ECE506 - Project 1 - Stevan Dupor - smdupor@ncsu.edu - id: 000211897
Due: Oct 4th, 2021, 11:59pm – Exported to PDF: 2021/10/03: 20:46

ABSTRACT: This project investigated the design and performance analysis of parallelization of Radix sort within a Graph analysis program, using three methodologies: OpenMP(Multi-Thread), MPI(Multi-Process/Host), and a tandem design using both. I will detail the program analysis, design choices, performance evaluation, and conclusions regarding these methodologies in the following sections.

Introduction and Task Analysis

The Graph Analysis program requires a sorted list of edges (connections between vertices) to implement the breadth-first search in a timely fashion. In the original revision of this program, a single-pass Count Sort is used to sort the edge list.

Terminology

Count sort consists of four loops: the first initializes an N-length vector with zeroes, where N is the size of the largest element in the collection (assuming only unsigned integers). The initialization loop (and any pre-configuration initialization) shall be referred to as **INIT** in this paper. The second loop proceeds through the list to be sorted, counting the quantity of instances of each integer, and placing a total-count-quantity at the index of each integer in the N-length vector. This loop shall be referred to as **COUNT**. The third loop iterates along the N-length vector, creating a cumulative sum. This loop will be referred to as **TRANSFORM**. The fourth loop iterates along the original collection of objects, looks up the index of each from the N-length vector, and moves the object to that index in a new collection. Finally, this third loop decrements the count value in the N-length vector by 1 as it sorts each element, such that the next instance of a matching object will be emplaced in the correct location. This fourth loop shall be referred to as **SORT**. Throughout the design sections of this paper, the qualities of, and changes to, these loops will be referred to extensively using these established shorthand names.

Radix sort uses multiple iterations through a specialized count sort, which counts elements by a single digit, or a subsection of digits (eg, tens-place or one-or-more bits-place counting), and emplaces counts into a M-length array equal to the number of possible unique values within the chosen "Radix Size". Eg, a radix size of two bytes would require a M-length array of size 256, and would require four iterations of the outer master loop. This outer loop shall be referred to as the **RADIX LOOP** for the remainder of this paper.

Parallelization Challenges Analysis: Count Sort

As an initial exploration of parallelizing this sort, each loop was analyzed for its parallelization properties. A simplistic OpenMP parallelization of a single-iteration Count Sort was implemented to analyze the performance gains and losses from adding parallelization to this general implementation case.

INIT: DOALL

The init loop contains no loop dependencies: as many threads may write to any given vector address as the same time without causing race conditions. As such, this loop is an excellent candidate for parallelization in the Count sort case, particularly when very large integers are present in the dataset (meaning, a very large N-length count vector).

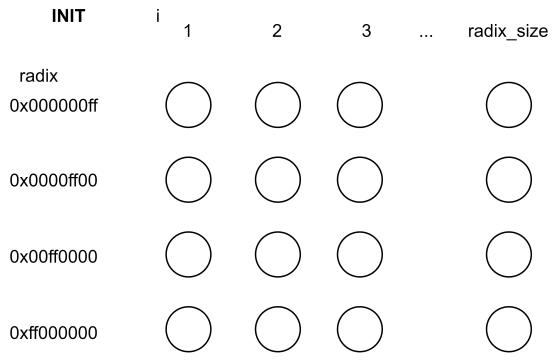

Figure 1: LDG For INIT Step (No Dependencies)

COUNT: DOACROSS

The count loop also presents a prime target for parallelization, but requires some synchronization. Different threads may attempt to increment a count vector address at the same time, so some form of synchronization must be present. For the Count Sort analysis, simple Atomic synchronization was used; however, notably, there was some performance loss. This portion of the parallelization problem may be considered a False dependency, as algorithmic transformation (see Design section) may be used to resolve this dependence.

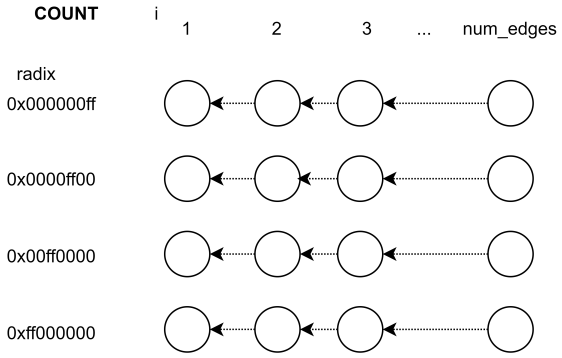

Figure 2: LDG For COUNT Step (AntiDependencies within each radix)

TRANSFORM: SINGLE

The Transform loop presents a significant challenge to parallelization, as a True dependence exists between iterations of this loop. Each iteration cannot perform the cumulative sum until the data is available from the previous iteration. However, as explored further in the Design section, there is opportunity for parallelization between whole-iterations of this loop using algorithmic transformation.

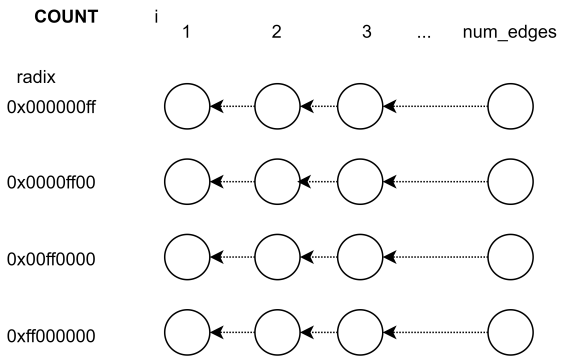

Figure 3: LDG For TRANSFORM Step (True Dependencies within each radix)

SORT: DOPIPE

In the Count sort implementation, there is limited opportunity for parallelization of the SORT loop. There exists an anti-dependence between iterations, where a subsequent iteration requires an updated count vector value to prevent over-writing already-sorted items in the destination collection. Furthermore, significant synchronization is required such that the existing order at levels intended to NOT be sorted (eg, the original order of Edge "destinations") is consistent and preserved while the "sources" are properly sorted.

