Improving Software RAID 6 Systems by the Means of Parallelization

Mihailo Vesović, *Graduate Student Member, IEEE,* Borislav Đorđević, *Member, IEEE*, and Aleksandra Smiljanić, *Member, IEEE*

***Abstract*—RAID storage technology improves the reliability of the computer storage systems. Hardware RAID 6 systems have been traditionally using Reed-Solomon codes. However, in software RAID 6 systems, different coding schemes can be employed. Parity array algorithms show satisfactory results, as they display significantly higher execution speed. In this paper, we have evaluated parity array EVENODD algorithm as a candidate for RAID 6. We have listed its main advantages compared to the similar algorithms. Additionally, we have shown the performance improvements that can be achieved through its parallelization on general multicore processors.**

***Index Terms*— Parallel Algorithms; RAID; Reliability;**

I. Introduction

The reliability of a system is an ability to run it for a long period of time without failures [1]. Reliability can be expressed through the MTTF (Mean Time To Failure) or MTBF (Mean Time Between Failures) factors. MTTF is the average time during which no equipment failures will occur. MTBF represents average time between unrecoverable errors on a drive of a given type [2].

In the recent years, MTTF and MTBF factors for a single drive have been improving. However, if we have a system with N drives, the resulting MTTF and MTBF factors would be N times worse. Today’s large-scale storage systems comprise thousands of drives, and if no additional redundancy is introduced, the resulting MTTF would be unacceptably low.

In order to improve the reliability, disks are organized in RAID systems (Redundant Array of Independent Disks). RAID is a data storage technology that combines multiple physical disk drives into single logical unit. Multiple RAID levels have been defined, with differences in reliability and performance. Standard RAID levels are numbered from 0 to 6 and the most common types are RAID 0, RAID 1, RAID 5 and RAID 6.

In a RAID 6 system, two coding nodes are used, usually labeled as the P and Q drives. This means that the system can recover from the state where two drives have failed simultaneously. The P drive is defined as a simple parity drive. The definition of the Q drive is not standard, which means that different coding schemes can be used. The requirement is that the resulting code must be maximum distance separable (MDS) [3-4].

Today, we can choose between hardware and software based RAID solutions. Hardware RAID solutions can provide higher speeds. On the other side, software solutions are cheaper and more flexible, especially in the case of RAID 6 where the coding algorithm is not predetermined [4].

The recovery time for one disk is dependent on multiple factors: number of drives, total capacity, sustained data rates, etc. In software RAID systems, the processor frequency also influences the recovery time. Performance of RAID 6 software system can be improved by the appropriate choice of an optimal algorithm. Its speed can be improved by executing the algorithm simultaneously on multiple processor cores, i.e. by parallelization.

In this paper, we aim to improve the performance of EVENODD algorithm [5]. EVENODD is a well-known algorithm belonging to the group of parity array algorithms. Parity array algorithms have advantage compared to the others because they are not based on multiplication, which is expensive operation. Furthermore, EVENODD is a good candidate for parallelization, as it inherently has little sequential dependencies.

II. Theoretical background

A. RAID 6

In a RAID 6 system, there are $m + 2$ drives in total: m data disks, labeled D_0, D_1, … , D_{m-1}, and 2 redundant disks, labeled D_m (or P drive) and D_{m+1} (or Q drive). Let us assume that each disk is divided into small data blocks of equal size. The size of each block is B symbols, where the symbol size can be arbitrary.

Group of blocks placed on the same position on different drives is called a stripe (see Fig. 1). Encoding and decoding are performed on the stripe level. This means that encoding of the k-th block on D_m or D_{m+1} disk will be based on only k-th blocks of the data disks D_i, $i = 0,...,m-1$.

B. Erasure Coding Algorithms

In RAID 6 system, Q disk is usually encoded by using Reed-Solomon coding scheme [6]. Reed-Solomon coding is based on the Galois Field GF(2^x) arithmetic. GF addition and subtraction are equivalent to exclusive OR (XOR, $\oplus$)

Mihailo Vesović is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: mikives@gmail.com).

Borislav Đorđević is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: bora@impcomputers.com).

Aleksandra Smiljanić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: aleksandra@etf.rs).

operation. Multiplication, on the other hand, requires the addition of exponents, which is expensive operation. Therefore, other coding techniques are being considered.

