Increasing the Energy Efficiency of Distributed Sorting by Avoiding Communication

-,-,-

ABSTRACT

Sorting is an essential building block for several algorithms and applications. As we ramp up to the first exascale systems, it is important to look beyond the usual metrics of performance and scalability and also consider the energy and power requirements of sorting algorithms. In this work we improve the energy efficiency of distributed sorting by proposing a new algorithm that avoids communication whenever possible. In addition, we analyze the most energy and power efficient sorting strategies on GPUs and Heterogeneous system architectures.

1. INTRODUCTION

Large-scale distributed sorting have focused on improving sort throughput (number of keys sorted per sec) and parallel scalability. However, for the next generation of supercomputers, additional metrics related to energy and power consumption are going to be equally if not more important. Since, sorting is entirely dominated by data movement, energy is equivalent to the total data communicated and power is proportional to the peak bandwidth utilized. This makes distributed sorting an ideal algorithm to study energy and power related optimizations. The motivation is to develop a distributed sorting algorithm that aims to minimize the overall data-movement, the peak bandwidth required without sacrificing either performance or scalability. The development of such a distributed sorting algorithm is the central contribution of this work. We propose both distributed as well as node-local strategies to achieve this goal. Our expectation is that this work will serve as guideline for the development of such energy and power efficient for a wider array of problems in the near future.

Relevance.

Motivate and connect to a larger set of problems. Highlight contributions and list limitations.

2. BACKGROUND

Keep related work here and separate from introduction.

some glue text here before we go into separate subsections on the energy aspects and the algorithmic aspects of sorting.

2.1 energy considerations on GPUs

2.2 Sorting Algorithms

The design and implementation of parallel algorithms for sorting is a well studied subject. In our discussion of prior work, we focus our attention to distributed memory algorithms that have been experimentally shown to scale to large core counts and large datasets. We will also discuss related sorting algorithms (albeit not at scale) that study the energy-efficiency of sorting. We begin by a formal definition of distributed sorting.

Given an array A with N keys and an order (comparison) relation, we would like to sort the elements of A in ascending order. In a distributed memory machine with p tasks, every task is assigned an N/p-sized block of A. Upon completion, task i will have the i^{th} block of the sorted array. Note that the overall ordering of the keys is determined by the ordering of the tasks as well as the ordering of the keys on each task.

Distributed sorting.

The most popular algorithm used in actual distributed sort implementations is SampleSort, originally proposed by Frazer and McKellar[**?**]. Given p tasks, we reshuffle the elements of A in p buckets so that the all the keys in the i^{th} bucket are smaller (or equal) than the keys in the $(i+1)^{\text{th}}$ bucket. Bucket i is assigned to task i and thus, once we have shuffled A, each task can sort its bucket of keys in an embarrassingly parallel manner using any local (shared-memory-parallel) sort algorithm. The challenge is to obtain good load-balancing, i.e., ensuring that each task has roughly the same number of keys, while minimizing communication costs.

One possible way to reshuffle A is to estimate the boundaries for each bucket, by selecting $p-1$ keys, which we call "splitters". This can be done by sampling a subset of keys in A, sorting them and selecting splitters from that set. Once these $p-1$ splitters have been selected, a global data exchange takes place to move the original keys of every task to their correct bucket. An additional local sort is invoked to finalize the output array. SampleSort is well understood. However, its performance is quite sensitive to the selection of splitters, which can result in load imbalance. Most importantly, the final data exchange—requiring $\mathcal{O}(p^2)$ messages—can congest the network. As a result SampleSort may scale suboptimally, especially when the communication volume approaches the available hardware limits [?].

Parallel HistogramSort [?, ?] is a variant of SampleSort that efficiently estimates the splitters. The authors presented one of the largest distributed comparison sort runs (32K cores on BG/P) with 46% efficiency. The algorithm overlaps communication and computation in all stages. During the splitter estimation stage, the iterative estimation of the ranks of the splitters is combined with partial local sorting of the data, by using the splitter candidates as pivots for quicksort. Once the splitters are estimated, the communication of data is staged and overlapped with local merging. In [?] the best throughput was obtained on 16,384 cores of Jaguar XT4 at ORNL for 8M (64-bit) keys per core; the sort took 4.3 seconds achieving a in-RAM throughput of 14.4TB/min.