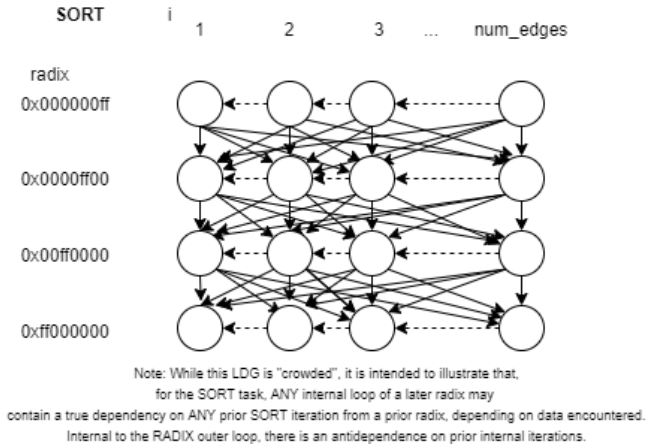

Figure 4: LDG For SORT Step (AntiDep. within each radix, True Dep. between Radices)

Parallelization Challenges: Radix Sort

Radix sort presents the same challenges from the above loops, while introducing a new True dependence: The active sorting (relocation of objects) from one radix to another (eg, least-significant byte to next-significant byte) cannot take places until the prior sort has completed. Allowing this race condition to occur unsynchronized will cause a loss of sorted order from previous iterations of the radix sort. As such, the project specification recommends running these iterations serially due to the challenges of parallelizing these; however, in the design section, I will explore three possible solutions to this issue. In the following Figure 5, I illustrate some early results of parallelization on the sections at hand across multiple datasets. Note that a time gap for MPI messaging is included on this plot, but will be zero as MPI was not used for the basic OpenMP case.

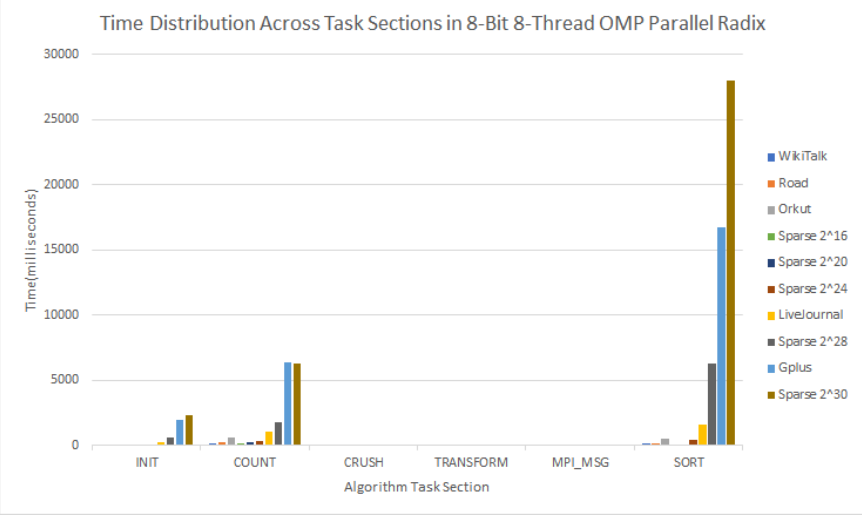

Figure 5: Algorithm Sections Performance Across All Datasets

Design Considerations

Broadly, the goal of this project is to increase performance of this algorithm as much as possible by parallelizing as many of the tasks as possible. More specifically, simply running the various loops in parallel (using synchronization directives when appropriate) is not enough, as the synchronization incurs significant performance cost thanks to cache misses and invalidations.

Timing Analysis

For all design considerations, a vector of timers was created and passed to each sort function. These timers were started at the beginning of the function and stopped after each subsection to analyze which subsections were causing gain/loss in performance. While this code is not included in the submission (in order to match the expected function arguments), the datapoints from these measurements is addressed in the Results section.

Key Design Task: Loop Splitting: INIT, COUNT, TRANSFORM from SORT

The root of my design for this parallelization task begins with loop splitting for the outer RADIX loop. In the final production version, I have separated the INIT, COUNT, and TRANSFORM tasks (manageable dependencies) from the SORT task (unavoidable dependencies). The Count Vector is transformed into a 2-D count array, and re-sized such that each independent thread and/or process has enough space to work independently of all other threads. Because each radix has it's own private count vector within the 2-d array, even the TRANSFORM step can be performed in reduced parallel fashion (as many threads as there are independent radices).

The result is that all RADIX iterations over these three steps are performed concurrently.

However, there are tradeoffs in overhead and memory cost. More threads will require more time to fork the threads into existence, and increased memory demand will cause more consumption of memory while these processes are running. Critically, a new synchronization step must be added where the private data from the independent threads is collapsed back into the expected size for the SORT loop. This additional overhead loop will be referred to as the **CRUSH** stage for the remainder of this paper.

