2018 17th International Symposium on Parallel and Distributed Computing

CISC: Coordinating Intelligent SSD and CPU to Speedup Graph Processing

Dongyang Li[1,2] Yafei Yang[2] Weijun Li[2] Qing Yang[1,2]
[1]Dept. of Electrical, Computer and Biomedical Engineering, University of Rhode Island, RI, US
[2]Shenzhen Dapu Microelectronics Co., Ltd., Shenzhen, China
Email: lidongyang@ele.uri.edu

***Abstract*—Minimum Spanning Tree (MST) is a fundamental problem in graph processing. The current state of the art concentrates on parallelizing its computation on multi-cores to speedup MST. Although many parallelism strategies have been explored, the actual speedup is limited, and they consume a large amount of CPU power. In this paper, we propose a new approach to the MST computation by coordinating computing power inside SSD storage with host CPU cores. A comprehensive framework of software-hardware co-design, referred to as CISC (coordinating Intelligent SSD and CPU), preprocesses MST graph edges inside storage and parallelizes the remaining computation on host CPU. Leveraging the special properties of modern SSD storage, CISC exploits a divide and conquer approach to reordering graph edges. We have implemented an FPGA circuit that reorders chunks of graph edges inside an SSD. The ordered chunks are then loaded to the system RAM and processed by the host CPU to build a B-Tree structure by repetitively picking up edges at heads of chunks. A working prototype CISC has been built using NVM-e SSD on a server. Extensive experiments have been carried out using real-world benchmarks to demonstrate the feasibility and performance of deploying CISC in NVM-e SSD storage. Our experimental results show 2.2$\sim$2.7$\times$ speedup for serial version implementation and 11.47$\times$ to 17.2$\times$ speedup for the parallel version with 96-cores. For the same number of cores, our parallel CISC outperforms the traditional software MST by up to 35%.**

***Index Terms*—Graph processing, Minimum Spanning Tree, Processing in Storage (PIS), NVM-e storage,**

I. Introduction

Processing of graph-structured data has become increasingly important and has brought to the forefront of computational challenges. Graphs with up to billions of vertices and trillions of edges are commonplace in today's big data era [1]. Minimum Spanning Tree (MST) is a fundamental problem in graph processing to compute a subset of a graph with the total edge weight being minimum. It is pervasive throughout science, broadly appearing in fields such as social network, biological science, transportation, VLSI and so forth. A classical way of computing MST is the well-known Kruskal algorithm which sorts edges in ascending order first and then merges them into a subset without overlaps. Extensive research has been reported in the literature to speed up Kruskal MST computation using parallel architectures [2$\sim$7]. On parallel architectures such as FPGA, multicores and GPUs, MST computation can be divided into multiple tasks that are executed concurrently in parallel, and hence speeding up MST computation.

However, effectively parallelizing MST faces many challenges. The most critical challenges are data dependency and potential deadlock problems. Li et al [2] used High-Level Synthesis (HLS) in FPGA to solve graph edge dependency in MST. The FPGA works as co-processor of CPU and aborts the conflict task when data dependency happens. Experimental results show that such CPU-FPGA co-processing achieves up to 2.2$\times$ speedup compared with the single core computation. Subramanian et al [4] presented FRACTAL to speed up MST using multi-cores. Their work is based on a cycle-accurate, event-driven simulator to model parallel system with 256 cores [7]. To avoid data dependency in MST, they modified task scheduler and used timestamps to determine which tasks execute in high priority. Their simulation shows 40$\times$ speedup when configured with 256 multicores. Manoochehri et al [6] proposed MST implementation on GPU. They used Software Transactional Memory based synchronization to alleviate data dependencies among GPUs. It outperforms MST running on single core CPU by 4.5$\times$.

While existing research made efforts on solving task deadlock and data dependency problems [2$\sim$7], the ultimate speedups obtained by the current state of art are still limited. In order to further improve MST performance, we carried out extensive experiments to study what is holding the MST from running faster. In the ordinary storage system, a large number of graph edges are stored in SSD or HDD without ordering. In most existing works, parallel computing architecture loads unsorted graph edges into local memory and sorts them before MST merge. Sorting data in the main memory of host is computation intensive, and it consumes enormous CPU resources. We observed in our experiments that edge sorting takes a significant portion of total MST execution time ranging from 36% to 75% of the total time. We therefore believe that there is a great potential for further performance improvement by leveraging the intelligence available inside SSD where a huge amount of graph edges is stored.

In this paper, we present a new approach to the MST computation by means of CISC (Coordinating Intelligent SSD and CPU). The idea is to exploit the controller logic inside the SSD to preprocess graph edges while being loaded to the main memory of the host. CISC divides the large amount of graph edges into chunks and sorts each chunk of edges in order using hardware. In this way, the edges loaded into the internal memory of the host consist of multiple sorted chunks. To allow software MST to efficiently use sorted chunks, we developed two software programs for the host

