Quadtree-Based Lightweight Data Compression for Large-Scale Geospatial Rasters on Multi-Core CPUs

Jianting Zhang
Dept. of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, USA
syou@gc.cuny.edu

Le Gruenwald
Dept. of Computer Science
The University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

Abstract— **Huge amounts of geospatial rasters, such as remotely sensed imagery and environmental modeling output, are being generated with increasingly finer spatial, temporal, spectral and thematic resolutions. In this study, we aim at developing a lightweight lossless data compression technique that balances the performance between compression and decompression for large-scale geospatial rasters. Our Bitplane bitmap Quadtree (or BQ-Tree) based technique encodes the bitmaps of raster bitplanes as compact quadtrees which can compress and index rasters simultaneously. The technique is simple by design and lightweight by implementations. Except computing Z-order codes for cache efficiency, only bit level operations are required. Extensive experiments using 36 rasters of the NASA Shuttle Range Topography Mission (SRTM) 30 meter resolution elevation data with 20 billion raster cells have shown that our BQ-Tree technique is more than 4X faster for compression and 36% faster for decompression than zlib using a single CPU core while achieving similar compression ratios. Our technique further has achieved 10-13X speedups for compression and 4X speedups for decompression using 16 CPU cores on the experiment machine equipped with dual Intel Xeon 8-core E5-2650V2 CPUs. Our technique compares favorably with the best known technique with respect to both compression and decompression throughputs.**

Keywords—Lightweight, Quadtree, Compression, Geospatial

I. Introduction

Advancements in remote sensing technologies and environmental modeling have generated huge amount of large-scale geo-referenced raster data (or geospatial rasters for short). There are three orders of magnitude of difference between CPU and disk I/O speeds and two orders of magnitude of difference between CPU speed and network bandwidth [1]. Disk I/O and network bandwidth are among the most significant bottlenecks in processing large-scale geospatial rasters.

Data compression is a popular approach to reducing data volumes and hence lowering disk I/O and network data transfer times. While several lossy data compression techniques have demonstrated excellent compression ratios, lossless data compression techniques are still among the most popular ones in geospatial processing due to data quality and accuracy concerns. The popular LZ77 algorithm [2] is considerably sophisticated which makes software tools that are based on it, such as zlib[1], heavyweight and less flexible to new hardware architectures. In addition, the LZ77 algorithm is asymmetric by design and it is much more expensive to compress than to decompress. While this is a good choice in many applications that involve more decompression than compression, it is not well suitable for quite some geospatial processing applications that symmetric algorithms and implementations are more preferable. For example, many intermediate results of environmental modeling are only compressed and decompressed once for the sole purpose of reducing disk or network traffic in distributed computing. The frequencies of compression and decompression are roughly the same and very often data compression and decompression are interleaved with other types of computation.

In this study, we aim at developing a lightweight data compression technique for large-scale geospatial rasters on modern CPUs, with respect to both algorithmic complexity and computing time, to effectively reduce data communication time due to disk and/or network I/Os. The proposed technique is based on our Bitplane bitmap Quadtree (BQ-Tree for short) that was originally developed for indexing geospatial rasters on GPUs [3]. The basic idea of BQ-Tree is to represent all the bits of all raster cells in a bitplane as a bitmap and then use a quad-tree structure to code the bitmap efficiently. For a K-bit raster, there will be K bitplane bitmaps and the resulting K BQ-Trees will be concatenated as the compressed data. Similar to many data compression techniques, large rasters can be segmented into chunks to facilitate parallelization and bound computation overhead. Comparing with zlib, our technique is conceptually simple and can be implemented in a few hundreds of lines of code (including both compression and decompression, *source code available at*[2]) and embedded in various applications.

The rest of the paper is arranged as follows. Section II introduces background, motivation and related work. Section III presents the BQ-Tree designs and implementations on multi-core CPUs. Section IV provides experiment results on NASA SRTM 30 meter elevation data and compares our technique with zlib. Finally, Section V concludes the paper and discusses future work directions.

II. Background and Motivation

Geospatial processing has been undergoing a paradigm shift in the past few years. Advancements in remote sensing technology and instrumentation have generated huge amounts of remotely sensed imagery from air- and space-borne sensors.