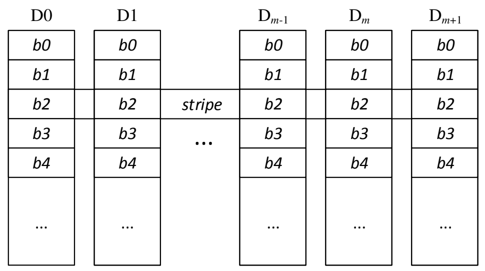

Fig. 1. Illustration of data (D_0-D_{m-1}) and parity (D_m, D_{m+1}) disks composed of the multiple equal-sized blocks. Stripe is a sequence of blocks at the same position on different disks.

Another important class of algorithms are parity array algorithms, in which all the coding can be expressed through the use of XOR operation. First parity array algorithm that was proposed was EVENODD algorithm [5]. In the following years, numbers of similar algorithms have been proposed, such as Liberation [4], RDP [7], etc. EVENODD algorithm performs better than Reed-Solomon due to the fact that XOR is much simpler operation than multiplication. Liberation algorithm is considered as a good alternative to EVENODD, due to the fact that it is not patented [4].

There were other numerous codes mentioned in the literature, like X-Code [8], STAR [9], etc. These algorithms are either not horizontal or MDS codes, and thus cannot apply to RAID 6 system. There are also algorithms which propose the use of 3 or more coding drives, like RTP [10]. Since RAID 6 uses only 2 coding drives, these algorithms are also out of scope for RAID 6.

1) EVENODD

EVENODD is a parity array algorithm designed for tolerating double disk failure in RAID 6 systems. In the case EVENODD algorithm, size of the block must be $B = m-1$, where m is the number of drives excluding coding drives. This restriction is not crucial, since m is small and the number of blocks is large.

Encoding procedure: The disk D_m is calculated as the simple parity of data disks, according to (1):

$$a_{l,m} = \bigoplus_{t=1}^{m-1} a_{l,t}, \quad 0 \le l \le m-2 \tag{1}$$

The content of disk D_{m+1} is calculated using (2):

$$a_{l,m+1} = \left(\bigoplus_{t=1}^{m-1} a_{m-1-t,t} \right) \oplus \left(\bigoplus_{t=0}^{m-1} a_{\langle l-t \rangle_m, t} \right), \quad 0 \le l \le m-2 \tag{2}$$

Decoding procedure: We differentiate 4 different recovery cases based on which two disks have failed.

In the case when D_m and D_{m+1} redundant disks have failed, i.e. $i = m, j = m+1$, the decoding procedure is effectively the same as the encoding procedure.

If the D_i and D_m disks have failed, i.e. $i < m, j = m$, the decoding procedure is performed according to the (3), (4) and (1):

$$S = a_{\langle i-1 \rangle_m, m+1} \oplus \left(\bigoplus_{l=0}^{m-1} a_{\langle i-l-1 \rangle_m, l} \right) \tag{3}$$

$$a_{k,i} = S \oplus a_{\langle k+i \rangle_m, m+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \ne i}}^{m-1} a_{\langle k+i-l \rangle_m, l} \right), \quad 0 \le k \le m-2 \tag{4}$$

In the case where D_i and D_{m+1} disks have failed, i.e. $i < m, j = m+1$, we use (1) to reconstruct the disk D_i, and (2) to reconstruct D_{m+1} disk.

The most complex case is when two data disks D_i and D_j have failed, $i < j < m$. In order to reconstruct the data, we must first find diagonal parity (5), and horizontal (6) and vertical syndromes (7):

$$S = \left(\bigoplus_{l=0}^{m-2} a_{l,m} \right) \oplus \left(\bigoplus_{l=0}^{m-2} a_{l,m+1} \right), \tag{5}$$

$$S_u^{(0)} = \bigoplus_{\substack{l=0 \\ l \ne i,j}}^{m} a_{u,l}, \quad 0 \le u \le m-1, \tag{6}$$

$$S_u^{(1)} = S \oplus a_{u,m+1} \oplus \left(\bigoplus_{\substack{l=0 \\ l \ne i,j}}^{m-1} a_{<u-l>m, l} \right), \quad 0 \le u \le m-1. \tag{7}$$

Afterwards, we do the following recursive procedure:

1. $s = \langle i-j-1 \rangle_m$; $a_{m-1,l} = 0, \quad 0 \le l \le m-1$,
2. $a_{s,i} = S_s^{(0)} \oplus a_{s,j}$, $a_{s,j} = S_{\langle j+s \rangle_m}^{(1)} \oplus a_{\langle s+j-i \rangle_m, i}$,
3. $s = \langle s+i-j \rangle_m$,
4. If $s = m-1$ stop. Otherwise go to the step 2.