CloudRAMSort [?] demonstrated good scalability on 256 nodes with shared memory parallelism using pThreads and SIMD vectorization. The best results are for sorting 1TB of data (10byte key + 90byte record) in 4.6 secs achieving an in-RAM throughput of 12.6 TB/min. They use a variant of Histogram Sort [?], where the samples are iteratively computed (in parallel) in order to guarantee a minimum quality of load-balance. Additionally, the communication of the records is split into two parts by first communicating the keys followed by the values overlapped with the merging of the keys.

In recent work [?], we have addressed the scalability of sorting on very large clusters. Hyksort[?] is an extension of quicksort on a Hypercube [?] to a k-ary Hypercube along with an efficient parallel (median) selection algorithm. The current work is a modification of Hyksort[?] allowing us to avoid communication in several stages of Hyperquicksort. We describe the new communication-avoiding distributed sort algorithm in detail in §**??**.

High-performance GPU sorting.

Several sorting algorithms tailored for GPU architectures have been proposed in the literature. The earliest sorting implementations were often based on Batcher's bitonic sort [?]. This includes work from Purcell et al. [?], and Kipfer et al. [?]. The advent of the CUDA programming model enabled the construction of more complex sorting algorithms. Harris et al. [?] implemented a split-based radix sort and a parallel merge sort. Le Grand [?] and He [?] described a histogram-based radix sort. Segmented scan-based implementations of radix and merge sorts were presented by Sengupta et al. [?]. Improved versions of radix and merge sorts were presented by Satish et al. [?] and by Merrill et al. [?]. In this paper, we use two state-of-the-art GPU sorting algorithms at the node level: radix sort from the CUB library [?], and merge sort from the ModernGPU library [?].

Code variant tuning.

Apart from Nitro [?], several programmer-directed autotuning frameworks support tuning of code/algorithmic variants. Petabricks [?] supports user specification of *transforms* that are analogous to functions. Transforms are automatically composed together to form hybrid algorithms using a compiler framework and an adaptive algorithm [?]. Petabricks, however, implicitly tunes variants for the size of the input data set. Ding et al. [?] propose extensions to the PetaBricks language to enable support for tuning based on arbitrary input features.

Multi-objective optimization.

3. DISTRIBUTED SORTING

Large-scale distributed sorting, such as those discussed in §**??**, have focused on improving sort throughput (number of keys sorted per sec) and parallel scalability. Since, sorting is entirely dominated by data movement, energy is equivalent to the total data communicated and power is proportional to the peak bandwidth utilized. This motivates us to design an algorithm that minimizes the peak-bandwidth requirement as well the overall data communicated. In this section, we first look at the theoretical arguments in this direction, leading to the actual arguments.

In [?], we proposed a variant of HyperQuicksort[?] that allowed us to control the algorithm behavior by adjust a parameter k. Selecting $k = 2$ made the algorithm behave as HyperQuicksort, whereas selecting $k = p$ resulted in behavior similar to Samplesort. The bandwidth term for the algorithm is,

$$k \frac{\log^2 p}{\log k} + \frac{N}{p}. \tag{1}$$

Ignoring the N/p term, we can see that for Samplesort($k = p$) the bandwidth required is $\mathcal{O}(p \log^p)$, whereas it is only $\mathcal{O}(\log^p)$ for Hyperquicksort. So clearly, Hyperquicksort is preferable from the bandwidth (and therefor power) perspective. An added benefit is that Hyperquicksort only sends $\mathcal{O}(p)$ message at each stage as opposed to $\mathcal{O}(p^2)$ for Samplesort. While Hyperquicksort does communicate $\mathcal{O}(n \log n)$ data, as compared with $\mathcal{O}(n)$, it's overall performance and scalability is comparable to Samplesort[?]. It is not possible to reduce the bandwidth term for Samplesort, so we modify Hyperquicksort to reduce the amount data it needs to communicate. This is achieved by selectively changing the task-role instead of moving data between tasks. We call this algorithm SwapRankSort. This is in principle similar to moving the computation to the data instead of moving the data to the computation. We now elaborate on SwapRankSort.

