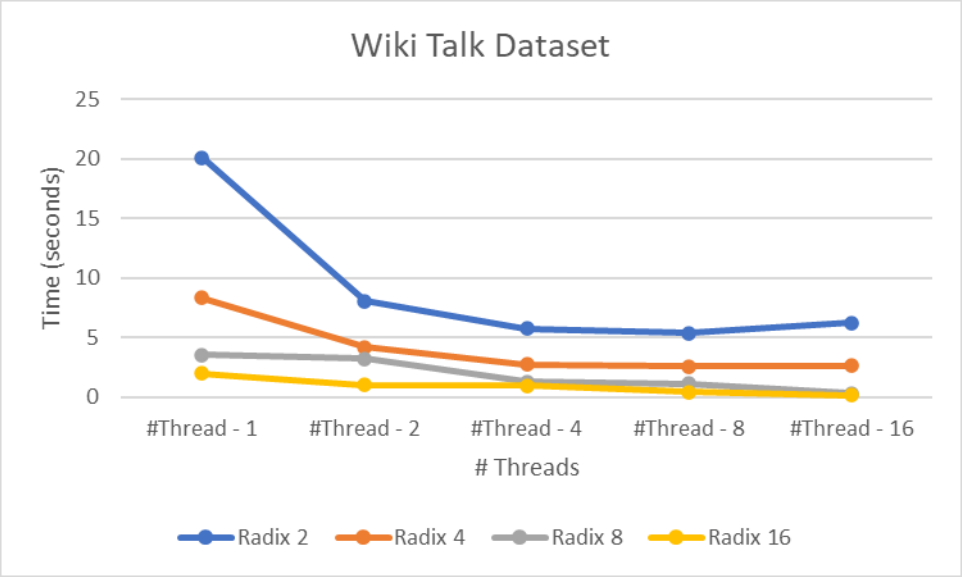
OpenMP					
Live Journal Dataset					
	#Thread - 1	#Thread - 2	#Thread - 4	#Thread - 8	#Thread - 16
Radix 2	130.308132	30.764551	64.631539	60.760064	122.265153
Radix 4	85.122786	12.635318	30.297129	24.996523	51.970183
Radix 8	57.135655	5.22461	4.4282	1.811434	1.667826
Radix 16	13.799681	4.087796	3.542577	1.541493	2.665556

Here we can observe that, for a particular radix, when the number of threads is increased there is a speedup in performance of the radix sort. We can also observe that, there is a significant increase in performance when added a single thread to the serial radix sort.

There is also a performance degradation for lower radix bits, since in that case the overhead of thread creation and operation dominates as for lower radix bits say 2, we need to create thread and destroy them after processing 16 times as we have 32-bit integer in our unsorted list. Thus, having higher number of threads for smaller radix causes a performance degradation.

We can also notice that for radix value of 4,8 and 16 we can see significant performance while using 4,8 and 16 threads, since the thread creation and destruction overhead becomes lesser as we iterate only for few times compared to radix of 2 to reach 32 bits.

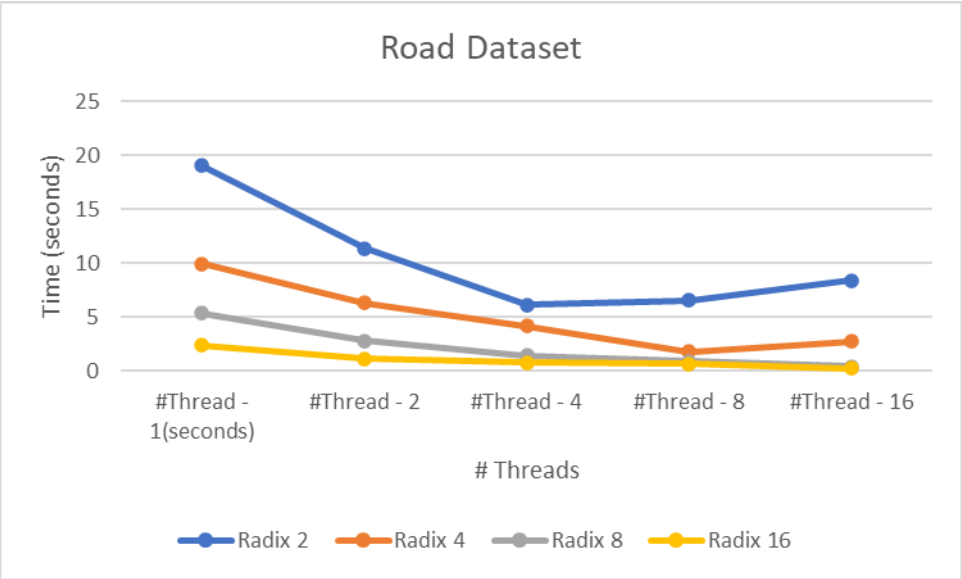
Plot of Time(seconds) Vs #Threads for each radix in Wiki talk dataset



