Parallel Radix Sorting

Zan Wills  
James Weber

April 12, 2023

Abstract

The sorting of data has been a subject of considerable focus for a great many entities throughout history. The advent of computer systems to handle this data has not alleviated the need for sorting data, but has resulted in a need for more efficient and reliable sorting. The data each company uses for their individual business ventures has grown in scope, volume, and in focus as the ability to gather and store data has increased in this modern world-connected environment.

With that in mind, the focus of this report will be the application of parallel numerical sorting, focused on parallel radix sort and how sorting is applied by Arkouda, a Python API, powered by the Chapel programming language [1].

1 Background Information

* 1. What is a Sorting Algorithm  
       
      A sorting algorithm is a method for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest value or shortest-to-longest distance [2]. The sorting criteria is very dependent on the data and application of the data. The decisions on how to sort the data are specific to the users and designers of the algorithm that deal with the target data. Due to the inherent need to analyze each individual piece of data and placing it correctly within the entire set of data, in the manner that results in the desired ordering of the data, the algorithm can take a great deal of time and involve a large amount of processing time.   
      The most efficient sorting algorithms have a normal, or best, run time of . Meaning that it takes more CPU cycles to sort the data than the number of data items in the dataset. When dealing with large datasets where each item must be analyzed in order to determine its correct placement within the entire dataset, the required run time can grow, sometimes exponentially.
  2. Serial Radix Sort  
       
      Radix sort does a digit-by-digit sort, starting from the least significant digit to the most significant digit. The run time for radix sort can be as quick as O(n) if we have log2n bits for every digit [3].   
      Radix sort is not applicable for sorting all kinds of data. There are some key restrictions to radix sort that have to be considered. The data digits must be between a certain range. The input array elements must have the same radix and width. Sorting is done on an individual digit or letter position. Sorting starts from the rightmost position and a stable algorithm must be used at each position. Radix sort uses a temporary count array.  
       
      Radix sort is done by sorting on each position of the data, from least significant to most significant position. In base 10, radix sort would start with sorting the digits in the 1’s position, preserving like digit’s order during the sorting. Radix sort typically uses counting sort or bucket sort to perform the sorting operation on each digit place. Meaning that, for a four-digit number, counting sort will be called 4 times to sort the numbers based on the number in each of the digit places.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5283 |  | 5283 |  | 3746 |  | 7269 |  | 1497 |
| 7269 |  | 3746 |  | 1668 |  | 5283 |  | 1668 |
| 1668 | → | 1497 | → | 7269 | → | 4387 | → | 3746 |
| 3746 | Sort1 | 4387 | Sort2 | 5283 | Sort3 | 1497 | Sort4 | 4387 |
| 1497 |  | 1668 |  | 4387 |  | 1668 |  | 5283 |
| 4387 |  | 7269 |  | 1497 |  | 3746 |  | 7269 |
| Stable Sort used for each digit | | | | | | | | |
|  |  |  |  |  |  |  |  |  |
| 5283 |  | 5283 |  | 3746 |  | 7269 |  | 1668 |
| 7269 |  | 3746 |  | 7269 |  | 5283 |  | 1497 |
| 1668 | → | 4387 | → | 1668 | → | 4387 | → | 3746 |
| 3746 | Sort1 | 1497 | Sort2 | 4387 | Sort3 | 1497 | Sort4 | 4387 |
| 1497 |  | 1668 |  | 5283 |  | 1668 |  | 5283 |
| 4387 |  | 7269 |  | 1497 |  | 3746 |  | 7269 |
| Non-Stable Sorting can result in incorrect sort | | | | | | | | |

As shown in the illustration of radix sort, a stable sort is required to maintain the ordering done in the previous sorting step. Both sorts correctly sorted the digits place at each step, but the lack of a stable sort resulted in the final sorting of the number being incorrect, even though the sorting of the digit place, at each step, was correct.

