Parallelizing Reed-Solomon Erasure codes for Storage Systems

ECE1747 Project Report

December 22,2015

Authors:

Qiannan Zhao

Jin Xu

Shehbaz Jaffer

Abstract

This report presented a common way of providing resiliency against disk loss in a storage system is RAID. A typical RAID group consists of a bunch of D data disks and P parity disks. Every data block written to disk is first buffered and its parity computed using XOR or Erasure Coding. Once this is done, the data is written parallelly to all data and parity disks. In this project, we investigate how to use Pthreads and OpenMP to parallelize the encode computation, thereby improving the performance of RAID compute and rebuild.

First of all, we abstract the encode procedure to the model of matrix multiplication, and then use several methods to reduce the implementation time. The parallel approaches of Pthreads and OpenMP were implemented in our project to be compared to sequential workload version. We analyzed the influence of CPU utilization, block sizes, file sizes, cache awareness to this project and did many evaluation and comparison experiments to verify the analyses. Through the results of evaluation experiments, we can find that Pthreads provided better speedup of encode computation. Last we analyzed and evaluated effects of OpenMP method to the performance.

Contents

1 Introduction

Huge data centers often employ multiple drives with varying media types for storing data. The drive type could be SSD drive for workloads that require low latency and high throughput, or SATA drives for workloads that can tolerate medium to high latency but require large amount of storage. To achieve reliability while storing data in media devices, storage admins often keep backup or mirror of data on another drive. If the primary drive is lost, the data can be read from the mirror drive that keeps a consistent copy of the data.

As the amount of data to be stored increased, we saw the advent of drive pools and drive aggregates, where data was scattered around a bunch of drives, all assigned to 1 drive pool. This helped storing more data in multiple drives, thereby increasing the total storage space. However, failure of one of the drives lead to loss of data on that drive. Hence, copies of data on each drive were kept. Therefore, with the increase in drive storage capacity, the amount of redundancy that had to be stored in the system also kept increasing.

To reduce the amount of redundancy in the system, researchers came up with RAID which provided multiple standards to vary the amount of redundancy in the system. The standards were fixed and did not change during the lifetime of a drive (unless explcitly done by user).

1. **RAID** :

   In this section we describe some of the existing resiliency techniques on a drive aggregate. Note that all these techniques provide static redundancy pattern to data. They do not dynamically change the amount of redundancy in the system.

   a. **Mirroring** This is the simplest form of redundancy (also called RAID1). The data written to one drive is replicated to the other drive. This provides data gurantee and higher performance (as we can fetch different data blocks from different drives concurrently). However, this leads to increased storage space consumption. Also, if we update one data block, data on both the drives will now have to be updated.
   b. **Striping** Also known as RAID0, this provides zero reliability by keeping no redundant data in the system. We stripe all the data in the system on available drive in the drive aggregate. This technique is storage optimal,

and gives high throughput (since multiple data blocks from multiple drives may be simultaneously fetched).

c. **RAID 4** In this form of resiliency, we keep 1 dedicated parity drive. Data is striped across multiple drives, and a parity of each of these drives is computed and stored as a stripe on the parity drive. This provides failure resiliency against 1 drive loss in the drive aggregate. Also, with an update to any of the data stripes in the system, the parity has to be recomputed and written to the parity drive, because of which the parity drive wears out soon. Furthermore, there are scalability issues since the drive is now the hotspot for all write and update workloads.

d. **RAID 5** To overcome the limitations of 1 parity drive, another standard, RAID5 distributes the parity blocks across multiple drives in the drive aggregate. This helps solve the problem of parity drive being the hotspot. However prior work [5] shows how uniform load distribution may render the complete drive array unusable if all the drives underneath wear out at the same rate. Furthermore, RAID5 also gives us the same reliability gurantees as that provided by RAID4, failure of more than 1 data drive may render the complete drive unusable.

2.**Erasure Coding**

Erasure coding is a forward error correction (FEC) code for binary erasure channel. It transforms a message of k symbols into a longer message (code word) with n symbols such that the original message can be recovered from a subset of the n symbols. To put it in a easy way, erasure code is a technique that lets you take n storage devices, and encode them onto m additional storage devices, as a result, it could have the entire system be resilient to up to m device failures.

a. **Erasure Coding** - Reed Solomon Erasure Coding By adding t check symbols to the data, a Reed–Solomon code can detect any combination of up to t erroneous symbols, or correct up to t/2 symbols. As an erasure code, it can correct up to t known erasures, or it can detect and correct combinations of errors and erasures.

2 Background

Here we describe the internals of Reed-Solomon Erasure Coding.

2.1 Reed Solomon Erasure coding

Codes are based on matrix multiplication in liner algebra. To begin with, define a M * K distribution matrix A, whose first K rows are identity matrix and the rest (M - K) rows are parity matrix. Matrix B is data matrix. A * B equals a M * N column vector composed of [data + parity] (the coding words)

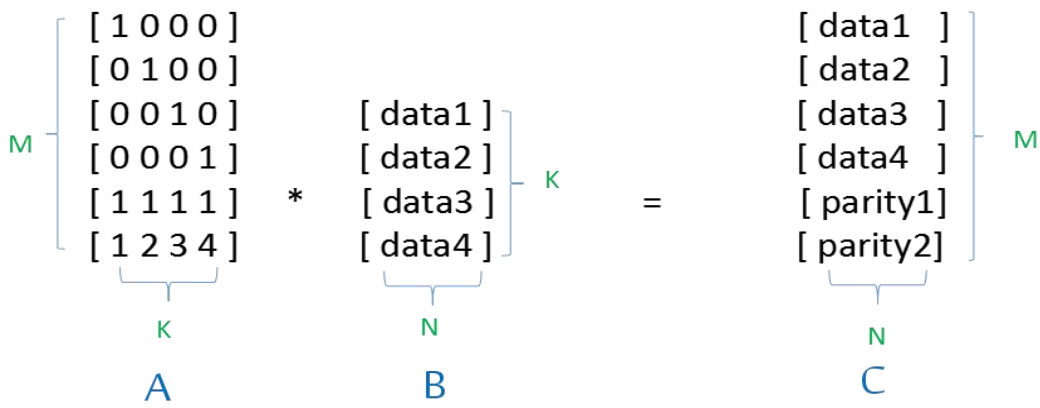

Figure 1. Erasure coding

Suppose there are errors in data 1 and data 3 to make them unavailable, we could delete corresponding rows in both B and A * B at the same time to return the data matrix. And we could use the Survivors matrix to restore data matrix B.