OpenMP					
Wiki Talk Dataset					
	#Thread - 1	#Thread - 2	#Thread - 4	#Thread - 8	#Thread - 16
Radix 2	20.093563	8.064075	5.753672	5.362359	6.241811
Radix 4	8.340958	4.184822	2.752227	2.613898	2.656163
Radix 8	3.567073	3.212795	1.322159	1.140718	0.341112
Radix 16	2.024242	1.038883	0.950326	0.426288	0.195582

Here also we can notice the similar behaviour for smaller radix size of 2, the thread overhead dominates while having 16 threads and so time for sorting increases in that case. The performance is optimal for radix size of 4 and 8, having number of threads of 8. For having a radix of higher number increases the memory required to perform the execution as we need to store 2^{16} elements and traverse through them, even though it has a benefit of very much lesser thread overhead as we iterate only twice to get to 32 bits.

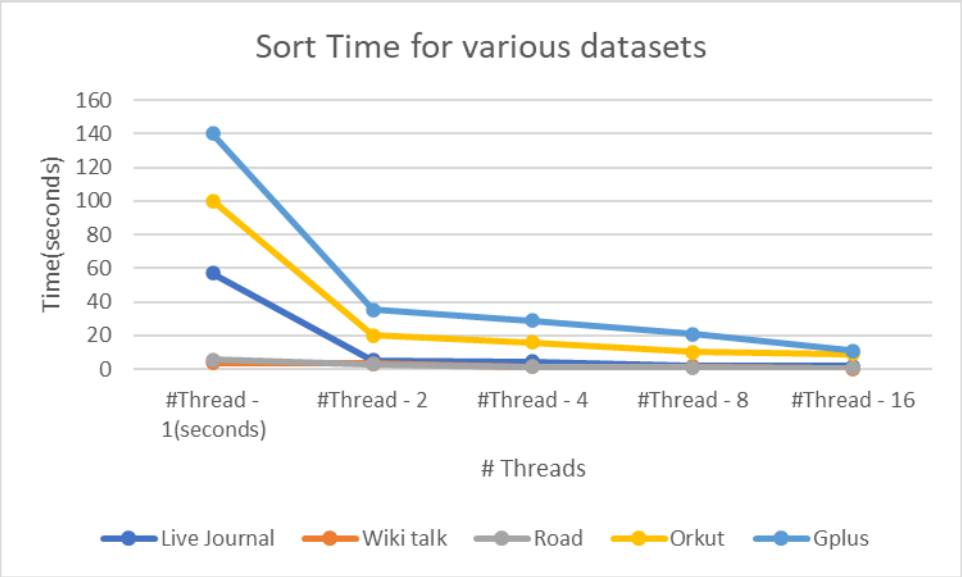
Plot of Time(seconds) Vs #Threads for each radix in Wiki talk dataset



OpenMP					
Road Dataset					
	#Thread - 1	#Thread - 2	#Thread - 4	#Thread - 8	#Thread - 16
Radix 2	19.057294	11.331084	6.113026	6.5428	8.364978
Radix 4	9.929949	6.271994	4.15234	1.754836	2.688969
Radix 8	5.311167	2.762284	1.409113	0.888027	0.362768
Radix 16	2.344186	1.126835	0.725082	0.614199	0.205048

The Road dataset graphs also shows the similar trend. The thread overhead dominates thereby increasing the total sort time for lower radix values. Even though having higher radix values is better, the memory required becomes an overhead.