* 1. Parallel Radix Sorting  
       
      Parallel computing can significantly speed up the sorting algorithm by dividing the data to be sorted among multiple processors to perform the sort on their part of the data concurrently. While this idea is attractive, especially when dealing with the time-consuming process of sorting, not all sorting algorithms are equally suitable for parallelizing the process.   
       
      Parallel radix sort is a non-recursive algorithm that follows the digit-by-digit strategy. It works by sorting the data based on the least significant digit first, and then moving to the next more significant digit, until all digits are processed. Parallel radix sort can use different methods to sort each digit, such as counting sort or bucket sort. Parallel radix sort can achieve a high throughput by exploiting the regularity and locality of the data. Parallel radix sort is stable, meaning that it preserves the relative order of equal elements [4].   
       
      In terms of complexity, parallel radix sort has a complexity of time and depth, where d is the number of digits. In terms of communication, parallel radix sort requires messages and data, where d is the number of digits and p is the number of processors.   
       
      In general, parallel sorts consist of various rounds of the serial sort, known as local sort, performed in parallel by each processor, followed by key mobility among processors, known as the redistribution step [5]. Parallel radix sort can look like the steps illustrated here. Step 1 divides the data evenly between the processing nodes, in this case 12 numbers divides into 3 processors. Step 2 performs a local sort on the ones place of the numbers. Step 3 redistributes the data to the processor needing the data, based on the number in the one’s column. Step 4 performs another local sort on the data, but on the 10s column. Step 5 redistributes the data to the needed processor, based on the numbers in the 10s column. The steps for local sort and redistribution would repeat until all of the number columns were processed and redistributed.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| {45, 26, 64, 83, 91, 72, 67, 57, 17, 81, 79, 35} | | | | | | | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 45 | 26 | 64 | 83 |  | 91 | 72 | 67 | 57 |  | 17 | 81 | 79 | 35 |  |  |
|  | ↓ | |  |  |  | ↓ | |  |  |  | ↓ | |  |  | Local Sort |
| 83 | 64 | 45 | 26 |  | 91 | 72 | 67 | 57 |  | 81 | 35 | 17 | 79 |  |  |
| |  | | --- | |  | |  |  |  |  |  |  |  |  |  |  |  |  | |  | | --- | |  | |  | Redistribute |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Data |
| 91 | 81 | 72 | 83 |  | 64 | 45 | 35 | 26 |  | 17 | 67 | 57 | 79 |  |  |
|  | ↓ | |  |  |  | ↓ | |  |  |  | ↓ | |  |  | Local Sort |
| 72 | 81 | 83 | 91 |  | 26 | 35 | 45 | 64 |  | 17 | 57 | 67 | 79 |  |  |
| |  | | --- | |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Redistribute |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Data |
| 17 | 26 | 35 | 45 |  | 57 | 64 | 67 | 72 |  | 79 | 81 | 83 | 91 |  |  |

The resulting data is ordered and can be distributed to be stored however required.

* 1. Serial Sort used within Parallel Radix Sort  
       
      There are two main sorting algorithms that are used with Radix sort. Counting sort and Bucket sort. These sorting algorithms are used due to their efficiency when the data being sorted has a limited range of possible values.  
       
      Counting sort is an efficient algorithm for sorting an array of elements that each have a nonnegative integer key, for example, and array, sometimes called a list, of positive integers could have keys that are just the value of the integer as the key, or a list of words could have keys assigned to them by some scheme mapping the alphabet to integers (to sort in alphabetical order, for instance) [6]. Here are the steps of counting sort:

1. Find the max value in the array (array A).
2. Create an array (array C) of length max + 1, setting all elements to 0.
3. Store the count of the elements in the original array in the index of their value in the new array.
4. Iterate through the new array and store the cumulative count at each index.
5. Iterate through A, find the value at C[A[i]], subtract 1 and store the value at A[i] in the sorted array (array B).
6. After placing each element in step 5, subtract 1 from the value in C[A[i]].

