OpenMP and POSIX threads Implementation of Jerasure 2.0

Suayb S. Arslan[1], Hoa Le[2], Joseph Landman[3] and Turguy Goker[2]

***Abstract*—In shared memory multiprocessor architectures, threads can be used to implement parallelism. POSIX threads (pthreads) is a low-level bare-bones programming interface for working with threads. Therefore, we have extremely fine-grained control over thread management (create/join/etc), mutexes, and so on. On the other hand, openMP, as a shared-memory standard, is much higher level and portable interface which makes it easier to use multi-threading capability and obtain satisfactory performance improvements. Since pthreads is more flexible, it helps programmers gain more control on performance optimizations. Jerasure 2.0 erasure coding library has encoding/decoding engines which comprise independent "for" loop iterations and hence possess huge potential for multi-threaded processing. In this short paper, we investigate multi-threaded implementations of encoder/decoder pair of Jerasure 2.0 using two different technologies: OpenMP and pthreads. We constrain our changes to a minimum possible and compare the pure encoding/decoding performance with respect to each other as well as against that of the original single-threaded version by running them on two different server systems.**

***Index Terms*—OpenMP, POSIX, pthreads, multi-threading, erasure coding, reliability.**

I. Introduction

Reed-Solomon (RS) coding [1] is based on Galois field arithmetic and used to typically protect storage systems from failures. However during basic encoding and decoding operations, repeated Galois field multiplication operations over the large data byte regions are deemed to be complex and prevented RS codes from being widely used in industry. There have been numerous studies to accelerate the execution speed performance of RS encoding and decoding operations [2]. Some of these methods focused on the simplification of the algorithmic processes [3] i.e., mathematics behind erasure coding while keeping loss correction capability identical, the rest of the studies are based on the efficient use of the parallel hardware by leveraging the modern Central Processing Unit (CPU) and Graphical Processing Unit (GPU) architectures [4], [5]. Jerasure 2.0 [6] is one of the well known open-source erasure coding library that is developed to utilize the instruction-level parallelism. However, such modern hardware also provides thread-level parallelism that encoding/decoding operations of Jerasure 2.0 can benefit [7].

*This work is supported by Quantum Corporation, Irvine, CA.

[1]S. S. Arslan is with the Faculty of Computer Engineering, MEF University, Maslak, Istanbul, Turkey. arslans at mef.edu.tr

[2]H. Le and T. Goker are with the Advanced Development Lab., Quantum Corporation, Irvine, CA 92612, USA Hoa.Le at Quantum.com and Turguy.Goker at Quantum.com

[3]J. Landman is with Joyent Inc., San Francisco, CA, 94111 USA. joe.landman at joyent.com

In this study, we focus on two known standards that modern compilers typically support to provide multi-threading: POSIX threads also known as *pthreads* and *openMP* libraries. One of the differences between these two standards is that openMP is task-based and pthreads is thread-based. In other words, in case hardware has available resources, openMP master thread can fire up dynamically more or less slave threads behind the stage and provide good amount of parallelism for the execution of the program. Although openMP may not lock the software into a preset number of threads, and have better potential of improved performance, it may happen that multiple files are encoded/decoded by the same server or more threads might be requested than needed and hence the CPU may run out of resources quickly. Thus, it is easy to write multi-threaded Jerasure encoder/decoder in openMP but it is not clear whether it shall perform faster than the pthreads multi-threaded implementations. In essence, when the multi-threaded algorithm is complex and there exists various serial regions inside the program, it becomes harder for openMP to optimize (although there are nice tools provided by the modern openMP standard). On the other hand, pthreads is a lower level approach to multi-threading and can provide the large enough design space to get the performance we want at the expense of more effort for optimization.

The organization is as follows. We shall introduce some of the mathematical basics of erasure coding, namely encoding and decoding engines of Jerasure 2.0 in the next section. We introduce the sequential programs of Jerasure 2.0 and discuss the parameters of the system. We demonstrate the potential for multi-threaded applications that enable significant speedups over the original programs. Section III summarizes the changes we made to the existing Jerasure 2.0 software. Section IV provides numerical results to support our arguments, quantify speedups/efficiency and compare performance in terms of throughput between different multi-threaded implementations as well as the original sequential programs. Finally, we provide a summary of our conclusions in Section V.