2) Liberation

Liberation algorithm is considered for standard coding procedure for RAID 6. Size of the block is $B = w$, where $w+1$ must be prime number and greater or equal than number of data disks, m.

The encoding procedure starts with the creation of binary matrix of size $w(m+2) \times wm$, called Binary Distribution Matrix (BDM). BDM is multiplied with vector column containing wm symbols from all the data blocks in a given stripe. As a result, we get vector column of $w(m+2)$ symbols, which represents the vector of all the symbols in a stripe. Last $2w$ symbols are written on P and Q drives. Decoding procedure relies on the creation of new $wk \times wk$ BDM' matrix using the data from surviving devices. The algorithms for generating BDM matrices are omitted for the sake of clarity, but may be found in [4].

BDM matrix is sparse, which means that there are small number of non-zero elements. Thus, matrix multiplication will have many zero dot products. Therefore, authors propose technique called bit matrix scheduling. From the BDM matrix, schedule is created. Rather than traversing the whole matrix, the 5-tuples list <*op*, *sd*, *sb*, *dd*, *db*> is created [11]. Field *op* specifies the operation (assignment or XOR), and other fields identify the data on which *op* operation is performed (on which drives are the source and destination operands located, and the position of the operands on the drives).

C. Jerasure

Jerasure [11] is a C library that implements multiple erasure coding algorithms. It supports horizontal mode of erasure codes. Among implemented, there are Reed-Solomon, Cauchy Reed-Solomon and Liberation codes. Jerasure is dependent on GF-Complete library, which implements Galois Field arithmetic.

D. Parallelization

In order to speed up any program, capabilities of the processors must be fully utilized. As the frequency scaling of the processors has stalled in recent years, trends have shifted towards integrating multiple cores on chip [12]. Therefore, execution of algorithms on the multiple cores is a way to improve their performance.

OpenMP is an API (Application Programing Interface) that allows high level parallelization of the programs. OpenMP is a collection of compiler directives, library routines, and environment variables for parallelism in C, C++ and Fortran programs [13]. It is intuitive, simple, and offers various other benefits to programmers (easy to read code, minimized race conditions etc.). It is intended for the systems with shared memory.

In order to further improve the performance, coprocessors and graphic cards can be used. Coprocessor cards have up to hundred processing cores which execute offloaded code. E.g., XeonPhi 31S1P coprocessor card [14] consists of 57 cores. XeonPhi cores also have VPU (Vector Processing Unit) which is ideal for the arithmetic or logic operations performed on large arrays of data [15]. Graphic cards comprise hundreds to thousands cores, however, their capabilities are much more limited comparing to the coprocessor cores. E.g., NVIDIA Tesla K20 GPU accelerator [16] has 2496 simpler cores and can easily cost ~ 3000 USD.

III. Implementation

We have chosen EVENODD algorithm because it is well-known parity array algorithm, easy to implement and understand, and has potential for parallelization. Parity array algorithms in general rely on exclusive OR operation which is standard logical operation and which can be easily vectorized if performed on the array data. Additionally, it allows easier parallelization, as it is the operation on which reduction can be done in OpenMP.

We have implemented parallelized version of EVENODD algorithm in the C programing language. Parallelization was done using OpenMP API. Program was compiled with the GCC 5.4 version with optimizations level 3 (-O3) turned on.

The main program is divided into four parts. The first part of the program is the initialization of data structures. The second part is the encoding of the P and Q disks, based on the chosen pattern (random, ordinary numbers, etc). Encoding times are measured and saved in output file. In the third part, we assume that two data drives have failed, based on the user input. The decoding procedure is initialized, and the decoding times are measured. In the last part, results are compared to check whether the simulation was successful.

Let us assume that the total number of data drives in the system is *m*. Data and parity disks are represented as 2D array of integers, called *data*, with m rows and $m+2$ columns. Columns $i = 0,1,...,m-1$ represent data drives D_i, while columns *m* and *m* + 1 represent the P and Q drives. Results are stored in the 2D array *rdisks*[*m*][2]. This array has only two columns, since we can tolerate simultaneous failures of two disks. The first column is reserved for the disk with smaller ID.