Discarded Design 1: A Synchronized-Tandem Parallelization of SORT Inner Loop

The first parallelization strategy involved running the inner sort loop in parallel for each single iteration of the outer RADIX loop. Prevention of race condition is necessary and critical as there exists a true dependence between shared addresses of the vertex_count vector. I found that the required synchronization made this technique perform slightly worse than working on this loop serially, like due to both cache invalidations and the need to synchronize each process at the read/decrement stage of the vertex_count loop. Hence, this design was ultimately discarded in favor of the better-performing serial SORT task.

Discarded Design 2: An Unsynchronized Responsibility-Split Parallelization of SORT Inner Loop

Next, a different parallelization technique for the inner SORT loop was explored to eliminate the overhead of thread synchronization. Multiple threads were spawned on each of these inner loops, but the threads were configured to only trigger sort actions based on certain criteria. As a proof-of-concept, two threads were utilized, one of which checked elements and only performed the sort when an ODD radix was encountered, while the other thread checked elements and only performed a sort on EVEN radix elements. When performance between this technique and a sequential-single-thread (with INIT-COUNT-CRUSH in parallel) was compared, I encountered approximately 15% performance loss. I hypothesize this loss is due to a combination of thread forking overhead and cache invalidations in the shared memory variables, as there is no stalling for synchronization in this design. Ultimately, this technique was also discarded in favor of improved performance.

Discarded Design 3: A Semaphore-Based Pipelined Parallelization of SORT Outer Loop

For the purpose of exploration, total parallelization of all tasks was attempted, including the SORT loop. In order to address the dependence between iterations of SORT (as part of iterations of the split RADIX loop), a separate private temporary collection array was instantiated in memory for each iteration of SORT, eg, for a 8-bit Radix, there existed the original unsorted array, and four copies of the increasingly-sorted destination array. Each of these copies were initialized with a semaphore value (-1), and all four iterations of SORT were started simultaneously. Custom synchronization logic stalled each subloop until the semaphore at the desired address was cleared (replaced by a valid Source value), and the subloop was permitted to continue.

However, this design presented multiple serious problems. The first, and most obvious, is the extreme pressure on system memory as a result of needing so many copies in memory. This could be addressed by

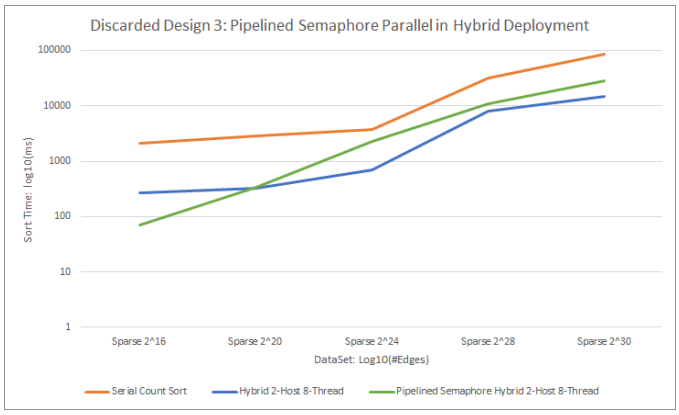

Figure 6: Performance of Semaphore Pipelined Sorts

running the simulation on hosts which provide ample memory for the task (eg, the Skylake Partition on the NCSU CSC ARC cluster, which provides 96GB per host), or splitting the task between multiple hosts with MPI to spread the memory pressure.

More pertinently, this technique introduces a new Initialization task which is extremely expensive, where two, four, or eight (depending on radix size) copies of destination edge arrays must be allocated and initialized with the required semaphores. While this task can be run in parallel, the cost of handling it was unavoidable.

But, the most problematic issue was in the relative payoff of this technique. A stalled parallel "second" iteration can only start the sort process when the "first" iteration has populated the object at the desired memory address (in our implementation, the last item in the collection). In the case of an 8-bit radix, there exists only a 1/256 chance that this address will be populated at (this) iteration. Furthermore, before this "second" loop can move on, another 1/256 chance is encountered, then another. Lastly, before the "second" loop can continue past all the items in the collection with radix value 0x000000ff (eg, continuing to the 0x000000fe, 0x000000fd, and so on), every item from the first loop which fits the 0x000000ff criterion must be sorted and emplaced in the shared temporary target array.

As such, the gains from this design were minimal, and the performance losses were significant. Ultimately, this design was abandoned in favor of the standard serial implementation of this section of the algorithm. Figure 1 details the performance of this design which justifies discarding it.

Correctness and Validation

When developing these algorithms, validation of correctness of the sorted data was of paramount importance prior to diving deep into performance tuning. To support these goals, in addition to the provided logic, the Graph struct was extended with a copy_graph(*) function which made an exact copy, in memory, of a given graph. For each technique, an in-memory copy of the un-sorted Graph was created and sorted using the provided count sort code. As this function is in graph.c, it is not included with this submission – but it is not required, nor was it called, when gathering performance data on the various techniques outlined in this paper.

Also, a validation_run(*graph1, *graph2) function was created in main.c, which runs an exact comparison between the edgelists of two Graph structures in memory. This function checks to ensure that, at each location, the "src" and "dest" values match exactly between the two graphs. If there is a mismatch, a counter is incremented, and the number of mismatches at conclusion of the algorithm is reported in varying levels of verbosity based on programmer's choice. While this function is never called in the submission code, and was not called when gathering performance data, it is included in main.c.

When validating correctness of the various techniques, a copy of the graph as-sorted with the original count sort code was compared against the results of various radix sort implementations. Only radix implementations which provided consistent validation PASS results (zero mismatches) were permitted to be performance-analyzed and submitted.

