3-Ranks for strongly regular graphs

Andrew Novocin
University of Delaware
Newark, DE, USA
andynovo@udel.edu

David Saunders
University of Delaware
Newark, DE, USA
saunders@udel.edu

Alexander Stachnik
University of Delaware
Newark, DE, USA
stachnik@udel.edu

Bryan Youse
University of Delaware
Newark, DE, USA
bryouse@udel.edu

ABSTRACT

In the study of strongly regular graphs, ranks of adjacency matrices (Laplacians actually) are extensively used to demonstrate inequivalence of graphs. Constructions have been given for several families of graphs. Formulas for the ranks in these families are an important tool for understanding their properties.

The first and computational challenge is to compute rank modulo 3 of some very large matrices. To our advantage is that the ranks are expected to be relatively small. Typically in these families, the matrix dimension is 3^k while the rank modulo 3 is in the vicinity of 2^k.

Here we discuss a high performance parallel solution to the problem. It involves parallelism at three levels: word-level vectorization of field elements, shared-memory multicore, and a multi-node distributed memory and file-system modulated level. The implementation has been applied to the case k = 16, wherein the matrix contains approximately 1.85 peta-entries.

The second challenge is to discern a formula for the sequence of ranks in a given graph family.. Our computations provide further evidence for an existing conjecture concerning the Dickson family of strongly regular graphs and provide a starting point towards finding a formula for the Ding-Yuan and Cohen-Ganley families of graphs.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*algebraic algorithms*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph Algorithms*

Keywords

strongly regular graph, 3-rank, Paley-type difference set, skew Hadamard difference set

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3599-7/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2790282.2790295

1. INTRODUCTION

The goal of this work is to compute the ranks in sequences of adjacency matrices of strongly regular graphs that are defined by skew Hadamard or Paley type difference sets, specifically the graphs produced by a construction based on Dickson semifields the construction of Cohen and Ganley, and the construction of Ding and Yuan [8]. A summary of these and other constructions is to be found in Xiang et al [20]. In these families, strongly regular graphs are defined for each odd prime p and exponent e, the graph having p^e vertices. The graphs are defined in terms of difference sets, either pseudo-Paley or Hadamard. In this paper we concentrate on the case $p = 3$.

The Dickson family is defined for each even exponent e, in terms of a Paley-type difference set over a semifield, and the Ding-Yuan family for each odd e, skew Hadamard difference set over GF(3^e). Previously ranks have been computed for the Dickson sequence for even e up to $e = 14$ [20, 14, 17], and for Ding-Yuan's sequence, odd e up to $e = 11$. Here we extend the known Dickson ranks to $e = 16$, providing further evidence for a conjectured recurrence relation in [17]. We also report rank computations on the Ding-Yuan family up to $e = 15$ and discuss the prospects for computations with larger cases.

These matrices are dense, with about half the entries non zero. The method used is a multi-level parallel implementation of the algorithm of [14]. It has two phases, (1) a projection of the given $n \times n$ matrix to a much smaller $m \times m$ block ($M = XAY$, where X is $m \times n$ and Y is $n \times m$) and (2) Gaussian elimination on the $m \times m$ block to determine the rank. Letting $A_{i,j}$ denote the (i, j)-th $m \times m$ block of A and blocking X, Y conformally, we have the formula

$$M = \sum_j M_j Y_j, \text{ where } M_j = \sum_i X_i A_{i,j}.$$

In our application, $n = 3^e$, while the rank (and thus m) is within a factor of 2 or 3 of 2^e. The projection phase has cost O(n^2) as described in section 2, while the elimination phase detailed in section 5 costs O(m^3). Thus, when e is increased by 1, the projection cost grows by a factor of 9 and the elimination phase grows by a factor of approximately 8. Projection cost is dominant and has received our greatest implementation effort.

Algorithm 1 Small rank via projection

```
for j from 1 to $n/m$ do
    for i from 1 to $n/m$ do                          ▷ Phase 1a
        Compute the contribution of $A_{i,j}$ to $M_j$.
    end for
    Compute the contribution of $M_j$ to $M$          ▷ Phase 1b
end for
                                                      ▷ Phase 2
Using Gaussian elimination, compute rank($M$).
Verify rank with probabilistic certificate.
```

The projection phase 1 uses all three levels of parallelism, word-level vectorization (bit-slicing), shared memory multi-processing (openMP), and multi-node distributed memory computation (a custom design). The word level vectorization is the "bit-sliced" scheme described in [4]. A brief example of this is illustrated in figure 1. A vector of 64 GF(3) elements are stored in a pair of 64 bit words, each element occupying one bit of each word. Bit logic operations on the words are used for the parallel execution of arithmetic on these packed vectors of elements.), Due to the compression inherent in bit-slicing, a single $m \times m$ block fits in main memory of a compute node. Thus it has been sufficient for each column block computation, phase 1a, and for the elimination, phase 2, to use just the first two levels and compute the rank on one shared memory multi-core node. In addition the activities in phases 1a and 1b must be suitably coordinated so that an excess of large intermediate results, the M_j, does not accumulate, exceeding memory and disk resources.