In recent years, numerous remote sensing platforms for Earth observation with increasing spatial, temporal and spectral resolutions have been deployed by NASA, NOAA and the private sector. The next generation geostationary weather satellite GOES-R serials[3] (whose first launch is scheduled in 2016) will improve the current generation weather satellites by 3, 4 and 5 times with respect to spectral, spatial and temporal resolutions [4]. With a temporal resolution of 5 minutes, GOES-R satellites generate 288 global coverages every day for each of its 16 bands. At a spatial resolution of 2km, each coverage and band combination has 360*60 cells in width and 180*60 cells in height, i.e., nearly a quarter of a billion cells. While 30-meter resolution Landsat TM satellite imagery is already freely available over the Internet from USGS[4], sub-meter resolution satellite imagery is becoming increasingly available, with a global coverage in a few days. Numerous environmental models, such as Weather Research and Forecast (WRF [5]), have generated even larger volumes of geo-referenced raster model output data with different combinations of parameters, in addition to increasing spatial and temporal resolutions. For example, the recent simulation of Superstorm Sandy[6] on NSF Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA) has a spatial resolution of 500 meters and a 2-second time step. Running at a 9120*9216*48 three-dimensional grid (approximately 4 billion cells), a single output file is as large as 264 GB [5].

The large volumes of geospatial raster data have imposed a significant challenge on data management. Despite storing large amounts of data is getting cheaper and cheaper due to fast decreasing cost of magnetic disks, moving such data through various processing pipelines are getting more and more expensive when compared with floating point computation. It is thus desirable to allocate more computing power to data compression and reduce data volumes that need to go through disk I/O controllers and network interfaces to improve overall system performance.

As a well established research discipline, many lossy and lossless data compression techniques have been developed in the past few decades, especially for text and image data [2]. However, many of these data compression techniques are optimized for compression ratios and/or data transmission over noisy data communication channels (e.g., wireless). The sophisticated designs not only require complex implementations but also make them difficult to parallelize. For example, to the best of our knowledge, despite several efforts (e.g., [6]), efficient implementations of the LZ77 algorithm on GPUs are still in early stages.

Although it seems to be straightforward to simply apply image compression techniques to geospatial rasters to achieve better compression ratios than the generic tools such as zlib, there are several concerns. First of all, many image compression techniques are lossy in nature. While they may perform well in terms of both compression ratios and computing overhead when operated under the default lossy compression modes, the performance can be significantly poorer if lossless compression is required even if the techniques may allow lossless compression as an option. Second, many geospatial rasters use special values (e.g., -1 or 65535) to represent outliers (e.g., no values, masked cells). These outliers may have significantly different values than their neighbors and may generate large coefficients for many transformation based techniques, such as wavelets and delta coding [2].

The most relevant work to ours is the In-Situ Orthogonal Byte Aggregate Reduction (ISOBAR) lossless compression technique for scientific data due to Schendel et al [7]. The technique performs favorably over zlib in terms of both throughput and compression ratio. ISOBAR and BQ-Tree share the similarity of searching for compressible blocks in a top-down manner. However, ISOBAR searches for arbitrarily shaped flat rectangular blocks at byte level while BQ-Tree searches for homogenous quadrants at bit level in a hierarchical manner. While it is possible to support query processing based on the derived block level metadata in ISOBAR by scanning all the blocks, the hierarchical tree structure derived by the BQ-Tree approach could be more efficient in query processing. Our BQ-Tree technique has comparable performance on a single CPU core based on the results reported in [7]. However, no parallel results have been reported in [7] and the ISOBAR's scalability on multi-core CPUs is thus unclear.

III. Bitplane Quadtree

A. The BQ-Tree Data structure

A K-bit raster is first split into K bitplanes and each bitplane becomes a bitmap. Given a bitplane bitmap of a raster R of size N*N (without losing generality, assuming $N=2^n$), as illustrated in Fig.1, it can be represented as a quadtree where black leaf nodes represent quadrants of presence ("1"), white leaf nodes represent quadrants of absence ("0") and internal nodes are colored as gray. The quadtree can be easily implemented in main-memory by using pointers or stored on hard drives as a collection of linear quadtree paths. However, while the storage overheads of pointers or the paths can be justified if the length of the data field is much larger than the length of the pointer field (4 bytes for 32-bit machines and 8 bytes for 64-bits machines), the overhead is unacceptable as the data field is intended to be only 1-bit long to encode a bitplane bitmap. Furthermore, as the memory pointers are allocated dynamically and can point to arbitrary memory addresses, they are known to be cache unfriendly. To overcome these problems, we have designed a spatial data structure called BQ-Tree to efficiently represent bitmaps of bitplanes of a raster [3].

Our BQ-Tree technique sequences nodes of a regular quadtree into a byte-stream through breadth-first traversals with sibling nodes following the Z-order [8] as shown in Fig. 2. Different from classic main-memory quadtrees that use pointers to address child nodes, the child node positions in a BQ-Tree do not need to be stored explicitly. As such, the pointer field in regular quadtrees can be eliminated which reduces storage overhead significantly. In addition to tree nodes, a BQ-Tree also includes a compacted "last level" quadrant signature array (to be described shortly).