Plot of Time(seconds) Vs #Threads for radix 8 for all datasets



Radix - 8					
Dataset	#Thread - 1	#Thread - 2	#Thread - 4	#Thread - 8	#Thread - 16
Live Journal	57.135655	5.22461	4.4282	1.811434	1.667826
Wiki talk	3.567073	3.212795	1.322159	1.140718	0.341112
Road	5.311167	2.762284	1.409113	0.888027	0.362768
Orkut	100.123479	20.127845	15.764322	10.152367	8.972367
Gplus	140.231234	35.28389	28.783657	20.778456	10.874677

The graph shows the performance in various datasets, for a fixed radix of 8 and by varying the thread number w can see speedup in performance.

Count Sort Vs Parallel Radix Sort

The count sorting algorithm takes directly the entire number and computes the count array and performs sorting. The main problem with this approach is that memory requirements become overhead. In this case we use 32-bit integers and we need 2^{32} elements to be placed in an array and traversed each time we update a value. On the other hand, Radix sort has lesser memory overhead as we divide the 32-bit number by radix and perform operation. This significantly reduces the memory and traversal overhead.

The processor can store this small amount of data in cache to further enhance the computation speed in radix sort algorithm as compared to count sort where we may face significant cache misses since we deal with large range of numbers.

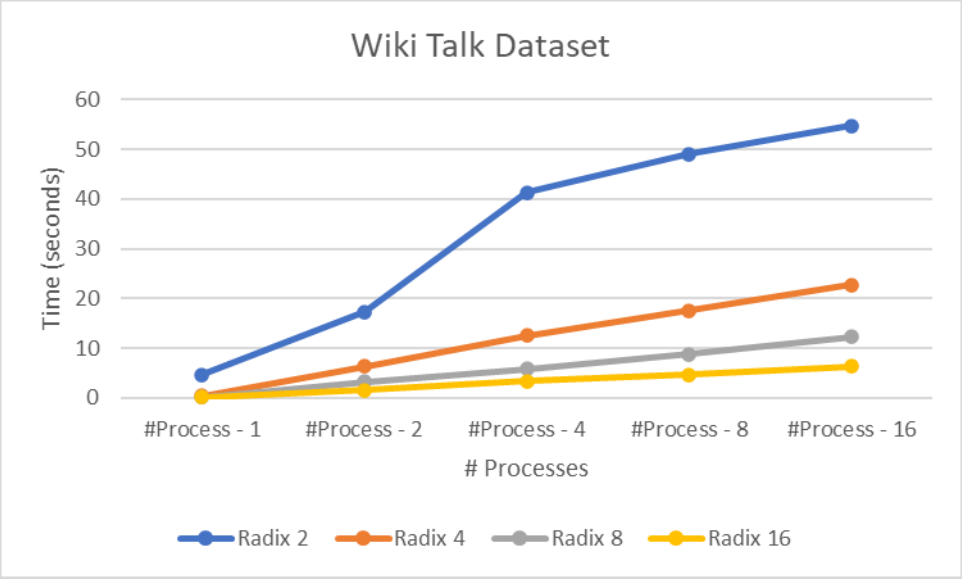
Also, for larger datasets, parallel radix sort performs faster data is split between various process and sorted. Even if count sort is parallelized, each processor needs to 2^{32} numbers wherein most of values are 0 thereby wasting huge amount of memory space. For smaller datasets, we can use count sort as parallel radix sort may have less performance due to thread overhead.

MPI Implementation:

This implementation parallelizes the radix sort algorithm by using a message passing paradigm. The workload is being divided into various process. Since each process has its own address space, communication between various process occurs through inter process communication protocols. Based on the ITG and LDG graphs, once when the required number of processes is being spawned, the entire graph elements is split and distributed among the various processes. Each process performs the counting, prefix sum and finally sorts its elements based on the count array. Finally, all the process sends the data back to sender and the sender merges everything into final result.

Plot of Time(seconds) Vs #Process for each radix in Wiki talk dataset.