978-1-5386-5330-2/18/$31.00 ©2018 IEEE
DOI 10.1109/ISPDC2018.2018.00029

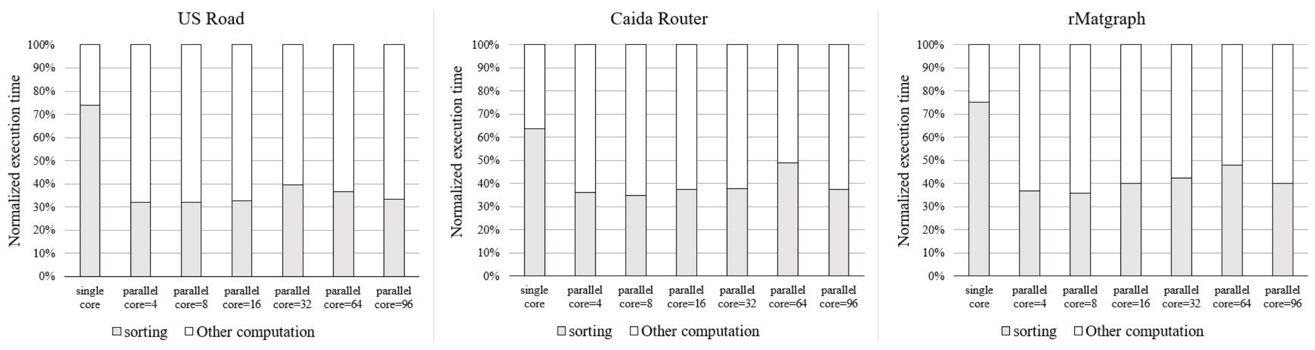

Fig. 1. Fractions of graph sort time over total execution time of the serial and parallel MST running on multi-cores.

servers of serial and parallel MST, respectively. The serial MST forms a B-tree holding the smallest edges of all the chunks and merges smaller edges into MST in high priority. In the multicore system, we optimized the classical sample sort algorithm [8] of parallel MST, and the remaining computation can be effectively parallelized on multicores. Such an efficient data distribution in the host main memory ensures smaller weight edges can be processed very efficiently. To demonstrate the feasibility and performance potential of CISC, we have implemented CISC using FPGA inside an SSD. A working prototype has been built that consists of both software running on the host and hardware circuit inside SSD. Using the CISC prototype, we run standard graph benchmarks to measure performances. Experimental results show that CISC outperforms pure Software MST substantially.

This paper made the following contributions:

- A pipeline structure of FPGA sort module has been presented that can provide wire-speed hardware sort of multiple edge chunks. We have designed and implemented the FPGA module alongside the I/O bus inside PCI-e SSD realizing a true processing in storage (PIS) for graph processing. It is also extensible to other sort-based software applications.
- A B-tree based selection algorithm and an optimized sample sort algorithm have been proposed that run on single core and multicore systems, respectively. CISC coordinates the chunk sorting inside SSD and selection/merging of minimum weight edges on CPU cores efficiently. The software and hardware co-design framework is the first of the kind for graph processing.
- A working CISC prototype has been built that works as expected. The prototype has been used to carry out extensive experiments for performance measurements. Our experimental results demonstrated the superb performance and effectiveness of CISC for MST over existing approaches.

The rest of this paper is organized as follows: In section II, we discuss the related work. Section III provides detailed design for in-storage sort module. Section IV describes the two MST software modules of CISC that run on single-core and multicores, respectively. Section V presents experimental results and discussions. Section VI concludes the paper.

II. Background

A. Overhead of Sorting in MST

MST computation consists of a number of tasks. The first time-consuming task is edge sorting. To understand how significant the sorting part contributes to the total computation time of MST, we measured the actual sorting time of Kruskal MST and estimated the proportion of sorting time in the entire MST computation. We set up the experiment environment with Intel Xeon processor having 96 cores running at 2.5 GHz. Linux system with kernel version 4.14 was installed on the server. We selected three benchmark datasets from [28∼30]. The parallel software code [9] uses OpenMP when configuring multicores.

Figure 1 shows the breakdown of edge sort time and other computation time of the MST running on 1 to 96 cores. It can be seen from this figure that edge sorting takes a significant proportion of overall MST execution time. For all three benchmarks, we observed consistent behavior. The fraction of time taken for edge sorting ranges from 36% to 75%. In addition to execution time, edge sorting consumes computation resources that could otherwise be used for other computation tasks. Examining the experimental results, we believe such expensive edge preprocessing can be offloaded to data storage device where the large amount of edges is stored.

B. Previous Work on Near-Data Processing

In many computer systems for the data mining, big-data, and database, the data movement becomes the bottleneck that it causes performance degradation and power waste [34]. Data processing is swiftly moving from computing-centric to data-centric. Inspired by these trends, the concept of NDP [10] (Near-Data Processing) has recently attracted considerable interest: Placing the processing power near the data, rather than shipping the data to the processor. The NDP computation might execute in memory or in the storage device where the input data reside [11], and it can be divided into two main categories: PIM and PIS.

PIM aims at performing computation inside main memory. Various PIM approaches have been proposed since the pioneering work by Gokhale et al. [12]. Recently, Yitbarek et al. [13] have reported accelerator logic for string matching, memory copy, and hash table lookups in hybrid memory cube (HMC) [14][15]. Ahn et al. [16] proposed a scalable PIM architecture for graph processing with five workloads including average teenage follower, conductance, PageRank, single-source shortest path, and vertex cover. They verified the graph processing performance by simulation.

