Construction of Extremal $(48, 24, 12)$ Doubly-Even Codes

Sheridan Houghten

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September, 1993

ISBN 0-315-87267-5

Canada

Abstract

Construction of Extremal (48,24,12) Doubly-Even Codes

Sheridan Houghten

An *extremal* error-correcting code is one that corrects the maximum possible number of errors for its length and dimension. A code's error-correcting capability is directly related to its minimum weight. According to Mallows and Sloane (1973), the largest minimum weight of a self-dual, doubly-even binary code of length n and dimension $n/2$ is $d = 4[n/24] + 4$.

Of such codes, there is one extremal code of length 24 and one known extremal code of length 48. There is no known extremal code of length 72. The search for other non-isomorphic extremal codes of length 48 may be divided into three cases. We completed a search assuming one of these cases, finding only codes that are isomorphic to the known code.

Acknowledgments

I would like to thank my supervisor, Dr. Clement Lam, for his guidance and support throughout the research for this thesis. I also thank Mr. Larry Thiel for his assistance both with ideas and with writing and debugging programs.

For allowing me access to their computing facilities, I thank the staff of Computer Services, the Department of Computer Science and the Centre interuniversitaire en calcul mathématique algébrique at Concordia University, and the staff of the Computer Science Department at Northeastern University, Boston.

I owe a great deal to my family and friends for being understanding and for encouraging me, and I thank them all.

Contents

List of Figures

List of Symbols

C	an (n, k, d) linear code
$C^{\perp}$	the dual of C
D	an extremal $(48, 24, 12)$ doubly-even code
RM	the $(24, 4, 12)$ code obtained by taking 3 copies of the Reed-Muller code $R(1, 3)$
row_i	the i'th row of a generator matrix
QR	the Extended Quadratic Residue code of length 48
v'	the codeword v complemented on the first 24 columns
$v \cdot w$	the inner product of codewords v and w
$v * w$	the number of intersections between codewords v and w
W_C	the weight enumerator of C
$wt(v)$	the weight of codeword v

Chapter 1

Introduction

This thesis is concerned with the construction of extremal self-dual $(48, 24, 12)$ doubly-even codes. This chapter gives some of the basic definitions used in coding theory. Some textbooks on the subject include [2, 5, 7, 8]. There is also a description of the organization of the thesis and a summary of contributions.

1.1 Error-correcting Codes

Let F be $GF(q)$, the finite field with q elements. A code C is a subset of F^n, the set of vectors of length n with components from F. The elements of C are called *codewords*. A code can be either *linear* or *non-linear*. It is linear if for any two codewords u and v, $u + v$ is also a codeword. If this is the case, then the code forms a linear subspace. In this thesis, we are interested only in linear codes.

The *weight* $wt(w)$ of a codeword w is the number of its non-zero components. A code is *even* if the weight of any vector in the code is divisible by 2, and *doubly-even* if the weight of any vector is divisible by 4.

The *minimum weight* of a code is the smallest weight of the non-zero vectors in the code. The *distance* between two codewords u and v is the number of components in which they differ. For linear codes, *minimum weight* and *minimum distance* are the same.

An (n, k, d) code is a k-dimensional subspace of F^n with minimum distance d. We can easily correct up to $(d-1)/2$ errors by using *maximum likelihood decoding*, which involves simply choosing the closest codeword to the one we receive. Because

of its error-correcting abilities, we obviously find high minimum distance to be an extremely desirable property of a code.

Two linear, binary codes are *equivalent*, or *isomorphic*, if one can be obtained from the other by a coordinate permutation of their codewords. Isomorphic codes have the same decoding properties.

We say that C is *cyclic* if for every codeword $v = v_0 v_1 \dots v_{n-1} \in C$, the vector $v_{n-1} v_0 v_1 \dots v_{n-2}$, obtained by cycling the elements of v one space to the right, is also a codeword.

1.2 Self-Dual Codes

Let C be an (n, k, d) code over $GF(2)$. If u and v are two vectors in C, then their *inner product* is defined as

$$u \cdot v = \sum_{i=1}^{n} u_i v_i \pmod 2.$$

If $u \cdot v = 0$ then u and v are *orthogonal*. The *dual* of C is defined as

$$C^{\perp} = \{u \in GF(2)^n | u \cdot v = 0 \; \forall v \in C\}.$$

A code C is *self-dual* if $C = C^{\perp}$.

The largest minimum weight of a self-dual, doubly-even $(n, n/2, d)$ code over $GF(2)$ is $d = 4[n/24] + 4$ [6]. A self-dual code that has the largest possible minimum weight is an *extremal code*. Of such codes, the *Golay code* is the only $(24, 12, 8)$ code and the Extended Quadratic Residue code QR is the only known $(48, 24, 12)$ code. There is no known $(72, 36, 16)$ extremal code. Such codes are of particular interest because for any non-zero weight w, the codewords of weight w form a 5-design [1].