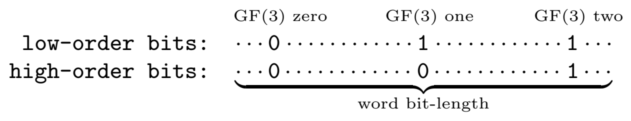
GF(3) zero
GF(3) one
GF(3) two
low-order bits: ···0············1············1···
high-order bits: ···0············0············1···
word bit-length

Figure 1: Conceptual overview of bit-slicing GF(3) values. Values occupy two bits, one in each of two contiguous machine words. We call this pair of words a *sliced unit.*

In section 2 we discuss the formula for the projection and the variants of it that we've had to adopt. Section 3 concerns the use of shared memory and word level parallelism in the projection phase 1a. In section 4 the primary distributed algorithm is explained. It manages processes carrying out instances of phase 1a and consuming and using their results for phase 1b. The method is organized around the files read and written by the processes. Section 5 concerns the use of shared memory and word level parallelism in phase 2, the rank computation via Gaussian elimination on the projected block. Finally section 6 reports our analysis of these ranks, i.e., the progress these computations have allowed us to make toward producing formulas for the ranks by graph family. We summarize in a concluding section 7.

2. PROJECTION

The projection scheme for rank computation is fully explained and analyzed in [14]. We review the basic features here in order to have the framework for discussion of the implementation strategy. Let $A \in \mathrm{GF}(3)^{n \times n}$ be the matrix whose rank is to be computed. Let b be chosen with the expectation that $b > \mathrm{rank}(A)$. Let c be chosen to set the strength of the probabilistic validation of the rank. Probability of error is bounded by 3^{-c}. Finally, let $m = b + c$. The projection we use computes $m \times m$ matrix $M = XAY$, where X and Y are $m \times n$ and $n \times m$, respectively.

The construction was oversimplified in the introduction for brevity. (There $c = 0$.) The additional c rows and columns in the constructed block are random linear combinations of the rows and columns of A, computed for the sake of probabilistic validation of the rank. We choose $c = 64$, not so much because we want the extreme assurance of success as because it is no more expensive than a smaller value in view of our sliced word vectorization discussed in section 3.

The decomposition is best understood in block form.

$$M = \sum_{j=0}^{n/b} \sum_{i=0}^{n/b} X_i A_{i,j} Y_j,$$

where M is $m \times m$ and $A_{i,j}$ are blocks of a subdivision of A into $b \times b$ blocks. The projectors X, Y are decomposed into nearly square blocks, the X_i being $b + c \times b$, and Y_j being $b \times b + c$.

Further, the projectors have the form

$$X_i = \begin{pmatrix} P_i \\ U_i \end{pmatrix} \text{ and } Y_j = \begin{pmatrix} Q_j & V_j \end{pmatrix}.$$

Thus M has leading $b \times b$ block $B = \sum_{i \in 0..\frac{n}{b}} \sum_{j \in 0..\frac{n}{b}} P_i A_{i,j} Q_j$. Here $P_i \in \mathrm{GF}(3)^{b \times b}, Q_j \in \mathrm{GF}(3)^{b \times b}$ are (sparse or structured) preconditioners, while $U_i \in \mathrm{GF}(3)^{c \times b}, V_j \in \mathrm{GF}(3)^{b \times c}$ are random. The point of the preconditioners is to make it likely that $\mathrm{rank}(B) = \min(\mathrm{rank}(A), b)$. The point of the dense random U_i, V_j is to sample the row and column spaces to provide certification of the rank. If the rank of the leading submatrix B of M is the same as the full rank of M when including these random samples, then the probability that this fails to be the rank of A is bounded by 3^{-64}.

For our largest rank computation to date, $e = 16$, we have $n = 3^{16} = 43046721$ and we used $b = 2^{17} + 2^{14} = 147456$, which turns out to be slightly larger than the rank. Observe that we do not have sufficient memory to store an $n \times n$ matrix nor even an $n \times m$ matrix. There are 292 blocks per row/column and 85264 blocks overall. It is crucial to work finally with only a few $m \times m$ blocks (each is 5GB) on each node at any given time.

To detect potential differences between the ranks of M and its submatrix B, we compute the rank of M with Gaussian elimination, checking if any row or column outside of B contributes to the rank.

In general, to ensure success, the blocks of P, Q have to be butterfly matrixes or similar preconditioners [14, 17]. However, in view of the random certification, we may heuristically choose much simpler preconditioners. Most notably we may choose P_i and Q_j as $b \times b$ random permutations or even the identity. We found that it sufficed for success on the ranks reported in this paper to let $P_i = I_b$ and have Q_j be a small amount of permutation of columns done at the granularity of permuting sliced units (blocks of 64 columns). The cost in the largest case $e = 16$ is about 3 seconds per block permuted. For simplicity in the sequel we will ignore the P_i and Q_j and refer to the leading submatrix B as a summation of the blocks $A_{i,j}$ of A.