PIS aims at processing in storage (PIS). Early PIS approaches include the Active Disks architecture proposed by Acharya et al. [17]. They perform the scan, select, and image conversion in storage system and provides a potential reduction of the data movement between disk and CPU. Patterson et al. [18] proposed an architecture (IDISK) which integrates the embedded processors into the disk and push computation closer to the data. Their results suggest that a PIS based architecture can be significantly faster than a high-end symmetric multiprocessing (SMP) based server. Choi et al. [19] implemented algorithms for linear regression, k-means, and string matching in the flash memory controller (FMC). BlueDBM [20] is a PIS system architecture for distributed computing systems with a flash memory-based FPGA. The authors implemented nearest-neighbor search, graph traversal, and string search algorithms by High-Level Synthesis (HLS) in FPGA. Morpheus [33] frees up scare CPU resources by using embedded processor inside SSD to carry out object deserialization. Recently, Biscuit [21] equipped with FMCs and processes pattern matching logic in storage which speeds up MySQL requests. Lee et al [35] proposed ExtraV, a framework for near storage graph processing such as Average Teenage followers, PageRank, Breadth-First Search and Connected Components. It efficiently utilizes a hardware accelerator at the storage side to achieve performance and flexibility at the same time.

Our focus in this paper is on speeding up graph processing that has become increasingly important in today's big data era. As will be evidenced shortly, the benefit is great to preprocess a huge amount of graph data inside SSD where the data is stored.

III. Hardware Architecture of In-storage Sort

A. System architecture

The large fraction of time that edge sorting takes in MST and the intelligence available inside modern SSD motivate us to propose a new and practical PIS architecture. Compared with the existing PIS approaches, CISC is unique in that it uses Verilog to generate RTL and provides wire-speed sort in hardware. A pipelined circuit structure was designed to tailor to high-speed storage data sort especially. Graph edge sort is done concurrently with data transfer on the bus. It minimizes sort overhead of the host server CPU which is computation intensive and time-consuming.

As shown in Figure 2, PIS augments a special functional logic to perform the desired function inside a storage device, in this case, SSD. All the storage control functions are implemented on an FPGA. Inside FPGA chip, major storage logic units include the flash controller, NVM-e interface, DMA engine and in-storage sort module. All modules are connected to AXI4 bus which is a bridge for data movement between host and flash memory. The data width of AXI4 bus is 8 bytes with clock speed of 250MHz. As shown in Figure 2, in-storage sort module is added between AXI bus and NVM-e interface along with storage read I/O path. It provides sort function that is activated by NVM-e command and is done while data is being read from the storage to the host.

B. In-storage sort module

A challenging problem of hardware sort is to sort the large-scale dataset. Due to the on-chip memory size limitations of FPGA, the existing work [22$\sim$25] partially buffers sorted results in the off-chip memory such as DRAM or SSD and reads them back when FPGA performs merge sort. Such off-chip buffer strategy causes multiple FPGA memory accesses and slows down the hardware sort performance of the large-scale dataset.

In order to eliminate the off-chip memory accesses in FPGA sort, CISC takes a divide and conquer approach. Instead of sorting the entire edge list that is huge, we divide the large edge list into chunks and sort these chunks using hardware. Each chunk can fit into FPGA on-chip memory. The pipeline architecture of in-storage sort module provides wire-speed sort of data streams. There are two benefits of dividing edges into chunks to sort. The first one is memory resource savings. It is impossible to hold and reorder large-scale data in FPGA alone. We choose the right chunk size to fit the internal memory space of the FPGA. The second benefit is to bound the in-storage sort latency to match the normal read I/O speed so that the host can read the sorted chunks as if they were directly read from flash memories with no interruption. Once the sorted chunks of edges are loaded to the system RAM, the software on the host can efficiently execute the remaining computation of MST.

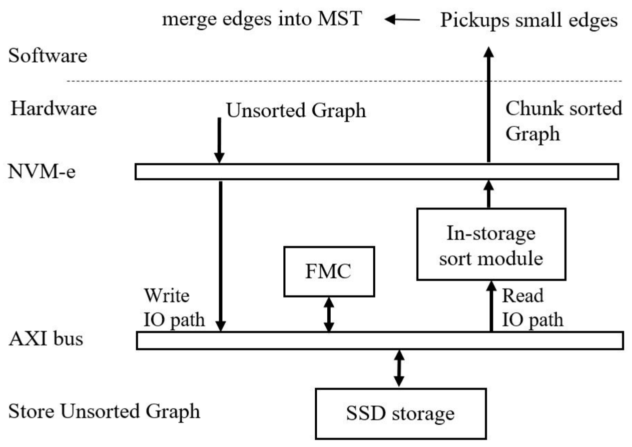

Fig. 2. System architecture of the CISC.

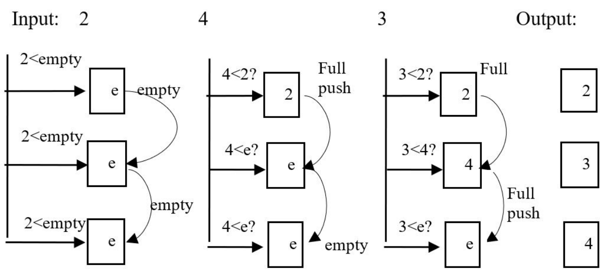

Fig. 3. The architecture of the linear-time sorter.