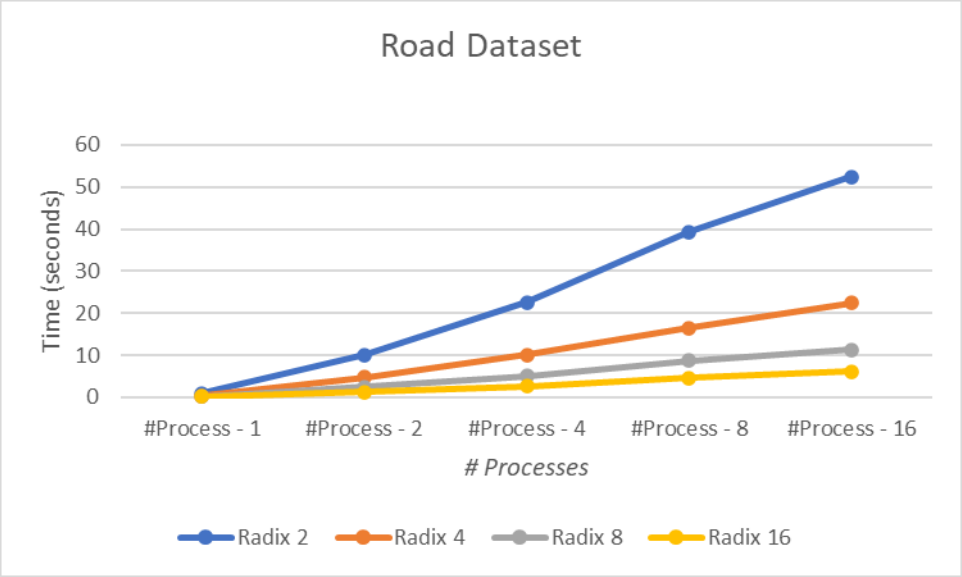
The graph is plotted by taking the time taken for sorting, for each radix value by increasing the number of processes to perform the sorting.



MPI					
Wiki Talk Dataset					
	#Process - 1	#Process - 2	#Process - 4	#Process - 8	#Process - 16
Radix 2	4.660057	17.298914	41.309507	48.985544	54.686195
Radix 4	0.403833	6.388268	12.513519	17.504721	22.769809
Radix 8	0.21132	3.277522	5.848805	8.822257	12.227839
Radix 16	0.117008	1.586534	3.400974	4.631877	6.359252

From the graph, we can observe that, for a particular radix value increasing the process counts have a nearly same or slight better performance. For lower value of radix of 2, we can see that the performance degrades as we increase the number of processes since we keep of iterating the loop which repeatedly performs process send and receive (communication overhead). But for higher radix values, we can see similar performances among various processes. The communication overhead and memory requirements for send and receive buffer are the significant overhead in using processes to divide the load.