Implementation Details

This section will explore the specific choices made and their justifications when exploring the OpenMP, MPI, and hybrid implementations of parallel radix sort.

OpenMP Parallel Radix

The OpenMP implementation investigated parallelization of the INIT, COUNT, and TRANSFORM loops, with a CRUSH synchronization step prior to the SORT step. The vertex_count array was transformed into a 2-d array, and each row was extended to permit $2^{radixSize} x \frac{numThreads}{bitbucket}$ elements to be simultaneously read/written to without synchronization. Nested parallelization was performed such that the COUNT step counts all radices at the same time, and given sufficient quantity of threads, multiple threads are assigned to each radix.

Notably, the INIT step was investigated both serially and in parallel, and it was discovered that the thread forking overhead of a parallel implementation overshadowed the gains from parallelization due to the small size of the data being addressed. As such, INIT was **specifically set to Serial** for improved performance. The COUNT and CRUSH steps are performed in nested parallel. The TRANSFORM step was implemented both serially and in parallel, with a tread working on working on all radices at the same time with a single thread per radix. The threading overhead overshadowed the quick execution time, and in the production version, TRANSFORM is also performed serially for (slightly) improved performance.

The CRUSH step was also investigated serially and in parallel, and it was determined that the serial case outperformed the parallel case due to thread forking overhead. Hence, this step is performed serially.

Finally, as discussed in the Design Considerations sections, a parallel SORT step was implemented and ultimately discarded due to poor performance. Hence, this step is performed serially – unfortunately, this is the most expensive step in the algorithm and continues to scale linearly. An analysis of the expense and scale of each step is discussed in the results section, as well as the effects of different thread quantities and radix sizes.

MPI Parallel Radix

MPI is a network protocol API which runs atop TCP or UDP (implementation dependent). Network transmission of data is inherently slow when compared to CPU-Memory and CPU-LocalPageFile, so the MPI protocol implementation was **specifically designed** to be as lightweight as possible. Notably, it is possible and, in many cases, recommended to load the data on one host and use MPI Scatter/reduce to transmit key points to remote nodes, and gather the results.

However, I chose to take a different route which takes advantage of all the available tools. Notably, the datasets are stored on the BeeGFS Parallel Filesystem in the ARC cluster, which provides multiple network entry points, and multiple disk spindles to handle heavy data read loads.

For this reason, my MPI implementation simultaneously loads the Graphs on all connected hosts using the usual code (having performed the **MPI initialization** at boot). Each MPI host is assigned one or more radix to work on, and only performs the INIT, COUNT, CRUSH, and TRANSFORM steps on the assigned radices. Then, at the synchronization point, each worker host transmits a **short MPI message using MPI_Ssend** containing the completed row of vertex_count which that process was assigned. The root process (which also counts one or more radices) waits to receive the missing data with **MPI_Recv**. Once the root process has a complete copy of vertex_count for all radices, the worker processes free their memory and exit. The root process completes the sort, and the rest of the BreadthFirst search algorithm.

Notably, this lightweight messaging protocol scales extremely well across a range of input sizes. For any given input size, from a few MiB to 15GiB, the algorithm need only transmit and receive $N * radixSize$ bytes of message. In the case of an 8-bit radix on two MPI hosts, only 2048B of payload need be transmitted during the sort operation, which easily fits in two TCP packets. Most of the heavy network traffic is handled in parallel in the pre-load steps. Further discussion is available in the results section, but this design choice takes advantage of the strengths of MPI (scalability) without susceptibility to the weaknesses (latency of excessive messaging).

Hybrid Parallel Radix

The hybrid solution combines the two techniques mentioned in the prior two subsections. Multiple MPI hosts load the data, and count the data for their assigned radices. The primary extension is, on each host, OpenMP is used to fork parallel processes to spread the load of the INIT, COUNT, and TRANSFORM sections. At the synchronization point, worker processes still transmit extremely lightweight messages to the root process to communicate the results of their assigned task, and exit. The root process finishes the sort task, performs the breadth first search, and exits. All of the conclusions and design choices which held true for the OpenMP section, and the MPI section, hold true and are implemented in this Hybrid solution.

Results

Datasets: Dense and Sparse Graphs

In early exploration of this project (prior to the production datasets becoming available), I developed code to generate my own datasets of arbitrary (large) size. In comparing results between early tests on small self-generated datasets and the provided small datasets, I noticed an interesting trend between the two types of set. The provided datasets tend to the "dense" in nature, where the largest vertex is much smaller than the number of edges in the graph. Conversely, the generated datasets (which create edges based on random numbers within the standard 32-bit int size range) hold vertices that are much larger (and greater in diversity) than the number of edges. While the differences between these sets are reduced for the radix sort implementation, the provided serial count sort suffers severe performance degradation on the sparse (generated) datasets because of the large sizes and traversals at the INIT stage.

These performance deltas were notable enough that all design techniques and test runs mentioned prior were tested both with the five provided datasets, and with five datasets I generated ranging from very small (2^{16} edges) to the bounds of what a standard int in the vertex_count array can hold (approximately 2^{30} edges). After confirming available space and quota, these new datasets were generated and stored on the BeeGFS Parallel Filesystem. The dataset sizes and relative sparseness (in terms of number of vertices) are illustrated in the table below, and discussion of the results across both types of dataset will follow in the subsequent sections.

NB: Investigation of the generated sparse datasets is unrelated to the context of the BFS task provided by the project architects. It is included and discussed here as a study of the general case of this parallel algorithm as it may apply to any given project.