The layout of BQ-Tree nodes is as follows. Each BQ-Tree node is represented as a byte (8 bits) with each child quadrant takes two bits. We term the two bits as a child node signature. The three combinations correspond to three types of nodes in classic quadtrees: "00" corresponds to white leaf nodes, "10" corresponds to black leaf nodes and "01" corresponds to gray nodes. The combination of "11" is currently not used. Child nodes corresponding to the quadrants with "00" or "10" signatures in their parent node can be safely removed from the byte stream as all the four quadrants in the child nodes are the same and their presence/absence information has already been represented in the respective quadrant signatures of the parent nodes (Fig. 2). By consolidating the information of the four child quadrants into a single node, the depth of a BQ-Tree can be reduced by 1 when compared with classic quadtrees. The technique can potentially reduce memory footprint to up to 1/4.

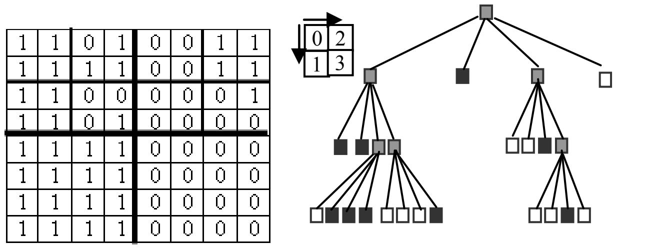


Fig. 1 Quadtree Representation of a bitplane bitmap

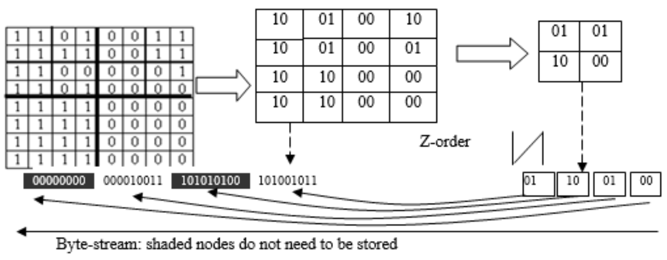
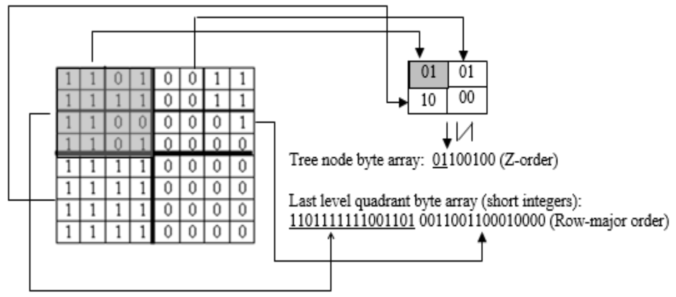


Fig. 2 Streaming BQ-Tree Nodes



01 01
10 00
Tree node byte array: 01100100 (Z-order)
Last level quadrant byte array (short integers):
1101111111001101 0011001100010000 (Row-major order)

Fig. 3 Generating LLQS Array Using a 4*4 Quadrant Size

While it is possible to use different last level quadrant sizes, in this study, we empirically choose a 4*4 last level quadrant size. We refer to [3] for the designs and experiments on using 2*2 last level quadrant size and the discussions on using 8*8 last level quadrant size in Section III.D. We term the concatenation of the bits of the 4*4 quadrant following a row-major order as the Last Level Quadrant Signature (or LLQS). An example to show the processes of compacting LLQSs in the right part of Fig. 3. The bitmap shown in the left part of Fig.3 represents a bitplane of an 8*8 raster, the resulting quad-tree has only one node and its signature is $(01100100)_2$. This is because the signature of the shaded quadrant is 01 (mixture of 0s and 1s) and the signatures of the rest of the three quadrants are 10 (all 1s), 01 (mixture of 0s and 1s) and 00 (all 0s), respectively. The quadrant signatures are concatenated to form the quadtree node for the 8*8 bitmap. Since the first and the third quadrants are mixed of 0s and 1s, their last-level quadrant signatures need to be streamed to the LLQS array, which results in the concatenation of two 16-bit short integers, i.e., $(1101111111001101)_2$ and $(0011001100010000)_2$. Note that the bits of last level quadrants follow row-major order instead of Z-order to minimize Z-order calculation as our experiments have shown that Z-order calculation can be expensive.

B. *BQ-Tree based Compression*

While the BQ-Tree structure was primarily designed for indexing large-scale geospatial raster and was not optimized for data compression [3], as nearby cells of geospatial rasters typically have similar values and their higher bits are often the same, it can be naturally used for lightweight geospatial raster compression.