II. Encoder/Decoder Implementations

A. RS Erasure Coding Basics

For a given $[n, k]$ block erasure code, k data blocks are encoded to generate $m = n - k$ redundant or parity blocks. For the given data blocks $\mathbf{d} = (d_0, d_1, \ldots, d_{k-1})$ and parity blocks $\mathbf{c} = (c_0, c_1, \ldots, c_{m-1})$, we can give the mathematical relationship as follows,

$$c_i = \sum_{j=0}^{k-1} g_{i,j} d_j, \text{ for all } 0 \leq i < m \tag{1}$$

which can be expressed in a matrix form as

$$[\mathbf{d}_{1\times k} \mid \mathbf{c}_{1\times m}]^T = \mathbf{G}_{n\times k} \times \mathbf{d}_{k\times 1}^T \quad (2)$$

where $\mathbf{G} = \{g_{i,j}\}$ is known as the generator matrix and defines the set of equations that needs to be satisfied. If we carefully inspect equation (2), the generator matrix $\mathbf{G} = [\mathbf{I}_{k\times k} \mid \mathbf{P}_{k\times m}]^T$ includes the identity matrix $\mathbf{I}_{k\times k}$ and hence, data appears at the output of the encoder without any change. These types of codes are known as *systematic* codes. Also from equation (2), we have the parities calculated as

$$\mathbf{c}_{m\times 1}^T = \mathbf{P}_{m\times k} \times \mathbf{d}_{k\times 1}^T. \quad (3)$$

The success of decoding operation is heavily dependent on the right selection of $\mathbf{P}_{m\times k}$ matrix, in which the matrix entries are w-bit words chosen from the corresponding Galois field $GF(2^w)$, i.e., $g_{i,j} \in GF(2^w)$. These field elements can be chosen in different ways and yet in order to reconstruct data from any k encoded data (m erased blocks maximum), these entries need be selected according to a set of constraints. Two well known choices are *Vandermonde* matrix with entries $(g_{i,j} = (\alpha^i)^{j-1})$ [1] and *Cauchy* matrix with entries $(g_{i,j} = 1/(x_i + y_i)$ for different $x_i, y_i \in GF(2^w)$. Both are supported by Jerasure 2.0). Selection of such special matrices leads to two different constructions of RS codes. Once the right $\mathbf{P}_{m\times k}$ is selected, data array is multiplied by each row of the matrix to create the parities according to equation (3). Due to independent parity block calculations, this implies that parity blocks can be computed concurrently using multi-threaded implementation.

While decoding, assume that we collect k' available data and m' available parity blocks (with $k = k' + m'$). Supposing available blocks are indexed by $i_1, i_2, \ldots i_{k'}, i_{k'+1}, \ldots, i_{k'+m'}$ satisfying $i_1 \leq i_2 \leq \cdots \leq i_{k'+m'}$ and available parity blocks are indexed by $i_{k'+1} - k \ldots i_{k'+m'} - k$, the decoder algorithm uses the rows of $\mathbf{G}$ indexed by the available blocks to generate the submatrix $\mathbf{G}'_{k\times k}$. After that, the decoder takes the inverse of the submatrix and multiplies it with the available data blocks $\mathbf{d}' = (d_{i_1}, \ldots, d_{i_{k'}})$ and parity blocks $\mathbf{c}' = (c_{i_{k'+1}-k}, \ldots, c_{i_{k'+m'}-k})$ and regenerates the lost data blocks as summarized below.

$$\mathbf{d}_{k\times 1}^T = {\mathbf{G}'_{k\times k}}^{-1} \times [\mathbf{d}' \mid \mathbf{c}']_{k\times 1}^T. \quad (4)$$

Finally, lost parity blocks are found using usual encoding process. In the decoding process, although the matrix inverse ${\mathbf{G}'_{k\times k}}^{-1}$ is computed only once, the equation (4) is computed iteratively for different sections of the file. Since these operations can be done concurrently, it is of interest to investigate multi-threaded decoder implementations.