Dense (Provided) Datasets			Sparse (Generated) Datasets		
Name	**num_edges**	**Largest Vertex**	**Name**	**num_edges**	**Largest Vertex**
WikiTalk	5.021×10^6	2.394×10^6	Sparse 2^{16}	6.554×10^4	1.242×10^9
Road	5.533×10^6	1.871×10^6	Sparse 2^{20}	1.049×10^6	1.678×10^7
Orkut	3.468×10^7	4.037×10^6	Sparse 2^{24}	1.678×10^7	1.242×10^9
LiveJournal	1.172×10^8	3.073×10^6	Sparse 2^{28}	2.684×10^8	1.242×10^9
GPlus	9.260×10^9	2.894×10^7	Sparse 2^{30}	1.074×10^9	1.845×10^8

Table 1: Datasets and sizes

OpenMP Parallel Radix vs Serial Count Sort

The openMP implementation was tested across both datasets, using a range of thread sizes (2 to 16 threads) and radix sizes (4, 8, and 16 bits). Once analysis of the now-discarded semaphore pipelined design was complete, all tests, and the results discussed below, were performed on the ARC Normal nodes. The results of the OpenMP test runs are displayed in Figures 6 and 7.

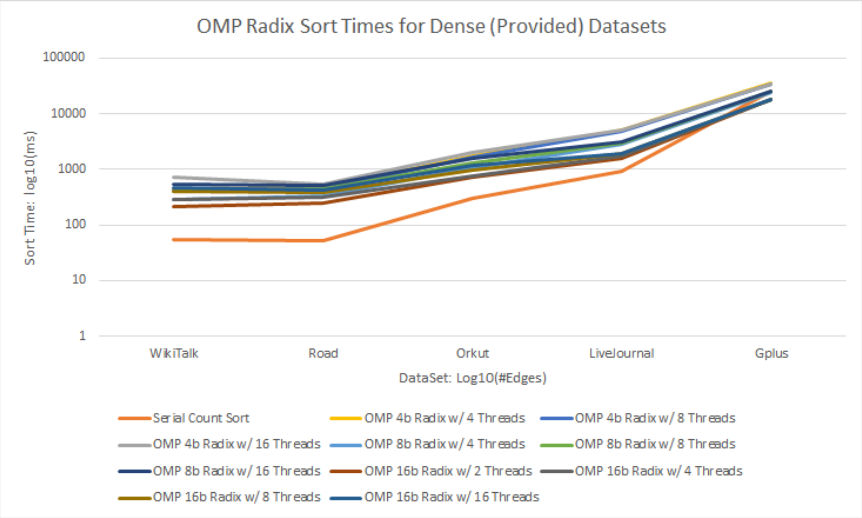

Figure 7: OpenMP Dense Results

Notably, in the case of the dense datasets, Serial count sort tends to outperform the openMP implementation for all but the largest datasets. This is primarily due to fixed initialization overhead based on threadsize, which tends to overshadow gains from paralellization until very large datasets are reached. As a further point, for the dense datasets, Count sort presents a cheaper initialization step due to the lower allocation

requirements for smaller "max" vertex sizes. Finally, I note that due to the final selection of a serialized SORT step, the larger radix sizes tend to overshadow the smaller sizes because the quantity of iterations through the SORT task is smaller.

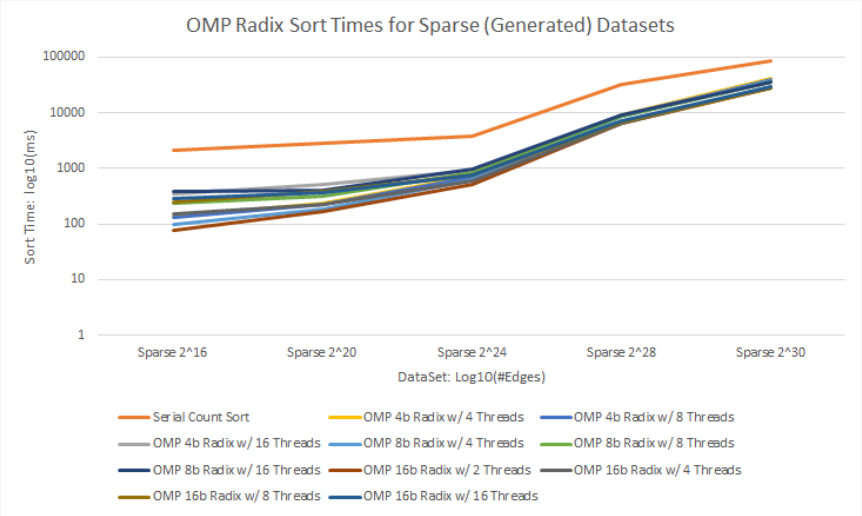

Figure 8: OpenMP Sparse Results

When examining the sparse datasets, I find similar trends across size, but significant improvements over Countsort for all sizes. Another interesting point is that I note a wider spread at small sparse dataset sizes, where the largest radix tends to far overshadow the gains from parallelization. As the size of dataset grows, the spread is narrowed. Again, I believe this to be due to a combination of larger initialization overhead across larger thread counts, and the effects of fewer SORT step iterations – particularly in the case of the smallest datasets.

MPI Parallel Radix vs Serial Count Sort

The MPI implementation was designed around, and deployed using, the **8-bit radix size**. This permitted 2 or 4 discrete MPI hosts to independently work on either two or a single radix before the synchronization step. Both configurations were deployed and tested on the ARC Cluster, using 2 or 4 ARC "normal" nodes. The results of these experiment runs are displayed in Figures 8 and 9. Please note that the lightweight MPI design was so fast and efficient, these plots use base-10 log of NanoSeconds, whereas other plots for OpenMP use base-10 log of Milliseconds.

