Quartile and Outlier Detection on Heterogeneous Clusters using Distributed Radix Sort

Kyle L. Spafford Jeremy S. Meredith Jeffrey S. Vetter
Future Technologies Group
Oak Ridge National Laboratory
1 Bethel Valley Road, Oak Ridge, Tennessee
spaffordkl, jsmeredith, vetter @ ornl.gov

***Abstract*—In the past few years, performance improvements in CPUs and memory technologies have outpaced those of storage systems. When extrapolated to the exascale, this trend places strict limits on the amount of data that can be written to disk for full analysis, resulting in an increased reliance on characterizing in-memory data. Many of these characterizations are simple, but require sorted data. This paper explores an example of this type of characterization–the identification of quartiles and statistical outliers–and presents a performance analysis of a distributed heterogeneous radix sort as well as an assessment of current architectural bottlenecks.**

***Keywords*-sorting, GPUs, performance analysis**

I. Introduction

One of the primary tools of descriptive statistics is the median, a numeric value which separates a distribution into upper and lower halves. When this approach to segmenting data is expanded by adding two additional points–one at the 25th percentile and the second at the 75th percentile–the three points are known as quartiles. More specifically, these two new points are known as the *lower* and *upper* quartiles, respectively.

Quartiles capture fundamental information about a dataset's distribution that complements other traditional metrics like the mean, mode, and standard deviation. Their use in data analytics has become widespread and is particularly evident in non-parametric characterizations of data such as the famous Bowley five-number (i.e. quartiles, minimum, maximum) and seven-number (first and last deciles) summaries. However, most will be familiar with quartiles from the ubiquitous Tukey boxplot, a well-known visualization in quantitative analysis, e.g. Figure 1, used primarily for comparing distributions across multiple data sets.

The second major feature of the boxplot is the outlier, any data point that markedly deviates from the others. In a quantitative sense, outliers are usually defined as points which lie more than 1.5x the interquartile range (the difference between upper and lower quartiles) below the lower quartile or above the upper quartile. This traditional definition is adopted in this work. More practically, outliers often represent an interesting observation or erroneous result. In this sense, quartiles and outliers provide one avenue for understanding and summarizing large datasets.

An important motivating application for this work is understanding data at the exascale, where it is anticipated that the amount of data produced by simulations will far exceed the bandwidth to long-term storage [1]. Instead of saving an entire dataset, it may be feasible for an application to only store descriptive information such as the five-number summary and outliers. Or, more advanced techniques could store an adaptive sample informed by these properties. In particular, this work targets data being produced by a parallel scientific application. The analysis focuses on problem sizes that are much larger than the memory of a single node. Also, the initial data is assumed to be resident in memory (i.e. just calculated), but distributed among many nodes.

Traditionally, this class of problems has not received as much attention as those that cannot fit into memory (dubbed external sorting and well covered by the database literature) or those that can fit into the memory of a single processor, which has been exhaustively covered by the architectural community.

A. Identification of Quartiles and Outliers

Several algorithms exist for the identification of quartiles and outliers, and these algorithms are sometimes embedded in other analyses such as histogramming. However, in many cases, data is first sorted to identify quartiles (and as a pre-processing step for further analysis). One notable exception is Tarjan's median-of-medians algorithm [2], although this approach tends to be less efficient when performing multiple analyses since each quartile must be found serially.

Sorting is one of the most studied algorithms in computer science, and a well-tuned sorting implementation can be leveraged to efficiently calculate quartiles and outliers. In this approach, depicted in Figure 1, the high level algorithm becomes:

1) Sort the data
2) Calculate the quartiles (Q_1, Q_2, Q_3) based on the number of observations. In cases where there is an even number of points (e.g. Figure 1), the quartile is