B. Jerasure 2.0 Encoder/Decoder as Sequential Programs

In RS encoding and decoding, computations are defined based on w-bit words. In real systems, encoding of consecutive data byte regions (for instance disk sector structures) are done with multiple fine grained Galois field additions and multiplications over w-bit words which can make the total workload a computationally intensive task. Thus, there are two major operations in both encoding and decoding processes, namely region XOR and multiply a region by an element of a Galois field, that need to be efficiently performed.

[1]The element α is known as the primitive element of the Galois field.

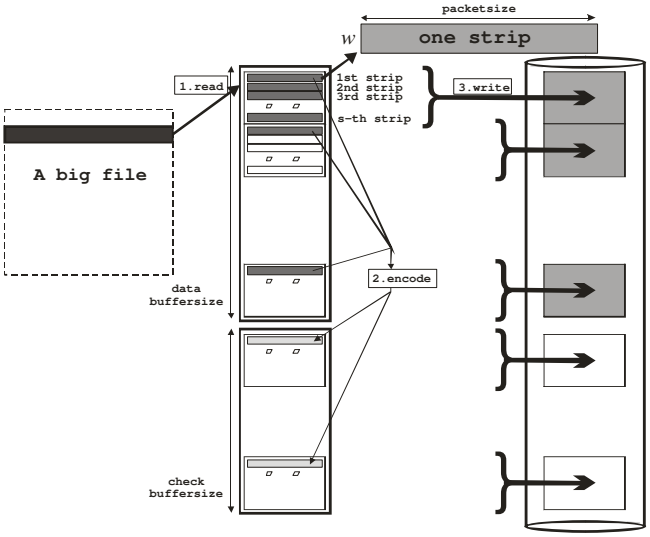

Fig. 1. Jerasure 2.0 library encoding process consists of three major phases as shown above.

Intel's Single Instruction Multiple Data (SIMD) support enabled processing multiple data sections using a single instruction, and it is shown that using such instructions, the execution of Galois field operations over large data regions can significantly be improved. Libraries such as GF-Complete [4] and ISA-L [8] are two well known implementations of Galois field operations using streaming SIMD instructions. Jerasure 2.0 comes with a set of routines that use GF-Complete to improve the performance of RS encoding/decoding matrix operations.

Jerasure 2.0 encoding has three phases as summarized in Fig. 1. In the first phase, the file is broken into equal size partitions and the partitions are copied to data *buffer* with *buffersize* capacity on main memory. Depending on the buffersize, multiple rounds of encoding are performed. The buffer is partitioned into k data blocks and with the start of the second phase, the data circulates between CPU caches in the form of *strip*s and is processed for encoding output. In order to determine the strip size, a parameter called *packetsize* is defined. Thus the strip size is given by $w \times$ *packetsize*. Note that the buffersize and packetsize parameters are used to optimize the memory usage and thereby the speed of execution in Jerasure 2.0. After encoding operation is over, the generated parities are written to the parity buffer on memory by traversing the same cache hierarchy. In the third i.e., the last phase, the data in the buffer is sent for permanent storage (*non-volatile*) by invoking necessary kernel I/O calls.

C. Multi-threaded applications

Although all three phases of the encoding/decoding operations can be multi-threaded, we particularly focus on the second phase that involves no I/O operations and report the pure encode/decode performance throughput as well as speedups

```
/* Encoder main for loop */
#pragma omp parallel for shared(data, coding, blocksize, matrix, k) private(init, j,  sptr, dptr)
    num_threads(m) schedule(dynamic, 1)
/* Decoder main for loop */
#pragma omp parallel for shared(erased, data, coding, blocksize, decoding_matrix, k,  dm_ids) private(init,
     j, matrix_row, sptr, dptr) num_threads(k) schedule(dynamic, 1)
```

relative to original Jerasure 2.0 encoder/decoder programs. The first phase can be multi-threaded as well, but this is observed to change the buffer/memory architecture i.e., more memory is needed to store and process data simultaneously which shall reduce performance due to increased data migration. In our approach, each parity block is computed by a distinct thread and the data buffer is shared among the threads and is used for read-only operations. This way we avoid mutexes for the pthreads implementation and avoid unnecessary performance overheads. In the openMP implementation, we may have to copy some data for local caches to eliminate the effect of false-sharing.