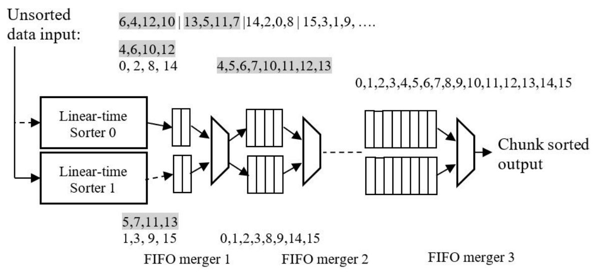

Fig. 4. The pipeline architecture of the in-storage sort module.

The in-storage sort pipeline is composed of the linear-time sorters [25] and several stages of FIFO mergers [22] [24]. We design this architecture especially for the in-storage graph processing with the minimal PIS latency and hardware cost.

As the first stage of the pipeline, the linear-time sorter uses *n* buffers to hold sorted graph edges. It compares each incoming edge's weight in parallel with all already sorted edges in the buffers and inserts the new graph edge into the appropriate location in the buffers to maintain the existing sorted order [25]. Figure 3 shows an example of *n* equal to 3 to demonstrate how the linear-time sorter works. Such linear-time sorter generates the sorted sequence of *n* edges after *n* clock cycles.

Upon a read I/O from the host, the storage data need to be continuously fed into the PIS function. In order to sort data stream in wire-speed, two linear-time sorters are deployed to work in parallel. As shown in Figure 4, the two linear-time sorters alternate working on the input data and switch functions after every *n* clock cycles with one sorting the incoming data stream and the other sending out the sorted results to the next pipeline stage.

The linear-time sorter requires buffers and parallel comparators for the parallel comparisons. Such buffers and comparators will become prohibitive costly if the sorted data size becomes very large. Our solution is again divide and conquer by dividing each chunk of data to be sorted into smaller segments. The dual linear-time sorters only sort the initial segment with a small data size. The in-storage sort module then doubles up such segment by FIFO mergers [24] that form the rest of the pipeline stages as shown in Figure 4. To connect the first pipeline stage (dual linear-time sorters) with the rest of pipeline stages (FIFO mergers), the *n* sorted edges from the linear-time sorter0 are immediately forwarded to one of the FIFO buffers of the next pipeline stage, FIFO merger1, as shown in Figure 4. During the next *n* cycles, the linear-time sorter1 fills up the other FIFO buffer of the next pipeline stage. The same process repeats when the storage data continuously flushes into the PIS module.

Each FIFO merger stage doubles up the segment of the previous pipeline stage [24]. For example, the size of data sort doubles up from 4 to 16 when the data stream passes through two stages of the FIFO mergers. As shown in Figure 4, each stage of the FIFO merger has two FIFOs. At any given time, the data stream from the previous pipeline stage flushes to one of the FIFOs. If the flush size reaches the size of the previous segment, a control logic switches the data stream to the other FIFO. After one of the FIFO has finished fetching data with the size of the previous segment, the data merge starts and the fetching data flushes into the other FIFO at the same time. Data in the two FIFOs are merged in ascending order to the next pipeline stage of the FIFO merger, that is, we always pick up the smaller data from two FIFOs to be flushed to the next stage [24]. In this way, the current segment merges two of the previous segments and doubles up the sort size. The sort size of the last segment is the chunk size that depends on the FPGA's internal resources (numbers of FIFO merger stages). After passing through the in-storage sort module, the graph edges are loaded into the host main memory in form of multiple sorted chunks.

The startup time of such pipeline of FIFO mergers depends on the data transfer delay of the last stage of the FIFO merger [24]. The delay is the data transfer time of the first chunk of the graph data. Therefore, PIS latency is only the pipeline's startup time when the host server reads the first chunk of a large number of sorted chunks from the storage.

IV. Software Design of CISC

To allow the MST application to use the in-storage sort module, we developed two CISC software modules running on the host, one for single core CPU and the other for parallel MST running on multicores. The following paragraph describes the software design of CISC.

A. Serial CISC software

As shown in Figure 5, a B-tree based selection algorithm has been developed in the serial CISC software and coordinates with the chunk sorting circuit. To initialize the B-tree, the serial CISC software picks up graph edges from all the chunk heads. It has n_{edge}/*chk_size* B-tree nodes holding the smallest edges of all the chunks, where n_{edge} denotes edge numbers and *chk_size* is the chunk size, i.e. the number of sorted edges per chunk. In order to avoid collision, each B-tree node adds the same weight edges into a linked list. After the initialization of B-tree, the serial CISC software trims the minimum edge from the B-tree by the rule of in-order traversal [26]. A new edge from the chunk head of the trimmed edge is the next B-tree candidate, and the software inserts it into the B-tree after

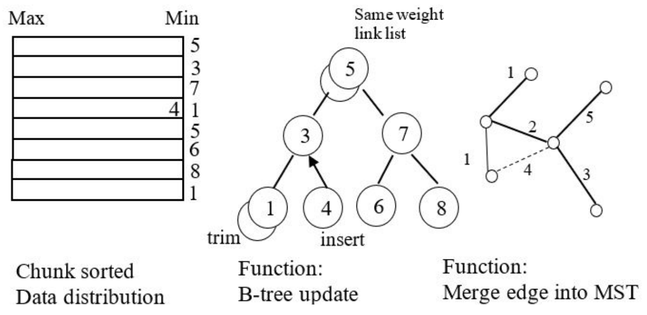

Fig. 5. MST software of CISC on the single-core system.

the previous minimum edge is trimmed. The size of the B-tree remains the same (n_{edge}/*chk_size*) during software execution.