We organized the computation of the block accumulation in two stages: first compute column accumulations using the row projectors and then combine the results using the column projectors.

$$M_j = \begin{pmatrix} B_j \\ R_j \end{pmatrix} := \sum_{i \in 0..\frac{n}{b}} \begin{pmatrix} I_b \\ U_i \end{pmatrix} A_{i,j}, \qquad (1)$$

$$\text{so that} \begin{pmatrix} B_j = \sum A_{i,j} \\ R_j = \sum U_i A_{i,j} \end{pmatrix}$$

and

$$M = \begin{pmatrix} B & S \\ R & T \end{pmatrix} := \sum_{j \in 0..\frac{n}{b}} M_j \begin{pmatrix} I_b & V_j \end{pmatrix} \qquad (2)$$

$$\text{so that} \begin{pmatrix} B = \sum\sum A_{i,j} & S = \sum\sum A_{i,j} V_j \\ R = \sum\sum U_i A_{i,j} & T = \sum\sum U_i A_{i,j} V_j \end{pmatrix}$$

This organization reduces the cost of applying the column projectors. The choice of column accumulation first rather than row accumulation is due to a greater efficiency of $R_i A_{i,j}$ over $A_{i,j} S_j$ computation in view of our word level row vectorization.

In the implementation presented here, the column blocks are computed using the vectorization and openMP parallelizations only, while their accumulation into a final block exploits multi-node distributed computation.

An earlier version of the implementation was motivated by a desire to saturate network connections for testing purposes. In that case we worked at a finer granularity. Producers on a collection of machines in one building computed $X_i A_{i,j} Y_j$ and consumers in another building combined these blocks to produce M. The coordination between producers and consumers exploits the file system for the consumers, which proved to be quite a strain with this large number of producers. However, the coarser granularity of producing and consuming only column blocks at the distributed memory level is more efficient especially of file system resources, while providing sufficiently many tasks for the available nodes in the cluster we used.

3. PHASE 1A: COLUMN BLOCK PROJECTION

The row projection of column blocks benefits from two forms of parallelism, the word-level vectorization and shared memory multi-threading.

For our word-level vectorization of arithmetic over GF(3) we use bit-slicing [4, 22], A row vector of 64 elements of GF(3) is stored in two 64 bit words, using one bit from each word per element. We call this pair of words a *sliced-unit.* Arithmetic is performed using bit operations, five of which are required for addition of 64 element vectors, while scalar multiplication is particularly easy, using a single xor for the case of multiplication by -1 modulo 3. We refer the reader to [4, 22] for details.

Our test machine is the Chimera cluster at the University of Delaware. The compute nodes of this energy-efficient machine contain 4 AMD Opteron 6164HE 12-core 1.7GHz CPUs, for a total of 48 cores per node. Chimera has 65 nodes, of which we used at most 18 at any one time. Users are allocated resources at the node level (we have exclusive use of our nodes). Out of the 18 nodes, eight were devoted to generating the M_j (phase 1a). Each of these nodes would check a queue of work for the first unbuilt M_j, and claim responsibility for projecting the j^{th} column of blocks to construct that M_j.

On this machine we found our arithmetic operations in matrices and vectors over GF(3) to be about 15 times faster using bit-slicing than using one field element per word with delayed modular reduction (the LinBox standard representation).

The rest of the algorithm is organized to make convenient use of the sliced units. Thus in the projection we use a block size which is a multiple of 64, and in the rank computation we emphasize row operations so that vector axpy is exploiting the sliced unit operations. Also a 64 column wide random block is used in the column projection (the M_i) so that it is one sliced unit wide. Incidentally this also provides a very good probability of correct rank, namely $1 - 1/3^{64}$ [14].

To take advantage of thread level parallelism in the computation of a column block, we have made straightforward use of openMP for blocking the the computation $B_j = B_j + A_{i,j}$. Here B_j is the leading b rows of M_j. We did not take the trouble to parallelize the production of the last 64 rows, $X_i A_{i,j}$, The primary cost here is the generation of entries of the block $A_{i,j}$ and the secondary, much smaller, cost is the addition of the block to M_j. and the $X_i A_{i,j}$ computation. The generation is with respect to a stored difference set, represented by an array D, such that $D[i-j]$ indicates the i, j entry of A. This array is determined by an underlying semifield, whose definition defines the family of matrices. Once the semifield is determined, we have $D[k] = 1$ if the k-th element of the semifield (of order 3^e) is a quadratic residue, while $D[k] = 0$ if not. Zero is a special case, $D[0] = 2$. The process is complicated by the fact that the difference indicated is semifield subtraction. To be correct we should say that the i, j entry of A is $A_{i,j} = D[\phi^{-1}(\phi(i) - \phi(j))]$, where $\phi()$ is a bijection between the semifield and the index range $[0, \ldots, 3^e - 1]$ for semifield of cardinality 3^e. For a given $A_{i,j}$ entry, i and j must be mapped to the corresponding semifield element, the two elements subtracted, and the resulting element mapped back to the index used to access D. The mappings ϕ and ϕ^{-1} are of nontrivial cost to compute. By contrast adding a block into the sum is being done multiple elements per clock cycle in view of the word level sliced unit parallelism. For example, in the Dickson matrix of dimension 3^{16}, with $b = 2^{17} + 2^{14}$, each thread spends 6.85s generating a $b \times b$ block, $A_{i,j}$, and .25s adding it into the sum B_j. Parallel efficiency is very high as shown in Figure 3.

