5-2014

Minimal-density, RAID-6 Codes: An Approach for w = 9

Bryan Andrew Burke
University of Tennessee - Knoxville, bburke@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Bryan Andrew Burke entitled "Minimal-density, RAID-6 Codes: An Approach for w = 9." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James S. Plank, Major Professor

We have read this thesis and recommend its acceptance:

Jian Huang, Bradley T. Vander Zanden

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

5-2014

Minimal-density, RAID-6 Codes: An Approach for w = 9

Bryan Andrew Burke
University of Tennessee - Knoxville, bburke@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Bryan Andrew Burke entitled "Minimal-density, RAID-6 Codes: An Approach for w = 9." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James S. Plank, Major Professor

We have read this thesis and recommend its acceptance:

Jian Huang, Bradley T. Vander Zanden

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Minimal-density, RAID-6 Codes: An Approach for $w = 9$

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Bryan Andrew Burke
May 2014

Abstract

RAID-6 erasure codes provide vital data integrity in modern storage systems. There is a class of RAID-6 codes called "Minimal Density Codes," which have desirable performance properties. These codes are parameterized by a "word size," w, and constructions of these codes are known when w and $w+1$ are prime numbers. However, there are obvious gaps for which there is no theory. An exhaustive search was used to fill in the important gap when $w = 8$, which is highly applicable to real-world systems, since it is a power of 2. This paper extends that approach to address the next theoretical hole at $w = 9$ by expanding upon the techniques used for $w = 8$ and adding customizations to allow for parallel processing.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 History

Erasure codes are algorithms or functions, usually represented by matrices, which provide fault-tolerance for data. They *encode* the data such that chunks of the encoded data may be lost, but the original data can still be recovered. In the modern computing environment characterized by big data – storage, retrieval, and processing of large quantities of data – data integrity is an increasingly important area of study and research.

In 1960, Irving Reed and Gustave Soloman published a paper [6] on what is probably the first construct we now refer to as an erasure code. It uses coefficients of a polynomial over the field $GF(2^w)$ to encode and decode symbols. To compute these coded symbols, a special form of arithmetic, known as *Galois Field Arithmetic*, must be used. Traditionally, multiplication in this field is computationally expensive, and so researchers looked to variants of this code for real storage systems. However, Reed-Solomon codes are general-purpose, and they are *MDS* (defined later), so they have seen wide adoption in many areas, despite their relative inefficiency.

1.2 Definition of a Code

To implement an erasure code, one must decide how much fault-tolerance is needed and how many independent data stores (traditionally disks) are to be used. With these in mind, an erasure code can be selected which provides the desired properties. Broadly, we define the data-centric properties of an erasure code by a triplet: (k, m, w). k is the length of the input data-vector (or the number of disks, if you prefer). m is the length of the coding-vector: the number of coding symbols generated for a given k data symbols. w is the *word-size* – the smallest unit of data on which the erasure code operates. Thus $k + m$ is the total output data-vector and usually corresponds to the total number of disks participating in an coded storage system. Figure 1.1 illustrates the coding operation for a classic, disk-based setup: RAID-5 with 3 disks. Each disk conceptually holds w bits, and thus the matrix is a $3w \times 2w$

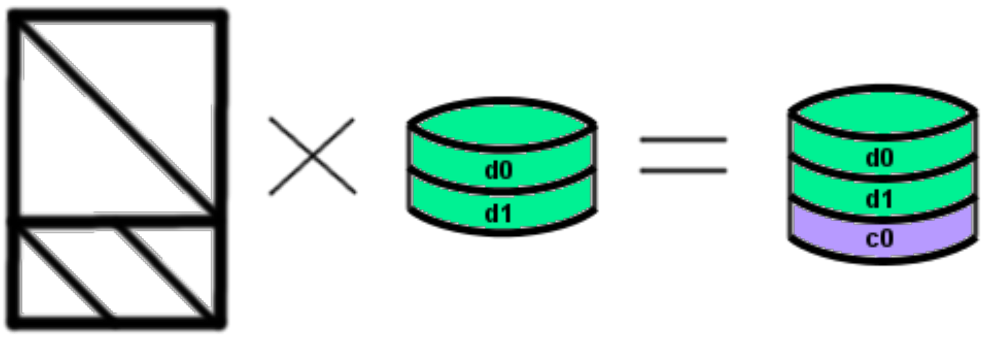


Figure 1.1: Example storage system using $k = 2$, $m = 1$; a standard RAID-5 setup

bit matrix. The matrix is divided into upper and lower halves, and the diagonals represent the ones in the bit-matrix. The matrix acts upon the live data, which is conceptually the two disks shown, and the result is a larger vector, conceptually an array of three disks consisting of the original data plus one disk of coding data. Now, if any one of the resulting three disks fail, the data from the original two disks can be recovered.