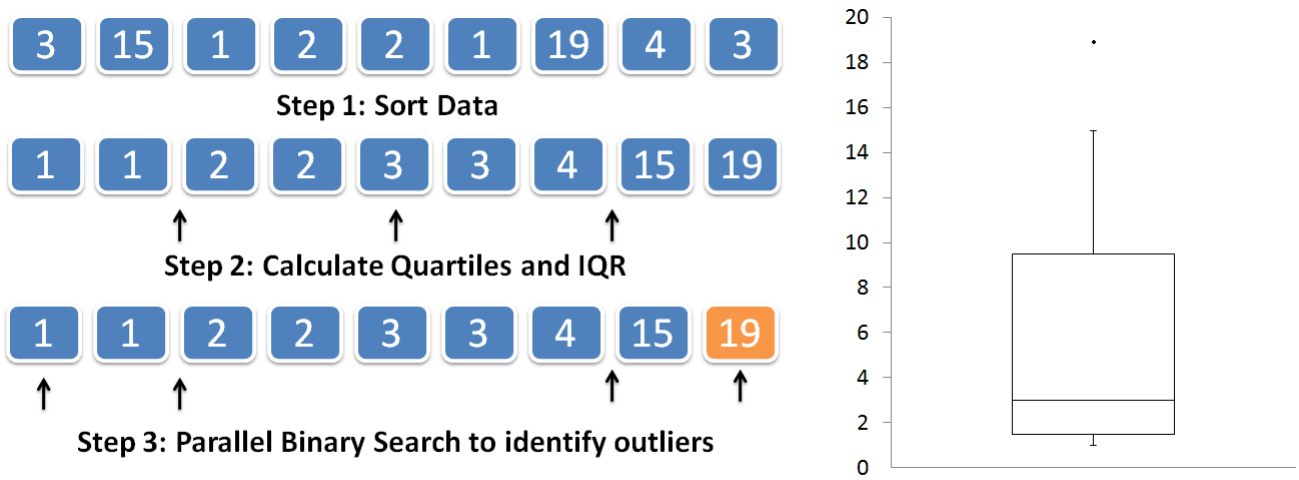

Figure 1. An example of the basic algorithm is shown on the left. First, data is sorted. The quartiles are then identified based on their position in the sorted dataset. The quartiles for this example are 1.5, 3, and 9.5 (the average between the two relevant points is used when there are an even number of samples). Finally, a parallel binary search is performed from the quartiles to the end of the dataset to identify outliers. Results for this dataset are shown in an example boxplot on the right, with one outlier (19). Whiskers (at 1 and 15) indicate the minimum and maximum observations among non-outliers.

defined as the arithmetic mean between the two most central observations.

3) Calculate the interquartile range by subtracting the lower quartile from the upper quartile, $R_{iq} = Q_3 - Q_1$.
4) Perform a binary search between the lowest observation and the lower quartile for the greatest element that is sufficiently less ($1.5R_{iq}$) than the lower quartile. This and all lower elements are outliers.
5) Repeat step four on the upper portion of the data, finding elements which are sufficiently greater than the upper quartile.

Sorting, then, is the dominant component of the algorithm. Steps 2 and 3 are accomplished in constant time, and the two binary searches can be performed in parallel. Binary search is bounded at $O(logn)$, although each search only covers one fourth of the data.

B. Sorting Primitives

While a full review of sorting is beyond the scope of this paper, we briefly note that Leischner recently analyzed the performance of sample sort on a single graphics processor [3], documenting the fastest known in-memory, comparison-based sorting. Merill et al. improves upon this result for some data types by using a radix sort [4]. A radix sort is a specific type of distribution sort that exploits the binary representation of data to sort in $O(n)$ time. Our distributed approach exploits the fast GPU parallel primitives identified by Merrill (e.g. reduce, prefix sum) involved in radix sort and extends them to a distributed environment.

II. Approach

Before presenting the approach in detail, it is important to note several constraining assumptions. This work is primarily concered with performance phenomena due to heterogeneity and data is assumed to be resident in GPU memory (that is, just calculated by the accelerated portion of a scientific application). Therefore, complexity arising from I/O, a common bottleneck for sorting large data, is not considered. For the state-of-the-art in external sorting, we refer the reader to Triton Sort, the winner of the 2011 Sort Benchmark competition [5]. Second, individual nodes are assumed to begin with the same number of keys. This is a common scenario based on the motivating example of a scientific simulation, and, in any case, work can be easily re-balanced in our approach during the first key exchange.

A. Distributed Sorting Algorithm

Radix sort is an iterative process which repeatedly partitions data, or keys, in a stable fashion, r bits at a time. These bits are known as a digit or radix and r is known as the radix width. r is usually fixed, and, given an input of n k-bit keys and values, radix sort will require $\frac{k}{r}$ passes. We consider the parallel case with a number of processors, p. p is, more specifically, the number of shared memory processing domains (in our case, a GPU and CPU core). The proposed algorithm is a reasonable extension of serial radix sort and uses an explicit counting method.