Step 1: for each of the $2^{m-q}*2^{m-q}$ last level quadrants
1.1 Gather the raster cells in the quadrant and calculate the Z-order code of the quadrant
1.2 For each of the K bits
1.2.1 Generate the last level quadrant signatures and write them to LA
1.2.2 Derive the child node signatures for four neighboring quadrants and form a leaf node; output the leaf node to the last level (level m-q) matrix of PA
Step 2: For each of the m-q-1 levels of PA, loop bottom-up along the pyramid and do the following
2.1 **For each** of the elements of the matrix at the level l
2.1.1 **For each** of the K bits do the following:
2.1.1.1 Examine the four child nodes at the level l+1 matrix and generate the signatures for the four quadrants of the node using the following rules: 0x00->"00", 0xaa->"10", all others ->"01".
2.1.1.2 Concatenate the four 2-bits signatures and write the node value to the level l matrix
Step 3: For all the elements in PA, output bytes that are not 0x00 or 0xaa
Step 4 For all the elements in LA, output short integer values (2 bytes) that are not 0x0000 or 0xFFFF (for LLSQ size of 4*4).

Fig. 4 BQ-Tree based Data Compression Algorithm

The inputs for the compression algorithm are a raster dataset (or a raster chunk) R, raster/chunk size C ($C=2^m*2^m$), last level quadrant size S ($S=2^q*2^q$) and K, which is the number of bits of R. The outputs of the algorithm include the compacted BQ-Tree node array CN and the compacted LLQS array CL. The initialization includes allocating a pyramid array for all bitplanes (PA) and allocating a LLQS array (LA). Clearly the size of PA should be $K*(1+4+16+\ldots+2^{m-q-1}*2^{m-q-1})=K*(4^{m-q}-1)/3$ and the size of LA should be $K*2^{m-q}*2^{m-q}$. The encoding algorithm is given in Fig. 4 which is straightforward to follow by using the examples provided in Fig. 3. First, a whole raster (or a raster chunk) is divided into quadrants based on the last level quadrant size. Both the signatures of the last level quadrants and the corresponding leaf nodes are then generated (Step 1). Second, for each bitplane, a pyramid is generated bottom-up by combining the child node signatures into the parent node signatures (Step 2). Third, starting from the root of the BQ-tree for each bitplane, all the nodes in the matrix correspond to a pyramid level are examined by following the Z-order. The pyramid is then compacted into a byte array for each bitplane by skipping 0x00 (all 0s) and 0xaa (all 1s) bytes (Step 3). Finally, the signatures of the last level quadrants are also compacted into a short integer stream by skipping signatures that are considered to be uniform, i.e., those correspond to "00" or "10" values in any of the four quadrants of leaf nodes of a BQ-Tree (Step 4).

C. BQ-Tree based Decompression

The decoding process is the reverse of the encoding process. A detailed procedure similar to Fig. 4 can be easily constructed and is skipped in the interest of space. Instead, we next present an overview of the steps of the decoding algorithm and discuss several implementation considerations. Starting from the root of a BQ-Tree, the pyramid PA is reconstructed level by level as follows. Each quadtree node is scanned and the signatures of the four child nodes are extracted and examined. Values of 0x00 and 0xaa will be used to update the corresponding matrix elements in the next level (i.e., child nodes in the pyramid layout) if the child node signatures are "00" or "10", respectively. Otherwise, a byte value is retrieved from the compacted BQ-Tree byte stream and used to update the corresponding matrix elements in the next level. After the pyramid is reconstructed, the elements of the last level matrix of the pyramid (correspond to the leaf nodes of the BQ-Tree) are then combined with the LLQS array to reconstruct the original bitplane bitmap by setting the LLQSs with either all 0s and all 1s if the quadrant signatures in the leaf nodes are 00 or 10, respectively; or with the values in the LLQS array if the quadrant signatures in the leaf nodes are 01. Finally, the reconstructed bitplane bitmaps are combined to reconstruct the raster cell values through bitplane level composition. To set the i^{th} bit of raster cell value v decoded from the i^{th} BQ-Tree of a raster chunk, the following bitwise operation can be applied: $v|=(b<<i)$ where b is the bit value of the i^{th} bitplane of the raster cell.

D. Discussions on Key Implementation Issues

It is clear that the algorithms for both compression and decompression are conceptually simple as only bit-level operations are involved, except Z-order related calculation which may require significant amount of arithmetic operations if it is implemented in a naive way. Fortunately, by using the efficient design proposed in [9], the conversion between (row, column) pairs and Z-order codes is mostly reduced to table lookups which are highly efficient, provided that the lookup tables can be accommodated in on-chip CPU caches. Since most reasonably up-to-date CPUs have at least 32KB L1 cache, despite that the cost of Z-order related computation is still significant, it does not dominate based on our experiments. By using row-major order instead of Z-order for all the raster cells in a last level quadrant, we have reduced the number of Z-order computation to 1/16 by using a 4*4 LLQS size. While it is interesting to use larger LLQS sizes to further reduce Z-order related computation overhead, we note that row-major order performs poorly when the raster cells in a last level quadrant cannot fit in a cache line, which is a well known issue in accessing multi-dimensional arrays [1]. Using a 4*4 last level quadrant size, 32 bytes are needed to store all the 16 raster cells (2 bytes each) in a last level quadrant. This is smaller than L1 cache line length in most CPUs. Although some CPUs may have 128-byte L1 cache line length and thus support the 8*8 last level quadrant size naturally, using a larger level quadrant size may potentially decrease compression ratio in non-homogenous regions in a raster, although it may reduce runtimes of both compression and decompression.