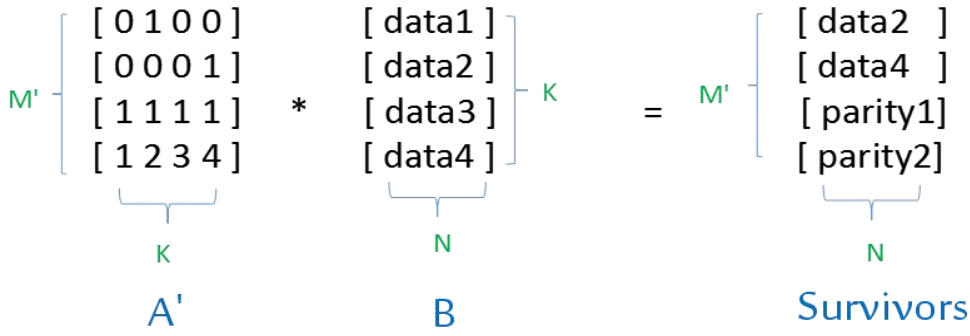

Figure 2. Erasure coding when data loss

Next part is the kernel of Erasure coding approach, explaining about how to restore data matrix B by treating with Survivors. First thing to do is invert A’ to get A’’, and multiply both sides of the equation by A’’. Since A’’ * A’ = I, we have just decoded data matrix B by multiplying A’’ * Survivors.

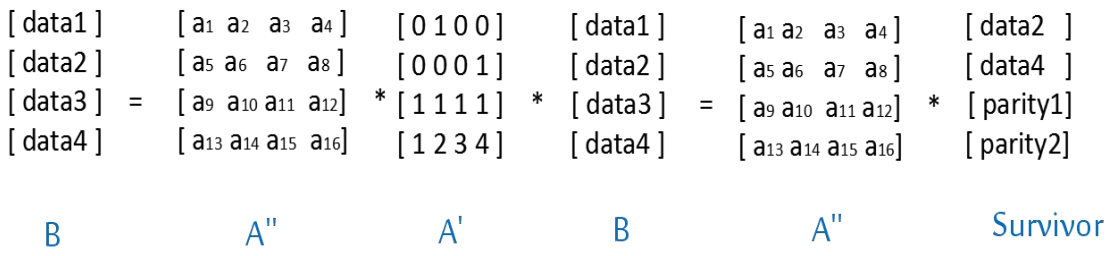

Figure 3. Erasure decoding

2.2 Parallelization opportunities

2.2.1 Opportunity 1 - 3D Matrix Multiplication

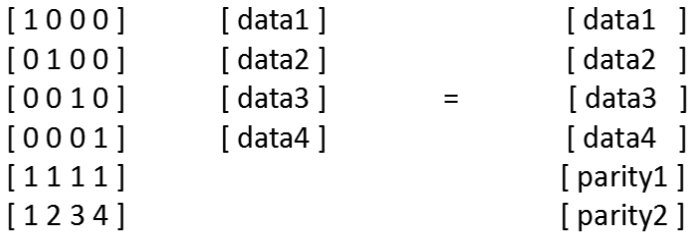

Figure 4. Encode workflow RS (4,2)

The figure above shows 3 matrices. First matrix to the left is called the Vandermonde matrix. It consists of two parts, the top part called identity matrix. it consists of 1s on each row, aiming to restore the second matrix to the third matrix at the right of equal sign; the second part consists of parity blocks, which means to encode the data to parity data stored in the third matrix at the right of equal sign.

The second matrix is the data matrix, from which we compute the parity matrix. The resultant matrix is the encoded data + parity matrix. We write the rows of the resultant matrix on different disks. This is a typical 3 dimensional matrix multiplication, hence we can use parallelism here to achieve speedup.

2.2.2 Opportunity 2 - Computation on Independent Data Sets

The second opportunity is achieving speedup through parallelism by prefetching more data simultaneously.

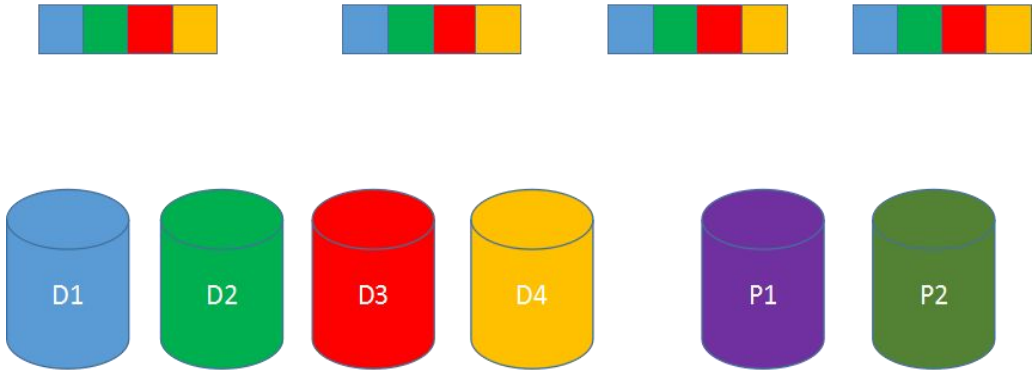

Figure 5. Sequential encoding of data blocks

Figure 5 above shows 6 disks, 4 data disks and 2 parity disks. On the top, we see data blocks - each block representing the color of the disk that it will get stored.

We typically start reading the data sequentially. Each data block is picked once , encoded, and written to its corresponding destination disk.If we want better speedup than reading and encoding blocks sequentially, instead, we parallelly fetch multiple blocks of data together once, and encode them as graph 6 shows below. This will avoid creating threads each time, thus saving much time. To be noted, this opportunity also explains that the number of times of creating pthreads are small in our approach,hence it count little if we create threads everytime after we fetch data.