Plot of Time(seconds) Vs #Process for each radix in Road dataset

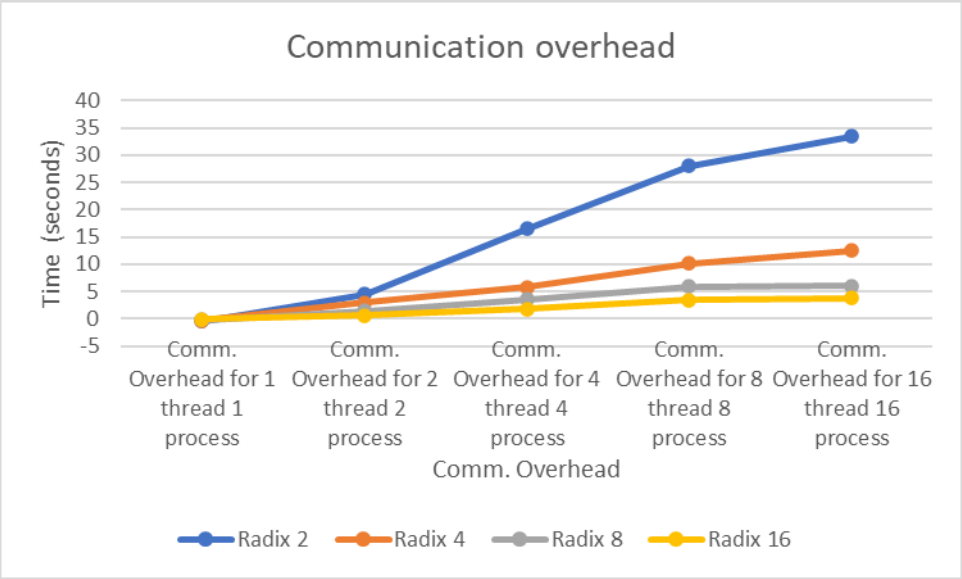


MPI					
Road Dataset					
	#Process - 1	#Process - 2	#Process - 4	#Process - 8	#Process - 16
Radix 2	0.945894	10.079685	22.586505	39.344141	52.449621
Radix 4	0.352088	4.728569	10.004083	16.482586	22.401806
Radix 8	0.192072	2.407125	5.020998	8.644111	11.324067
Radix 16	0.117638	1.215187	2.576997	4.582393	6.147156

We can observe the similar performance here as well, for smaller radix sizes we have high communication overhead when the number of processes is being increases. Thus, having an optimal radix size of 4 or 8 or 16 we can better performance while having higher number of processes to divide the load.

Processor Communication Overhead

Plot of Time(seconds) Vs Com for each radix in Road dataset



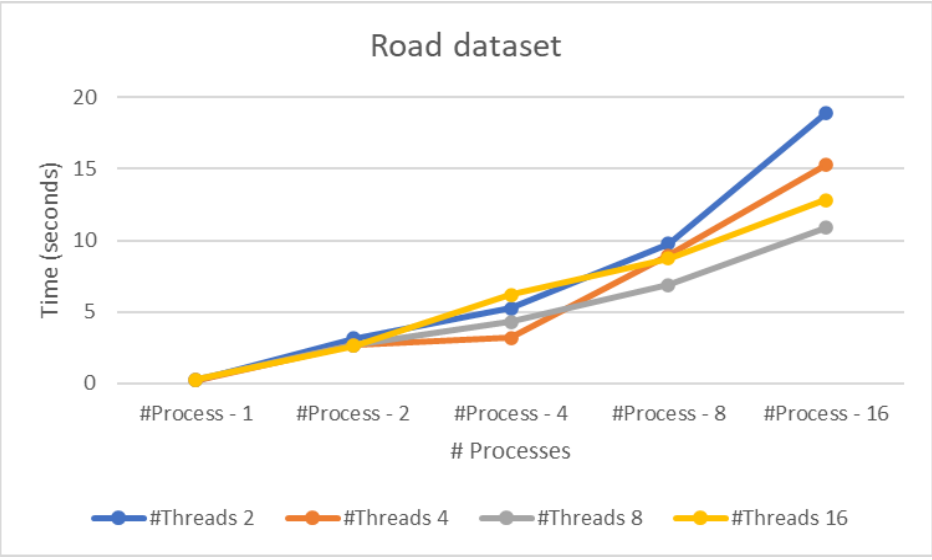
Road Dataset	Comm. Overhead for 1 thread 1 process	Comm. Overhead for 2 thread 2 process	Comm. Overhead for 4 thread 4 process	Comm. Overhead for 8 thread 8 process	Comm. Overhead for 16 thread 16 process
Radix 2	-0.41908	4.536885	16.47348	28.01306	33.39233
Radix 4	-0.33688	2.973733	5.851743	10.21059	12.47186
Radix 8	-0.1707	1.519098	3.611885	5.881827	6.0129
Radix 16	-0.08741	0.600988	1.851915	3.455558	3.80297

The communication overhead is being calculated by taking the difference between the sorting time of MPI (processes) and sorting time using openmp (threads). The positive indicates the communication overhead. We can see that the communication overhead for lower radix values of 2 is higher as compared to higher radix since we iterate many times to get to 32 bits thereby increasing the number of send and receives.

Hybrid Implementation

This approach uses both the above methods of parallelizing the radix sort by exploiting the advantages of both the methods. The MPI implementation is being used to divide the workload among the various processes. The OpenMP implementation is being used inside each process to divide the work among various cores, thereby parallelizing the work load of each process.