4. PHASE 1B: COARSEST GRAIN: ACCUMULATION OF COLUMN BLOCKS

As mentioned, an earlier parallel decomposition involved writing each $A_{i,j}$ block to an intermediate file upon computing it. The considerable computational problem involved the number of blocks, and thus the number of 5GB files that were needed using this decomposition. Imagine a grid formed by logically laying these $b \times b$ blocks out on a plane. Such a grid must of course, provide full coverage over A. Therefore the number of blocks in each dimension of this grid is $\lceil 3^{16}/(2^{17} + 2^{14}) \rceil = 292$. A is composed of around

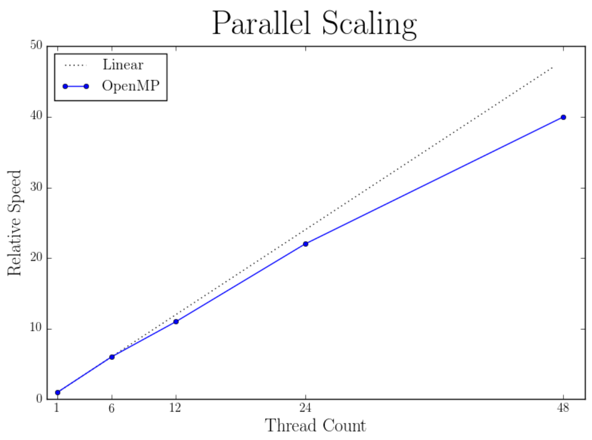

Figure 2: Phase 1a multi-threaded parallel scalability for building a column block, `buildColBlock`. Speedup relative to the single-threaded application is charted with the linear speedup line for comparison. The multi-core scalability of this algorithm is quite favorable, further illustrating the embarrassingly parallel, CPU-bound nature of generating matrix entries.

292^2, or eighty-five thousand such blocks. Continuing to speak approximately, even with bit-sliced compression we are still occupying 425 terabytes. This amount of data is obviously impractical to work with concurrently, as was done for the smaller matrices in the sequence. At the current time, this is too much data for any of our systems to store simultaneously, even out-of core.

Since we cannot use or store all of A simultaneously, it is clear that the solution is to work individually with blocks of A. The vital step from a storage perspective is in discarding the blocks after accepting their contribution to M. Because A is defined by a formula, the $A_{i,j}$ blocks of A can be conveniently created by independent processes running concurrently. This course-grained, "outer" parallelism would be characterized by Dijkstra as "loosely connected processes" [7]. That is to say processes that, while related to one another, can each operate entirely autonomously with no restrictions on the relative speeds the processes. This flow of work lends itself to a producer/consumer relationship between groups of processes. These processes can, and in this case do, individually employ a second tier of parallelism: a fine-grained multi-threading to get the most out of multi-threading on each node, a shared memory multi-core. The producers perform the phase 1a computation discussed in the previous section. The consumers handle the output from the producers in phase 1b.

The phase 1b consumption stage performs the summation and column space random sampling on the column blocks produced by phase 1a processes. This amounts to combining the column blocks, M_j, into our final block M. Just as the phase 1a process can delete the block A_{ij} after adding into B_j and contributing to R_j, so too can these consumers delete each M_j after it has been been incorporated into M. In this case it is a matter of removing a file created by a phase 1a column block producer. The M_j, if all stored, would occupy over 1.4 TB. While we could handle this particular amount of data, the code is robust enough to handle deleting the M_j in a timely way. This would be vital in computing, say, the order 3^{18} Dickson rank.

Phase 1a and phase 1b are coordinated by a set of Python scripts. These scripts use remote procedure calls for communication, which is conveniently abstracted to shared objects using the Python Remote Objects (PyRO) module. These objects manage two work queues, one indicating which M_j need to be built, and another indicating which M_j have been consumed. This approach allows for a modular design regarding producers and consumers, allowing heterogeneous parallelism. Any machine can contribute to either production or consumption by requesting work from the queues. Computing resources can be added or removed at any time during the course of the algorithm. The script structure, along with the standalone programs that perform the matrix generation and arithmetic, are packaged together in a publicly available repository [21].

In order to discuss a few details, we next give the code for a step of column block accumulation into a final block.

```
void addColBlock(int b, int c, int j){
    int m = b + c;

    // The final block
    Matrix M;
   M.init(m,m);

    // M: get current state
   M.readBinaryFile();

    // M_j: j-th column block computed
    Matrix M_j;
    M_j.init(m,b);
    M_j.readBinaryFile(j);

    // V_j: to make colspace sample
    Matrix V_j;
    V_j.init(b,c);
    random(V_j);

    // Incorporation of M_j, in two parts
    Matrix M_Sum_k, M_CertCols;
    // M_Sum is (B | R)
   M_Sum.submatrix(M, 0, 0, m, b);
    // M_CertCols is (S | T)
    M_CertCols.submatrix(M, 0, b, m, c);
    addin(M_Sum, M_j);
    axpyin(M_CertCols, M_j, V_j);

    // M: save new state
   M.writeBinaryFile();
}
```

As in phase 1a, addition of a block involves an addition of large blocks and a multiplication (axpyin) which in this case adds several random linear combinations of the given (group j) columns to the column space sample.

One difference from phase 1a is that $m \times b$ blocks are being combined into an $m \times m$ block, whereas in the earlier phase it was $b \times b$ blocks to form the $m \times b$ inputs here. This is a small difference, since $m = b + c$, and c can be very small. The c is chosen for random verification and the probability of error decreases exponentially with c [14]. We use $c = 64$ which fits well with our row vector slicing.