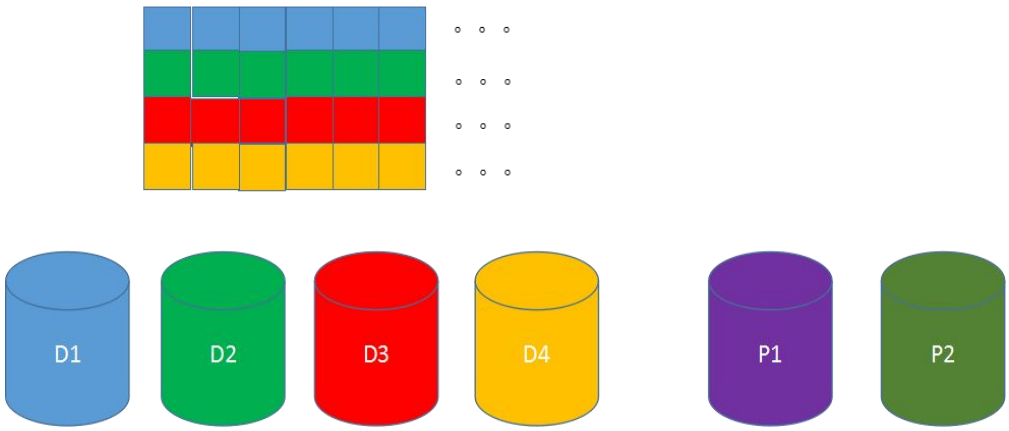

Figure 6. Parallel encoding of data blocks

2.2.3 Opportunity 3 - pipelining

Since the opportunities above solved speedup by paralleling several threads to

implement fetching data and computing matrix, we took a deeper analysis that could help speedup. Each thread mentioned before will experience two stages, one is fetching data and the other is computing data in matrix. So on the basis of parallelly fetch data and compute data, pipelining of threads that corresponds to certain data or certain data array could be applied here to separate fetching data phase and computing data phase. This could bring extra speedup.

3 Implementation

3.1 Setup

We do the evaluation experiments based on Core i7, and there are 8 cores,16 GB RAM for each ug machine.

3.2 Sample files

The input file that was used to be encoded was a random generated character file created from /dev/urandom. The block size of the file was 4096 bytes. We changed the count for dd command to get files of different sizes.

For example,
To create 4MB file, dd if=/dev/urandom of=4M bs=4096 count=1000
To create 400MB file, dd if=/dev/urandom of=400M bs=4096 count=100000

3.3 File size ranges (4M to 400 M)

If we set file size too small, then the overhead of managing parallelism like creating threads will account for a big part relatively. On the other hand, we are only supported to generate a at most 400MB file. So size 4M to 400M is a proper range.

3.4 Cores

We used command “cat /proc/cpuinfo output” to check out 8 cores in ug machine. The L1 + L2 cache size is 8MB.

4 Experiments

4.1 code profiling

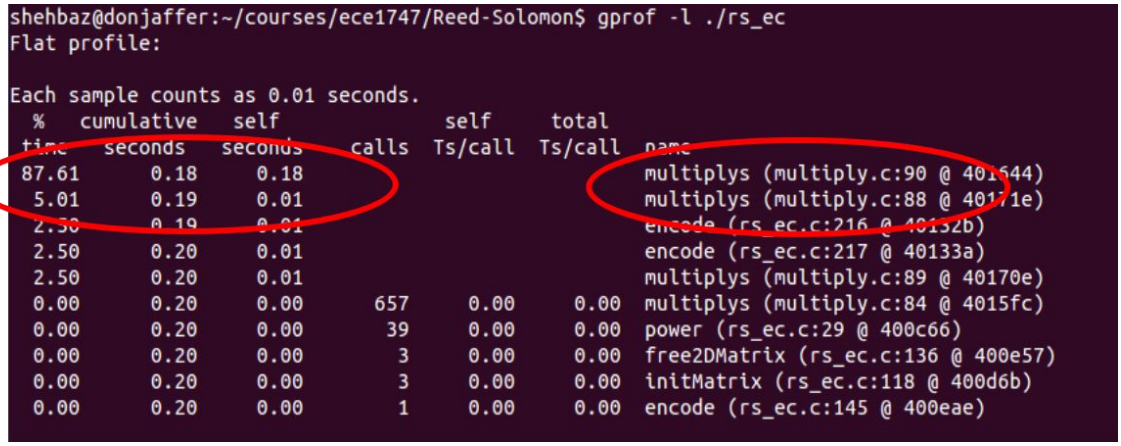

Figure 7. Time taken in each part of the process

GPROF is a GNU Profiling Tool. It is used to analyse the amount of time each part of the code takes during execution. We can see that most of the time is consumed during the matrix multiplication operation. The other parts of the code take much less time as compared to the matrix multiplication operation. Hence, there is a massive space to allow for parallelisation opportunity to speedup while doing Reed Solomon Erasure coding.

4.2 Effect of Caching

To improve the performance further, we could take advantage of a CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter.

In this report, we focus on the cache aware approach where items that are accessed together are stored together in the same memory block. Knowledge of the memory block size is needed to accomplish this. In this approach the items can be stored without the use of explicit pointers, but the layout of the items in memory does not constitute a sorted array.

We do the following experiments in the sequential workload case to check how using cache awareness approaches could help reduce the process time compared to cache unawareness method.

4.2.1 Multiplication loop order i,j,k vs i,k,j

The following graph shows a typical 3 dimensional matrix multiplication. And the top line is how data layout in cache. It consists of 3 loops. The first matrix is M * K, the second is K * N. Note that in the following part, we are going to show how accessing the second matrix will affect the CPU Utilization.