Different from compression that only a single Z-order computation is required for a last level quadrant, if we combine the K decompressed signatures of a last level quadrant directly to the output array, we will need K individual Z-order computations, each for a bitplane. An optimization we have adopted is to use a temporal array to hold all the bitplane-combined last-level quadrants in the order of quadtree nodes, i.e., Z-order. Note that raster cells within a last-level quadrant are still in row-major order. We compute Z-order values only when the temporal array is copied back to the output array where Z-order is transformed to the desired row-major order. The optimization essentially reduces the number of Z-order computation at the bitplane level to 1/K at the raster cell level. Our experiments have shown that this optimization is very effective in improving the overall system performance. One disadvantage of the optimization is that, in a way similar to merge sort, a temporal array as large as the output array is required and copying the temporal array to the output array may require additional memory bandwidth. This might explains that, despite our decompression technique performs better than zlib using a single CPU core, it becomes slower than zlib when all 16 cores are used, possibly due to memory bandwidth contention. However, we believe that further optimizations, for example, better reordering of accesses to raster cells and data pre-fetching, can potentially improve the decompression performance significantly for both serial and multi-core implementations of our BQ-Tree based decompression design. Similar improvements can also be applied to compression, although the compression performance of our technique is already significantly faster than zlib for both serial implementation (4.1X) and 16-core parallel implementation (6.5X) using a chunk size of 1024*1024, as reported in Section IV.B. In general, as our technique is likely to be memory bandwidth bound, more efficient data accesses can further significantly improve the performance for both compression and decompression.

E. Parallelization on Multi-Core CPUs

For a large geospatial raster, a natural way to parallelize compression and decompression is to split it into multiple chunks and let each processing unit (i.e., a CPU core in this case) to process a chunk independently. Although we are aware of more sophisticated parallelization techniques might perform better with respect to load balancing (e.g., work stealing implemented in Intel Thread Building Blocks -TBB[7]), we have decided to use the simple "parallel for" parallelization directive (or pragma) for a simple implementation as we aim at developing a lightweight technique. Since most computing systems support OpenMP compliers and runtime systems, the extra overhead of parallelization is minimized. As discussed in the experiment section, we will also use OpenMP to parallelize zlib for compressing and decompressing large-scale rasters after they are chunked for fair comparisons.

During compression and decompression, each raster chunk will have its own input and output arrays as well as local variables. As such, the parallelization process is designed to be embarrassingly parallelizable. However, after all chunks are compressed, the output sizes need to be combined so that the compressed data can be written to the proper memory location and complete the final concatenation process. While the combination process can be easily implemented in parallel through a prefix-sum process [10], since the number of chunks is typically limited and moving memory-resident data in large chunks is fairly fast on modern CPUs, we simply perform the final concatenation process sequentially with negligible overhead. We note that, it might be more beneficial to use a linked list to virtually concatenate these compressed chunks which can be effortlessly done on CPUs, instead of actually moving the chunks in memory. When the compressed chunks need to be written to disks, they can simply be output sequentially without physical concatenation. The compressed chunks can naturally be used to support parallel I/Os in cluster computing environment by distributing the chunks into multiple disk strips. This is left for our future work.

IV. Experiments and Results

A. Data and Experiment Environment Setup

Although our technique can be applied to virtually any type of geospatial rasters (integer/float and signed/unsigned), in this study, we will use NASA SRTM 30 meter resolution elevation data in the CONUS (CONtinental United States) region. The SRTM data was obtained on a near-global scale in February 2000 from a space-borne radar system and has been widely used in many applications since then. The CONUS raster dataset has a raw volume of 38 GB and is about 15GB when compressed in TIFF format in 6 raster data files. There are about 20 billion raster cells in these 6 files and we refer them collectively as the NASA SRTM raster hereafter. Like many geospatial rasters, cell values in this dataset are 16-bit short integers. To accommodate computer systems with lower memory capacities, we have further split the 6 raster files into 36 tiles to ensure that each tile requires limited memory capacity. Given that our experiment system has 16 CPU cores, such data partition also guarantees that the workload of each core can be roughly balanced at the raster chunk level using different chunk sizes. Clearly, the smaller the chunk size and the larger number of chunks, the better chance for load balancing. However, this is at the cost of higher metadata and scheduling overheads.