As shown in Algorithm 1, the serial CISC software merges the trimmed edges from the B-tree into a graph subset. Once the growing subset forms a cycle, the software abandons the currently selected edge and picks the next edge from the B-tree to grow MST. Such a process stops when MST traverses all n_{vertex} vertices of the graph, which takes n_{merge} iterations in Algorithm 1. The B-tree selection avoids sorting all the graph edges because the smaller edges are placed at the heads of chunks and merge process always picks up the smallest from the B-tree into MST.

For the same graph, both the software MST and the CISC MST will select the same set of graph edges to form the MST and take same number of iterations, n_{merge}, to merge the small edges into the MST. The superiority of serial CISC MST comes from the graph sort. Instead of sorting n_{edge} edges, the serial CISC MST picks up a minimum graph edge from the B-tree and merges the graph edge in every iteration. The number of iterations (n_{merge}) is related to n_{vertex} and much smaller than n_{edge} in most graphs in practice.

Table 1 shows the time complexities of the sorting part of the MST algorithm of traditional software MST and our CISC. While the best time of software sort is $n_{edge} \times log_2(n_{edge})$, CISC sort time is $n_{merge} \times log_2(n_{edge}$/*chk_size*). During the CISC software execution, the B-tree size remains the same (n_{edge}/*chk_size*). For B-tree updates, the time complexity is $log_2(n_{edge}$/*chk_size*). Such B-tree update is performed concurrently with the merge operations of MST. CISC sort finishes when all of the graph vertices (n_{vertex}) are merged, taking n_{merge} iterations. From the comparison of these two formulas, we can see CISC takes advantages of both smaller value of n_{merge} and the efficient data distribution of sorted chunks.

TABLE I
COMPARISON OF THE SERIAL SORT BETWEEN SOFTWARE AND CISC

	Execution time of serial sort
Software	O($n_{edge} \times log_2(n_{edge})$)
CISC	O($n_{merge} \times log_2(n_{edge}$/*chk_size*))

Algorithm 1: The serial CISC software of MST

```
0:  $n_{merge}$ = 0;
1:  Initial B-tree size = $n_{edge}$/chk_size;
2:  for $k < n_{vertex}$
3:      select edge = trim (B-tree);
4:      update (B-tree);
5:      if (merge_MST (select edge) == success) $k$++;
6:      else $k$ = $k$;
7:      $n_{merge}$++;
8:  end for
```

B. Parallel CISC software

The parallel CISC software cannot use the B-tree selection algorithm because of data dependency. Each edge selection depends on the previous updates of the B-tree, and it may cause task deadlocks wasting the multicores' computational resources.

In order to speed up MST in the multicore system, we optimized the classical sample sort algorithm [8] of parallel MST. The concept of the sample sort is to divide the dataset into segments, and the data values within each segment have a range. The ranges among segments are non-overlapping. CPU cores sort these segments in parallel and complete the sample sort after combining all of the sorted segments. However, in most cases, the unsorted data does not follow the above segments' data distribution. The sample sort algorithm needs reshuffle the dataset by selecting samples and partition segments. Figure 6 shows a sample sort example of the n_{total}=24 sorting elements with p=3 parallel tasks. There are four major steps of the sample sort algorithm:

(1) Local sort: Multiple tasks divide the n_{total} elements into p chunks of the size n_{total}/p each and sort these chunks in parallel.

(2) Select & sort samples: The sample sort algorithm chooses m=2 samples evenly from each sorted chunk and then sorts all these selected samples with the total number of $m \times p$.

(3) Segment partition: From the above $m \times p$ samples, the

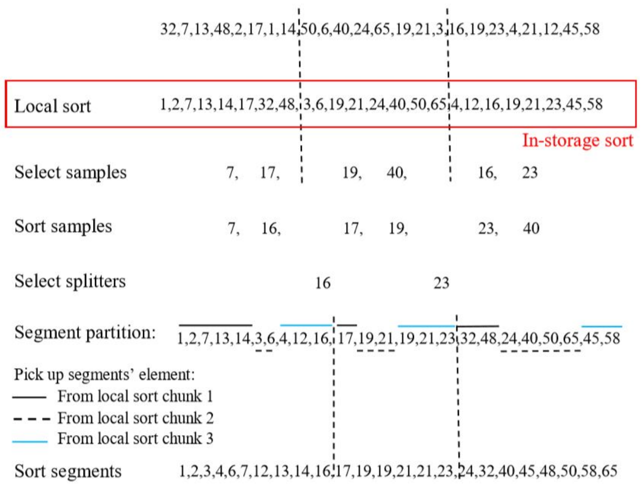

Fig. 6. CISC optimizes sample sort algorithm in MST.