However, a key difference from phase 1a is that our call to axpyin() in this stage takes longer than in the previous phase. The essence of axpyin is matrix multiplication. The gain in multiplication cost due to row bit-slicing is proportional to the the number of columns in the right multiplicand. Here a $m \times b$ is multiplied by a $b \times c$. In phase 1a a $c \times b$ is multiplied by a $b \times b$. As before, both multiplicands are compressed with bit-slicing. The number of word-vector operations is approximately the same, b^2 in phase 1a and mb in phase 1b. However, in this case the left multiplicand, whose entries are extracted one by one, is considerably larger. Thus we have greater overhead due to the cost of extracting bit-sliced entries. Fortunately, this kernel need only be performed once per column block ($\frac{n}{b}$ times in total). The IO calls in the subroutine readBinaryFile() and writeBinaryFile() account for roughly one-sixth of the running time.

In this computational layout the consumer is written to take advantage of multiple workers, each of which could be consuming a column block simultaneously. Consuming a column block means writing to the block sum. Without taking care, workers could overwrite each other's partial work, leaving the master block in an unknown (but almost certainly meaningless) state. To solve this problem in practice, each worker creates its own master block to write to. This distinction involves the call to addColBlock taking as input the worker identification number. Upon the consumpion stage's end, the final block M can be created by a simple summing of these intermediary column-block sums produced by the workers. This summing is a quite fast operation with bit-sliced matrices, and has negligible effect on overall running time.

5. PHASE 2: GAUSSIAN ELIMINATION FOR RANK

For the elimination phase, we perform an echelon form computation on the final projected block, taking advantage of only the word level and shared memory multi-core parallelism of the row operations. Each time a pivot is found, elimination (by way of row vector axpy — exploiting the word level vectorization) must be performed on all subsequent rows. This step can be performed entirely independently on each row, yet another embarrassingly parallel routine. Therefore these rows can be evenly pooled into t groups, where t is the number of OpenMP threads available. Each thread then eliminates the rows for which it is responsible. The resulting speedup was sufficient to make the elimination phase not a computational bottleneck. For example, with the Dickson matrix at size 3^{14} the runtime was 0.13 hours, a tenth of the cost of the block projection, 1.3 hours. This elimination time is also 10 times faster than a 2009 [17] echelon form computation on this block. These times are not directly comparable. The 2009 non-parallel computation used a faster processor but less efficient word level vectorization (packing rather than slicing).

Although we haven't precisely quantified it, clearly the parallel efficiency is not so high in the elimination phase as in the projection phase.

We did not (yet) pursue further tuning for performance on this segment. However, for attempting still larger instances of the problem (as we intend to do) it will be necessary to tune the elimination phase. As an indication of this we note that for our largest computation to date, $n = 3^{16}$, the elimination phase took 8.7 hours, as shown in table 1. The elimination cost ratio, $8.7/0.13 \approx 67$, is very close to the expected factor of 64 slowdown as you go from $n = 3^{14}$ to $n = 3^{16}$ The additional factor of 64 to compute at $n = 3^{18}$ would bring us to about 560 hours (more than 3 weeks). Faster elimination algorithms will help as will introduction of multi-node computation.

Table 1: Running time for computing the ranks of Dickson adjacency matrices: fully parallelized implementation (2015) versus sequential implementation (2009). The algorithm running time is broken into two components: the 'b' columns denote time to project the matrix into a rank-sized block, and the 'r' columns denote the time taken to perform elimination, thus computing rank. The parallel build step used 8 compute nodes acting as producers and 10 consumers. The time units are 's' for seconds and 'h' for hours.

Dickson SRG						
e	dimension	Rank	2009b	2009r	2015b	2015r
6	531,441	7283	1.2h	80s	.05h	21s
7	4,782,969	32064	96.4h	1.2h	1.3h	.13h
8	43,046,721	141168	-	-	27.9h	8.71h

Regarding faster elimination, note that block decomposition is not done (thus far). It would be beneficial to do so, using for example the Strassen-Winograd algorithm [19] and/or the 4 Russians method [2]. M4RI [1] is an effective implementation of this approach over GF(2). Preliminary work of Lambert [11] suggests good gains from this for our setting (over GF(3)).

Regarding use of the third level of parallelism, observe that the time of the projection for $n = 3^{16}$ was 28 hours, a factor of only $28/1.3 \approx 22$ over the $n = 3^{14}$ case. The arithmetic cost here actually increases by a factor of 81. The smaller increase in time is explained by the introduction of the distributed memory third level of parallelism to this phase. In contemplating the 3^{18} we must address an 80-fold increase in work along with a 4-fold increase in memory needed for each block. Handling several 20GB blocks at a time will strain memory resources on harware we are likely to use. However, it is straightforward further block decompose our blocks as they flow through the computation. In the end we need work with only one $m \times m$ block, our final projected block. Such decomposition, together with speedup of the Gaussian elimination and enough time and processors should make the $n = 3^{18}$ rank computation feasible.

6. RANK SEQUENCE RECURRENCES

In the situation we are exploring, we calculate ranks of families of matrices. The dimensions and rank computation costs increase exponentially in each family. Given that computing larger and larger terms is ultimately infeasible we would like to find patterns in the computed ranks which might be provable mathematically.