The E5-2650V2 CPU running at 2.6 GHZ used in our experiments, which are released in the third quarter of 2013, are reasonably up-to-date. The memory bandwidth shared by the eight cores in a CPU is 59.7 GB/s. As we focus on end-to-end performance with respect to runtimes, we assume all data are memory resident and we do not include disk I/O times. All programs are compiled with –O3 optimization level using gcc 4.9 and linked with zlib 1.2.3 when the "uncompress" or "compress" API in the library is used. We have also experimented two raster chunk sizes, i.e., 1024*1024 and 4096*4096, and we have obtained similar results. While we will report the overall results using the summations of the 36 raster tiles for both chunk sizes, due to space limit, we will only report the runtimes of individual tiles for the chunk size 1024*1024.

B. Overall results

The experiment results for our BQ-Tree (bqtree) and Zlib/LZ77 (zlib) based techniques are listed in Table 1, where RT stands for runtime (in milliseconds). From Table 1 we can see that, our BQ-Tree technique is significantly faster than zlib for compression. The speedup is about 4.1x for the serial implementation (1264697/311746) and 6.5x for 16-core implementation (157341/24120), respectively, for chunk size 1024*1024. The speedups change moderately for chunk size 4096*4096, which are 3.5x and 4.5x for the serial implementation and the 16-core implementation, respectively. On the other hand, for decompression, while our BQ-Tree technique is considerably faster (124442/91794 - 1=36%) for the serial implementation for chunk size 1024*1024 and slightly faster (124445/117289-1=6%) for chunk size 4096*4096, the 16-core implementation is 73% slower (22318/12859 - 1) and 124% slower (28857/12882 - 1) for the chunk size 1024*1024 and 4096*4096, respectively. Clearly the asymmetric design of the underlying LZ77 algorithm makes zlib 10X+ faster for decompression than for compression. Although our BQ-Tree technique is also 3X faster for decompression than for compression in the serial implementation for both chunk sizes, the asymmetry is mostly due to implementation and hardware characteristics, not algorithmic design.

Table 1 Compression/Decompression Runtimes (in milliseconds) of BQT and Zlib Techniques Using Two Chunk Sizes

	1024*1024		4096*4096	
	RT-serial	RT-16-core	RT-serial	RT-16-core
bqtree-comp.	311,746	24,120	365,220	35,046
bqtree-decomp.	91,794	22,318	117,289	28,857
zlib-comp.	1,264,697	157,341	1,264,959	156,495
zlib-decomp	124,442	12,859	124,445	12,882

Comparing the runtimes of the serial implementations and the 16-core implementations, while the speedup for bqtree-compression, zlib-compression and zlib-decompression are 13.0X, 8.0X and 9.7X, respectively, the speedup for bqtree-decompression is only 4.1X, using chunk size 1024*1024. Similar results can be obtained for chunk size 4096*4096. It seems that bqtree has higher scalability than zlib for compression while has lower scalability for decompression.

The relatively lower performance and lower speedup of the 16-core decompression implementation of our BQ-Tree technique motivate us to consider the interaction between our algorithms and the multi-core CPU architecture of the experiment machine. Given that our algorithms are memory-bound, we attribute the lower performance to memory contentions when all 16 cores are fully utilized for parallel decompression. Despite the memory bandwidth on the Intel Xeon E5-2650V2 CPU (59.7 GB/s) in our experiment system is significantly higher than previous generation CPUs, it is still much lower than what is needed. Since 8 cores in a CPU share the memory bandwidth, the share of each core is very limited. We expect that high CPU memory bandwidths are likely to improve the performance of our BQ-Tree technique for decompression.

When comparing the BQ-Tree based compression and decompression, the compression has much better scalability (10-13X using 16 cores) than decompression (~4X). While we are still in the process of fully understanding the realized implementation asymmetry, we suspect that the result is due to a non-optimized memory access pattern in the decompression implementation. While a raster cell is only accessed once from memory and its value can be kept in a register during compression, the raster cell may need to be accessed K times during decompression to combine bits at the K bitplanes. Since it is non-trivial to synchronize decompression across K bitplanes and keep the resulting raster cell in a register, we leave such potential optimization for future work.

We next turn to analyzing the overall performance by including compression and decompression and their use frequencies into consideration. Given the measured runtimes of compression and decompression for a particular technique (BQ-Tree or zlib) using a specific implementation (serial or 16-core) and a chunk size, assuming the compression and decompression are equally frequent, the total runtime can be easily computed and compared. Despite that our BQ-Tree technique is slower than zlib for decompression for the 16-core implementation, when the frequencies of compressions and decompressions are the same, overall, our BQ-Tree technique is 3.66X and 3.44X better than zlib, for the serial and 16-core implementations using a chunk size of 1024*1024, respectively. The speedups are 2.88X and 2.65X using a raster size of 4096*4096.