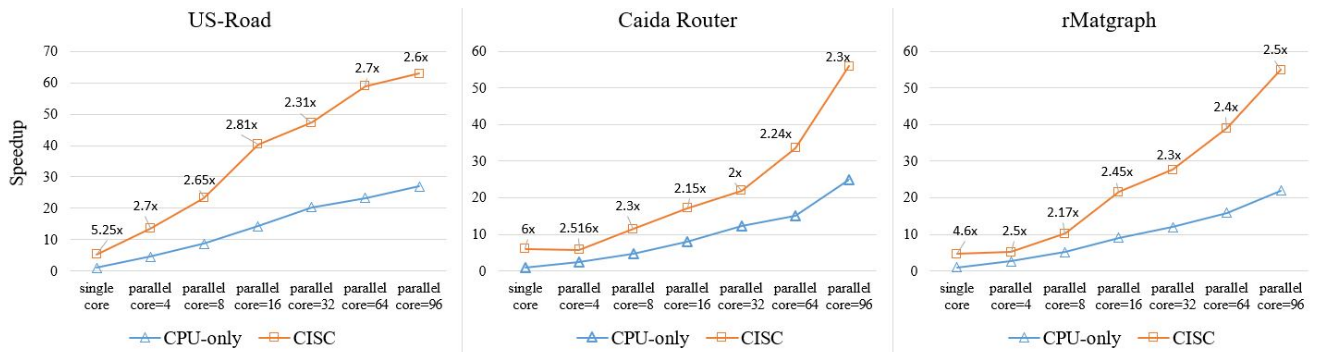

Fig. 7. The sort speedup of CISC, the baseline is serial software sort running on single-core.

sample sort algorithm evenly selects *p-1* samples as splitters. These splitters partition the dataset into *p* segments with non-overlapping ranges.

(4) Segment reorganization: The multiple tasks pick up the segments' candidates from each local sorted chunk according to the value ranges and then sort each segment in parallel. The sample sort completes after combining all the sorted segments.

The sample sort algorithm is suitable for the multicores system because the local sort (step 1) and segment reorganization (step 4) can be executed in parallel. However, each parallel task still sorts a large number of graph edges, which is computation intensive and time-consuming. It also has a synchronization problem of multiple tasks because the sample sort waits for all the parallel tasks to be finished before the next step of processing.

The parallel CISC software optimizes the sample sort algorithm by skipping the local sort (step 1). The in-storage sort circuit divides a large amount of graph edges into chunks of size n_{total}/*p* each and sorts each chunk of edges in order using hardware. As shown in Figure 6, CISC provides an efficient data distribution for the rest of the sample sort's steps and avoids the local sort of parallel tasks in the host main memory.

In the parallel CISC software of MST, we did not change the original design of graph merge. According to the benchmark baseline [9], the parallel MST starts to merge after graph sort (sample sort) is completed. It merges sorted edges into the graph subsets with multiple tasks and grows by several sub-trees in parallel. The parallel MST computation finishes when all the sub-trees join together and MST traverses all the graph vertices. As will be shown latter in our experiment, CISC offers overall speedup of MST due to the optimized sample sort.

V. Evaluation

In order to evaluate how CISC performs in comparison with traditional approaches, we have built an NVM-e SSD prototype that implements CISC. The hardware chunk sort module is augmented inside the FPGA controller of the PCIe SSD. The PCI-e SSD card is inserted to a multi-core server to carry out a performance evaluation of CISC. This section discusses the prototype setup and evaluation results.

A. Experimental Platform and Benchmark Selection

We set up the experimental environment on an Intel Xeon processor with 96 cores. It runs at 2.5 GHz and hosts a Linux system with kernel version 4.14. The system contains a PCI-e 3×4 that connects our CISC storage and other peripherals.

We built our CICS prototype on top of the Open-SSD platform [27]. All storage logic fits into Xilinx Zynq-7000 series FPGA, including a dual-core ARM processor, DRAM/flash controller logic, NVM-e interface and CISC's in-storage sort module. The ARM processor runs at 1GHz clock speed, and this platform contains 1GB DDR2 and 256GB flash memory. To evaluate CISC, we store MST benchmark files on SSD before the host starts the MST application. The in-storage sort module is set to sort 128K edges (*chk_size*) per chunk.

Three benchmark datasets are chosen from [28∼30], including transportation, Internet data analysis and Graph Mining, as listed in Table 2. The PBBS benchmark [9] source code is used in our design as the baseline to evaluate the performance difference between CISC and the traditional software. We compose CISC software to replace the sample sort and serial MST in the baseline. The parallel software code uses OpenMP configured for multicores.

TABLE II
The Benchmark datasets we used in this paper

	n_{vertex}	n_{edge}	Description
US-Road	23,947,347	58,333,344	Transportation
Caida Router	12,190,914	34,607,610	Network
rMatgraph	10,000,000	50,000,000	Graph Mining

B. Numerical Results and Discussions

Since edge sort is the main part that CISC offers performance advantages for MST computation, we first carried out experiments to measure the execution times of edge sort using CISC and traditional software approach (CPU-only).

Figure 7 plots the speedup of CISC sort over the traditional software sort. As shown in the figure, CISC achieves as much as a 4.6∼6× speedup compared to the pure software sort on the single core. The B-tree algorithm can process smaller edges effectively and finishes MST as soon as the software