In the encoding process, as can be seen from equations (1) and (3), each parity block c_i is processed by the i-th thread with $0 \leq i \leq m-1$. Multi-threaded encoding example is shown in Fig. (2) for $k = 6$ and $m = 4$. The decoding process has more serial parts compared to encoding. Thus, the performance of multi-threaded decoder will heavily depend on the number and the distribution of lost blocks between data and parity sections. For instance, if all the lost blocks are from the data section, each lost data block can be generated by independent and distinct threads using $\mathbf{G'}^{-1}$. This way, maximum number of threads would be used to complete the decoding process.

On the other hand, if $k'' = k - k'$ data blocks are lost, and m'' parity blocks are lost (a total of $k''+m'' \leq m$ blocks), the decoding operation shall comprise of two sequential parallel regions. The first parallel region consists of computation of k'' data blocks and partial computation of m'' parity blocks. Since there are independent computations, this workload can be handled by a total of $k'' + m''$ threads. In the second parallel region, m'' independent threads compute the other half of the parity blocks and finally XOR them with the first half (previously computed partial parity blocks). Using two separate consecutive parallel region processing will be able to compute parities correctly because underlying RS codes are linear codes.

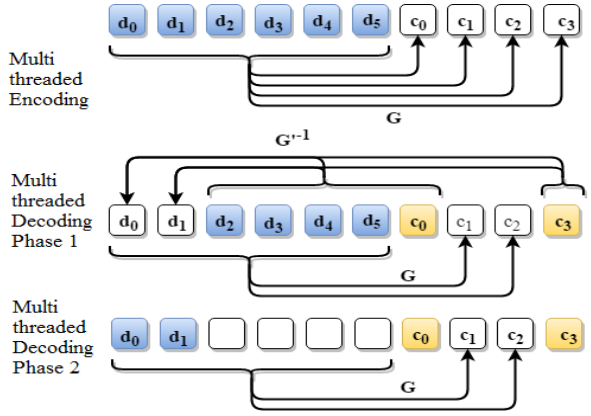

Fig. 2. Jerasure 2.0 encoder/decoder block operations. Empty blocks in the figure are considered to be NULL.

Fig. 2 presents an example where $k'' = 2$ data blocks and $m'' = 2$ parity blocks are lost and decoded using two parallel regions. In the same figure, light fonts show intermediate results whereas the bold font represents the final result.

III. Summary of Changes to The Software

First and foremost, we generated a sequential program before we made any attempt to generate multi-threaded encoder and decoder implementations. Essentially, we call GF-Complete library functions such as $galois\_region\_xor$ and $galois\_x08\_region\_multiply$ in a *for* loop repeatedly. This means to call lower GF-Complete functions such as gf_w32 that are optimized through SIMD instruction calls. These low level GF-Complete functions are normally called by $jerasure\_matrix\_dotprod$ in Jerasure 2.0. However, we bypassed it and called low level functions directly inside the encoder/decoder to help get explicit looping so that the iterations of these loops can be parallelized without making various changes to different Jerasure files. For pthreads, we needed to define a *structure* data type to hold the inputs of a function which shall be called by $pthread\_create$ and $pthread\_join$ functions from $pthread.h$. We also need to create, assign individual threads to independent vector multiplications and join those threads manually based on their IDs. This of course led to more lines of code relative to the openMP implementation in which we only used few pragmas to be able to utilize parallel/worksharing *for* loop constructs. An open source of these changes can be found on a new fork of master Jerasure github repository[2].