1.3 Objective and Contribution

Our objective is to determine whether the Extended Quadratic Residue Code is the only $(48, 24, 12)$ code. The method we use is exhaustive enumeration. The search can

be divided into three cases. In this thesis we report the results of a search assuming one of the cases.

1.4 Organization of Thesis

The layout of the remainder of the thesis is as follows: Chapter 2 gives techical definitions, mathematical preliminaries and background information about the research problem. Chapters 3, 4, 5 and 6 describe the methodology used to solve the problem. Chapter 7 gives the results and conclusion.

Chapter 2

Mathematical Preliminaries and Notation

This chapter gives some technical definitions and additional information that we require to solve the problem of generating a $(48, 24, 12)$ doubly-even self-dual code. We refer to this code as D.

2.1 Generator Matrix

A *generator matrix* of an (n, k, d) code C is a $k \times n$ matrix that contains k basis-vectors. It is possible to have more than one basis, so obviously there can be more than one generator matrix.

2.2 Weight Enumerator

The *weight enumerator* of C is

$$W_C(x, y) = a_0 x^n + a_1 x^{n-1} y + a_2 x^{n-2} y^2 + \cdots + a_n y^n, \tag{2.1}$$

where a_i is the number of vectors in C of weight i. The MacWilliams Identity relates the weight enumerator of a code to that of its dual: [5, page 127]

$$W_{C^\perp}(x, y) = \frac{1}{|C|} W_C(x + y, x - y). \tag{2.2}$$

The only possible non-zero coefficients in the weight enumerator of D are a_0, a_{12}, a_{16}, a_{20}, a_{24}, a_{28}, a_{32}, a_{36} and a_{48}.

Define z as the length-48 vector whose components are all 1. Because z is in $D^\perp$ and D is self-dual, z is in D. For each vector of weight i in D there is a corresponding vector of weight $48 - i$ obtained by adding z. Thus the coefficients of the weight enumerator are related as follows: $a_0 = a_{48} = 1$, $a_{12} = a_{36}$, $a_{16} = a_{32}$, $a_{20} = a_{28}$.

From Eq. (2.1) and (2.2) we have

$$W_D(x,y) = x^0y^{48} + a_{12}x^{12}y^{36} + a_{16}x^{16}y^{32} + \cdots + a_{16}x^{32}y^{16} + a_{12}x^{36}y^{12} + x^{48}y^0$$

and

$$W_D(x,y) = W_{D^\perp}(x,y) = \frac{1}{2^{24}}W_D(x+y, x-y).$$

By expanding $\frac{1}{2^{24}}W_D(x+y, x-y)$ and equating the coefficients of x^2y^{46}, x^4y^{44}, x^6y^{42} and $x^{10}y^{38}$ to zero we get a system of equations which, when solved, give:

$$\begin{aligned} a_0 &= a_{48} = 1, \\ a_{12} &= a_{36} = 17\ 296, \\ a_{16} &= a_{32} = 535\ 095, \\ a_{20} &= a_{28} = 3\ 995\ 376, \\ a_{24} &= 7\ 681\ 680. \end{aligned}$$

2.3 Miscellaneous Identities

Let $dim\ C$ denote the dimension of a code C. From linear algebra, we have

$$\dim C + \dim C^\perp = n, \tag{2.3}$$

for any linear (n, k, d) code C [7, page 8].

An *intersection* of two codewords is a position in which both codewords contain a 1. We define $a * b$ as the number of intersections between codewords a and b. We have the following identity relating $wt(a+b)$ and $a * b$: [7, page 14]

$$wt(a+b) = wt(a) + wt(b) - 2(a * b). \tag{2.4}$$

Since all codewords of D must be doubly-even, $v * w$ must be even for all codewords v and w in D.

2.4 Quadratic Residue Codes

As previously mentioned, the only known $(48, 24, 12)$ doubly-even, self-dual code is the Extended Quadratic Residue code of length 48, QR. Here we give a brief description of this code. If we find any codes during our search, we must check whether they are isomorphic to the known code. For further information on quadratic residue codes, see [2].

An element $a \in GF(p)$ is a *quadratic residue* modulo p if $x^2 = a$ has non-zero solutions in GF(p) [2, page 171]. For $p = 47$ the set of quadratic residues is

$$S = \{1, 2, 3, 4, 6, 7, 8, 9, 12, 14, 16, 17, 18, 21, 24, 25, 27, 28, 32, 34, 36, 37, 42\}.$$

We define the Hall polynomial as $\theta(x) = \sum_{s \in S} x^s$. The generator polynomial for the $(47, 24, 11)$ quadratic residue code can be obtained by

$$g(x) = \gcd(1 + x + \cdots + x^{46}, \theta(x)).$$

We find that

$$g(x) = 1 + x + x^2 + x^3 + x^5 + x^6 + x^7 + x^9 + x^{10} + x^{12} + x^{13} + x^{14} + x^{18} + x^{19} + x^{24}.$$

From $g(x)$ we construct a generator matrix for the $(47, 24, 11)$ Quadratic Residue code. Label the columns of the generator matrix from left to right as 0 to 46. Row 1 of the matrix has a 1 in column i if x^i is in $g(x)$, and a 0 otherwise. Any other row j, $2 \leq j \leq 24$, is obtained by cyclically shifting row $j - 1$ one column to the right.