Bucket Sort is a sorting algorithm that divides an array’s elements into several buckets. The buckets are then sorted one at a time, whether by using a different sorting algorithm or by recursively applying the bucket sorting algorithm. In the case of radix sort, each bucket sort is merely identifying the bucket for each digit place. Since digits are from 0 to 9, we have 10 possible buckets that each number will be placed into. Since placing the numbers into their respective buckets also sorts them based on the current digit position, there is no more sorting needed. The order of placement into the buckets is retained as each bucket sort is executed. As each digit position is bucket sorted, the entire array will be sorted when all digit positions go through bucket sorting.

1. Arkouda
   1. Exploratory Data Analysis [7]

Exploratory data analysis (EDA) is a prerequisite for all data science, as illustrated by the ubiquity of Jupyter notebooks, the preferred interface for EDA among data scientists. The operations involved in exploring and transforming the data are often at least as computationally intensive as downstream applications (e.g. machine learning algorithms), and as datasets grow, so does the need for HPC-enabled EDA. Arkouda allows a user to interactively issue passively parallel computations on distributed data using functions and syntax that mimic NumPy, the underlying computational library used in the cast majority of Python data science workflows. The computational heart of Arkouda is a Chapel interpreter that accepts a pre-defined set of commands from a client (currently implemented in Python) and uses Chapel’s built-in machinery for multi-locale and multithreaded execution. Arkouda has benefited greatly from Chapel’s distinctive features and has also helped guide the development of the language.

Arkouda breaks the shared memory paradigm and scales its operations to data frames with over 200 billion rows, maybe even a trillion. In practice we have run Arkouda server operations on columns of one trillion elements running on 512 compute nodes. This yielded a >20TB data frame in Arkouda.

* 1. The Arkouda Sort

The core of the sorting algorithm is a parallel sort written in the Chapel programming language. The code performs a stable, least significant digit, radix sort. The sorting algorithm can handle non-uniform data that can be communication intensive [8].   
  
 The sort is performed by a serial process of identifying the bucket for each integer position through analysis of the binary representation of the integer. The sorting of the integers, based on their identified buckets, is performed in a parallel bucket sort. The sort leverages the idea of chapel locales and tasks to partition the bucket sort among parallel processes or nodes. A locale, in the chapel programming language and in the context of the x86 CPU architecture, is a single node of a multi core node architecture [9], and a task is a thread or process. The bucket sort is parallelized among the available locales and further parallelized among the available tasks on each locale.  
  
  
  
 The entry point for the Arkouda sort is the Radix Sort Least Significant Digit Core function. The function begins by generating a counting block domain to account for the total buckets across all parallel processes, locales, and tasks. Let this count be . The resulting domain counts from . This domain is cast to an array of integers for the global counts used for all locales and tasks.  
  
 The main loop of the function sorts the radix of each number by iteratively right bit shifting. For a given digit, the code parallelizes to the available locales, then again to the available tasks for each locale. Each task generates a two-dimensional array, which contains an array of buckets for each task. The task then retrieves a reference to its block of the overall array. The lower bound of each task’s block is determined by multiplying the task number by the number of times a list can be divided by the number of tasks, then adding the lower bound of the task’s subdomain. The upper bound of the domain is calculated by adding the sub division size to the calculated lower bound. Each task’s subdomain is calculated on each locale via the aD.localSubdomain() function.  
  
  
 Once the buckets on each task’s subdomain have been properly calculated, each task calls the destination aggregator in parallel to copy their respective local counts to the global counts array. The destination aggregator copy() function allows the local data to be copied to a global buffer. For each bucket in the local buckets, the bucket is mapped to a global count index using the formula   
   
which is implemented by the calcGlobalIndex() function. When complete, the aggregator is then flushed to apply the result to the global counts buffer.  
  