So how does one hunt for patterns in sequences of integers?

We have reason to suspect a linear recurrence for cases such as the Cohen-Ganley and Ding-Yuan families. In a number of other cases of families based on difference sets, the 3-ranks were each given by linear recurrences. For example, Paley by $x - 4$ (proven) [5], Pseudo-Paley by $x^4 - 10x^2 + 9$ (proven) [20], and Dickson by $x^3 - 4x^2 - 2x + 1$ (conjectured) [14].

The first step should be to run the Berlekamp-Massey algorithm [13, 3] on an even number of terms. From $2n$ terms of a sequence one will unequivocally find a unique monic polynomial of degree n that generates those terms. We may get a best guess polynomial or a generator that can be ruled out due to rational coefficients, failure to generate the $2n+1^{st}$ term, or some other consideration.

The next step might be to search the OEIS [18] for known sequences or nearby sequences. This is not a traditional algorithm but when the next term is very expensive to compute this trick has unlocked more than a few patterns. If this still does not yield results there is a result [12] which can compare your sequence with compositions of sequences in the OEIS and Sloane database. In the Ding-Yuan and Cohen-Ganley case none of these approaches yielded any satisfactory results.

6.1 Lattice-based recurrence-finding recipe

Now we give our strategy for coping with linear recurrences when there are not enough terms to uniquely identify a generator using Berlekamp-Massey alone. In our case there are three simple observations which can help significantly:

1. Ranks are integers.
2. Matrix dimension bounds the rank.
3. Rank is non-negative.

Observation 1 has two impacts. One is that for each possible value of d, the degree of the generator, there will be a lattice of integer-coefficient polynomial generators which must contain a correct generator if one exists. The second impact is that we need only consider monic generators.

Observation 3 allows us to rule out any possible generators which ever produce a negative value.

Observation 2 along with the following proposition allows us to rule out, as generators, any polynomials with roots too large.

PROPOSITION 6.1. *If $f \in \mathbb{Z}[x]$ generates the rank sequence (r_k) of the matrix sequence of dimension $n_k = c3^{2k}$ for some constant c, then $|\lambda| \leq 9$, for any root λ of f.*

PROOF. The generating function of a homogeneous linear recurrent sequence is rational with denominator the reverse polynomial of the characteristic polynomial. Let R be the maximum of $|\lambda|$ for roots λ of the characteristic polynomial. The Exponential Growth Formula [Theorem IV.7 [9]] gives $\limsup r_k^{1/k} = R$. Since, by observation 2, $r_k \leq c9^k$ it follows that $|\lambda| \leq R \leq 9^k$. $\square$

In the case when one strongly suspects a linear recurrence and additional terms are not readily obtained, we suggest the following computer-aided approach exploiting the above observations.

1. Find the lattice of possible generators for the degree you wish to explore.
2. Use LLL to create a reduced basis of this lattice.
3. Determine if there is a finite list of polynomials with small roots centered at the origin of this lattice.
4. Filter this list to be only those polynomials that extend the sequence by generating non-negative terms.

6.1.1 *Details of this approach*

Let d be a candidate degree for the sequence generator and suppose $d+k$ sequence elements $(a_0, \ldots, a_{d+k-1})$ are known. In the cases under consideration, $d+k$ is at most 8. Since the generator is known to be monic, the possible generators must solve the linear equation $M\vec{f} = \vec{b}$ where M is a $k \times d$ Hankel matrix given by $M_{i,j} = a_{i+j-2}$, $\vec{b}$ is given by $b_i = a_{i+d-1}$ and $\vec{f}$ is the vector of generator coefficients f_i. Since the solution must be integer valued we have a Diophantine linear system. For very efficient Diophantine solvers see [15] for dense systems and [10] for sparse. For our small systems we used the method of Chou and Collins [6] which proceeds via a direct reduction to Smith form. This gives us a diophantine solution and nullspace basis which we may reduce with LLL. First compute the Smith normal form of M as $U^{-1}DV^{-1}$ where U and V are unimodular square matrices and D is diagonal. Then $D\vec{x} = U\vec{b}$ can easily be solved and $\vec{f} = V\vec{x}$. At least $d-k$ components of $\vec{x}$ are free to vary and these correspond to the degrees of freedom of the linear system. Let $R = \left(\alpha\vec{f_0}, V_{k+1}, \ldots, V_d\right)$ be the matrix formed by adjoining any particular solution $\vec{f_0}$ of this system to the last columns of V which are in the nullspace of M. We observe that this system has at least $d-k$ degrees of freedom.

The LLL algorithm can then be used to find a reduced lattice of solutions. To do so, first form the matrix $R' = \left(R^T | \alpha\vec{e}_1\right)$ where $\vec{e}_1$ is the unit vector $(1, 0, \ldots, 0)^T$ and α is a large integer. By choosing a sufficiently large α, the LLL algorithm will avoid adding the row containing the particular solution to the system to other rows. Thus after running LLL and resorting the rows, the matrix will be of the form $\left(S^T \mid \alpha\vec{e}_i\right)$ where S is a reduced matrix with columns $\vec{S}_0, \vec{S}_1, \ldots, \vec{S}_{d-k}$. The possible generators of the sequence are then $\vec{S}_0 + \sum_{i=1}^{d-k} \beta_i \vec{S}_i$ for integer constants β_i. This lattice fully describes the space of possible generators of degree d for the given sequence elements.