In order to help improve performance in the openMP implementation and avoid false-sharing, we use *private* and *shared* storage attributes and dynamic scheduling at the run time. Note that we do not know a priori the number of parities to be generated (for the encoder) or the indexes and the number of lost blocks to be recovered (for the decoder). In other words, different iterations will take different amounts of time, and hence dynamic scheduling with a default chunk size of 1 is preferred. We compiled all the source code using *gcc* compiler with both *-fopenmp* and *-pthread* linker flags. In addition to multi-thread support, we added helper functions to the original library and added support for covering file sizes as large as 4GiB. Unless otherwise stated, we use m threads in the encoding process and a total of $k'' + m''$ threads in the decoding process by setting the argument of $num\_threads(.)$

[2]Available online for download: https://github.com/suaybarslan/jerasure

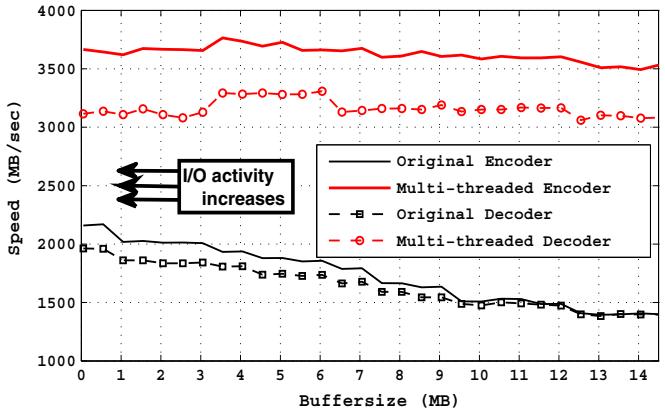

Fig. 3. Single and Multi-threaded encoder/decoder (using pthreads) implementations with pthreads ($k = 8$ ve $m = 4$) using Server I. As we move along the x axis from right to left, we increase the number of I/O. The size of the file in simulation = 128MiB.

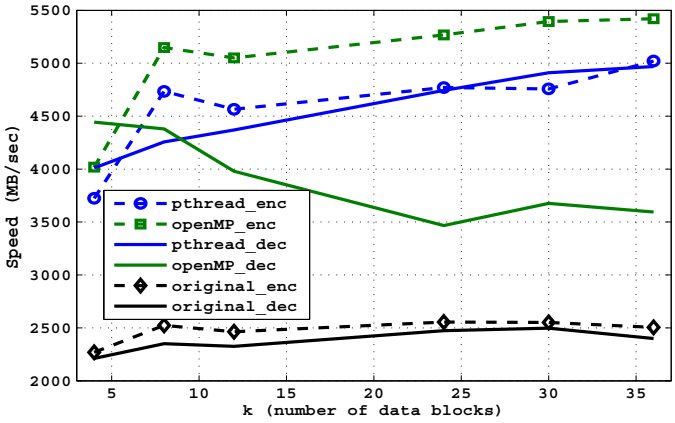

Fig. 4. Performance of encoder/decoder of Jerasure 2.0 original as well as multi-threaded implementations using openMP and pthreads using $m = 4$ and varying k.

run time function. We did minimum changes to the original Jerasure 2.0 variables and used for instance the above *pragmas* in our openMP encoder and decoder implementations. The variables in the statement above are identical to the original variable definitions of Jerasure 2.0. Note that the pragma given for the decoder only covers the data block losses, we also have a similar pragma to cover for the parity block losses.

In the original serial decoding process, the last data block recovery is typically based on region XOR operation ($galois\_region\_XOR$) if and only if the first parity block is intact/unerased or unmodified. This is due to the simplicity of XOR operation and the serial program can take advantage of it particularly when we think about the most likely error case being the single data block loss. When the application is multi-threaded however, this little trick can still be used but it will even further serialize the whole process. Thus, we entirely get rid of it and used the inverse of the generator matrix as well as the standard $galois\_region\_multiply$ function simultaneously (with other function calls) to balance the workload among threads each handling a specific data block recovery process.

IV. Numerical Results

We obtained our numerical results based on two Xeon machines as detailed in Table I and use $w = 8$ throughout this section. Due to space limitations, we focus on fully parallel encoder and decoder performance in this section.

TABLE I
TWO SERVERS USED IN OUR SIMULATIONS.