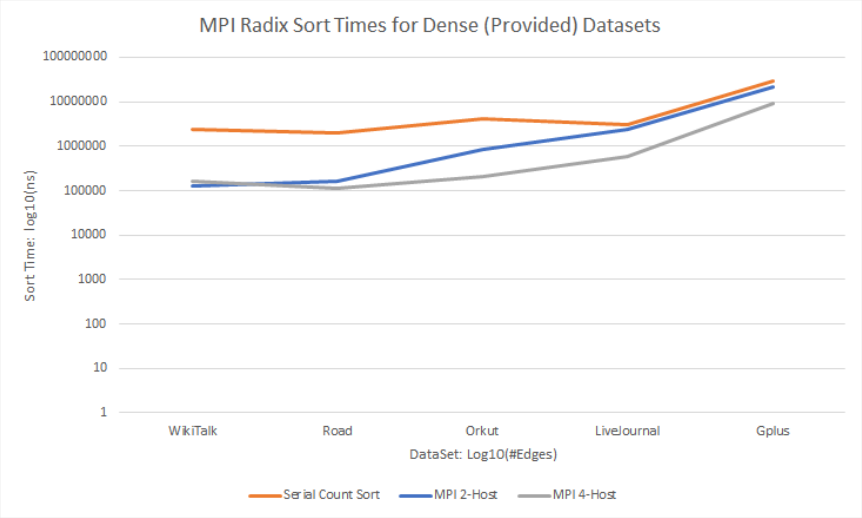

Figure 9: MPI Dense Results

Of immediate note is the fact that the lightweight MPI design performs well in comparison to Count Sort at all dataset sizes. I believe this is due to several key factors: chief among them is that the MPI implementation avoids the need for shared memory, and the additional overhead in forking threads and eliminating

synchronization between threads working on shared memory. Each process has it's exclusive cache and memory hierarchy, are there are no cache invalidations from other threads. Finally, the MPI design itself is specifically intended to keep network traffic to a minimum, and therefore to avoid network latency from excessive network traffic.

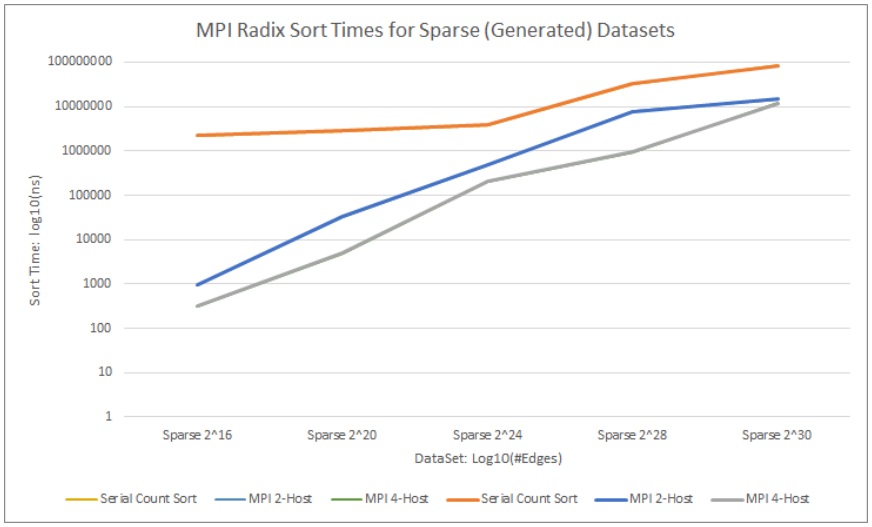

Figure 10: MPI Sparse Results

Notably, the MPI performance for the sparse graphs shows even greater improvement over the serial count sort. This is due primarily to the higher initialization requirements of Count sort as compared to the relatively smaller graph sizes of the smallest datasets.

Hybrid OpenMP+MPI Radix vs Serial Count Sort

The hybrid implementation was tested with an 8-bit radix size on two MPI hosts, and 8 threads per MPI host using OpenMP. This strategy combines the "best efforts" of both the individual OpenMP and MPI designs, namely, elimination of synchronization delays where possible, and extremely lightweight MPI protocols with parallel load-from-disk to reduce sort time. The results of the hybrid experiment runs are displayed in Figures 10 and 11.