We can further define an erasure code by the fault-tolerance it provides. Intuitively, one may think of this as being equal to m, but this is not always the case: m is only an upper-bound on the guaranteed fault-tolerance a particular code provides. In addition, even if the fault-tolerance is equal to m, it is not certain that *every* combination of m failures – or *erasures* – can be tolerated. Yet, a special class of codes, known as *MDS* codes, provide exactly this property.

1.3 Implementation

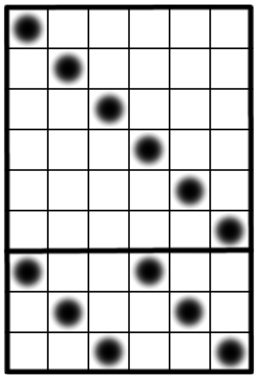
As mentioned earlier, erasure codes can usually be represented by a matrix or set of matrices. Such a matrix looks as follows:

$$\left(\begin{array}{c} I_k \\ \hline \text{CDM} \left\{ \begin{array}{cc} \alpha_{1,1} & \cdots \\ \alpha_{2,1} & \cdots \\ \vdots & \end{array} \right. \end{array}\right)$$

Figure 1.2: Matrix representation of a Reed-Solomon Code

In this matrix, the top part, I_k, is the $k \times k$ identity matrix, needed for preserving the original (input) data vector. The bottom part of the matrix, labelled the *CDM*, for *code distribution matrix*, is the part that defines the properties of the erasure code outlined in section 1.2. It is an $m \times k$ matrix whose elements are from $GF(2^w)$.

Operating on bits rather than elements of a Galois Field is advantageous for a number of reasons, chief among them is speed. When the elements of a coding matrix are in $\{0, 1\}$,


(a) RAID-5 Code

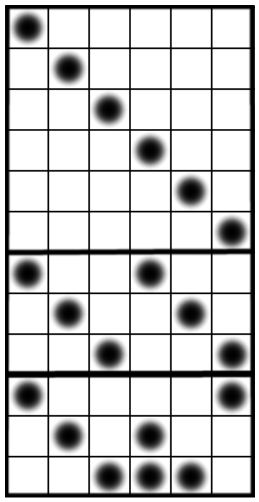

(b) RAID-6 Code

Figure 1.3: Example RAID-5 and RAID-6 Matrices for $w = 3$, $k = 2$

arithmetic in $GF(2^w)$ collapses to XOR operations only, which computers can execute very quickly. In order to extend the example in figure 1.2 for a bit-coding matrix, we increase the dimensions of the matrix, $(k + m) \times k$, by a factor of w: $(kw + mw) \times kw$. Now the CDM can be partitioned into $w \times w$ bit-matrices, which we will call *panels*, which in turn are grouped together to form *super-rows* – rows of panels – and they can operate on w-sized units of data. In particular, this means our input data-vector consists of k w-sized chunks.

1.4 RAID-6 and Minimal-density Codes

We have arrived now at the specific codes this paper concerns itself with: minimal-density, RAID-6 codes. Minimal-density just means that the CDM achieves a lower-bound on the number of 1's, defined by Blaum and Roth [1] to be $kw + k - 1$. Since relatively sparse matrices perform well under multiplication, these have certain performance advantages [5].

RAID-6 codes are a superset of RAID-5 codes: RAID-5 are MDS and defined by $m = 1$, $k > m$. RAID-6 are defined by $m = 2$ and $k \geq m$. Note that RAID is not a code but a specification. By that specification, a RAID-6 code's first super-row must be identical to the first (and only) super-row of RAID-5. In other words it must provide simple XOR parity in its first coding chunk/disk. Its second super-row can be anything, *as long as the resulting code is MDS.* Figure 1.3 shows an example RAID-5 and RAID-6 code for $w = 3$, $k = 2$.