CPU feature	Server I	Server II
	Number/Value	Number/Value
Sockets	1	1
Cores	4	6
Threads	8	12
Arch.	IA64	IA64
Freq.	2.4GHz	2.4GHz
QPI Speed	5.86GT/s	7.5GT/s
L1 cache	32KiB	32KiB
L2 cache	256KiB	256KiB
L3 cache	12288KiB	15360KiB
Main memory	$\approx$ 49GiB	$\approx$ 8GiB

Let us first use Server I, a 128MB test file and pick pthreads implementation and observe how the performance changes based on the varying packetsize and buffersize parameters. For the decoder, we assumed randomly selected m block losses at the same time from the data section to fully parallelize the decoding operation. We have run encoder/decoder for different combinations of k and m, however we only show a subset of our results in Fig. 3 for $k = 8$ and $m = 4$ to convey the main message. In this figure, x axis shows the buffersize, whereas y axis shows the pure encode/decode speed in MB/sec while packetsize is optimized through exhaustive search. Same figure also includes the single threaded application performance (original Jerasure functions) for comparison. Clearly, we can observe upto $3\times$ speedup relative to single-threaded version. In addition, we observe the multi-threaded version to maintain almost the same performance across different buffersizes. This is a desirable property for big files to optimize the I/O (minimize the number of accesses made to the slower storage units) and other operations of the system. In other words, buffersize can be selected for I/O performance rather than the pure encode/decode performance since the pure throughput is least affected by buffersize variations. On the other hand, as we move along the x axis from right to left, we observe that single threaded version's performance increase. However, small buffersize means more data accesses on disk i.e., more I/O kernel calls which are time costly operations. One final interesting observation, results associated with the multi-threaded version demonstrates better results than that of [5], where GPUs are considered to improve the overall throughput performance.

In our next simulation, we used a 512MB test file along with Server II which is a bit more capable system and compared two different multi-threaded implementations of Jerasure 2.0. As observed from the previous simulation result, we fixed $packetsize = 2KB$ and $buffersize = 5MB$ bytes to be able to ensure satisfactory results for both implementation approaches. In Fig. 4, we compare the performances of encoder/decoder of Jerasure 2.0 original as well as multi-threaded implementations

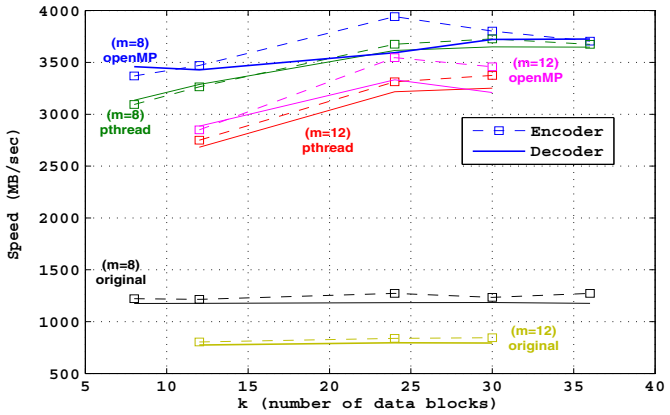

Fig. 5. Performance of encoder/decoder of Jerasure 2.0 original as well as multi-threaded implementations using openMP and pthreads using $m = 8$ and $m = 12$ and varying k.

using openMP and pthreads using $m = 4$ and varying k along the x axis. As can be observed, multi-threaded implementations are able to provide above $2\times$ speedup improvement. Another observation is that since there are more than $m = 4$ threads available in the system, openMP encoder provides better performance than that of pthreads. On the other hand, due to more complex operations of decoder, openMP decoder implementation does not perform as well as that of pthreads implementation. This may require more elaborate changes to the openMP implementation to improve its performance. Also, due to serial nature of decoder, openMP suffers more than pthreads does when it comes to frequent data movements between caches though openMP copies data to avoid false-sharing.