Algorithm 1: LSB Distributed Radix Sort

For each r-bit digit, starting with the least significant:

1) On each processor, compute a local histogram, H_{Local}, which counts the occurrences of each of the 2^r possible values of the current digit.
2) Perform a global, exclusive scan across all ranks' histograms to compute $H_{PriorRank}$. For each i (where $0 \leq i < 2^r$), $H_{PriorRank}[i]$ now contains the number of occurrences of keys with digit value i on all "lower" ranks.
3) On the highest rank GPU, perform a local exclusive scan on $H_{Local}+H_{PriorRank}$ to compute $H_{PriorDigit}$. $H_{PriorDigit}[i]$ now contains the number of occurrences of all keys with digit value less than i across all ranks.
4) Broadcast $H_{PriorDigit}$ to all ranks. Within each rank, $H_{PriorRank}[i] + H_{PriorDigit}[i]$ is now the global position where that rank will start placing any keys with digit value i.
5) As each rank may have more than one key containing digit value i, we locally calculate C_{Digit}, an accumulating count of the local occurrences of keys with value i.
6) Explicitly calculate each key's final global position. For the jth key, whose digit value is i, its destination position is $H_{PriorRank}[i]+H_{PriorDigit}[i]+C_{Digit}[j]$.
7) Perform a global key exchange and a local sort to place each key in its calculated final position.

Distributed sorting algorithms tend to be replete with tradeoffs, and radix sort is no exception. This variant has some extremely desirable properties, including the linear $O(n)$ asymptotic bound for work, provided r is fixed. This approach also exhibits a high degree of parallelism–each key can be processed in a data parallel fashion, and synchronization among processors is only required for scans during the indexing computation and the key exchange. The depth of each pass is also fixed, with a $log_2 \frac{n}{p}$ term for the local histogramming, $log_2 n$ for the global scan, $log_2 p$ to scan process totals, and finally, a constant depth of one for the exchange itself.

Furthermore, the choice of the radix width provides an important "knob" to decrease the number of passes. In effect, this decreases total depth, increases arithmetic intensity, and decreases communication among processors. The major drawback to distributed radix sorting is its high requirements for bandwidth–both for memory and for the interconnect. In each pass, keys must be read from memory once and, in the worst case, may move globally in each iteration (although this is data dependent). Depending on the width of the key and radix, this could result in substantially more data movement than a comparison-based method such as merge sort or sample sort.