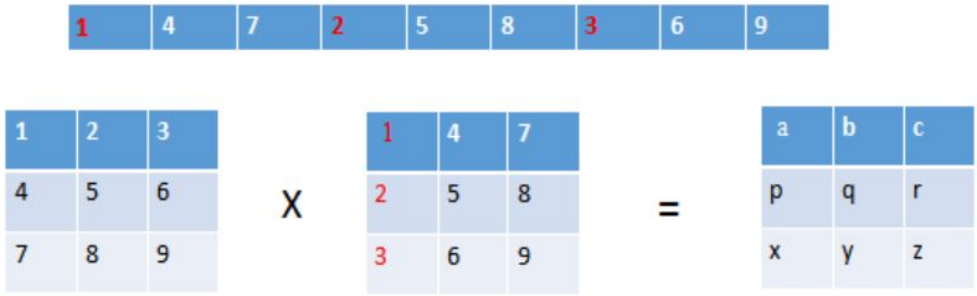

Figure 8. Fetching data in loop order i,j,k

Notice that usually the innermost loop in matrix multiplication is K, as shown below. When the innermost loop is being executed, the second matrix will be accessed column wise, column by column, as shown in the above graph. The positions of the column elements 1, 2 and 3 are shown in the cache. The elements are wide spread from each other, meaning that if we want to find elements 1,2 and 3 in cache, we need to find 1 and then jump over some elements to reach 2 and then 3, as with the second column to get elements 4,5, 6 , and the third column to get elements 7,8,9.

```
// multiply. and store product in C
for(i = 0 ; i < M ; i++){
        for(j=0; j< N ; j++){
                for(k = 0; k < K ; k++){
                        C[i][j] += C[i][j] + A[i][k] * B[k][j];
                }
        }
}
```

Figure 9. Matrix multiplication code in order i,j,k

Now, data layout in cache could be made use of to avoid complicity. If we switch the inner 2 loops of the code, as shown below, we can access data in the cache sequentially and hence reduce this cache thrashing.

```
// multiply. and store product in C
for(i = 0 ; i < M ; i++){
                for(k = 0; k < K ; k++){
        for(j=0; j< N ; j++){
                        C[i][j] += C[i][j] + A[i][k] * B[k][j];
                }
        }
}
```

Figure 10. Matrix multiplication code in order i,k,j

We could analyze how it works by switching these loops. Since j<N loop becomes the innermost loop after switching, and N is the number of columns of the second matrix, so now when the innermost loop is executed, it means the second matrix is accessed row wise, row by row. To get elements 1,4,7, we could go to cache and fetch them together, since they lied adjacent to each other, followed by elements 2,5,8 to access the second row and elements 3,6,9 for the third row. After doing that, multiplication result is computed in the third matrix. Making use of the layout of data in cache will help reduce process time and improve CPU utilization since there is no need to bother jumping over elements in cache to reach certain elements.

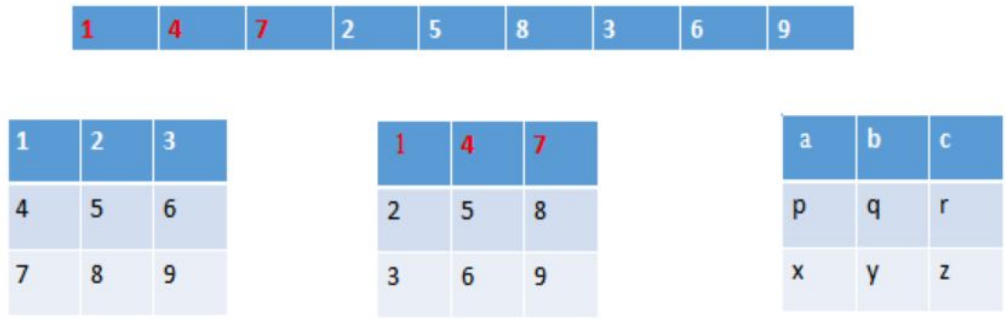

Figure 11. Fetching data in loop order i,k,j

4.2.2 Transposition of data matrix

When considering deeper of the switching loops method, we find it will bring overhead of frequently accessing positions in the third matrix. For example, some designated elements in the first matrix multiplying with [1,4,7] and stored in some positions in the third matrix, to be plus by next loop’ s results that are supposed to be stored in these positions. Without computing and storing results for one position in the third matrix once

and for all, but leaving results unfinished will cause overhead of accessing those positions again and again.

So another method of speedup for matrix multiplication is introduced as following shows. First of all, we make transposition of the second matrix(data matrix), and then still use the loop order of i,j,k, but change the computing code to

C[i][j] += C[i][j] + A[i][k] * B[j][k]

This will give us correct results with only accessing the position in the third matrix once and storing it before moving on to compute the result for next position. Avoiding the overhead of frequently accessing positions and plusing the new computing value to the last value till getting the correct results, the approach of transposition of data matrix will bring even less processing time.

4.2.3 Effects of Cache awareness

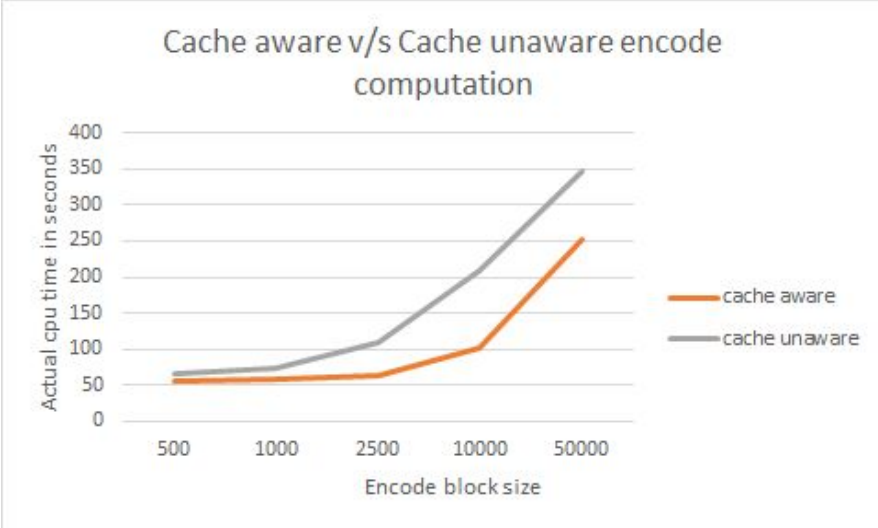

Figure 12. Cache aware v/s cache unaware encode computation time

To be noted, the values in the following graphs have been averaged over 5 runs. The figure above shows the comparison of actual time taken for loop order of i,j,k(cache unaware) and loop order of i,k,j(cache aware) as N of matrix B (which manifests number of column of the data matrix) increases. As obvious as the figure shows, when N of matrix B increases, cache awareness method will bring better speedup as compared with cache unawareness method. Especially when N hit 10000, where cache size reaches

8MB(L1+L2 cache), cache L2 starts getting thrashed, cache awareness method will give us more significant speedup.