 Using the values in the global counts array, which enumerate the counts of the bucket set from each task on each locale, the code generates a permutation, in parallel, which dictates the values to place into each index of the sorted list. Every task, across all locales, uses a source aggregator copy() function to copy its values from the corresponding global index using the calcGlobalIndex() function discussed previously. In the next parallelizing thread section, each thread gets a reference to its bucket within the taskBucketPosition array. The local subdomain and block, within the global array, is calculated again on each thread in the same manner, using the localSubdomain() and calcBlock() functions. Using the calculated block range, each task uses a destination aggregator to copy each of the values within its domain to the proper bucket in the global list ‘a’.

* 1. Code Performance

To evaluate the runtime performance of the Arkouda radix sort, the library was built and installed on an x86 Linux machine running Ubuntu 20.04. The processor used was an Intel i7-6700k with 4 physical and 4 logical cores and 32Gb of DDR4 memory. In the chapel programming language, this testing set up correlates to one locale, with a max of eight tasks.

To contextualize the performance of the Arkouda radix sort, we implemented a serial radix sort in C++ [10] using the standard template library (STL). The purpose of using C++ and the STL was not only to compare the parallel sort to a serial version, but also to compare a program written in the specialized chapel language to a program written with common language and employing standard libraries. Further context is provided via a parallel quicksort implemented in C with the Open Message Passing Interface (MPI) [11]. This allows comparison between the parallel radix written in chapel and another well know sort fundamentally different from radix sort, implemented with a well-established language and parallelization paradigm. Runtime performance was tested on three different, independent variables; number of threads, list size, and number of digits per integer. Each independent variable was varied by hard coding different values in the driver files and changing runtime environment variables.

A graph with a line

Description automatically generated with low confidence

*Plot 1*

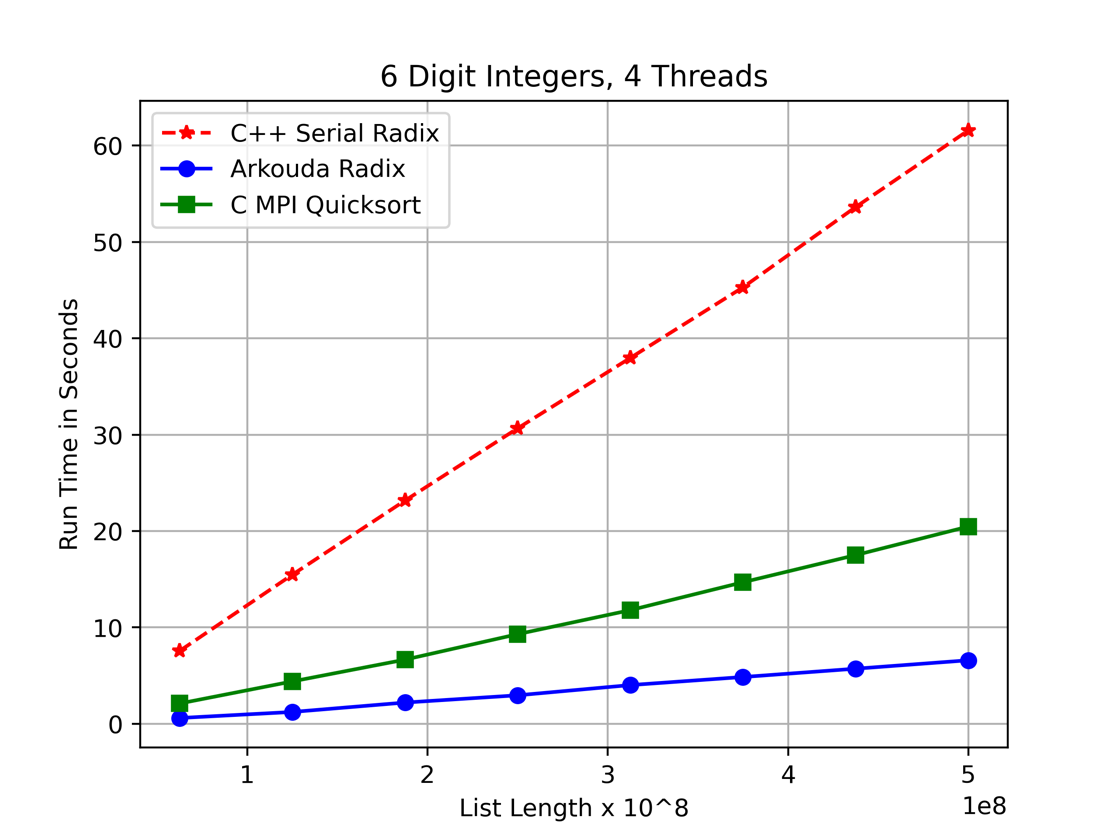
Plot one shows the run time of the Arkouda radix sort on a list of 500 million, six-digit integers with respect to the number of threads deployed by the sort. Adding threads up to the number physical cores on the processor (4 cores), results in an exponential decrease in run time. Adding a fifth thread, causes a small spike in runtime, as the thread is mapped to a logical core, causing a physical core to thrash between the two processes, forcing a process to wait and lagging communication. The addition of a sixth thread, up to eight, minimizes thrashing and sees a slight decrease in runtimes which can be attributed to the suppression of main memory latency stalls via the logical cores.