3.1 SwapRankSort

During each stage of Hyperquicksort, each task exchanges data with another task whose *rank*[1] differs from its own at bit-k, where k is the current stage of Hyperquicksort. How much data is exchanged between the tasks depends on the distribution of local-data on each task and the global median of the data (at this level). Analogous to quicksort, the lower-ranked task retains the keys smaller than the median (pivot) and the higher-ranked task retains the higher keys. Depending on the local-distribution of the keys, all the keys (N/p) might be exchanged between the tasks. In SwapRankSort, we propose a minor modification, where we evaluate the cost of exchanging data for the default case as well as if the ranks of the two tasks were swapped. A swap simply means that a task that would have retained the smaller keys will now retain the larger keys. Clearly that maximum amount of data exchanged in this case is $N/2p$.

Given that we run this in a distributed setting implies that even a single pair of tasks having a skewed distribution will cause the overall algorithm to exhibit poor performance. Also note that in case of uniformly distributed

[1]a unique identifier assigned to each task.

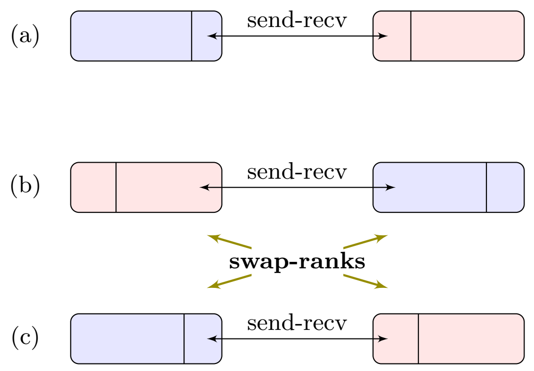

Figure 1: Illustration of the central idea behind SwapRankSort. In (a) the distribution of the keys results in the tasks exchanging a small number of keys. Ranks are not swapped in this case and the program behavior is the same as Hyperquicksort. In (b), we illustrate the case where the smaller ranked task (in blue) has a large number of keys greater than the pivot, and the higher ranked task(red) has a large number of keys smaller than the pivot. This results in a lot of data exchange. In (c), we illustrate SwapRankSort where the tasks swap ranks and exchange the smaller set of keys.

(amongst the tasks, independent of the data distribution) keys, SwapRankSort performs the same as Hyperquicksort with no tasks swapping ranks. The actual swapping of the ranks takes place at the end of each stage when we split the communicator. This is a step that is needed for Hyperquicksort as well, so there is no significant overhead to swapping ranks.

4. ENERGY-EFFICIENT NODE SORTING

The previously-described distributed sorting algorithm relies on a node-level sort of the subset of data assigned to each node at each stage of the sort. This section describes how we arrive at a node-level sorting algorithm that is both energy efficient and high performance.

4.1 Overview of Approach

We first present an overview of the node-level sorting approach, looking at factors such as resource selection in a heterogeneous platform, sorting algorithms, and how we extend an existing framework for performance optimization to also take into account energy and power consumption.

GPU vs. Parallel CPU Sorting.

Establish that GPU is highest performing and most energy efficient for node sort. Show performance (and possibly energy) difference for both Titan and Jetson compared with OpenMP?

Sorting algorithms as code variants.

It is well established that the best algorithm for sorting is dependent on input data set (type, size and distribution), target architecture and implementation details. Therefore, the best-performing and most energy-efficient implementation cannot be determined until run time without a prior knowledge of these factors. We refer to different sorting algorithms or implementations as *code variants*. A number of techniques for code variant selection and algorithm selection have been described in the literature, and our approach will rely on recent advancements in this area as described in this section.