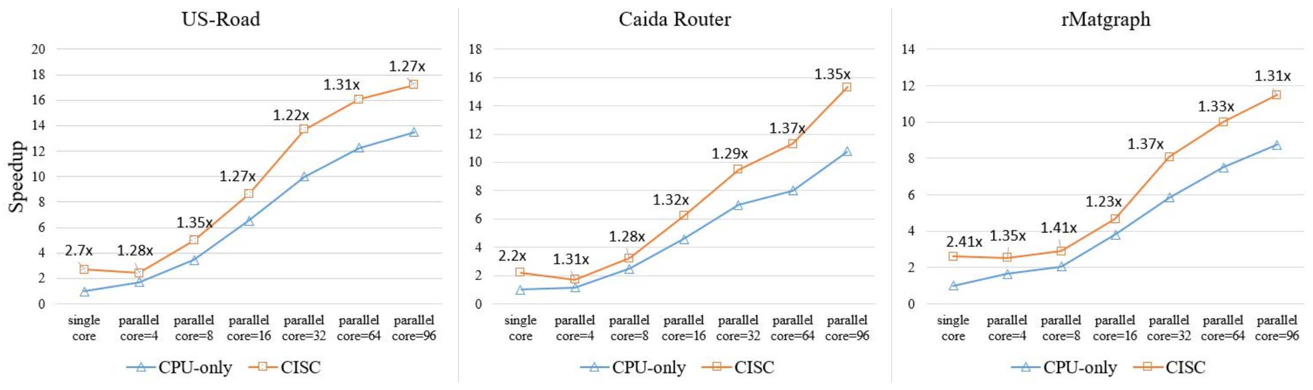

Fig. 8. The MST speedup of CISC, the baseline is serial MST running on single-core.

traverses all the graph vertices. The traditional software sort, on the other hand, needs to sort all the graph edges before the MST merge can start.

The speedup of parallel software sort increases with the increase of the number of cores on the host server. Compared with single core, the speedup increases to 22∼27× as the number of cores increases to 96, as illustrated by the blue line plots in Figure 7. For the same number of cores, our parallel CISC outperforms the traditional software sort. For all the benchmarks considered, we observed 2∼2.81× speedup compared to the traditional software sort with the same number of cores. These speedups can be mainly attributed to the elimination of parallel local sort tasks and partially offloading of computational resources from multi-cores to the SSD. As shown in Figure 7, the parallel CISC sort on 96 multicores shows 55× to 62× speedup compared to the traditional software sort on a single-core.

The overall speedup of the MST application depends on the fraction of sort time over total execution time. For a comparative analysis, we consider the baseline as running MST on a single-core with the in-storage sort module disabled. Figure 8 shows measured results for the benchmarks considered. We observed speedups of 2.2∼2.7× on single-core and a 1.3× speedup on multicores on average. The speedup ratio of a single-core is more significant than multicores because of the time fraction difference of edges' sort. The larger the fraction of graph sort time it takes, the more speedup CISC can obtain. As shown in Figure 2, the sort execution time on single-core consumes 65% to 75% of the overall MST execution, and parallel MST takes 31% to 46% execution time for the graph sort. Thus, the speedup ratio of multicores' MST is less significant than for single-core.

The speedup of parallel MST increases when using more CPU resources of the host server. As shown in Figure 8, CISC always runs faster than the traditional software with the same number of cores. It outperforms purely multicore systems because CISC obtains performance gains from both multicores and the in-storage sort. Compared to a single-core MST baseline, CISC outperforms traditional software by 11.47 to 17.2 times on 96-cores systems.

C. Hardware Cost Analysis

CISC partially offloads the expensive computation from the host server to the SSD. The additional hardware cost of implementing CISC inside an SSD controller includes logic cells, LUT, Flip-flops, and RAM. Table 3 lists the usage of hardware resources of CISC's in-storage sort module, as a fraction of total available hardware resources of FPGA chips.

Our CISC prototype is built on top of the open-SSD platform [27] with Zynq XC7z045 chip. It is not the latest FPGA with limited hardware resources. As shown in Table 3, the in-storage sort module takes 11% of LUT and 41% of RAM resources on Zynq XC7z045 chip. More recent FPGAs doubled and even quadrupled the on chip resources. The Ultra-Scale series FPGA of Zynq and Virtex are commonly used in the modern SSD controllers [31][32]. The hardware cost of CISC becomes insignificant on the latest Ultra-Scale series FPGAs. As shown in Table 3, the hardware resource utilization on such FPGAs is very low. The in-storage sort module of CISC takes 9.3% of LUT and 20% of RAM on the Ultra-scale Zynq. It only takes 1.3% of LUT and 1.7% of RAM on the Ultra-scale Virtex. Therefore, the hardware cost of CISC can be considered negligible on modern FPGAs. The sort module can also be extensible to many sort-based PIS functions and storage ASIC design.

TABLE III
HARDWARE RESOURCE UTILIZATION OF CISC ON DIFFERENT FPGAS

	CISC	Zynq dev: XC7z045		Zynq Ultra-Scale dev: ZU9CG		Virtex Ultra-Scale dev: VU13P	
		Total	Used	Total	Used	Total	Used
Logic cells	21.7K	350K	6.2%	600K	3.6%	3780K	0.5%
LUT	25.7K	218K	11%	274K	9.3%	1728K	1.4%
Flip-flop	17.3K	437K	4%	548K	3.1%	3436K	0.5%
RAM	1MB	2.4MB	41%	5MB	20%	57.5MB	1.7%

VI. Conclusion

In this paper, we have presented a new approach to the MST computation by coordinating computing power inside SSD storage with the host CPU, referred to as CISC. CISC exploits the controller logic inside the SSD to sort graph data while being loaded to the main memory of the host. In order to achieve wire speed, CISC takes a divide and conquer approach by partitioning MST graph edges into chunks and sorts each chunk using hardware. In this way, the MST can then proceed by selecting the smallest edge among the chunks and ensures smaller weight edges can be processed efficiently. To demonstrate the feasibility and performance potential of CISC, we have built a working prototype that consists of both software running on the host and hardware sort module inside the SSD. Extensive experiments have been carried out using real-world benchmarks to demonstrate the feasibility and performance of deploying CISC in NVM-e SSD storage. Our experimental results show 2.2∼2.7× speedup for serial version implementation and 11.47× to 17.2× speedup for the parallel version with 96-cores. For the same number of cores, our parallel CISC outperforms the traditional software MST by up to 35%. We believe the PIS function of CISC can be extended to other applications requiring data sort with an addition of similar CISC software module running on the host.