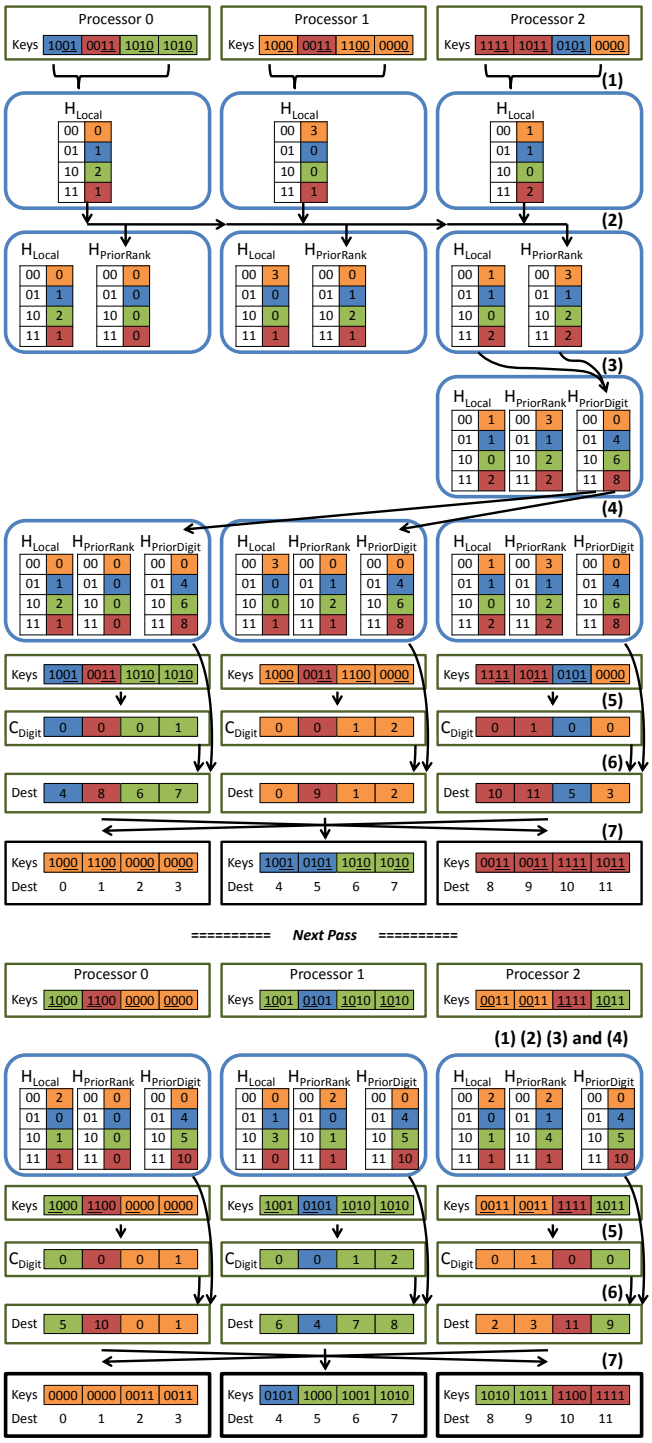

Figure 2. The full sorting algorithm. This example shows the steps in sorting twelve four-bit keys distributed among three GPUs using a radix width of two bits per digit. Due to the given radix width, local storage for each histogram array will be four (2^r) entries, and two passes ($\frac{4}{r}$) are will be required for four-bit keys. Note that in this figure, the second pass condenses steps 1-4 to conserve space. A detailed description of each step enumerated in this figure is in Algorithm 1.

And, as the disparity between CPU throughput and memory bandwidth increases, these data movement constants play an increasing role in the actual runtime of sorting alorithms. Indeed, Satish et al. make projections that, for some key/radix configurations, radix sort should be abandoned in favor of merge sort [6] for the foreseeable future for single node sorting. It remains to be seen, however, if radix's sort guaranteed (but high) bandwidth requirements will be preferable over the decreased parallelism of merge sort at the exascale. In other words, for large enough n, the merges used by comparison-based methods may have similar bandwidth requirements to radix sort.

B. IQR Computation and Outlier Identification

Compared to sorting, the requirements for interquartile range computation and outlier identification are quite low. Once data has been sorted, quartiles and the IQR can be computed in constant time and then broadcasted to all processors, adding roughly $logp$ steps to the total depth. Following the broadcast, both binary searches can be done in parallel. Since the binary searches contain both task-level and data-level parallelism, the depth of the entire algorithm is increased by only $log_2 \frac{n}{4p}$.

III. Evaluation

We evaluate the performance of the proposed approach for quartile detection on the NSF Track2D Keeneland System [7], a heterogeneous cluster comprising Intel Westmere CPUs and NVIDIA Fermi GPUs. More details of Keeneland's configuration are available in Table I. The prototype implementation is written in Thrust, a C++ template library for productive GPU programming similar to the Standard Template Library, and MPI. The implementation operates on 64-bit key-value pairs and and takes the radix with, r, as a parameter. Results are presented using Thrust 1.3.0, CUDA 3.2, and MVAPICH2 1.6rc2 running in CentOS 5.5. Our evaluation aims to characterize effects from data movement and identify current architectural bottlenecks. Since the sorting step strongly dominates runtime (and may be of wider interest to the community), we provide its analysis first.

A. Experimental Platform: Keeneland

Figure 3 shows the Hewlett Packard SL390, the architecture for a Keeneland node. This dual socket system based on Intel Westmere processors features a dual I/O hub design, which provides a full sixteen lane PCIe 2.0 connection to three NVIDIA Tesla M2070 GPUs (codenamed Fermi). Keeneland nodes are connected using single-rail QDR Infiniband, in a fat-tree topology. Theoretical peak unidirectional bandwidths for each of the SL390's datapaths are shown in Figure 3.