Such codes are known for certain values of w: Blaum and Roth found codes when $w + 1$ is prime [1]. Plank found codes when w is prime [2]. These notably do not specify a code for the practical $w = 8$. Without a much of a theoretical framework, these matrices were largely a mystery. As such, Plank performed an optimized search for these codes [3].

There are still gaps in the values which w can take on for minimal-density RAID-6 codes, and this paper describes one set of techniques for enumerating the matrices of the next unknown value, $w = 9$.

Chapter 2

Terminology

Erasure Code – An algorithm, representable as a matrix in two parts: the identity matrix which makes up its first k rows (or *super-rows*, in the case of a bit-matrix) and the *CDM* which, when applied to a chunk of data, produces redundant data appended to the original. This extra data can be used to reconstruct certain amounts of the original data if some is lost.

k – The number of data blocks.

w – The "word-size"; the smallest unit of data which the erasure code operates on.

m – The number of coding blocks; the extra (coding) data used to reconstruct lost data-blocks.

Coding Disk/Block – A logical disk (or block) which holds coding-blocks (or coding data).

P/Q Disks – When $m \in \{1, 2\}$, it is common to refer to the first logical coding disk (or coding block, or anything associated with the first block) as the *P-disk*, or simply *P*, particularly when this holds simple parity (XOR-based) data. Additionally, when $m = 2$, it is common to refer to the second coding disk as the *Q-disk*, or *Q*.

MDS – *Maximum-Distance Separable* – A property of certain erasure codes. When an erasure code is MDS, it can lose *any* combination of disks/blocks in a unit (or *stripe*), up to its defined fault-tolerance, and still recover all of the original data.

Stripe – The smallest independent unit of of an erasure code, which contains k blocks of data and m blocks of coding data. A single disk, or set of disks, may have a great many stripes.

CDM – The part of an erasure code matrix which defines how that code creates its coding blocks from the data blocks.

RAID-4/5 – A simple MDS erasure code for $k > 1$ and $m = 1$ that uses a XOR (or parity) calculation to provide fault-tolerance. In RAID-5, the stripes are rotated across the disks, i.e. there is no dedicated P-disk.

RAID-6 – The erasure code that is the primary subject of this paper. It builds upon the RAID-5 specification to provide a second coding calculation ($m = 2$, and stored on the Q-blocks) which provides a fault-tolerance of 2.

super-row – When discussing bit-matrix erasure codes, a super-row refers to a collection of w rows, aligned on a w boundary, which collectively act on a single w-length block of data, or produces a single w-length block of data.

panel – A $w \times w$ bit-matrix which makes up a single unit of a super-row.

SMP(CPU) – *Symmetric Multi-Processing* – The processing paradigm where multiple, concurrent units of operation (tasks/threads/processes) operate on parts of a problem to more efficiently solve it.

Chapter 3

Methods

There are several values of w for which there are no known constructions to produce the erasure code matrices for. A few of these word-sizes are 8, 9, and 14. Plank discovered all known code matrices for $w = 8$ [4] through an optimized search of the solution space [3], and I attempt here to extend the work to $w = 9$.

As these search spaces grow increasingly large, an exhaustive search relies more and more on intelligent pruning and raw computational power. Having examined Plank's techniques for reducing his search, I applied nearly all of his techniques and a few techniques of my own to prune the search space. Yet, I guessed that pruning alone would not be enough to traverse the $w = 9$ space. Therefore, I decided to design my code from the beginning to facilitate parallel processing of the solution space, and I adjusted the pruning techniques as necessary to enable parallel computation. The following sections detail the pruning techniques used, as well as the design decisions made, to achieve this goal. In the next section, I address the size of the search space in more detail.

3.1 Borrowed Techniques

While a full description of his work can be found in his paper [3], I will here briefly describe the techniques of his which I used without modification.

3.1.1 General Structure

A minimal-density, RAID-6 code is one which reaches the lower-bound on the number of ones in the CDM, determined theoretically in [1] to be $2kw + w - 1$. Explained intuitively in [3], this is equivalent to all panels of the first super-row of the CDM consisting of I_w identity matrices (like RAID-5), and the second super-row containing all $w \times w$ permutation matrices, with an extra '1' in all but one of the panels.