There are 4 decoding functions, depending on which drives have failed:

- *decode_c0_c1_lost* (Fig. 2) is called when both parity drives have failed;
- *decode_di_c0_lost* (Fig. 3) is called when one data drive and the P parity drive have failed;
- *decode_di_c1_lost* (Fig. 4) is called when one data drive and the Q parity drive have failed;
- *decode_di_dj_lost* (Fig. 5) is used when two data drives have failed, which is the most common case.

Function arguments *f*, *f1* and *f2* specify what the indices of the lost data drives are. All the decoding functions are based on the equations from the section II.

```
DECODE_C0_C1_LOST (int $m$, int** $data$, int** $rdisks$)
  $dpar = 0$
  for t = 0 to $m - 1$:
      $dpar$ ^= $data[m - 1 - t][t]$
  #pragma omp parallel for
  for $l = 0$ to $m - 2$
      $rdisks[l][0] = 0$
      $rdisks[l][1] = dpar$
      for $t = 0$ to $m - 1$:
          $rdisks[l][0]$ ^= $data[l][t]$
          $rdisks[l][1]$ ^= $data[\mathrm{mod}(l - t, m)][t]$
  return
```

Fig. 2. Pseudo-code of the decoding function when the P drive and the Q drive have failed simultaneously

Generally, each decoding function can be split into three parts – calculation of diagonal parity (variable *dpar*), reconstruction of the first disk and reconstruction of the second disk. The diagonal parity calculation is performed by using the for loop which contains instruction with XOR operation. Since OpenMP has a reduction mechanism for the XOR operation, parallelization is possible. It is not advisable to perform parallelization in all cases, because of the large overhead. Therefore, we have parallelized only the *dpar* calculation in the function *decode_di_dj_lost*. Reconstruction procedures and the calculation of horizontal (*hsyn*) and vertical syndromes (*vsyn*) are implemented as nested for loop. The outer for loop of all the functions can be parallelized, as its iterations can be performed in any order, i.e. there are no sequential dependencies between them.

In the function *decode_di_dj_lost*, the last part represents the recursive procedure. In order to calculate the data *rdisks*[*s*][1] it is necessary to calculate the data *rdisks*[mod($s + f2 - f1$, m)][0] first. The variable s must be traversed in specific order, starting from the $s = \text{mod}(f1 - f2 - 1, m)$. Therefore, it is impossible to parallelize this section, which consequentially yields higher execution times.

```
DECODE_DI_C0_LOST (int m, int f, int** data, int** rdisks)
    dpar = data[mod(f − 1, m)][m + 1]
    for t = 0 to m − 1:
        dpar ^= data[mod(f − t − 1, m)][t]
    #pragma omp parallel for
    for k = 0 to m − 2
        rdisks[k][0] = dpar ^ data[mod(k + f, m)][m + 1]
        for l = 0 to m − 1:
            if (l == f) continue
            rdisks[k][0] ^= data[mod(k + f − 1, m)][l]
    #pragma omp parallel for
    for l = 0 to m − 2:
        rdisks[l][1] = 0
        for t = 0 to m − 1:
            if (t == f):
                rdisks[l][1] ^= rdisks[l][0]
            else:
                rdisks[l][1] ^= data[l][t]
    return
```

Fig. 3. Pseudo-code of the decoding function when the P drive and one data drive have failed simultaneously

```
DECODE_DI_C1_LOST (int m, int f, int** data, int** rdisks)
    dpar = 0
    for l = 0 to m − 2:
        rdisks[l][0] = data[l][m]
        for t = 0 to m − 1:
            if (t == f) continue
            rdisks[l][0] ^= data[l][t]
    for t = 0 to m − 1:
        if (t == f):
            dpar ^= rdisks[m − 1 − t][0]
        else:
            dpar ^= data[m − 1 − t][t]
    #pragma omp parallel for
    for l = 0 to m − 2:
        rdisks[l][1] = dpar
        for t = 0 to m − 1:
            if (t == f):
                rdisks[l][1] ^= rdisks[mod(l − t, m)][0]
            else:
                rdisks[l][1] ^= data[mod(l − t, m)][t]
    return
```

Fig. 4. Pseudo-code of the decoding function when the Q drive and one data drive have have failed simultaneously

Encoding and decoding times were measured using *omp_get_wtime ()* OpenMP function. Encoding and decoding procedures were ran multiple times, and the mean values were calculated.