To look for candidate solutions choose a small constant c and explore the finite list of generators for each choice of $\beta_i \in [-c, c]$. Exclude any generator with a root of sufficiently large magnitude or which produces a negative term after some constant number of steps. This is a heuristic based on the reasoning that a reduced basis implies that large β_i correspond to large coefficients in the polynomial and large coefficients in the polynomial correspond to large roots which, by proposition 6.1, cannot minimally generate the sequence.

This method relies on lattice reduction which has a larger complexity than Berlekamp-Massey but the dimension of the lattices will be $d-k$. This can be accomplished in time $\mathcal{O}(\beta^{1+\epsilon}(d-k)^5 + \beta(d-k)^{4+\epsilon})$ [16].

6.2 Example candidate recurrence generators

We report on the results obtained using the method just described.

6.2.1 *Ding-Yuan*

Given the sequence $S = [2, 8, 42, 226, 1232, 6646, 35362, 185868]$ (these are the 3-ranks of the Ding-Yuan family for dimension 3^{2k+1}) we came up with the lattice $f_1 + x \cdot f_2 + y \cdot f_3$ of possible degree 5 generators where $f_1 = x^5 - 11x^4 + 38x^3 - 38x^2 - 14x + 13$, $f_2 = x^4 - 5x^3 + 34x^2 - 205x + 55$, and

Table 2: The 8 possible degree 5 generators given the first 8 ranks of the Ding-Yuan family, together with predicted rank for the 9th rank ($n = 3^{17}$ case)

Predicted	Polynomial
967082	$x^5 - 8x^4 + 6x^3 + 61x^2 - 54x - 138$
967650	$x^5 - 11x^4 + 38x^3 - 38x^2 - 14x + 13$
968218	$x^5 - 14x^4 + 70x^3 - 137x^2 + 26x + 164$
968786	$x^5 - 17x^4 + 102x^3 - 236x^2 + 66x + 315$
972190	$x^5 - 7x^4 + x^3 + 95x^2 - 259x - 83$
972758	$x^5 - 10x^4 + 33x^3 - 4x^2 - 219x + 68$
977298	$x^5 - 6x^4 - 4x^3 + 129x^2 - 464x - 28$
977866	$x^5 - 9x^4 + 28x^3 + 30x^2 - 424x + 123$

$f_3 = -3x^4 + 32x^3 - 99x^2 + 40x + 151$.

From there we could find a list of 85 (x, y)-pairs which had roots bounded by 9. Out of those 85, only 8 remained non-negative through the next 20000 terms. Thus with 8 ranks to work with, we have narrowed the field of degree 5 generators to a list of 8. These 8 reveal that the predicted next rank is between 967082 and 977866, and that the largest root of a degree 5 generator would be between 5.05 and 6.16 in absolute value. It follows from the proof of proposition 6.1 that these values bound the asymptotic growth rate of the sequence assuming a degree 5 linear recurrence. The predictions and possible generators are given in table 2. When the ninth rank is computed, if it matches one of the unique predictions of those 8 polynomials, then it will be the only possible generator of degree 5. If the ninth term does not match any of the predictions then no degree 5 linear recurrence is possible (up to the discovery of a small-rooted polynomial in the lattice with large x and y values).

Note that a degree 6 generator may turn out to be the case. We can currently create a large but finite list of candidate generators for degree 6.

6.2.2 Cohen-Ganley

In the Cohen-Ganley case we have computed 7 ranks. Based on the first 6 we were able to narrow down the number of possible degree 4 generators to 55. With the 7th rank we could determine that no degree 4 recurrence is possible. With the same 7 terms there are more than 30000 possible degree 5 generators, but a finite list is computable. Note that the negative result for degree 4 generators is achieved earlier than the 8 terms needed by Berlekamp-Massey to produce a candidate minimal generator.

6.2.3 Dickson

In the Dickson case a degree 3 recurrence has been conjectured based on the first 7 terms (6 for Berlekamp/Massey and one more to suggest validity) [14]. The proposed generator is $x^3 - 4x^2 - 2x + 1$. Our method here can prove uniqueness given only 5 terms. There is at most one generator of degree 3 generating these terms. Thus the 6th, 7th, (and now 8th) terms may be seen as further "evidence" for the conjectured generator.

Further exploring the method on this example, even basing on only knowledge of the first 4 terms we were able to narrow the field to 178 possible generators, with the conjectured generator appearing as the smallest vector produced by the search.

Table 3: Known and computed ranks for adjacency matrices of certain strongly regular graph families. Ranks in bold are first reported here so far as we know.

Order	Paley	$\mathcal{P}^*$	Dickson	C-G	D-Y
3^1					2
3^2	4	4	4	4	
3^3					8
3^4	16	16	20	20	
3^5					42
3^6	64	52	85	94	
3^7					226
3^8	256	160	376	448	
3^9					1232
3^{10}	1024	484	1654	2084	
3^{11}					6646
3^{12}	4096	1456	7283	**9652**	
3^{13}					**35862**
3^{14}	16384	4374	32064	**44650**	
3^{15}					**185868**
3^{16}	65536	13120	**141168**	-	
Formula:	known	known	conjecture	?	?