In Fig. 5, we also provided the same set of performance curves with $m = 8$ and $m = 12$ cases. One of the interesting observations with these results is that openMP decoder does not show the same performance degradation with $m = 8$ and $m = 12$. This is due to the fact that the number of threads available in the system is constrained by the requested threads m and thus, more controlled thread assignment is performed by openMP implementation. Overall, it seems that openMP implementation which is easier in many cases to implement compared to pthreads might serve well our computation needs, thanks to a decade-long standardization process and recent advances in open source community.

Finally, let us do cross-system comparison by looking at the speed up and efficiency metrics which are defined as

$$\text{Speedup} = \frac{\text{latency w/ singled-threaded App.}}{\text{latency w/ multi-threaded App.}} \quad (5)$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{\# of threads}} = \frac{\text{Speedup}}{m} \quad (6)$$

In Table II, we present the encoder/decoder speedups relative to original decoder for different values of k, m and file sizes (in terms of KiB) using both types of multi-threaded implementations. We pick parameter values so that openMP performs at least as good as that of pthreads implementation. As can be seen, as the file size grows, the gained speedup performance rises. On the other hand, if m grows, we observe

TABLE II
SPEEDUP PERFORMANCES OF ENCODER/DECODER PAIR WITH OPENMP AND PTHREADS IMPLEMENTATIONS

m	k	File size(MiB)	Type(enc/dec)	
			openMP	pthreads
4	8	64	1.53/1.66	1.63/1.6
		128	1.72/1.7	1.71/1.65
		256	1.9/1.78	1.71/1/75
		512	2.12/1.87	1.79/1.85
4	32	64	1.56/1.31	1.58/1.62
		128	1.69/1.63	1.66/1.91
		256	1.9/1.56	1.75/1.9
		512	2.14/1.49	1.9/2/01
8	32	64	2.17/2.75	2.03/2.54
		128	2.4/3.11	2.28/3.01
		256	2.77/2.98	2.55/3.00
		512	2.98/3.05	2.9/3.1
12	32	64	2.94/3.69	2.85/3.69
		128	3.27/4.10	3.15/4.24
		256	3.77/3.96	3.59/3.96
		512	4.01/4.06	3.96/4.06

better speedup performance but worse efficiency. The reason for the latter is that as we increase m i.e., the number of working slave threads, we are not getting the same increase in the speedup due to shared variables, data copy operations and unavailablity of CPU resources.

V. CONCLUSIONS

In this study, we demonstrated up to $3\times$ to $4\times$ improvement over the original programs using both multi-threading approaches. We have also shown similar throughput performance maintained across different buffersize selections. This is particularly useful for big data files and for reduced I/O and kernel performance optimizations. If there are available hardware resources, openMP is shown to be capable of creating easier multi-threaded applications and potentially exhibit better performance. Otherwise, fine granular programming skills are needed (for instance using pthreads) for better/satisfactory speedups.

REFERENCES

[1] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[2] J. Luo, K. D. Bowers, A. Oprea, and L. Xu, "Efficient software implementations of large finite fields $GF(2^n)$ for secure storage applications," *ACM Transactions on Storage*, 8(2), Feb. 2012.

[3] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong and S. Sankar, "Row Diagonal Parity for Double Disk Failure Correction," *FAST-2004: 3rd Usenix Conference on File and Storage Technologies*, San Francisco, CA, Mar., 2004.

[4] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast Galois Field arithmetic using Intel SIMD instructions", *In FAST-2013: 11th Usenix Conference on File and Storage Technologies*, San Jose, February 2013.

[5] X.Chu, C.Liu, K.Ouyang, L.S.Yung, H.Liu, and Y.-W.Leung, "PErasure:A parallel cauchy Reed-Solomon coding library for GPUs", *IEEE International Conference on Communications*, London, UK, 2015.

[6] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. W. OHearn, "A performance evaluation and examination of open-source erasure coding libraries for storage", *In 7th USENIX FAST*, pp. 253265, 2009.

[7] S. S. Arslan, "Implementation of multi-threaded erasure coding under multi-processing environments," 24th IEEE Signal Processing and Communication Application Conference (SIU), pp. 1773-1776, 2016.

[8] Intel's Intelligent Storage Acceleration Library (ISA-L). Address: https://01.org/intel®-storage-acceleration-library-open-source-version.