Node Architecture	HP SL390
CPU	Intel Xeon X5660 (Westmere)
CPU Frequency	2.80 Ghz
CPU cores per node	12
Host memory per node	24 GB
GPU Architecture	NVIDIA Tesla M2070 (Fermi)
GPUs per node	3
GPU memory per node	18 GB (6GB per GPU)
CPU/GPU ratio	2:3
Interconnect	InfiniBand QDR (Single rail)
Total Number of Nodes	120
Total CPU Cores	1440
Total GPU Cores	161,280

Table I
Keeneland Hardware Details

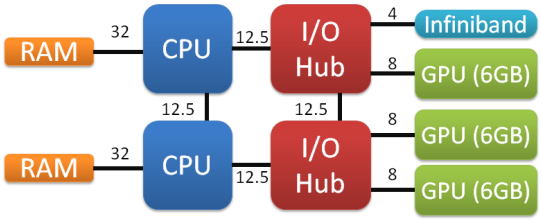

Figure 3. Block Diagram of HP SL390–An example of a dual-IOH system. Numbers indicate the peak theoretical unidirectional bandwidth of the system's datapaths in gigabytes per second.

B. Implementation Details

1) Local Histogramming and Scanning: Due to the well-characterized speedups for scan and radix sort within a single GPU [8], [9], the prototype implementation relies heavily on these operations, which are built into Thrust. First, a bitmask is applied to the keys to select only the relevant bits for the current pass. Then, using only these bits as a key, key-value pairs are sorted within a node. Then, the local histogram is computed with Thrust's vectorized binary search. This approach has the advantage of appropriately reordering the keys for a specialized all-to-all communication, which is used in the key exchange.

2) Key Exchange: Perhaps the most flexible portion of the sorting algorithm is the global key exchange. Due to the explicit nature of the index calculation, each processor contains information on how many keys it will send, the corresponding destinations, and how many it will receive (but not which processor will send them). Since radix sort is not an in-place algorithm, one has several options for implementing the exchange including: many point-to-point messages, three all-to-all collective operations (counts, keys, values), or numerous one-sided puts. The optimal choice is highly dependent on the MPI implementation, the network topology, and the interconnect hardware itself, and a detailed exploration of this space is an opportunity for future work when more GPU-based clusters are available for testing. The prototype implementation uses the all-to-all based scheme.

3) Division of Work Between CPU and GPU: Another key aspect of the design of our implementation is the division of work between the CPU and GPU. For most of the subtasks, including histogramming, scanning, and local sorting, the GPU is faster. Indeed, the CPU was left with only three major tasks: MPI communication in the indexing calculation, MPI communication in the key exchange, and all computation to prepare for those exchanges (MPI specialized all-to-all required two prefix sums for offset calculation). It should also be noted that our approach presents an opportunity to overlap communication and computation. In particular, the MPI indexing calculation (and associated CPU scanning) is overlapped with the asynchronous copy of the keys and values back from the GPU (just before the exchange).

C. Results

Our analysis aims to address the following questions:

- What is the relative importance of computation versus data movement? What data movement steps account for most of the runtime?
- How does the performance vary with problem size scaling, different data distributions, and parameter settings (radix width)?
- How does performance compare with sorting within a single GPU?
- How does performance scale when more nodes are added?

1) Computation vs. Communication: In order to understand how the ratio of computation to communication changes across problem sizes, the prototype implementation was instrumented with timing code to report the running time spent in each of the components of the algorithm. Results are shown in Figure 4 for two standard distributions of sixty-four bit key-value pairs: already sorted data (sequential) and random data with a uniform distribution. This experiment involves varying the problem size (keys and values) using a fixed processs count of twenty four (eight Keeneland nodes).

MPI communication ranges between 26%-30% of runtime. When including PCIe transfer time, data movement between ranks only accounts for up to 40% of total runtime. In relative terms, MPI strongly dominates PCIe transfer, by a factor of at least two for all problem sizes and both data distributions.

2) Problem Size Scaling and Data Distribution: Figure 4 also confirms the expected linear relationship between problem size and runtime. It provides insight into the relative insignificance of the global indexing calculation and CPU computations, at least for small node counts.

Averaged over one hundred random trials, already sorted data performs worse than random data with a uniform distribution; since the radix width is smaller than the key length, already sorted data is guaranteed to require passes which will perform full scatters, leading to the greatest MPI all-to-all times. Still, runtime for the two data distributions was fairly consistent, with the largest disparity at any problem size of roughly 5% of runtime, resultant from varying MPI all-to-all performance. This amount of variation is in line with expectation–kernels are expected to be essentially consistent, but MPI all-to-all time should vary slightly based on data distribution. In other words, depending on the data, keys may or may not move between nodes during the exchange.