To obtain the Extended Quadratic Residue code QR of length $n + 1 = 48$, we annex an overall parity check column to the matrix. We label this column as ∞. Fig. 2.1 gives the resulting generator matrix.

2.5 Reed-Muller Codes

The linear subspace of $GF(2)^8$ generated by the matrix shown in Fig. 2.2 is the first-order Reed-Muller code, $R(1, 3)$, of length 8 [7, page29]. The automorphism group of this code, $A\Gamma L(3, 2)$, is known to be triply-transitive [2, page 366].

Figure 2.1: Generator matrix of QR

			1	111	111	111	222	222	222	233	333	333	334	444	444
∞01	234	567	890	123	456	789	012	345	678	901	234	567	890	123	456
111	110	111	011	011	100	011	000	100	000	000	000	000	000	000	000
101	111	011	101	101	110	001	100	010	000	000	000	000	000	000	000
100	111	101	110	110	111	000	110	001	000	000	000	000	000	000	000
100	011	110	111	011	011	100	011	000	100	000	000	000	000	000	000
100	001	111	011	101	101	110	001	100	010	000	000	000	000	000	000
100	000	111	101	110	110	111	000	110	001	000	000	000	000	000	000
100	000	011	110	111	011	011	100	011	000	100	000	000	000	000	000
100	000	001	111	011	101	101	110	001	100	010	000	000	000	000	000
100	000	000	111	101	110	110	111	000	110	001	000	000	000	000	000
100	000	000	011	110	111	011	011	100	011	000	100	000	000	000	000
100	000	000	001	111	011	101	101	110	001	100	010	000	000	000	000
100	000	000	000	111	101	110	110	111	000	110	001	000	000	000	000
100	000	000	000	011	110	111	011	011	100	011	000	100	000	000	000
100	000	000	000	001	111	011	101	101	110	001	100	010	000	000	000
100	000	000	000	000	111	101	110	110	111	000	110	001	000	000	000
100	000	000	000	000	011	110	111	011	011	100	011	000	100	000	000
100	000	000	000	000	001	111	011	101	101	110	001	100	010	000	000
100	000	000	000	000	000	111	101	110	110	111	000	110	001	000	000
100	000	000	000	000	000	011	110	111	011	011	100	011	000	100	000
100	000	000	000	000	000	001	111	011	101	101	110	001	100	010	000
100	000	000	000	000	000	000	111	101	110	110	111	000	110	001	000
100	000	000	000	000	000	000	011	110	111	011	011	100	011	000	100
100	000	000	000	000	000	000	001	111	011	101	101	110	001	100	010
100	000	000	000	000	000	000	000	111	101	110	110	111	000	110	001

Figure 2.2: Generator matrix of $R(1,3)$

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{matrix}$$

We find it convenient to use the generator matrix shown in Fig. 2.3. One can obtain Fig. 2.3 from Fig. 2.2 by first replacing the first row of Fig 2.2 by the sum of its first and fourth row, followed by some row and column permutations.

Figure 2.3: Modified generator matrix of $R(1,3)$

1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1
1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0

Taking three "copies" of the above, we obtain RM, a $(24,4,12)$ code with generator matrix shown in Fig. 2.4. Note that there are eight blocks, each three columns wide; any three of these blocks may be mapped to any three others.

Figure 2.4: Generator matrix of RM

111	111	111	111	000	000	000	000
000	000	000	000	111	111	111	111
111	111	000	000	111	111	000	000
111	000	111	000	111	000	111	000

Chapter 3

Search Strategy

To conduct our search, we first attempt to find the basic structure of the generator matrix. Next we try to "fill in" the matrix by finding the possibilities row by row.

3.1 Finding the Structure of the Generator Matrix

Because the number of weight-12 codewords in D is non-zero, we use as many of these as possible when trying to form the generator matrix G. We label the rows of G as $row_1, \ldots, row_{24}$. We have $row_i \cdot row_j = 0$ since D is self-dual.