7. CONCLUSIONS AND FUTURE WORK

Many families of strongly regular graphs have been defined in terms of difference sets, either Paley type or skew Hadamard type. For several of these families, formulas for the adjacency matrix ranks have been found, the formula being in each case a homogeneous linear recurrence.

We have contributed to the discovery of linear recurrences for additional families in two ways, computing more terms of the rank sequence, and expanding the analysis strategy for making formula conjectures beyond the Berlekamp/Massey algorithm.

Several families with a known linear recurrence generator have it proven in the literature. However for Dickson's family, the "known" minimal generator is only conjectured. We have computed the 8th rank in this family, further strengthening the conjecture. To this end, we have developed, and here discussed, a parallel implementation of a projection technique for matrix rank computation suitable for cases when the rank is much less than matrix order. The approach uses 3 levels of parallelism: (1) a word level row vector parallel storage scheme, slicing, specific to computation over GF(3), (2) multi-threaded computation contributing to both projection to a smaller matrix and Gaussian elimination on that smaller matrix, (3) distributed and file system based parallelism to manage the large amount of data going into the coarsest grain of the projection. Levels (2) and (3) are applicable over any field. The implementation has successfully computed rank of a matrix of order 3^{16} having rank about 2^{17}.

For other sequences, specifically the Cohen-Ganley and Ding-Yuan families, the algorithm had provided so far ranks up to and including the 7th for C-G and the 8th for D-Y. We have established that C-G has no recurrence of degree less than 5. For D-Y our analysis shows that there is no recurrence of degree 4 or less and that a 9th term would establish a unique candidate for degree 5 recurrence or eliminate the possibility of degree 5. Future work is planned to compute the 8th rank in the Cohen-Ganley sequence, which is a relatively straightforward application of the parallel method of this paper. The 9th rank in the Ding-Yuan family is also within reach. The method of this paper may be used with sub-blocking to control memory and disk space demand. Additionally, for this skew Hadamard construction, rank may be found by use of an alternative (and sparse) matrix construction (not the adjacency matrix). We are exploring algorithms for that situation as well.

8. REFERENCES

[1] M. Albrecht and G. Bard. *The M4RI Library – Version 20100817*. The M4RI Team, 2010.

[2] M. Albrecht, G. Bard, and W. Hart. Efficient multiplication of dense matrices over gf(2). *CoRR*, abs/0811.1714, 2008.

[3] E. R. Berlekamp. Factoring polynomials over large finite fields. *Math. Comp.*, 24:713–735, 1970.

[4] T. J. Boothby and R. W. Bradshaw. Bitslicing and the Method of Four Russians Over Larger Finite Fields. *ArXiv e-prints*, Jan. 2009.

[5] A. Brouwer and C. Van Eijl. On the p-rank of the adjacency matrices of strongly regular graphs. *Journal of Algebraic Combinatorics*, 1(4):329–346, 1992.

[6] T.-W. J. Chou and G. E. Collins. Algorithms for the solution of systems of linear diophantine equations. *SIAM Journal on Computing*, 11(4):687–708, 1982.

[7] E. W. Dijkstra. Cooperating sequential processes. In P. B. Hansen, editor, *The origin of concurrent programming*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[8] C. Ding and J. Yuan. A family of skew Hadamard difference set. *J. Comb. Theory, Ser. A*, 113:1526–1535, 2006.

[9] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 2009.

[10] M. Giesbrecht. Efficient parallel solution of sparse systems of linear diophantine equations. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, PASCO '97, pages 1–10, New York, NY, USA, 1997. ACM.

[11] M. Lambert. private communication, U. of Delaware PhD Preliminary Research Project Report, 2015.

[12] G. Levy. *Solutions Of Second Order Recurrence Relations*. PhD thesis, Florida State University, Tallahassee, Florida, 2010.

[13] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory*, IT-15:122–127, 1969.

[14] J. May, B. Saunders, and Z. Wan. Efficient matrix rank computation with application to the study of strongly regular graphs. In *In Proc. of ISSAC 2007*, pages 277–284. ACM Press, 2007.

[15] T. Mulders and A. Storjohann. Diophantine linear system solving. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, ISSAC '99, pages 181–188, New York, NY, USA, 1999. ACM.

[16] A. Novocin, D. Stehlé, and G. Villard. An lll-reduction algorithm with quasi-linear time complexity: Extended abstract. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 403–412, New York, NY, USA, 2011. ACM.

[17] B. D. Saunders and B. Youse. Large matrix, small rank. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, In Proc. of ISSAC 2009, pages 317–324, New York, NY, USA, 2009. ACM.

[18] N. Sloane. The on-line encyclopedia of integer sequences. http://oeis.org.

[19] V. Strassen. Gaussian elimination is not optimal. *Numer. Mathematik*, 13:354–356, 1969.

[20] G. Weng, W. Qiu, Z. Wang, and Q. Xiang. Pseudo-paley graphs and skew hadamard difference sets from presemifields. *Designs, Codes and Cryptography*, 44(1-3):49–62, 2007.

[21] B. Youse. Finite field/rank code suite. https://bitbucket.org/bryouse/finite-field-rank-suite.

[22] B. Youse. *High Performance Exact Linear Algebra*. PhD thesis, University of Delaware, Newark, DE, 2015.