In this paper, we select among two algorithms to use for the node-level sort.

- *Merge Sort:* Merge Sort sorts a list of data by recursively splitting the list in half, sorting each half, and then merging the two sorted lists together. The Merge Sort implementation we use is part of the ModernGPU [**?**] library of GPU primitives.
- *Radix sort:* Radix Sort achieves a sorted list by grouping keys by individual digits that have the same position and value. The Radix sort implementation is provided in CUB [**?**].

May want to also say when one might be preferable to another.

Managing energy and power on the GPU.

We use two mechanisms to adjust energy and power usage on the GPU. We can monitor energy or power usage for each of the two sort algorithms, and together with performance measurements, select the preferred algorithm. In addition, the target Nvidia GPUs allow adjusting of the clock frequency, or frequency of the memory bus. Through monitoring energy or power at different frequency settings, we can select the preferred frequency(ies).

Since we would prefer an implementation that is both high performing and power or energy efficient, we must develop a *selection criteria* that considers multiple optimization goals in selecting the node-level sorting algorithm. The next subsection will describe a number of different selection criteria we explore in this paper and their overall impact on performance, energy and power.

Code variant selection using Nitro.

The system described in this paper used for code variant selection extends the Nitro autotuning framework [**?**]. Nitro provides a library interface that permits expert programmers to express code variants along with meta-information that aids the system in selecting among the set of variants at runtime. Figure **??** illustrates the approach in Nitro. A learning algorithm – Support Vector Machine (SVM) classifier by default – co nstructs a code variant selection model on the target architecture as a result of an offline training phase on the same architecture. For each architecture, training data has the form $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_M, y_M)\}$, where each $\mathbf{x}_i$ represents an input feature vector and each y_i represents the best variant for that input. When presented with a new, unseen input at runtime, the model predicts the best variant to use. For sort, prior work has used data type, data set size and presortedness as features [**?**]. In this paper, we omit presortedness, and replace it with the distribution of key values. These three features are available when the sort is invoked, and can be used in consulting a model for code variant selection at run time. In this paper, we extend Nitro in two ways: (1) we treat different clock frequencies as code variants, in addition to the different sorting algorithms; and, (2) the model is trained using both performance and energy/power data, according to the selection criteria outlined in the next subsection.

4.2 Code Variant Selection Criteria

Application tuning that looks at multiple optimization criteria is referred to as *multi-objective tuning*. A challenge with multi-objective tuning is that the solution must encompass a tradeoff space between different optimization objectives. Any solution among the *Pareto frontier* is valid; these points are ones for which there is no other solution that has better metrics among all the set of objectives. CLEAN UP AND ADD CITATIONS!!! Techniques for multi-objective tuning resolve this selection in several ways: (1) treat one objective as independent (e.g., an equivalence class as in PetaBricks paper); (2) weight one objective above others; (3) ask users; (4) use heuristics to drive optimization; or, (5) come up with combined metric.

In this paper, we have chosen to use a set of fairly standard combined metrics. We explore which leads to the best reduction in energy or power with the least impact on performance. These metrics are as follows:

- MKeys per Joule: Define
- MKeys^2 per Joule: Define
- MKeys per Watt: Define
- MKeys^2 per Watt: Define

These were selected because they capture the relationship between throughput and energy or power. Further, it is straightforward to build a model for code variant selection by consolidating on a single metric. In our experiments we will show some indication that this works well.

5. EXPERIMENTAL METHODOLOGY

This section describes the input data for sort and the three target architectures used in our experiment.

Input Data.

As the performance of sort is dependent on its input data, we use a variety of data types, distributions and sizes in our experiments. We consider XXX data types: integer, long integer, float, double, fill in the rest. The distribution of key values includes a uniform distribution, a Gaussian distribution, fill in the rest. Sizes are motivated by each experiment and the capacity of the target architecture. For Titan, ... For K20c ... For Jetson ... Perhaps this goes in the results section.