C. *Results of individual raster tiles*

To further understand the distributions of compressed sizes of our BQ-Tree technique, the three components of the resulting compressed bytes are plotted in Fig. 5. The metadata sizes are proportional to chunk sizes as we need to record lengths of the resulting quadtree arrays and LLQS arrays so that processing units can access the respective chunks in parallel during decompression. From Fig. 5, it can be seen that the metadata sizes are negligible and the LLQS sizes dominate. Fig. 6 shows the compressed sizes (in bytes) of our BQ-Tree technique and the zlib, which are comparable.

The runtimes of the serial and the 16-core implementations of both compressions and decompressions of BQ-Tree technique on the 36 raster tiles are plotted in Fig. 7 through Fig. 10. It can be seen that our BQ-Tree technique compresses much faster than zlib for both the serial and the 16-core implementations. While zlib vary significantly and performs much faster for raster tiles that have large portions of no-data values (especially for those that are in the border of the CONUS region where ocean raster cells are marked as no-data). In contrast, our technique is largely data-independent, which can be both advantageous and disadvantageous.

The data-independent feature of the BQ-Tree technique can be advantageous for datasets whose raster cells vary significantly, although it may need to do more work than zlib for datasets whose cells do not change much (e.g., large portions of raster cells have no-data values, as in several among the 36 raster tiles). We note that, despite the performance of our BQ-Tree technique and zlib are very close for raster tiles with large portions of no-data values in both the serial implementation and the 16-core implementation, our technique is much faster for other tiles (Fig. 7 and Fig. 8).

Although BQ-Tree is overall 36% faster than zlib in the serial implementation for decompression, as shown in Fig. 9, there are many cases that zlib is actually faster in decompression. Furthermore, the BQ-Tree technique performs worse in all cases for the 16-core implementation which explains the overall 1.7X slow down. As discussed in Section IV.B, memory bandwidth contention in the 16-core implementation might be an important reason. The different comparison results between the serial implementations and the 16-core implementations may suggest the importance of the coupling between algorithmic design and hardware architecture, rather than algorithm or CPU alone, in determining realizable performance.

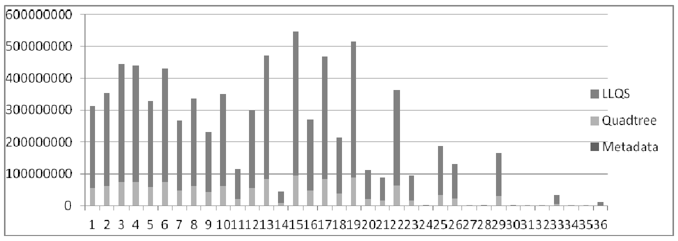


Fig. 5 Compressed Sizes for the 36 Raster Tiles of the BQ-Tree Approach

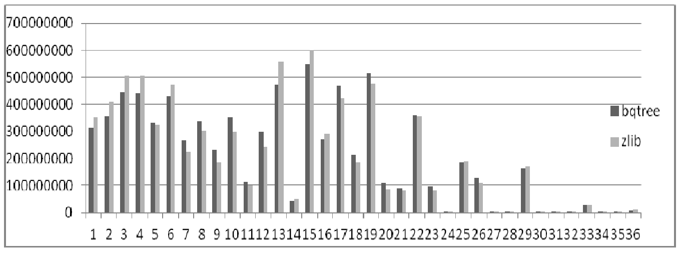


Fig. 6 Comparisons of compressed sizes of 36 raster tiles using BQ-Tree and zlib

While we leave further optimizations for future work, we would like to stress that, since we target at the applications that require balanced compression and decompression, our BQ-Tree technique performs better than zlib when overall performance is measured, in addition to being lightweight and portable by design.

V. Conclusions and Future Work

In this study, we aim at developing a lightweight lossless data compression technique for large-scale geospatial rasters to support efficiently streaming data from disks to CPU memories as well as among distributed computing nodes and file systems. Despite that the idea on using quadtrees to encode images rasters is quite simple and has been exploited before, to the best of our knowledge, we are the first to demonstrate that the quadtree based technique can perform several times better than the popular generic zlib technique not only for the serial implementation but also the overall performance for the multi-core implementation. Given that our implementations are still under intensive optimizations, we are optimistic in achieving comparable or even better performance than zlib for decompression implementation after reducing memory bandwidth contentions in the multi-core CPU implementation. The simple design and implementations of our technique make it easier to port to new hardware, including VPUs on CPUs, GPUs and Intel Xeon Phi devices, when compared with zlib.

For future work, we would like to perform more experiments on more large-scale geospatial rasters to further understand its strengths and weaknesses for performance improvements. Another research direction is to explore different ways in assembling and disassembling rasters from and to bitmaps for BQ-Tree coding, in addition to the bitplane bitmaps that are used in this study. Finally, while we have opted for simplicity in this study, it is interesting to explore the tradeoffs among computation workloads, compressed sizes and memory bandwidth utilizations.