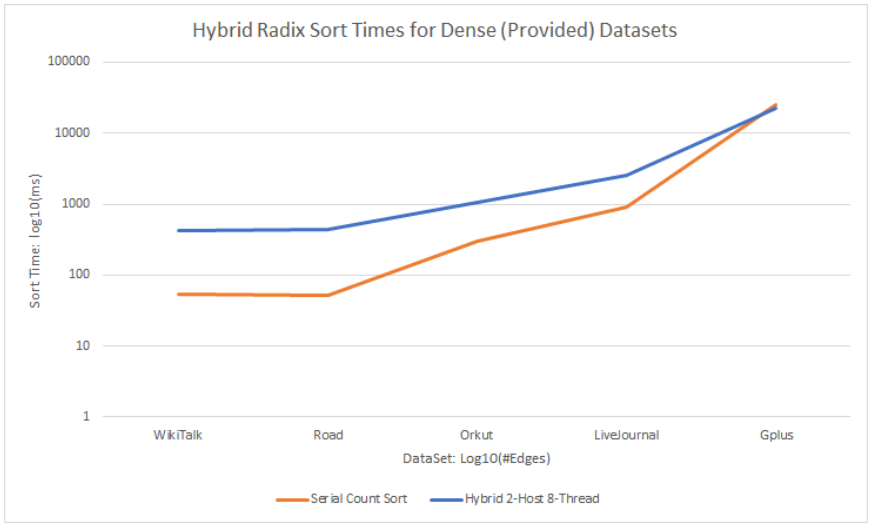

Figure 11: Hybrid Dense Results

Of note in the dense graphs, I find that inclusion of the same OpenMP strategies as in the OpenMP section introduces similar latencies for small-size dense graphs, with a crossover at the largest graph size where the hybrid technique overtakes the serial count sort. I believe the cause of this latency is threefold, including: introduction of higher initialization time to support the higher quantity of threads on each host, introduction of cache invalidations due to the shared memory on each host, and the significantly smaller latency introduced by thread forking.

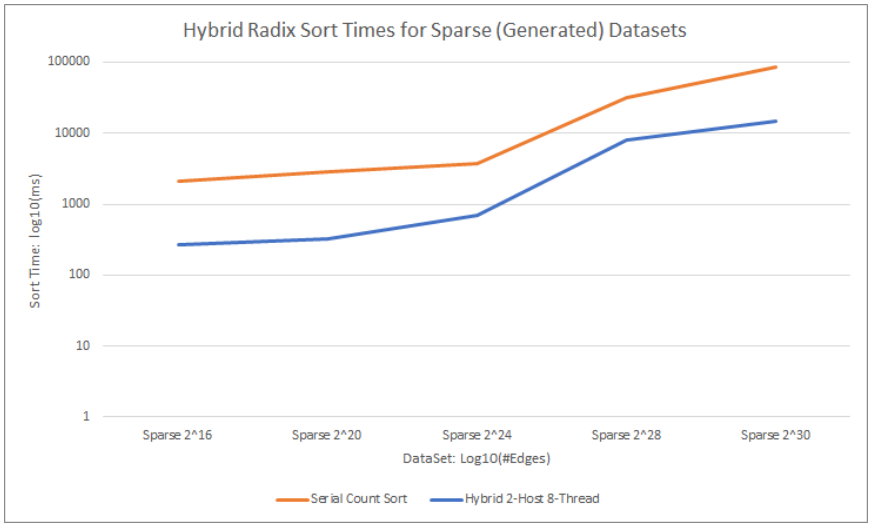

Figure 12: Hybrid Sparse Results

For similar reasons to the OpenMP section, I note that the sparse results present a higher speedup differential as compared to the dense results. This is primarily due to the algorithmic advantage of radix sort over count sort, where radix presents smaller initialization requirements, and will generally perform better across sparse graphs.

Conclusions and Methods Comparison

The experiment results from all three methodologies were compared and plotted. These plots are displayed in Figures 12 and 13.

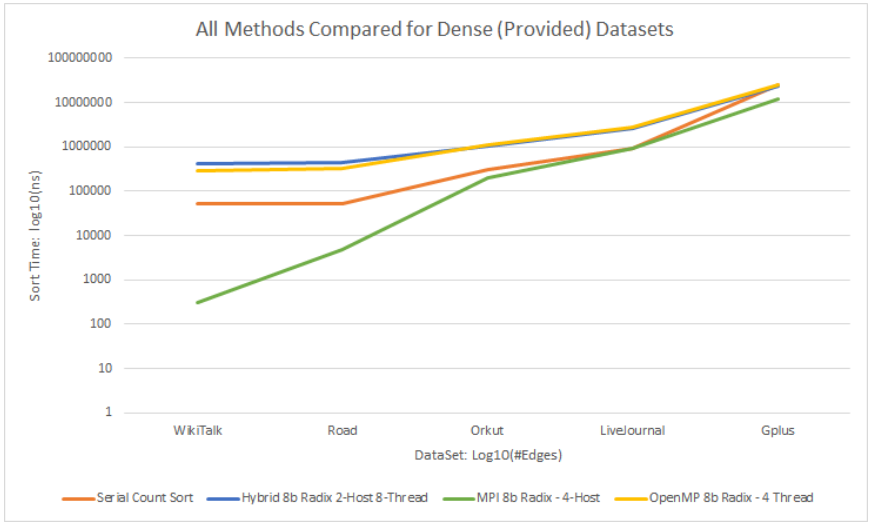

Figure 13: Method Comparison: Dense Results

In the dense case, I note that the lightweight MPI method is the clear winner as a result of eliminating thread overhead, low network traffic, and cache invalidations/thrashing from the multi-threaded approaches. Specifically, for the **smallest task, WikiTalk, the MPI implementation presents a 57.9% speedup** as compared to serial count sort. For the largest task, **GPlus, MPI presents 22.6% speedup** as compared to serial count sort. Notably, this holds true even when accounting for network latency, as these figures were gathered on discrete hosts connected via MPI+TCP/IP.

In the sparse case, I note expected similar trends towards MPI being the fastest implementation. Surprisingly, MPI is exceedingly effective at dealing with small sparse graphs in a timely fashion. The **medium sparse graph(Sparse 2^{24})** which is 3 times as large as the smallest dense graph, provides a **massive 87.3% speedup** as compared to the serial count sort. The speedup for the smaller graphs is even greater (approaching 98%).