Titan (ORNL).

We are interested in achieving scalable energy efficiency for distributed sorting on supercomputers, and therefore the target architecture is the Titan system at Oak Ridge. Titan has ... CPUs ... GPUs ... compiler/OpenMP/MPI installations? We would like to measure performance on the GPUs on Titan, but encountered an obstacle in the software installation. The CUDA version used on Titan is CUDA XXX, which is out-of-date with respect to sorting implementations we are using. We are also unable to measure power and energy in the same way on Titan as on the other platforms. Therefore, we will focus our Titan experiments on using the OpenMP implementation of sorting, and focus data gathering on the communication and scaling aspects of our optimizations. We will then extrapolate Titan GPU results using other representative clusters.

Nvidia K20c standalone GPU.

For the node-level experiments, we used an Nvidia K20c (Kepler generation) standalone GPU, representative of the Titan nodes. This machine has 13 GPU streaming multiprocessors, for a total of 2496 cores, 4.8 GBytes of memory and an L2 cache of 1.25 GBytes. It uses CUDA 6.5 and nvcc compiler version XXX. We use this machine in our experiments because we have complete control over its installation, software tools and configuration for each run.

For energy and power measurements, we use ... identify software power measurement tool and other details of methodology. The K20c has five clock frequency settings ranging from 614MHz to 758MHz, with 705MHz as the default setting. The clock frequency can be adjusted ... say how. It has two memory frequencies as well, but we do not adjust memory frequency in these experiments because the lower memory frequency of say what it is is far lower than peak of say what it is and is therefore going to perform poorly in a bandwidth-limited algorithm such as sort.

Jetson TK1 cluster.

We also measured performance of the distributed sorting on an Nvidia Jetson TK1 cluster; the nodes of the Jetson are low-power and lightweight, consisting of a single GPU streaming multiprocessor (Kepler generation) with 192 cores, and four-plus-one ARM cores, where the fifth ARM core is used as a master processor. The nodes have a unified DRAM of 2 GBytes, which is shared between CPUs and GPUs, and separate cache structures for CPU and GPU. The cluster we use in this experiment has XXX nodes, and the nodes are connected with ... network details. The software installation uses CUDA 6.5, nvcc compiler version 6.5.35, MPI version 1.6.5, and OpenMP version 3.1.

The power and energy reported for Jetson are physical measurements using the BK Precision's 2138e 4-1/2 ¡ digital multimeter. We measure the voltage drop across a known precision resistance in series with the Device Under Test (DUT). With a known resistance and measured voltage on that resistance, the current can be obtained with I=V/R. Here, the resistance is 0.020 ohms with a 1% variation. To determine the power, P=IV where I is the value calculated above, and V is 12V. The Jetson has fourteen core clock frequencies ranging from 72MHz to 852MHz, and twelve memory frequencies from 12.75MHz to 924MHz; because collecting physical measurements on all 1728 combinations of core/memory frequency per data set would be prohibitively time-consuming, we sampled for this experiment. Perhaps say what data we have or save for the results.

While not capable of the high GPU performance of the K20c since it has only one-thirteenth of the SMs, the Jetson GPU still outperforms the OpenMP node-level sort by XXX for a ... explain experiment.... Therefore, while not necessarily representative of Titan, the Jetson cluster looks to the future of high-performance and embedded GPU platforms. The unified memory allows us to look at power and energy without the data movement required to copy from CPU to GPU, and the large number of frequency adjustments allow us to examine how the large number of degrees of freedom in energy management impacts energy, power and performance in code variant selection.

6. EXPERIMENTAL RESULTS

Things to answer.

- Are different algorithms affected differently by frequency adjustment?
- How does frequency affect performance/energy/power?
- End with SCI cluster and Titan results, and extrapolate from prior measurements.

7. DISCUSSION

Discuss results and what the key findings are and what it means for future architectures.

8. ACKNOWLEDGMENTS

Thank Christopher Strong, nvidia?, Funding agencies.