3.1.2 Setting the Extra-bits' Rows

One requirement for remaining MDS is ensuring that all panels in the CDM are invertible; therefore, we must constrain any operations on the panels to preserve this property. Row and column swaps are two such operations, and if applied uniformly to all panels, obtain an equivalent CDM [3]. Further, if we swap two rows, and the identity matrices in the P super-panel are destroyed, then we may swap two columns to restore the identity matrices.

Yet another property is that the sum of any two panels in the Q super-row must be invertible as well. In particular, note that the extra '1' in each panel must occupy a separate row, else the sum of the two panels will contain an even number of ones in each row and thus not be invertible.

Combining the above with the operation of swapping panels – while still maintaining our constrained structure – it is clear that every code generated can be converted to one where panel i has its extra '1' in row $w - i$.

3.2 Modified Techniques

The following sections will discuss the methods which I borrowed but modified slightly in order to address potential issues with, or to optimize for, parallelization.

3.2.1 Permutation Matrix Generation

Generating permutation matrices – equivalently, generating permutations of the set $\{0, ..., w-1\}$ – is a long, expensive operation when applied recursively. In [3], permutation matrices were generated one row at a time, then tested for validity before moving on. However, this is still a recursive operation and is therefore difficult to break out of at arbitrary points. It is more difficult still to designate a unit of work, i.e. to have well-defined starting and ending points that can easily be duplicated.

In an effort to more easily separate out the search of the $w = 9$ space into many, independent units, keep state, and simplify the logic, I opted for a slower permutation enumeration (because it does not perform verifications on partially-generated matrices), but one which produced whole permutations at each cycle. This simplified some of the top-level enumerations and allowed me to separate out the permutation-matrix-enumerations into their own API/library. Finally, since there is a well-defined number of these – that being $w!$ – I can more easily count and segment the search.

3.2.2 Partial Verification

As mentioned in section 3.2.1, I was not able to utilize the full power of the $I1$ function from [3] for row-level verification. This function works by taking the first x rows of two $w \times w$ permutation matrices, summing them, and determining the resulting matrix' rank.

Without getting into too much detail, it determines if this *could* be the rows of a potential pair of panels, in particular, that they could still sum to an invertible matrix. If not, then the logic would eliminate the current row being generated. In this way panels could be built row-by-row, only creating panels that will be MDS with respect to the already existing ones.

Instead, I make use of a similar idea, but applied to whole panels; those whose base permutation matrix, as well as the extra-1, are already defined. Once created, I sum this panel with all those generated so far to verify the MDS property. If it fails, then we abandon this panel.

While this is certainly not as efficient as a row-by-row verification, it still results in an enormous improvement, since MDS matrices are the exception rather than the norm.

3.3 Original Techniques

This section describes the motivation and design decisions unique to my project, which allow parallel processing.

3.3.1 `simple_enumerator`

In attempting to implement some of 3.2.1, I also had to consider one other design choice: I needed the ability to stop the enumeration of permutation matrices at arbitrary points, copy them, then continue. This was needed both to have a master thread or process which could seed the (independent) workers at different starting points in the enumeration, and to enable a different enumeration mechanism, which over-constrained the search and was thus abandoned, but is discussed in the conclusion.

After outlining, implementing, and testing a traditional (purely recursive) approach, I moved the state information to a stack stored in the `simple_enumerator` object. Now, every call to the library's `enumerator_next()` function operates on the stack, popping and pushing new frames as appropriate. Now, making a copy of the enumeration at a point-in-time is possible with a deep-copy of the `simple_enumerator`.

3.3.2 Extra-bits' Column-Uniqueness

Above, we used the fact that all the extra-bits' rows must be unique in order to set them without searching to $r_i = w - i$, for panel $X_i, i \geq 1$. The same principle applies to the column of the extra bits. While we cannot simply define them like we do for the rows, we can use their uniqueness to avoid collisions, i.e. as we build panel X_j, we can ensure that we choose only columns not already taken by the extra bits in all panels $X_i, i < j$. In doing so, we reduce the search for each whole solution from $(w-1)^w$ to $w!$.