A picture containing text, line, diagram, plot

Description automatically generated

*Plot 2*

Plot 2 shows the same run time of the Arkouda radix sort plotted in plot 1, in the context of the runtime of the C++ serial radix sort and the MPI parallel quick sort. The serial radix and parallel quick sort were run with a list of 500 million 6-digit integers. The plot demonstrates that the Arkouda radix sort far outperforms the other two sorts. On average, the parallel radix sort is over six times faster than its serial counterpart, and over twice as fast as the parallel quick sort. Further, comparing the three sorts running on one thread, the chapel implementation of the radix sort outperforms the implementations using established industry tools and languages.



*Plot 3*

Plot 3 shows the run time with respect to the list size. The benchmark was performed with 4 threads on lists with 6-digit integers. The list sizes started at 62.5 million elements then increased by 62.5 million for eight list sizes up to 500 million. Despite the Arkouda radix and the quick sort both being parallel, the runtimes diverged as the list length increased, with radix outperforming quick sort.

A picture containing text, line, diagram, plot

Description automatically generated

*Plot 4*

Plot 4 shows the runtime with respect to a varying number of digits per integer. The benchmark was run with 4 threads and a list size of 500 million. As expected, the run time of the serial radix sort grows linearly with respect to the number of digits in each integer. The parallel radix sort’s runtime has two distinct steps at 5 and 9 digits, with the runtime remaining constant in between. The x86 architecture supports 16-bit short integers and 32-bit integers. The max value of a signed 16-bit integer is 32,767, which falls in the first quarter of the range of a 5-digit integer. The Arkouda radix sort performs a bit-wise sort of the numbers, for roughly three quarters of the five-digit range, the sort must look at 32 bits rather than 16 which doubles the work compared to the lists of numbers with less than 5 digits. Of particular interest though is the run time of the parallel quick sort with respect the varying digit amounts.

The parallel quick sort would not complete on a list 500 million numbers with less than 4 digits. The 5-digit sort took 53 seconds, and the 4-digit sort took 404 seconds, forming a vertical asymptote at 4-digit numbers. Starting at 6-digit number though, the sort’s run time hits a horizontal asymptote, which is expected from a quick sort given that the sort is not directly dependent on the number of digits in the list elements. The exponential increase in runtime on numbers with less than 4 digits can be attributed to the ratio of possible element values to the number of elements increasing the probability of quick sort picking a bad pivot. With each additional digit, the range of numbers increases by a power of which exponentially decreases the chances of a pivot being picked that would cause the majority of the current sub list to go to one side of the partitioning. When quick sort picks bad pivot, one side of the partition gets more data than the other, driving the run time towards . This phenomenon explains the asymptotic run time of the quick sort on the small number ranges corresponding to numbers with less than 5 digits. Radix sort has an advantage over quick sort when the data has a small range of values and a large number of elements.