4.2.4 Effects of transpose of data matrix

Figure13,14 shows the effect of transposing matrix B, the data matrix for RS erasure coding. We compute the parities for 1000 data blocks and 10 parity blocks. We note that for sequential implementation, there is no visible improvement in the speedup. This is primarily because when we are performing encode operation sequentially, the elements of matrix A, B and C are being accessed in row-major order. Although for a value of I in matrix computation, the Row of A and B will be same, the rows being accessed in C will constantly be churned. This leads to disparity and does not lead to speedup.

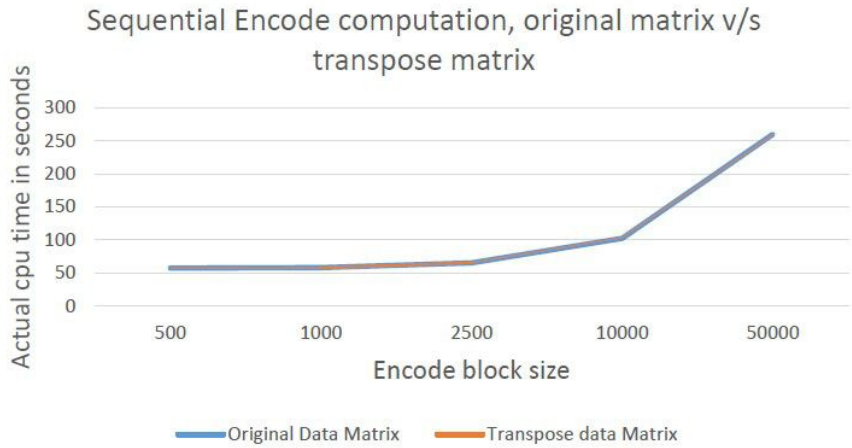

Figure 13. Sequential encode computation, original matrix v/s transpose matrix

In the parallel case as figure 14 shows however, with the increase in the number of rows in matrix C, we start seeing speedup. Although C is being shared among various processors, the degree of parallelism overcomes the amount of churning that takes place for matrix C. this leads to better speedup for parallel workloads.

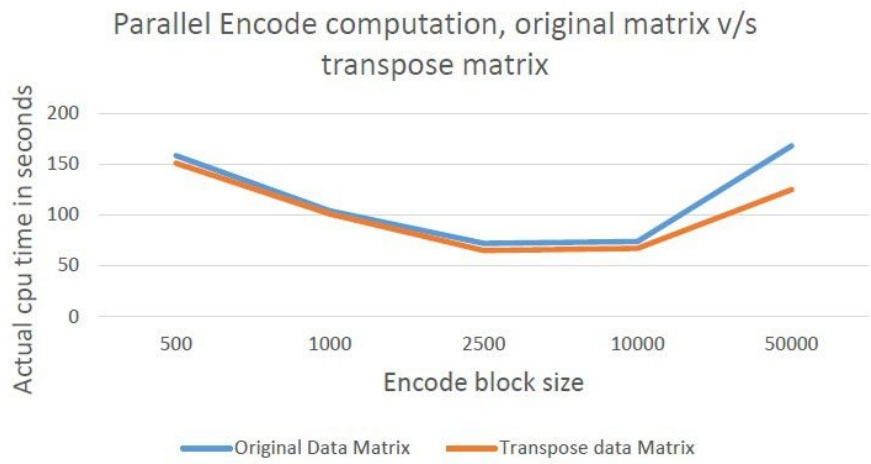

Figure 14. Parallel encode computation, original matrix v/s transpose matrix

4.3 Effects of increase in data files

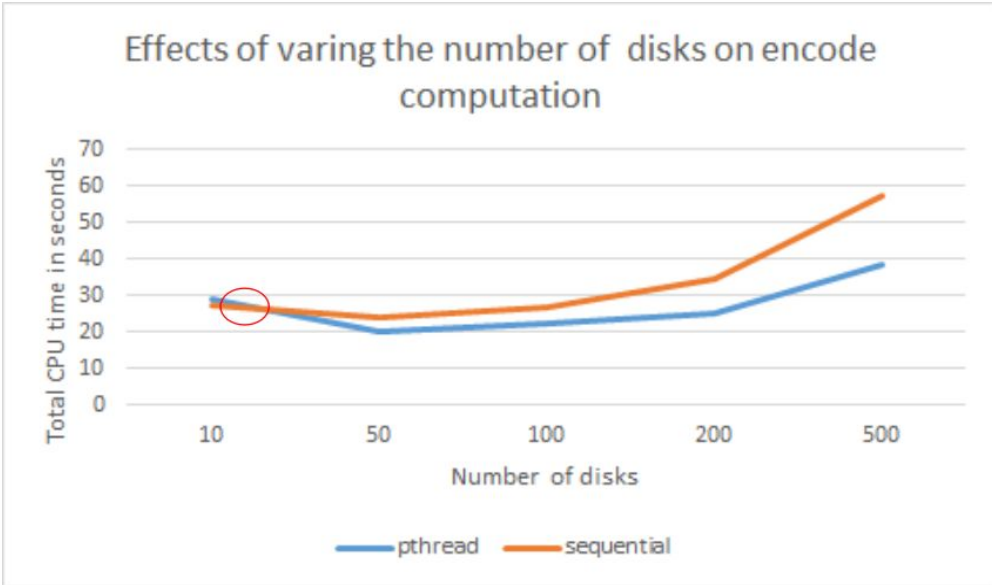

Figure 15. Effects of varying number of disks on encode computation time

Figure 15 above shows comparison of pthreads workload and sequential workload as number of disks increases. As we can see, when number of disks are small, parallelism has higher overheads. For number of disks greater than 14, however, we see pthreads outperform sequential computation. With larger increasing number in disks, pthreads implementation takes considerably less time than sequential computation.