3.3.3 Permutation Limits

At the highest level of the permutation, my `raid6_enumerate()` accepts a `size_t limit` parameter, which tells it how many time to enumerate through the first panel's (X_1's) permutation matrix. With this, and the easy counting from 3.2.1, I was able to efficiently break the search-space into mutually-exclusive components based on the first panel's current permutation matrix. In my final program, the workers are given only so many permutations to go through for X_1: $worker\_permutations = (w!)/number\_of\_workers + 1$

3.3.4 MDS Back-propagation

In section 3.2.2, we saw a technique to heavily prune the search space, based primarily on the idea that MDS matrices are much less frequent than all other matrices with the form described in this paper. However, there is still a chance for further improvement.

Consider the following case: the base permutation matrix of the panel which is currently being generated is identical to the previous panel in their first 2 rows, and the extra-1 is not in those rows. Certainly the MDS test will fail on these two panels, since the first two rows will sum to all zeroes. However, given how the MDS test prunes, the algorithm will continue trying every possible position for the extra-1, even though there is no position which will come to the aid of the first two rows.

Further, suppose the algorithm which enumerates the permutation matrices is allowed to run again on our problematic panel, but the next permutation has the same first two rows as the last. Then we will try every column position of the extra-1 even though none can succeed.

The back-propagation technique seeks to eliminate some of these unnecessary searches. It works as follows: During the testing of the MDS property, we deterministically test the rank of the sum of two panels by making the sum lower-triangular (we make it lower-triangular instead of upper-, because the `simple_enumerator` works from the first rows down, so the final rows have longer periods of stability). If the test fails, it did so because a particular row became all zeroes; suppose this is the row with index $x \leq w$. Now, if the index of the row of the extra-1 is strictly less than x, then changing columns will not change the last $w - x$ rows. So, we abort changing columns and immediately go back up a level of recursion to generate the next permutation matrix for the current panel.

Taking this further, if the next permutation matrix' last $w - x$ rows are identical to the previous permutation matrix', then we can go ahead and skip it, and any others, until something in the last $w - x$ rows changes.

Chapter 4

Analysis

Herein, I will crunch the numbers to illustrate exactly the degree to which my methods (outlined in chapter 3) pruned the the search-space to put an otherwise intractable problem within reach.

4.1 Raw Space

First, we need to understand how big this space is. The specification for RAID-6 leaves only so much liberty to the designer: the first super-row of the CDM *must* consist only of I_w identity matrices. Thus, if you were to just remove the second super-row of the CDM, you'd be left with one that works for RAID-4/5 (simple XOR parity). The second super-row, however, can be specified in any way that yields a MDS code.

According to Blaum and Roth [1] a minimal-density, MDS, RAID-6 bit-matrix consists of $2kw + w - 1$ 1s. Since I am trying to find all possible solutions for $w = 9$, and $k \leq w$, I set $k = w$. Then, the minimal number of 1s is $2w^2 + w - 1$. Also, this formula includes the super-row corresponding to the P disk, i.e. w^2 1s are not available to our search. Thus, we arrive at $w^2 + w - 1$ 1s to choose from for our last super-row. This super-row consists of $k = w$ $w \times w$ bit-matrices, so it has w^3 bits, and so our total, unrefined search-space is:

$$\begin{aligned} \binom{w^3}{w^2 + w - 1} &= \binom{729}{89} \\ &\approx 1.3538 \cdot 10^{116} \end{aligned} \tag{4.1}$$

4.2 General Structure

Yet, most of these will not yield an erasure code of any sort, let alone an MDS one. As previously pointed out, any MDS code will be equivalent to one with panel $X_0 = I_w$, and the remaining panels will be some permutation matrix plus an extra 1.

There are $w!$ ways to choose a permutation matrix, and there will be $w^2 - w = 72$ ways to choose the extra 1 in said matrix. Therefore, for $w = 9$, with $w - 1 = 8$ panels to fill, we get:

$$(9! \cdot 72)^8 \approx 2.1715 \cdot 10^{59} \tag{4.2}$$

But really, since two panels sharing the same permutation matrix would immediately violate the second MDS condition, that being that the sum of any two panels is invertible, we can reduce this to a permutation:

$$\binom{9! \cdot 72}{8} \approx 5.3857 \cdot 10^{54} \tag{4.3}$$

5 orders of magnitude still are not a great reduction on this scale, but every little bit helps.

4.3 Extra-bits' Rows

This one is simple: since we can set the row for the extra 1, we do not have to search for it. This reduces our search on the extra 1 from $w^2 - w$ to $w - 1$, and we get:

$$\binom{9! \cdot 8}{8} \approx 1.2511 \cdot 10^{47} \tag{4.4}$$

This reduced our search space by a factor of $4.3047 \cdot 10^7$.