Number of disks, blocks, threads, IDs of failed disks and number of measurements are all passed as arguments to the program. Results are written to the textual output files. Output files are interpreted by python language, and graphs were created. For the graphs creation, we have used *matplotlib* library.

```
DECODE_DI_DJ_LOST (int m, int f1, int f2, int** data, int** rdisks)
    dpar = 0
    #pragma omp parallel for reduction(^:dpar)
    for l = 0 to m − 2:
        dpar ^= data[l][m]
        dpar ^= data[l][m + 1]
    #pragma omp parallel for
    for u = 0 to m − 1:
        hsyn[u] = 0
        vsyn[u] = dpar ^ data[u][m + 1]
        for l = 0 to m − 1:
            if ((l == f1) || (l == f2)) continue
                hsyn[u] ^= data[u][l]
                vsyn[u] ^= data[mod(u − l, m)][l]
        hsyn[u] ^= data[u][m]
    s = mod(f1 − f2 − 1, m)
    while (s != (m − 1)):
        rdisks[s][1] = vsyn[mod(f2 + s, m)] ^ rdisks[mod(s + f2 − f1, m)][0]
        rdisks[s][0] = hsyn[s] ^ rdisks[s][1]
        s = mod(s + f1 − f2, m)
    return
```

Fig. 5. Pseudo-code of the decoding function when two data drives have failed simultaneously

We set the number of threads with *omp_set_num_threads* () function. Parallelization is done by using pragma *#pragma omp parallel for*. This pragma divides the iterations of the following FOR loop, and they will be executed on different threads. The way in which iterations are divided is specified through *schedule* () option. In our case, we have used *schedule (static, 16)* or *schedule (auto)* options. *Schedule (static, 16)* option divides all iterations of the following FOR loop into groups of 16, and gives to each thread different group of iterations to execute. By doing so, we can minimize the false sharing. If we calculate diagonal parity, it is better to use *schedule (auto)* option, in which the compiler will decide about the best possible scheduling technique. We have also tried dynamic scheduling options, and static schedule options with different numbers of iterations in group, but the results were poorer.

IV. Testing

Characteristics of testing machine are presented in Table I. We have measured execution times of encoding and decoding procedures, and then converted them into bit rate. For each test, measurements were done 10 times in total, and the mean value was calculated.

TABLE I
Testing Machine Characteristics

Processor	Intel Xeon E5-2620v3 @ 2.4 GHz
Number of cores	6 per processor, 2 threads per core
Cache size	L1 = 32K + 32K; L2 = 256K; L3 = 15M
RAM size	64GB DDR3 @ 1866 MHz
OS	RHEL 7.2

We have measured encoding and decoding times for sequential and parallelized EVENODD algorithm for different number of threads. Decoding times were measured for 4 cases, depending on which 2 drives have failed in the system – two data drives (D_i and D_j), two parity drives (P and Q), or one data and one parity drive (D_i and P or D_i and Q). Decoding procedure in the case where P and Q drives have failed is identical as the encoding procedure, and therefore we have included only one graph.

Parallelized versions were tested for different number of threads, up to the total number of cores. Higher number of threads would be suboptimal, as the threads cannot be executed simultaneously. Each thread was bound to the separate core. We have also tested the impact of hyper threading, but the results showed only minor performance improvement.

V. Results

In the Fig. 6 we can see the encoding performance of EVENODD algorithm for different number of threads. Sequential algorithm is better than all parallel versions if number of disks is smaller than 19. For larger number of disks, parallel versions are more than 1.8 times better. For the 6 threads, parallel version is up to 4.6 times faster.

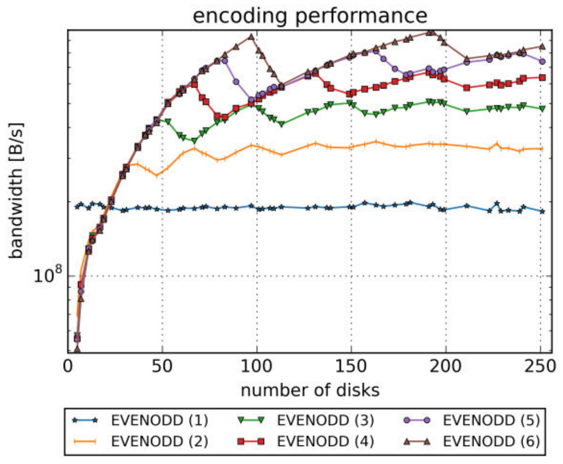

Fig. 6. Encoding performance of EVENODD algorithm for different number of threads

Fig. 7 shows the decoding performance in the case when two data drives have failed. This is the most common situation, as we have much more data disks than parity disks. Sequential version is faster for the number of disks smaller than 29. For the number of disks larger than 127, the best option is to use parallelized EVENODD with 6 threads, as the algorithm is around 5 times faster than sequential version. If the number of disks is between 29 or 127, the speed is optimized by different number of threads.