A picture containing text, line, plot, diagram

Description automatically generated

*Plot 5*

Plot 5 shows the performance of all three sorts on 5-digit integers as opposed to the 6-digit integers used in plot 2. Both the Arkouda parallel radix sort and the C++ serial radix sort perform well, whereas the quick sort struggles with pivot selection and on average performs worse than the serial radix sort.

1. Conclusion

Radix sort’s bounded linear runtime makes it an ideal sort for sets which have a large ratio between the number of elements and the number of digits in the elements. As this ratio decreases, the product of d and n in decreases, which translates to a lower run time as on numbers with less digits. As seen in plots 1 – 5, the Arkouda parallel sort outperforms the other sorts used for comparison, even when run on a single thread without parallelism. The sort even exhibits relative resistance to runtime growth with respect to increasing the number of digits in the numbers being sorted, and only increases marginally at the boundary between 16- and 36-bit integers.

Per the results of testing and code analysis, there are 3 main factors which give the Arkouda sort its uniquely high performance. The first of these factors is that the sort is performed bitwise rather than digit-wise. Not only does this greatly reduce the possible values of d, stabilizing the runtime growth with respect to the values of the numbers being sorted, but also reduces the overhead of comparisons at the assembly level. The second factor which gives the Arkouda radix sort is extraordinary performance is the implementation’s use of memory. The sort is performed in-place, and the code only declares one data structure with significant size, the global counts array, which is reused. Further, the lack of data allocation allows the program to leverage spatial locality and cache usage. In comparison, the STL queue utilized in the serial radix sort allocates non-contiguous heap memory for each element entered which makes for poor locality. Additionally, the MPI library use in the parallel quick sort deploys entirely separate processes so there is no spatial locality. The final factor which makes the Arkouda radix sort so performant is the chapel programming language and complier. Even running on a single thread, the chapel radix sort performed over six times faster than the serial radix sort implemented with the C++ STL. This jump in performance can be attributed to optimizations which exist in chapels libraries, or those optimizations which are made by the chapel compiler.

# References

|  |  |
| --- | --- |
| [1] | M. M. a. W. Reus, "Arkouda," 2020. [Online]. Available: https://bears-r-us.github.io/arkouda/#. |
| [2] | "Sorting Algorithm," October 2017. [Online]. Available: https://www.techtarget.com/whatis/definition/sorting-algorithm. |
| [3] | "Radix Sort," Geeks for Geeks, 12 April 2023. [Online]. Available: https://www.geeksforgeeks.org/radix-sort/#. |
| [4] | "Parallel Quick Sort and Paralled Radix Sort," LinkedIn, 5 April 2023. [Online]. Available: https://www.linkedin.com/advice/1/how-do-you-compare-contrast-parallel-quick-sort. |
| [5] | "Parallel Radix Sort Algorithm," Medium.com, 14 November 2021. [Online]. Available: https://himnickson.medium.com/parallel-radix-sort-algorithm-using-message-passing-interface-mpi-31b9e4677fbd. |
| [6] | K. Moore, "Counting Sort," Brilliant, [Online]. Available: https://brilliant.org/wiki/counting-sort/. |
| [7] | "Bears-R-Us," Arkouda, [Online]. Available: https://github.com/Bears-R-Us/arkouda. |
| [8] | Arkouda, "Arkouda.sorting," [Online]. Available: https://bears-r-us.github.io/arkouda/autoapi/arkouda/sorting/index.html. |
| [9] | "Chapel Documentation/Locales," Hewlett Packard Enterprise Development LP, 2023. [Online]. Available: https://chapel-lang.org/docs/primers/locales.html. |
| [10] | Z. Wills, 5 2023. [Online]. Available: https://github.com/wwills2/Radix\_Sort/blob/master/SerialRadix.cpp. |
| [11] | J. Weber, 4 2023. [Online]. Available: https://github.com/wwills2/Radix\_Sort/blob/master/jw\_parallel\_quick\_sort.c. |