3) Radix Width: The theoretical analysis asserted that changing the radix width provides an important mechanism for increasing arithmetic intensity and decreasing data transfer by controlling the number of passes. In this study, we focus on sixty-four bit data, given that is the precision used by most scientific applications. As the radix width increases, the number of passes decreases, leading to a reduction in PCIe, MPI, and kernel time due to decreased memory requirements (in the worst case, keys traverse GPU global memory, a PCIe link, and an MPI link in each pass). This decrease is depicted in Figure 5. On the other hand, the size of the histogram increases exponentially, making radix widths greater than sixteen impractical.

4) Strong Scaling: In the next scaling experiment, parallel efficiency was measured in a strong scaling scenario, shown in Figure 6 for a 1 GiB problem of already sorted data. As expected, efficiency drops dramatically when data must be moved off node (greater than three processes), but then levels off at around 60%.

However, it must be noted that this measurement includes PCIe transfer during each pass for the single process result, which is not strictly necessary. While this makes it easier to understand scaling behavior (in terms of the interconnect performance), it understates the penalty for using more than one GPU. In fact, compared to Merrill's optimized radix sort (kernel only), the overheads of the parallel version result in 18x slower performance when run on one GPU. This penalty due to PCIe transfer and other overheads makes a distributed approach unreasonable for problem sizes which can fit inside the memory of a single GPU.

5) Weak Scaling: Next, we examine the performance behavior in a typical weak scaling experiment to evaluate scalability (e.g., at what concurrency level overhead dominates and collective communication time reaches parity with computation). For this experiment, depicted in Figure 7, we use sixty-four bit key-value pairs, with a sixteen bit radix width ($r = 16$), and choose the already sorted distribution for input since it stressed the interconnect more than the random distribution in the previous experiment.

As the number of processes increases from 12 to 96, the total exchange time gradually increases from 1.74 to 2.12 seconds, with all MPI communication spanning 36% to 41% of total runtime. Each additional node (or 3 GPUs) adds roughly eight milliseconds to the exchange at low concurrency levels, dropping to approximately three milliseconds

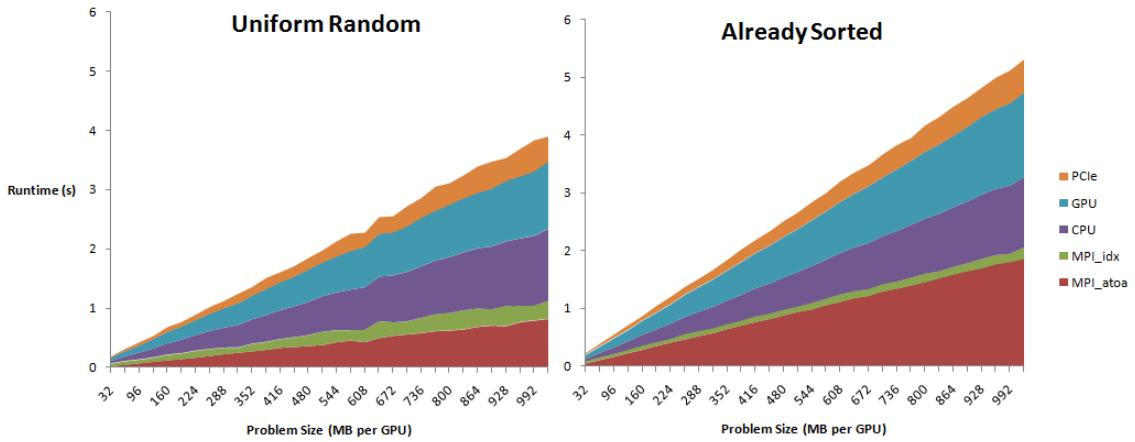

Figure 4. This figure displays total runtime for varying problem sizes, broken down by category. GPU and CPU indicate kernel computation time and others identify communication or data movement. MPI_atoa refers to the three collective operations during the key exchange and MPI_idx refers to all broadcasting and global scanning operations. All measurements use eight Keeneland nodes (24 GPUs) with a radix width of sixteen bits.