4.4 Effects of increase in file size

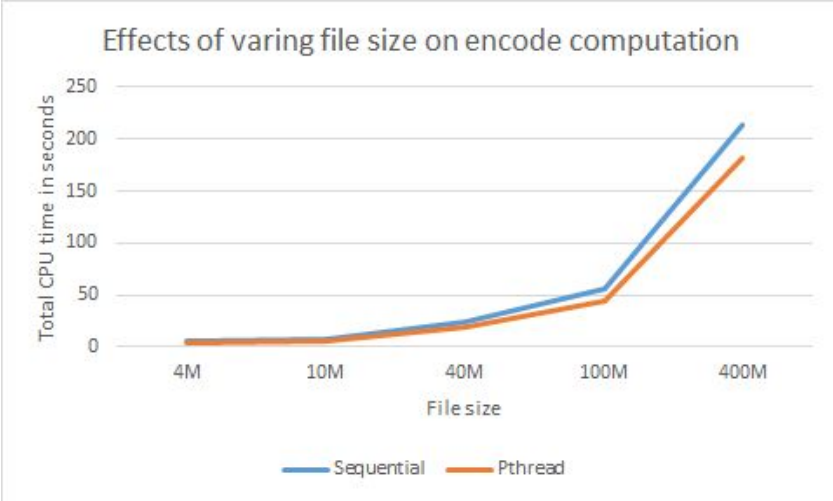

Figure 16. Effects of varying file size on encode computation time

Figure 16 above shows comparison of pthreads workload and sequential workload as increasing file size. When file size is small, times of fetching data and computing are few, thus effect of parallelism is not that notable. But when file size reaches 100MB, large amount of data is to be dealt with, so parallelly computation begins to show its advantages. Due to restrictions of ug machine, we are allowed to generate a 400MB file at most, but along with the lifting trend, we can tell that speedup effects will be more prominent when file size are large enough.

4.5 CPU Analysis

4.5.1 CPU Utilization for sequential workload

Figure 17. shows 2 lines, the orange line shows the total CPU time for the process completion. The Blue line shows the actual time cost when the thread was scheduled in the CPU. Values of x axis are N of matrix B(which manifests number of columns of the data matrix).

For lower number of columns N, we see that the total time is very large. this is because a lot of time is spent fetching the data block from disk, and the CPU is underutilized, as seen in next graph. However, the actual time the CPU is utilized exhibits a tendency of increase as we increase the columns of data matrix. CPU starts getting used more

efficiently, and percentage of CPU utilization grows gradually. And the gap between actual computing time and total processing time is decreasing correspondingly.

Analysis of CPU utilization also implies that if we aim to use processors more efficiently, and reduce extra time as possible, we could do it through increasing number of columns in data matrix. Since file size is large, in this way, rather than wasting time fetching data then computing time and time again, the processor will deal with more data once, thus reducing overhead and improving performance.