Fig. 8 shows the decoding performance of EVENODD algorithm when one data drive have failed along with the P parity drive. Sequential algorithm is better to use if the number of disks is smaller than 17. For larger number of disks, it is always better to use the parallel version with 6 threads. In the best case, parallel algorithm is around 5.6 times better than the sequential algorithm.

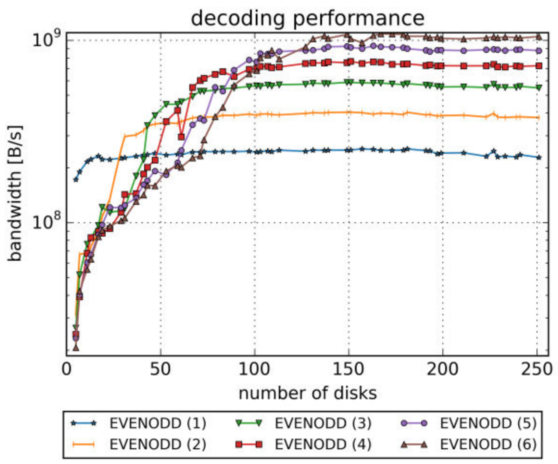

Fig. 7. Decoding performance of EVENODD algorithm for different number of threads in the case when two data drives have failed

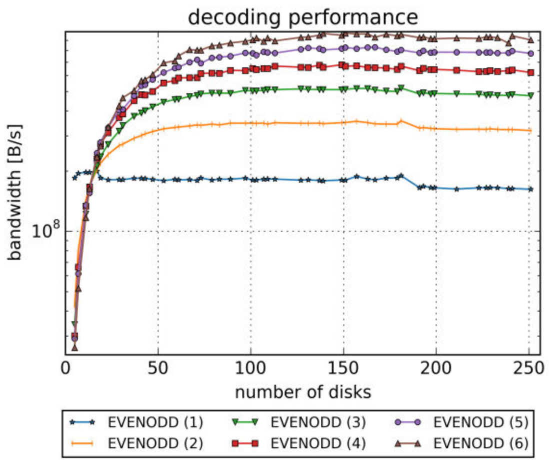

Fig. 8. Decoding performance of EVENODD algorithm for different number of threads in the case when the P drive and one data drive have failed

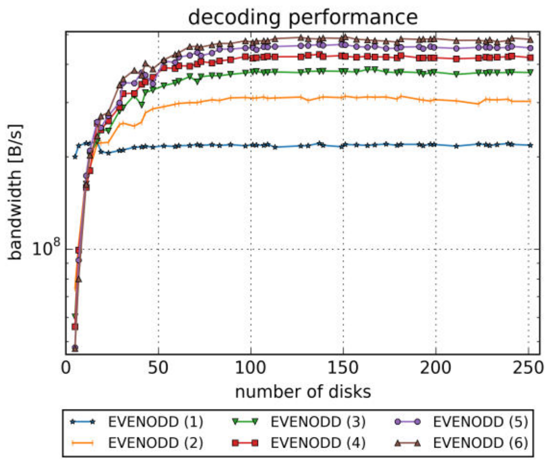

Fig. 9. Decoding performance of EVENODD algorithm for different number of threads in the case when the Q drive and one data drive have failed

Results for the case when one data drive and the Q drive have failed are presented in Fig. 9. Sequential version is the best option for the number of disks smaller than 19. For larger number of disks, it is better to use parallelized algorithm with 6 threads. However, improvements are slightly worse than in previous cases. In the best case, parallel EVENODD algorithm is better than the sequential one 2.2 times.

We have also considered the Liberation code as the alternative to the EVENODD algorithm. We have compared speeds of sequential Liberation algorithm and sequential EVENODD algorithms. For the Liberation algorithm implementation, we have chosen the jerasure library. However, performance of the sequential Liberation algorithm was poorer, as it can be seen in the Fig. 10.