Plot of Time(seconds) Vs #Process for each radix in Road dataset



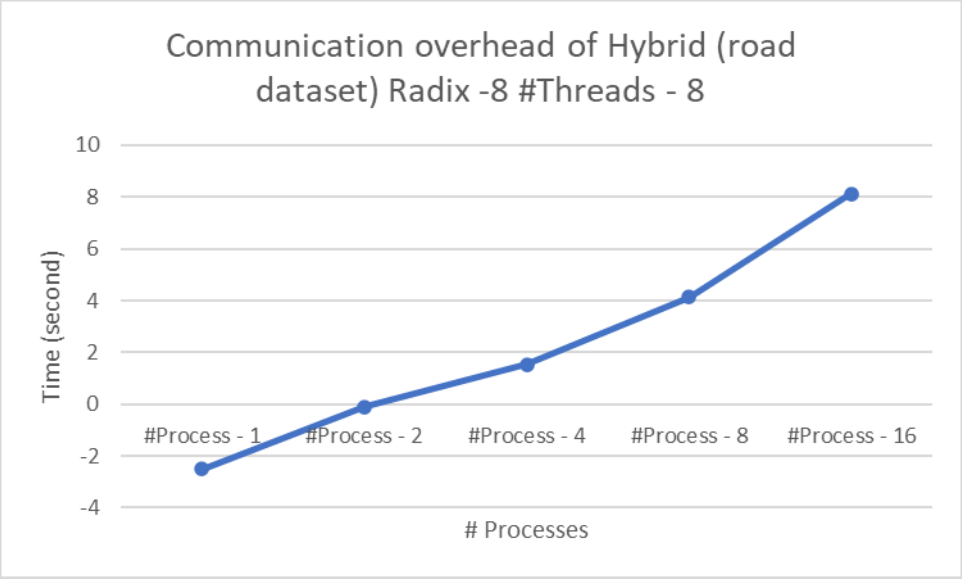
Hybrid					
Road Dataset Radix 8					
	#Process - 1	#Process - 2	#Process - 4	#Process - 8	#Process - 16
#Threads 2	0.222127	3.136498	5.246789	9.763566	18.892967
#Threads 4	0.223718	2.689925	3.1728989	8.969274	15.278891
#Threads 8	0.248732	2.664465	4.291809	6.892567	10.892832
#Threads 16	0.24409	2.646609	6.199214	8.726389	12.83789

We can observe from the graph that the behaviour is similar to MPI but the communication overhead is being reduced since parallelization is introduced inside various processes. For a radix value of 8 we can see performance improvement compared to MPI value. Due to communication overhead, there is a slight degradation of performance with increase in number of processes.

Communication Overhead of Hybrid:

Communication overhead of Hybrid					
Road Dataset Radix 8					
	#Process - 1	#Process - 2	#Process - 4	#Process - 8	#Process - 16
#Threads 8	-2.513552	-0.097819	1.529525	4.13028276	8.130548

The communication overhead is determined by finding difference between the sort time of hybrid algorithm with fixed 8 threads and fixed radix of 8 and the corresponding sort time using openmp (thread) implementation.



From the graph, we can see the communication overhead is lesser than the normal MPI implementation. For 8 threads, a radix of 8 and for 8 process the communication overhead for MPI is 5.881827. For similar configuration we have 4.13028276 for hybrid implementation. Thus, we can clearly see that hybrid implementation reduces the communication overhead by parallelizing the work among many cores inside a processor.

Comparison summary for OpenMP, MPI and Hybrid Implementation

Parameters	OpenMP	MPI	Hybrid
Paradigm	Shared memory model	Distributed memory model	Using both models
Parallelizing entity	Thread	Process	Thread and process
Overheads	Thread creation, allocation and free up	Communication overhead	Both
Implementation Complexity	Less complex due to usage of shared memory	complex due to message passing implementation	More complex due to integration of both
Speedup	Occurs with increasing threads	Occurs with increase in process up to a point	Occurs with increase in process, but threshold point is more than MPI
Dataset	Large	Moderate	Large
Memory requirements	Less (Shared memory for all threads)	More (Separate memory for each process)	More (Separate memory for each process. Threads share the memory)