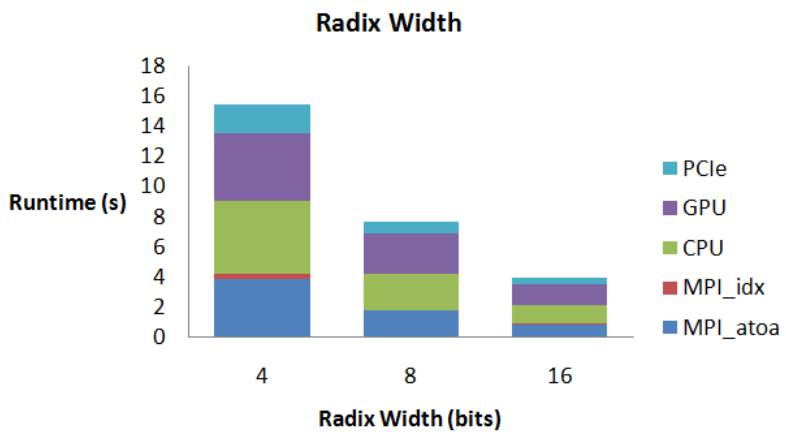

Figure 5. This figure captures the effect of changing the radix width. Runtime decreases substantially as radix width is increased. Widths beyond sixteen bits become impractical due to histogram size. Results shown on eight Keeneland nodes with 2GB problem size (256 MB per node).

within the higher concurrency levels we reached. We expect scaling to large node counts should present no performance obstacles, though of course further study would be required when considering more extreme concurrency levels.

D. Other Operations

Note that in Algorithm 1, the only other operations which require more than constant time are a global broadcast of the interquartile range and parallel binary search for outliers. And, despite their nontrivial $logn$ and $logp$ bounds, these two kernels represent only a fractional amount of the total runtime, on the order of .01%. For both these operations, our implementation relies on the optimized versions included in Mvapich2 and Thrust.

E. Architectural Bottlenecks

As mentioned in Section III-C4, the performance we observed on a single GPU, after adding the pieces necessary to enable a distributed algorithm, was 18x slower than the heavily optimized single-GPU Thrust implementation. This is a sobering reality, and it highlights the bottlenecks we encounter in developing algorithms for heterogeneous systems. In this case, some work which could previously be combined into a single pass was separated into multiple kernel calls. This introduces overheads: first from the mere fact that we are launching more kernels, and second because some bookkeeping calculations must be repeated in each phase.

The PCIe transfers in between these kernel invocations add further overheads. In the case of an architecture like Keeneland, and in particular systems with multiple CPU

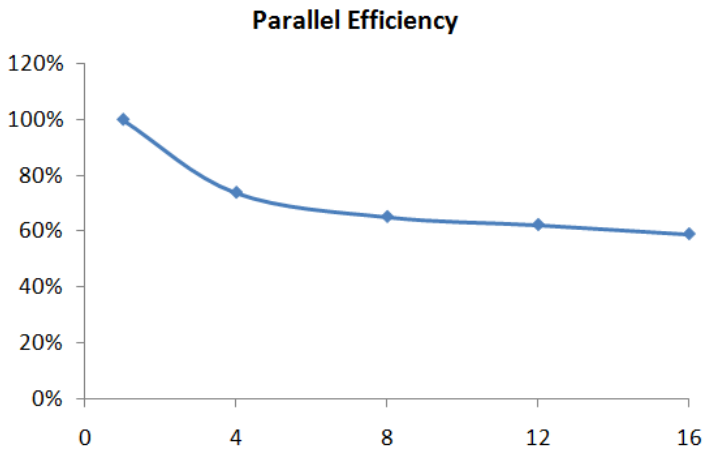

Figure 6. Parallel efficiency drops abruptly when moving off node (> 3 processes), then levels out around 60%. Results shown for strong scaling with a 1 GiB problem using 64-bit key/value pairs. This drop illustrates the performance penalty of off-node communication.

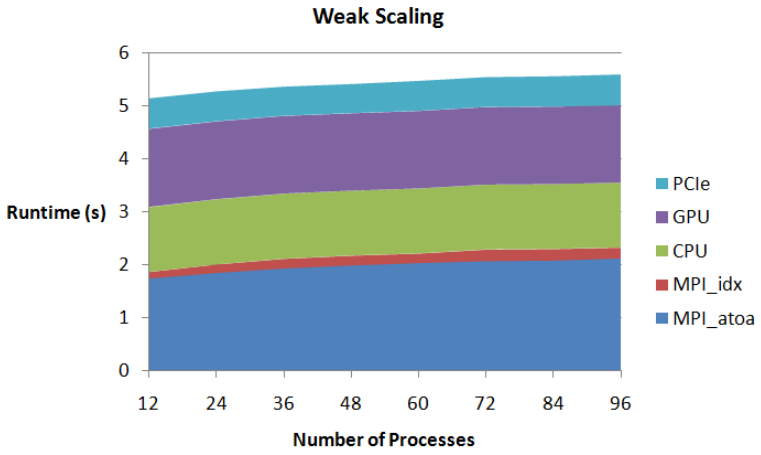

Figure 7. This figure depicts results from a weak scaling experiment using 512MB of 64-bit key/value pairs per GPU. While PCIe transfer and compute times remain fairly constant, time spent in the key exchange (MPI_atoa) rises from 1.74 to 2.12 seconds.