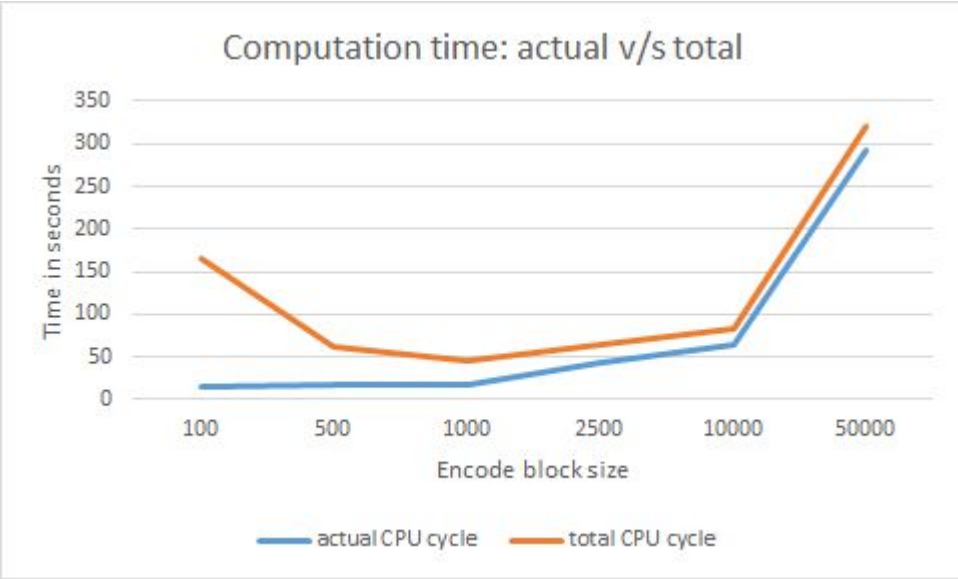

Figure 17. Sequential computation time: actual v/s total

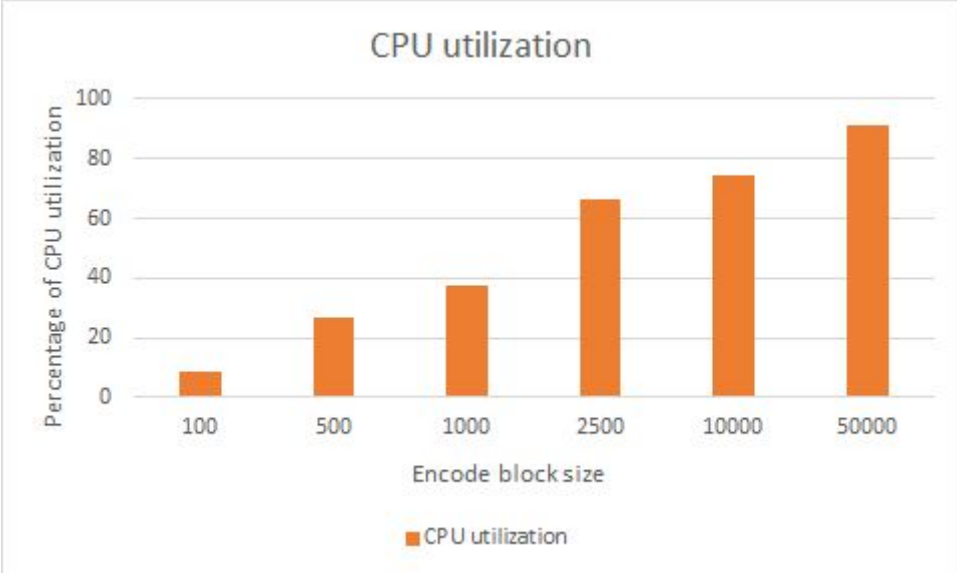

Figure 18. Percentage of CPU Utilization

4.5.2 CPU Utilization for parallel workload

We've also done CPU utilization experiments for pthreads workload. As we can see the figure below, orange line shows the total CPU time for the process completion, while blue line shows the actual time cost when the thread was scheduled in the CPU. What brings our notice is that actual time cost exceeds total time cost. Reason for this is that actual time is recorded by making a sum of each thread's time while total time is recorded from the beginning to the end of process. So it makes sense that summing each thread's time will be more than process completion time period since multiple threads are parallelly operating. It also explains why we basically use total time cost for comparison of sequential workload and parallel workload.

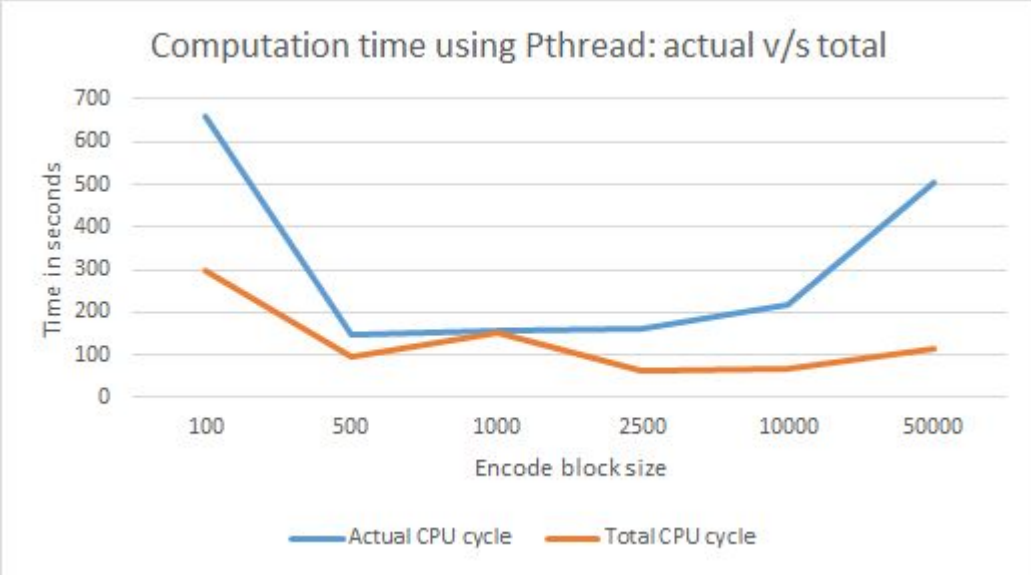

Figure 19. Pthreads computation time: actual v/s total

4.6 Effects of parity blocks

```
[ 1 0 0 0 ]     [ data1 ]          [ data1   ]
[ 0 1 0 0 ]     [ data2 ]          [ data2   ]
[ 0 0 1 0 ]     [ data3 ]     =    [ data3   ]
[ 0 0 0 1 ]     [ data4 ]          [ data4   ]
[ 1 1 1 1 ]                        [ parity1 ]
[ 1 2 3 4 ]                        [ parity2 ]
[1 4 9 16]                         [ parity3 ]
[1 8 27 64]                        [ parity4 ]
…
```

Figure 20. An example with more parity blocks

As the amount of data increases, the chance of data error is also increasing. In most cases,the data model of workflow RS (4,2) is not proper for encoding. More parity blocks are needed to make up errors. To meet this requirement, we want to check if more parity arrays in the first matrix will affect computing time.

We increase the number of parity arrays while keeping the size of the first matrix constant to see if it will bring extra overhead on computation. As the following figure 21 shows, there are basically no big difference of time when varying number of parity disks for both sequential workload and parallel workload.

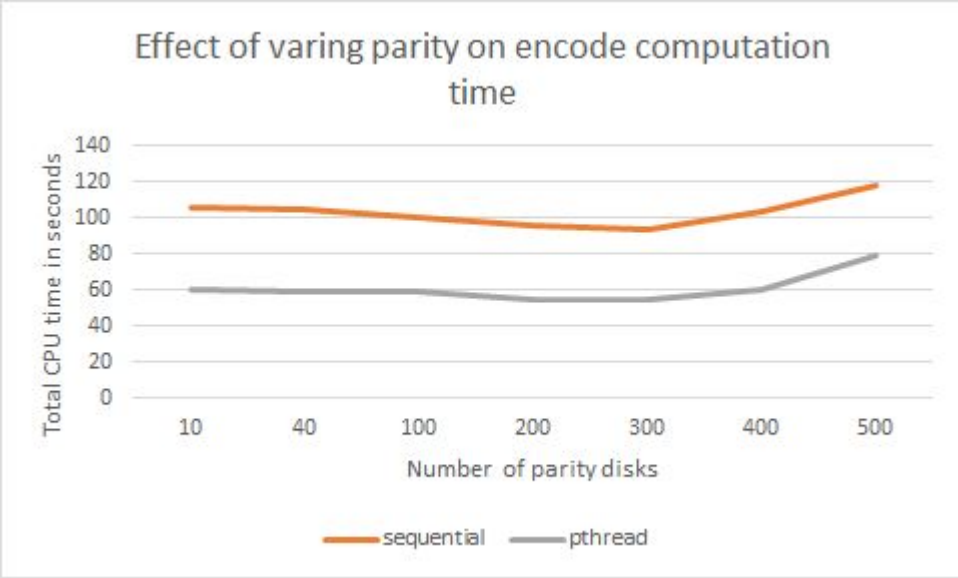

Figure 21. Effects of varying number of parity disks on encode computation time

4.7 Effect of pthread creation v/s keeping a constant pthread pool

If we compare the overhead of pthread creation and management, we find that the overall effect of thread creation is quite low as compared to the actual computation. As shown by gprof before, the amount of time spent while creation of pthread is negligible. We show the number of pthreads that are created varying the encode block size for multiple files. For a given file size, the number of threads created decreases as the encode block size increases (for a constant number of data files).