4.4 Extra-bits' Columns

This is the final well-defined, generic pruning technique (the rest will be heuristic in nature, and thus not easily analyzed mathematically). This is again similar to the preceeding section, but applies to the columns, once the row and permutation matrix are chosen. In fact, this choice can be made after and somewhat independent of those.

The observation is that, there are not actually $w - 1$ choices for the column of the extra 1 in every panel; it is conditional on the other panels' extra-1s' choices. Therefore, at worst, there are $w - 1$ choices for the column of the extra 1 in panel X_1, $w - 2$ choices for X_2, etc. With this in mind, we get:

$$\binom{9!}{8} \cdot (8!) \approx 3.0066 \cdot 10^{44} \tag{4.5}$$

Table 4.1: Number of times recusion was halted based on the partial-MDS verification

Panel Index	$w = 6$	$w = 7$
1	2,970	25,782
2	1,462,752	97,584,480
3	56,725,720	24,769,800,832
4	125,696,422	377,356,040,546
5	10,183,040	213,912,399,554
6	–	3,267,451,101

4.5 SMP

My methods aim to utilize the SMP capabilities of modern CPUs, as well as many CPUs or physical computers, to break the search into many small chunks, each handled in its own, independent thread or process. While only on a large scale will this logically break the search space down into significantly smaller chunks (say on order of 10^4 or more), it can still net real savings in the amount of time needed to process the space.

4.6 Partial Verification

This is, by its nature, a heuristic pruning technique. Therefore, without more substantial knowledge on the theory of the bit-matrices being discovered, it is impossible to define symbolically the degree to which this technique prunes the space.

Yet, as part of my code, I keep counters related to how often this technique halts further recursion, as long as it does not lead to further pruning as outlined in section 3.3.4. Below, table 4.1 shows some of the numbers for $w = 6$ and $w = 7$.

4.7 MDS Back-propagation

Interestingly, after applying this optimization to my project, I was able to see empirically that the enumeration was actually taking *longer* than without it.

Yet, there was still some value in a partial application of this optimization. Testing the optimization one panel at a time, I was able to determine that the full application of this optimization was primarily beneficial when applied to the first panel. Specifically, the extra overhead of skipping over permutation matrices (each involving a `memcpy(3)`, arithmetic, and an additional function call) was overcome by the speed gains of skipping over permutation matrices, if only on the first panel.

For the remainder of the panels, the overhead of skipping column-enumeration of the extra-1's was negligible and still netted significant improvements in running time; approximately a 15% speed improvement for $w = 6$, and 22% for $w = 7$. Table 4.2 illustrates several

Table 4.2: Comparing the run-times of several pruning techniques in seconds

Pruning Technique	$w = 6$	$w = 7$
MDS-partial	82	$2.92 \cdot 10^5$
Full Back-propagation	100	$3.11 \cdot 10^5$
Optimal Back-propagation	70	$2.29 \cdot 10^5$

of these differences.

Chapter 5

Conclusions

5.1 Results

As of this paper, the highest word size for which my work has produced complete results is 7. This run produced 1680 solutions, and ran approximately $2.29 \cdot 10^5$ seconds total across 8 cores. This means that with full utilization of the 8 cores, this run takes a lower-bound of 7.95 hours. This ran on an Intel®Core$^{\text{TM}}$i7-3770 CPU @ 3.40GHz (4 real cores, 4 logical cores).

For $w = 9$ the time-to-solution is too long to be included in this paper even with a framework to distribute the task across many nodes. In the coming months, a separate paper will outline and publish the final results of this work.

5.2 Future Work

Over the months spent working on this project, I have had time to ruminate on several aspects of the problem. This section briefly outlines some of the problems left, and how we might address them.

5.2.1 De-duplication

Once a complete set of solutions to my work is produced, there will undoubtably be duplicates with respect to the three equivalence operations mentioned in 3.1.2 (swapping rows/columns among the panels, and swapping panels themselves). Before reviewing the solutions it will be necessary to de-duplicate them for more efficient study.