sockets and I/O hubs, we observe an additional complexity where the PCIe bus comes into play: non-uniform memory access, or NUMA. As seen in Figure 3, the bandwidth and number of hops differs among choices of CPU and GPU; therefore, one must choose a CPU affinity for the host tasks assigned to each GPU carefully. [10] Here, the difference between a poor choice and proper choice of affinity was a performance penalty of approximately 12%.

IV. Related Work

Sorting algorithms capable of running on distributed memory parallel computers have been explored for some time. Examples include bitonic sort [11], [12], [13], distribution sort [14], and radix sort [15]. Similarly, some of these have been developed for graphics processors, namely bitonic [16], radix [4], sample [3], and hybrid combinations [17], [18]. While these GPU-based algorithms are inherently data-parallel due to the nature of the GPU architecture, most of these algorithms are limited to graphics processors shared by a single shared-memory host system, or more often just one GPU, and have not explored the combination of data-level and distributed memory parallelism required for large-scale heterogeneous systems.

V. Conclusion

As we move to an era where analysis techniques become increasingly necessary as an aid to data reduction, and as the machines supporting these calculations become more heterogeneous, reevaluating these techniques is crucial. In this paper, we present an extension of a fast GPU-based in-memory sorting algorithm to a distributed heterogeneous system and analyze its performance. We conclude that

significant sacrifices in performance must be made to enable distributed support as long as fast GPU memory is disparate from main memory and lies across a PCI-Express bus. However, once these overheads are paid, the approach allows applications to continue to scale up large datasets despite limited bandwidth to stable storage.

VI. Source Code

Source code for the distributed radix sort is open source and available at http://bitbucket.org/kyle_spafford/radix_sort

VII. Acknowledgments

This research is sponsored in part by the Office of Advanced Computing Research, U.S. Department of Energy, the National Science Foundation award OCI-0910735 and DARPA contract HR0011-10-9-0008. The work was performed at Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] J. Dongarra, P. Beckman, and et al, "The international exascale software roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, 2011.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Linear time bounds for median computations," in *Proceedings of the fourth annual ACM symposium on Theory of computing*, ser. STOC '72, 1972, pp. 119–124.

[3] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1 –10.

[4] D. Merrill and A. Grimshaw, "Revisiting sorting for GPGPU stream architectures," University of Virginia, Tech. Rep., 2010. [Online]. Available: http://www.cs.virginia.edu/~dgm4d/

[5] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat, "TritonSort: a balanced Large-Scale sorting system," in *Proceedings of NSDI*, 2011.

[6] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10, 2010, pp. 351–362.

[7] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *IEEE Computing in Science and Engineering*, vol. 13, no. 5, 2011.

[8] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Aire-la-Ville, Switzerland: Eurographics Association, 2007, pp. 97–106.

[9] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," University of Virginia, Tech. Rep., 2009. [Online]. Available: http://www.cs.virginia.edu/~dgm4d/

[10] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying NUMA and contention effects in multi-GPU systems," in *Proceedings of The Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011.

[11] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring), 1968, pp. 307–314.

[12] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Transactions on Computers*, vol. 28, pp. 2–7, 1979.

[13] Y. C. Kim, M. Jeon, D. Kim, and A. Sohn, "Communication-efficient bitonic sort on a distributed memory parallel computer," *Parallel and Distributed Systems, International Conference on*, vol. 0, p. 0165, 2001.

[14] M. H. Nodine and J. S. Vitter, "Deterministic distribution sort in shared and distributed memory multiprocessors," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '93, 1993, pp. 120–129.

[15] "Partitioned parallel radix sort," *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 656 – 668, 2002.

[16] I. Buck and T. Purcell, "A toolkit for computation on GPUs," in *GPU Gems*, 2004, ch. 37.

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06, 2006, pp. 325–336.

[18] "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381 – 1388, 2008, general-Purpose Processing using Graphics Processing Units.