VII. Acknowledgements

This research is supported in part by the NSF grants CCF-1439011 and CCF-1421823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. It is also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd, Shenzhen Peacock Plan (KQTD2015091716453118).

References

[1] Zhu, X, Han, W, Chen, W. "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning." In Proceedings of the Usenix Annual Technical Conference (2015), USENIX Association, pp. 375386.

[2] Zhaoshi Li, Leibo Liu "Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware", proceeding of the 44th Annual International Symposium on Computer Architecture ISCA 17.

[3] David A. Bader, Guojing Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," in Parallel Distrib. Comput. 66 (2006) 13661378.

[4] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for ne-grain nested speculative parallelism," in ISCA-44, 2017.

[5] S. Rostrup et al., "Fast and memory-efcient minimum spanning tree on the GPU," IJCSE, vol. 8, no. 1, pp. 21-33, 2011.

[6] S. Manoochehri , B. Goodarzi , D. Goswami "An Efficient Transaction-Based GPU Implementation of Minimum Spanning Forest Algorithm" High Performance Computing & Simulation (HPCS), 2017 International Conference

[7] C. Luk, R. Cohn, R. Muth et al., "Pin: building customized program analysis tools with dynamic instrumentation," in PLDI, 2005.

[8] Sample sort algorithm: http://parallelcomp.uw.hu/ch09lev1sec5.html

[9] PBBS benchmark suit of Minimum spanning tree baseline code reference: http://www.cs.cmu.edu/ pbbs/benchmarks.html

[10] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a MICRO-46 workshop," Micro, IEEE, vol. 34, no. 4, pp. 36-42, 2014.

[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016, pp. 27-39.

[12] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," Computer, vol. 28, no. 4, pp. 23-31, 1995.

[13] S.F. Yitbarek, T. Yang, R. Das, and T. Austin, "Exploring specialized near-memory processing for data intensive operations," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016. IEEE, 2016, pp. 1449-1452.

[14] T.S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of ash translation layer," Journal of Systems Architecture, vol. 55, no. 5, pp. 332-343, 2009.

[15] J.T. Pawlowski, "Hybrid memory cube (HMC)," in Hot Chips 23 Symposium (HCS), 2011 IEEE. IEEE, 2011, pp. 1-24.

[16] J. Ahn et al, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in ISCA-42, 2015, pp. 105117.

[17] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in ACM SIGOPS Operating Systems Review, vol. 32, no. 5. ACM, 1998, pp. 81-91.

[18] K. Keeton, D. Patterson, and J. Hellerstein. "A Case for Intelligent Disks (IDISKs)," SIGMOD Record,27(3):42-52, September 1998.

[19] I.S. Choi and Y.S. Kee, "Energy efcient scale-in clusters with instorage processing for big-data analytics," in Proceedings of the 2015 International Symposium on Memory Systems. ACM, 2015, pp. 265-273.

[20] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu et al., "Bluedbm: an appliance for big data analytics," in Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on. IEEE, 2015, pp. 1-13.

[21] B. Gu, A.S. Yoon, D.H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho et al., "Biscuit: A framework for near-data processing of big data workloads," in Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016, pp. 153-165.

[22] Wei Song, Dirk Koch, Mikel Lujan and Jim Garside. "Parallel Hardware Merge Sorter." In FCCM 2016

[23] Sang-Woo Jun, Shuotao Xu, Arvind. "Terabyte Sort on FPGA-Accelerated Flash Storage." In FCCM 2017

[24] Dirk Koch, Jim Torresen. "FPGASort: A High-Performance Sorting Architecture Exploiting Run-time Reconguration on FPGAs for Large Problem Sorting." In FPGA 2011

[25] Linear sort FPGA design: https://hackaday.com/2016/01/20/a-linear-time-sorting-algorithm-for-fpgas/

[26] In-order traverse: https://en.wikipedia.org/wiki/Tree_traversal

[27] Open-SSD+ platform: http://www.openssd.io/.

[28] Benchmark: https://www.cc.gatech.edu/dimacs10/downloads.shtml

[29] DIMACS 10 challenge graph collection Graph Partitioning and Graph Clustering: http://www.cc.gatech.edu/dimacs10/downloads.shtml.

[30] D. A. Bader and K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," Technical Report, 2006.

[31] Zynq FPGA: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[32] Virtex series FPGA product page:https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[33] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, S. Swanson, *Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing*, in Proceedings of the International Symposium Computer Architecture, 44(3): 53-65, ACM/IEEE,2016.

[34] H. Choe, S. Lee, H. Nam, S. Park, S. Kim, E. Chung, S. Yoon, *Near-Data Processing for Differentiable Machine Learning Models*, arXiv:1610.02273v3.

[35] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, Peter Hofstee, Gi-Joon Nam, Mark Nutter, Damir A. Jamsek, *ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator*, PVLDB 10(12): 1706-1717 (2017)