ACKNOWLEDGEMENT
This work is supported through NSF Grants IIS-1302423 and IIS-1302439.

VI. References

1. J. Hennessy, D. A. Patterson (2011). Computer Architecture: A Quantitative Approach (5th ed.), Morgan Kaufmann.
2. Salomon, D. (2006). Data Compression: The Complete Reference (4^{th} edition). Springer.
3. Zhang, J., You, S. and Gruenwald, L. (2011). Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs. In Proc. ACMGIS'11, pp. 457- 460
4. Schmit, T.J., Li, J., et al. (2009). High-spectral- and high-temporal resolution infrared measurements from geostationary orbit. Journal of Atmospheric and Oceanic Technology, 26(11), pp. 2273-2292.
5. Peter, J., Straka, M., et al. (2013).Petascale WRF Simulation of Hurricane Sandy Deployment of NCSA's Cray XE6 Blue Waters. In Proc. ACM SC'13, #63.
6. Ozsoy, A., Swany, M., Chauhan, A. (2014). Optimizing LZSS compression on GPGPUs. Future Generation Computer Systems, vol. 30, pp. 170-178.
7. Schendel, E. R., Jin, Y., et al. (2012a): ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression. In Proc. IEEE ICDE'12, pp.138-149.
8. Morton, G.M., 1966. A computer oriented geodetic data base and a new technique in file sequencing. IBM Technical report.
9. Raman, R. and Wise, D.S., 2008. Converting to and from Dilated Integers. IEEE TOC, 57(4), pp. 567-573.
10. McCool, M., Robison, A. and Reinders, J. (2012). Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann.

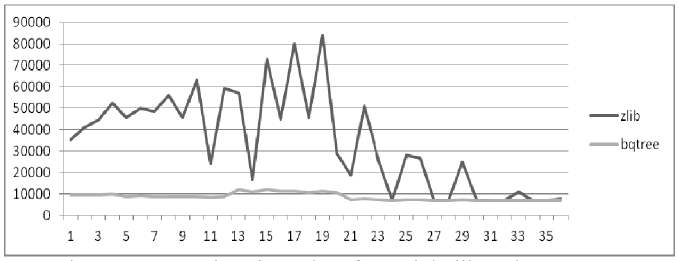


Fig. 7 Compression time plots for serial zlib and BQ-Tree implementations

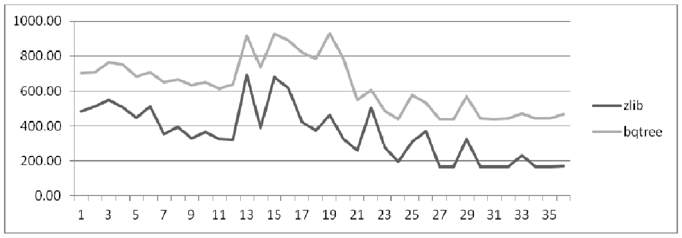


Fig. 8 Compression time plots for the 16-core zlib and BQ-Tree implementations

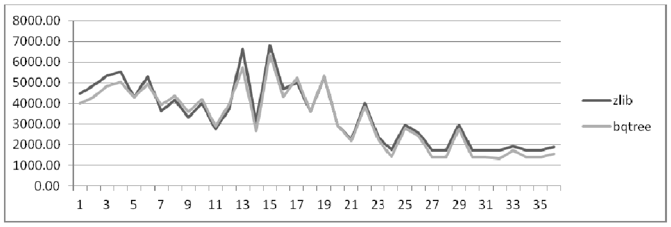


Fig. 9 Decompression time plots for the serial implementation

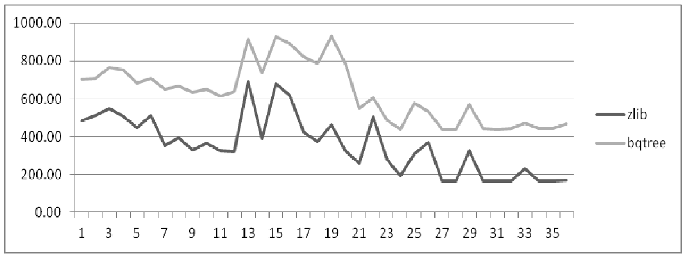


Fig. 10 Decompression time plots for the 16-core implementation

[1] http://www.zlib.net/
[2] http://www-cs.ccny.cuny.edu/~jzhang/bqtree_coding.htm
[3] http://www.goes-r.gov/
[4] https://lta.cr.usgs.gov/
[5] http://www.wrf-model.org
[6] http://en.wikipedia.org/wiki/Hurricane_Sandy
[7] https://www.threadingbuildingblocks.org/