The given table shows the number of threads that get created for 1000 data files, with varying file sizes (4M to 400M), and varying encode block reads (N = 1000 to 10000).

Encode Block Size \ File Size	4M	40M	100M	400M
1000	1	10	25	100
2000	1	5	13	50
5000	1	3	5	25
10000	1	1	3	13

Table 1. Number of threads for different file sizes and different N

4.8 Open MP Analysis

OpenMP provides two forms of scheduling - static and dynamic. In static scheduling, all threads are scheduled based on a fixed sequence to each of the processor. Dynamic scheduling on the other hand, assigns different threads to different processors depending on the amount of CPU utilization on the processor. We did not observe a lot of difference between the two forms of scheduling.

Moreover, we observe that the encode computation outperforms the sequential computation only upto 4 cores. Above this, for 6 and 8 cores, our encode performance drops considerably and performs worse than sequential code. We notice high CPU utilization for 6 and 8 cores. However on increasing the threads to a larger value, we notice better performance than sequential code. We think that OpenMP has its own optimizations that it performs which are quite obfuscated to a programmer.

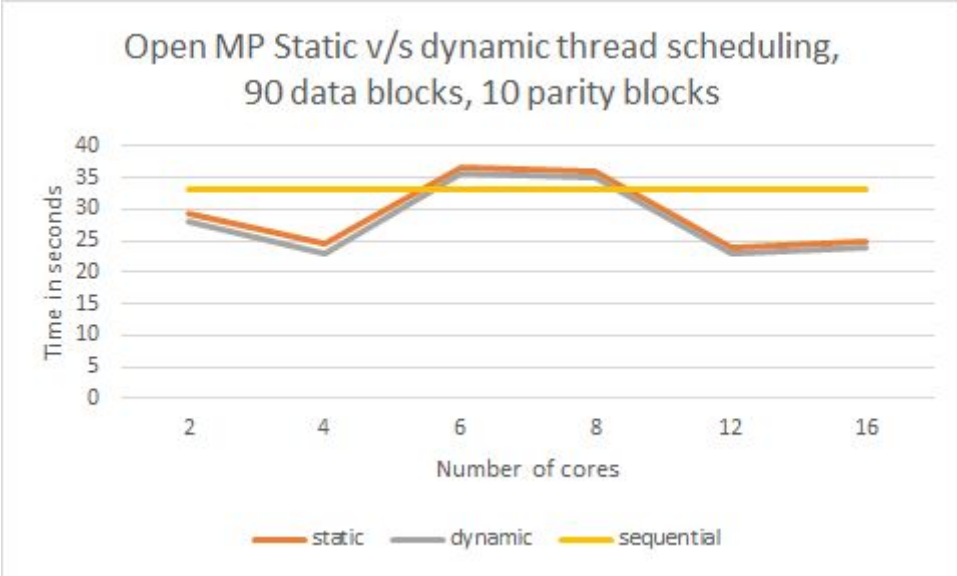

Figure 22. Open MP static v/s dynamic thread scheduling

5 Conclusion

In this project, we investigate how to use Pthreads and OpenMP to parallelize the encode computation, thus improving the performance of RAID computation and rebuilding. We abstract the encode procedure to the model of matrix multiplication, and then use several methods to reduce the implementation time. The parallel approaches of Pthreads and OpenMP were implemented to be compared to sequential workload version. We found that larger data prefetching basically means better parallelism performance.If small blocks are fetched, more time is spent in fetching the blocks than actual computation. Only when saturating CPU can we expect speedup.

We also analyzed the influence of varying block sizes, file sizes, disk numbers, cache awareness to encode computation time and did many evaluation and comparison experiments to verify the analyses. Through the results of evaluation experiments, we can find that ① we are able to achieve 3x speedup by making encoder "cache aware"; ②we are able to achieve 1.3x - 1.7x speedup by using pthreads over sequential workflow; ③we are able to achieve 1.4x speedup by using OpenMP.

6 Future Work

1. Increasing parallelism with GPUs. (e.g. OpenCL, CUDA) :

Further ,we intend to investigate if GPUs can be used to parallelize the parity computation, thereby improving the performance of RAID compute and rebuild. Maybe we could study on CUDA

2. Galois Field Arithmetic for XOR Computation.

Currently we only use Galois Field for matrix multiplication, but we could also use the advanced Galois Field Arithmetic to compute parity XOR data. This will do better to error detection or error correction.

3. Analysis for larger files and larger data blocks:

We were now restricted to 400MB files, we intend to do more experiments based on even larger files to see the effects of parallelism compared to sequential version.

4. Decode workflow:

We only implemented relatively complete encode procedure. However, to check if the code could be correctly restored back to data, and if the parity truly works, we still need

to implement decode phase, which could be done by computing the inverse matrix of the the first matrix and multiply the inverse matrix with the third matrix to the right of equal sign. Then, by making use of parity, we could get the correct data returned.

7 Additional Findings

1. Pthread library on i5 processor -

We've found that for i5 processor, multiple threads will be scheduled to one single core by default. To properly distribute threads to multiple cores, we need to explicitly pin threads (using set_cpu_affinity). Yet we did not observe same behaviour with i7 machine.

2. OpenMP -

We've also found that OpenMP were using obfuscated scheduling policy.