From the Fig. 10, we can see that the encoding and decoding speeds of sequential EVENODD are around 5 times faster than the Liberation algorithm from jerasure library. For the decoding, we have considered the case where two data drives have failed. Results for the other combination of drives were similar.

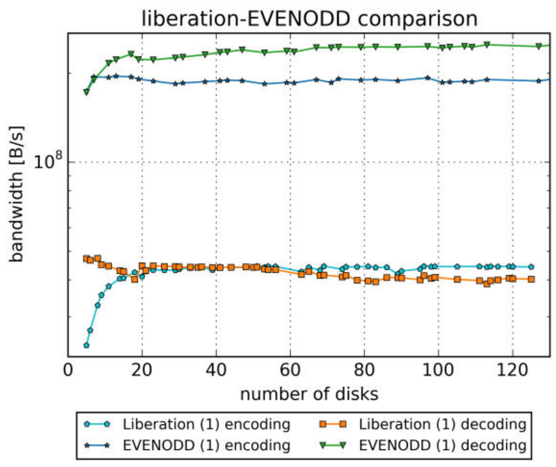

Fig. 10. Encoding and decoding speeds of sequential Liberation and EVENODD algorithms in the case when two data drives have failed

VI. Conclusion

In order to improve the reliability of the computer storage systems, RAID 6 is becoming popular as it allows the recovery from two simultaneous disk failures. The first recovery drive is defined to be a simple parity drive, while the definition of the second drive is left open. Therefore, multiple solutions have emerged, and the most promising candidates to be used are the parity array algorithms.

Liberation algorithm is a parity array algorithm based on EVENODD, which is considered as a good candidate for RAID 6. In this paper, we have compared execution times of sequential liberation and EVENODD algorithms, and concluded that EVENODD performs five times better. Additionally, EVENODD is easier to parallelize, as it is based on XOR operation and has little sequential dependencies.

As the final contribution, we have parallelized the algorithm by using OpenMP API. Parallel EVENODD shows improvement for the large number of disks. In the most common case when the two data drives have failed, parallel EVENODD is the best option when the number of disks is larger than 29. If the number of drives is larger than 127, parallel EVENODD can be more than five times faster.

Acknowledgment

This work was supported by the Serbian Ministry of Science and Education (project TR-32022), and by companies Telekom Srbija and Informatika.

References

[1] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2007.

[2] S. Stanley, "MTBF, MTTR, MTTF & FIT Explanation of Terms," *IMC Networks*, pp. 1-6, 2011.

[3] Intel corporation, "Intelligent RAID 6 Theory Overview and Implementation," Intel, 2005.

[4] J. S. Plank , "The RAID-6 Liberation Codes," in *FAST '08: 6th USENIX Conference on File and Storage Technologies*, 2008.

[5] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Transactions on computers,* vol. 44, no. 2, pp. 192-202, 1995.

[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics,* vol. 8, no. 2, pp. 300-304, 1960.

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong and S. Sankar, "Row-Diagonal Parity for Double Disk Failure Correction," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004

[8] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Transactions on Information Theory,* vol. 45, no. 1, pp. 272-276, 1999.

[9] C. Huang and L. Xu, "STAR: An efficient coding scheme for correcting triple storage node failures," *IEEE Transactions on Computers,* vol. 57, no. 7, pp. 889-901, 2008.

[10] A. Goel and P. Corbett, "RAID triple parity," *ACM SIGOPS Operating Systems Review,* vol. 46, no. 3, pp. 41-49, 2012.

[11] J. S. Plank, S. Simmerman and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2.," 2008.

[12] R. Buchty, V. Heuveline, W. Karl and J. Weiss, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators", *Concurrency and Computation: Practice and Experience*, vol. 24, no. 7, pp. 663-675, 2011

[13] OpenMP Architecture Review Board, "OpenMP Application Program Interface," 2005.

[14] Intel corporation, "Intel® Xeon Phi™ Coprocessor 31S1P," Intel, [Online]. Available: https://ark.intel.com/products/79539/Intel-Xeon-Phi-Coprocessor-31S1P-8GB-1_100-GHz-57-core. [Accessed 28. 4. 2017].

[15] J. Jeffers and J. Reinders, Intel Xeon Phi coprocessor high-performance programming, Newnes, 2013.

[16] NVIDIA corporation, "Tesla K20 GPU Accelerator Board," NVIDIA, [Online]. Available: https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. [Accessed 28. 4. 2017].