Table 5.1: The search-space sizes, after some optimization, for various word-sizes

word-size	Search-space
4	$1.214 \cdot 10^{4}$
5	$1.971 \cdot 10^{8}$
6	$1.908 \cdot 10^{14}$
7	$1.634 \cdot 10^{22}$
8	$1.731 \cdot 10^{32}$
9	$3.007 \cdot 10^{44}$
10	$1.091 \cdot 10^{59}$
11	$1.027 \cdot 10^{76}$
12	$3.046 \cdot 10^{95}$
13	$3.399 \cdot 10^{117}$
14	$1.680 \cdot 10^{142}$

5.2.2 $w = 14$

This work attempted to fill in the next-smallest gap in the coding theory for minimal-density, RAID-6 matrices, however there are others: $w = 14$ is the next such gap. Yet, the search space for this word-size is enormous. Using the same math as in equation 4.5 (with all the corresponding optimizations up to that point), table 5.1 shows the search-space sizes for various values of w. Given the sizes, it may be a fruitless effort to attempt to solve this same problem for $w = 14$.

5.2.3 Unique Sets

In light of that, solving $w = 14$ may simply require more theory. If, however, a search needs to be done, then it may be worth asking ourselves the question: Do we need to know *every* solution? Perhaps a partial search of the space would be sufficient information to begin probing at what other solutions might be.

In this case, a technique, which I am calling the *unique-sets property*, might be a worthwhile endeavor, since it can drastically reduce the size of the space being searched. The idea of imposing this property is that I do not want to consider the order of the final set of permutation matrices selected for a RAID-6 candidate. In other words, if we represent permutation matrices by π_i, and assuming only 4 panels, we only want to generate a RAID-6 code with the set $\{\pi_i, \pi_j, \pi_k, \pi_l\}$ once, regardless of the order which the panels use them.

By imposing an arbitrary order on the enumeration of the permutation matrices, and relying on it for the panel-by-panel enumeration, we will actually skip over valid solutions, however I saw a reduction in running-time for $w = 7$ (single-threaded) to approximately 30 minutes without the back-propagation optimization; much better than the 9 hours or so it took find all solutions (with 9 threads and optimal back-propagation).

Bibliography

[1] Blaum, Mario and Ron M. Roth. *On lowest density MDS codes.* IEEE Transactions on Information Theory, January 1999.

[2] Plank, James S. *A New MDS Erasure Code for RAID-6.* Technical Report UT-CS-07-602. Department of Electrical Engineering and Computer Science, University of Tennessee, September 2007.

[3] Plank, James S. *A New Minimum Density RAID-6 Code with a Word Size of Eight.* 7th IEEE Symposium on Network Computing Applications, Cambridge, MA, July 2008.

[4] Plank, James S. *The* 48 *Sets of Minimal Density MDS RAID-6 Matrices for a Word Size of Eight.* Technical Report UT-CS-08-611. Department of Electrical Engineering and Computer Science, University of Tennessee, March 2008.

[5] Plank, J. S. and A. L. Buchsbaum and B. T. Vander Zanden. *Minimum Density RAID-6 Codes.* ACM Transactions on Storage, 6:4, May 2011.

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

Vita

Bryan Burke is a student and employee of the University of Tennessee. He was born in Nashville, TN to Joanne Burke. As a youngster, he attended several K-6 schools, the most interesting and formative of which were *Country Day Montessori School* and *Buena Vista Jones Paideia Magnet School*, both of which were in the greater Nashville area. It was in these schools and years that he developed a passion for education in general, and mathematics in particular.

Going forward, Bryan was selected to attend the *Martin Luther King Jr. Magnet High School for Science and Engineering.* Here, he learned of the wonderful world of calculus while attending local and regional math competitions. In addition, in his spare time, he began to dabble in computer programming. Before the end, he was teaching not only himself, but others to write assembly code for *Texas Instruments' TI-83+* calculators.

Bryan attended the University of Tennessee in part because of its Computer Science program's reputation as a theory- or research-focused one. He completed the Bachelor of Science degree with majors in Mathematics and Computer Science, as well as a minor in Latin.

As an undergraduate, Bryan acquired a job working for the math department as a tutor in its *Math Tutorial Center*, and in the department of Computer Science – eventually Electrical Engineering and Computer Science – first as a teaching assistant, then as a member of the departmental IT Staff. As a graduate student in this department, he eventually took a job working for the IT Staff full-time, and continues working there today.