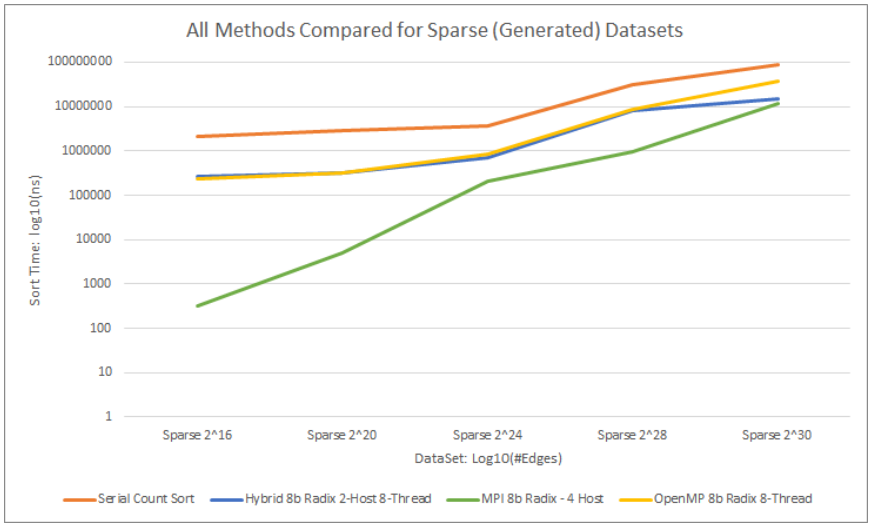

Figure 14: Method Comparison: Sparse Results

Broadly, I find MPI to perform the best, with Hybrid a close second, and OpenMP trailing. I acknowledge that this is likely due to shortcomings with the OpenMP implementation, and will present discussion on the facts behind this conclusion in the proceeding section.

Key Conclusion 1: Issues with SORT Task

As noted in the subtask analysis, the fact that the 3 methods analyzed (Synchronized, Unsynchronized, and Pipelined) for parallelizing the SORT task failed to produce faster results is the most serious issue with this implementation. The SORT task presents an exponential growth trend with no parallellization applied, and is also orders of magnitude more expensive than the other algorithm tasks. The second-most-expensive task, COUNT, was relatively-easily parallelized, and presents a linear growth trend as dataset sizes grow multiplicatively. As such, this task is the most interesting case for potential future study, as overcoming this growth pattern with parallel threading would yield deep rewards in algorithm speedup.

Key Conclusion 2: Improving OpenMP Efficiency

While the MPI design was highly effective at providing algorithm speedup, the two designs (OpenMP and Hybrid) that used OpenMP both suffered from issues with high overhead at the INIT stage and high overhead from managing the threads. Although I explored numerous strategies for parallelizing these stages, ultimately settling on a higher degree of manual thread control than simply appling Pragmas, I believe there is room for improvement in these areas of latency. This issue would be an excellent case for further study, in particular after gathering more insight on OpenMP "best practices" for efficient computation. Where possible, I did use static scheduling as I understand that to be the fastest scheduling strategy, but I believe there may be a case or procedure for forking off threads at the head of the sort function, and only terminating those threads at the TRANSFORM step.

Key Conclusion 3: Scalability Discussion

Another point for future exploration would be deeper scalability of this algorithm. While parallelization can provide some immediate speedup across small-to-medium datasets, the real power of a system like MPI is the ability to massively scale the operation across many hosts, which eliminates memory constraints encountered on a single host. I believe that, as the datasets are scaled from then tens-of-gigabytes into the hundreds of gigabytes, or terabytes, the trends will continue to show improvement over the serial count sort implementation. Unfortunately, there is a limit to the scale of this project (specifically, the limit of the 32-bit int which is used throughout the existing code). For this reason, my generated Sparse 2^{30} dataset was sized such that it comes very close to the maximum capacity of the 32b ints used in the counting arrays. An interesting further case study for MPI and Hybrid would be addressing very large datasets that go well beyond the scope of the sets in this study, and restructuring the algorithms to sort portions of the datasets on individual MPI hosts, and merge the resultant sorts at completion. Naturally, this investigation is beyond the scope of a simple class project.

Final Conclusions

The strategies detailed in this paper broadly explore the speedup and scalability of a parallel algorithm implementation when sorting a large graph. The project has provided solid context to dealing with the

architecture of parallel compupting, and cache coherence issues being discussed in lectures. While this project has provided an excellent introduction to OpenMP, MPI, and the NCSU CSC ARC HPC Cluster, there is clearly opportunity for more work on this implementation of the algorithm, to reach expected performance metrics.

Addenda

This project was compiled and tested on the NCSU ARC Cluster. Numerous implementation changes between the GCC version (5.x) on the ARC and those shipped with latest-update GCC (10.x) were observed; as such, compilation and testing is only guaranteed at this time for the NCSU ARC System.

Housekeeping Notes: All custom datasets discussed in this paper have been deleted from the BeeGFS parallel filesystem, as well as the directory under my Unity ID in BeeGFS.

References:

[1] **“OpenMP/MPI Reference and Tutorials,”** RookieHPC. [Online]. Available: https://www.rookiehpc.com/. [Accessed: 04-Oct-2021].

[2] **“OpenMPI Documentation,”** Open MPI documentation. [Online]. Available: https://www.open-mpi.org/doc/. [Accessed: 04-Oct-2021].

[3] **“Individual pragma descriptions,”** IBM. [Online]. Available: https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=pragmas-individual-pragma-descriptions. [Accessed: 04-Oct-2021].

[4] K. Zhang and B. Wu, **"A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess,"** 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 989-994, doi: 10.1109/H-PCC.2012.144.