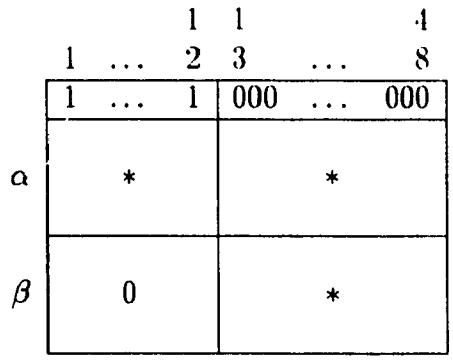
Due to column permutations, we can take any weight-12 word for row_1. Let us choose

$$row_1 = 111\ 111\ 111\ 111\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

Consider the largest subspace D_1 of D with zeros in the first 12 positions. Suppose $\dim D_1 = \beta$. Choose β basis vectors of D_1 and place them at the bottom of the matrix. Define $\alpha = 23 - \beta$. By permuting the rows, we obtain a generator matrix G_1 of the form shown in Fig. 3.1, where $*$ represents unknown entries.

We shall show that $\alpha = 10$ and $\beta = 13$. First consider the subspace of length 12 formed by restricting the vectors to their first 12 coordinates. We call these restricted rows $row_i(12)$, where $1 \leq i \leq 24$. The vectors $row_1(12), \ldots, row_{\alpha+1}(12)$ are linearly independent, since if they were not, then some linear combination of $row_1, \ldots, row_{\alpha+1}$ would be a length-48 vector with zeros in the first 12 positions; however, all such

Figure 3.1: Matrix G_1



vectors are in D_1 so such a linear combination is not possible. Let V_1 be the subspace generated by $row_1(12)$. Since $row_1(12) \cdot row_i(12) = 0$ for $i = 1, \ldots, \alpha + 1$, and by using Eq.(2.3), we have

$$1 + (\alpha + 1) \leq \dim(V_1) + \dim(V_1^{\perp}) = 12. \tag{3.1}$$

Next consider the subspace of length 36 formed by restricting the vectors to their last 36 coordinates. We call these restricted rows $row_i(36)$, where $1 \leq i \leq 24$. Let V_2 be the subspace generated by $row_{\alpha+2}(36), \ldots, row_{24}(36)$. We know that $row_{\alpha+2}(36), \ldots, row_{24}(36)$ are linearly independent since $row_{\alpha+2}, \ldots, row_{24}$ are also linearly independent. Also $row_2(36), \ldots, row_{24}(36)$ must be linearly independent since if they were not, then some linear combination of $row_2, \ldots, row_{24}$ would be a vector in the subspace generated by row_1. Furthermore, $row_i(36) \cdot row_j(36) = 0$ for $i = \alpha + 2, \ldots, 24$ and $j = 2, \ldots, 24$. Therefore

$$\beta + (\alpha + \beta) \leq \dim(V_2) + \dim(V_2^{\perp}) = 36. \tag{3.2}$$

Adding (3.1) and (3.2) we obtain

$$2 + 2\alpha + 2\beta \leq 48.$$

So $\alpha + \beta \leq 23$. However we know that $\alpha + \beta + 1 = 24$ since 24 is the total number of rows in the generator matrix. Hence (3.1) and (3.2) are actually equalities. Solving for α and β, we find that $\alpha = 10$ and $\beta = 13$.

To choose the later rows of the generator matrix, we have to be more careful. The new vector must be linearly independent from all previously-chosen vectors; also all linear combinations of the new vector with previously-chosen vectors must be valid codewords.

When trying to choose row_2, only row_1 has been previously chosen. We want to choose a weight-12 word that has zeroes in the first 12 positions. If we are unable to choose this word, then the last 13 rows and 36 columns of the matrix form a code with parameters $(36, 13, 16)$. Using the MacWilliams Identity to find the weight enumerator for this code, we obtain fractional and negative coefficients, so such a code is impossible. Hence by column permutations we can choose

$$row_2 = 000\ 000\ 000\ 000\ 111\ 111\ 111\ 111\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

We permute the rows so that the generator matrix, G_2, has the form shown in Fig. 3.2.

Figure 3.2: Matrix G_2

	1 ... 12	13 ... 24	25 ... 48
	1 ... 1	0 ... 0	000 ... 000
	0 ... 0	1 ... 1	000 ... 000
α_1	*	*	0
α_2	*	*	*
β_1	0	*	*
β_2	0	0	*

Consider the largest subspace D_2 of D with zeros in the last 24 positions. Suppose $\dim D_2 = \alpha_1 + 2$. Choose row_1 and row_2 of G_2 as the first two basis vectors of D_2, and then choose a further α_1 basis vectors and place them in rows 3 to $2 + \alpha_1$. Now consider the largest subspace of D_3 of D with zeros in the first 24 positions. Suppose

$\dim D_3 = \beta_2$. Choose β_2 basis vectors of D_3 and place them at the bottom of the matrix. Define $alpha_2 = 10 - \alpha_1$ and $\beta_1 = 12 - \beta_2$. Let $\gamma = \alpha_2 + \beta_1$. We know that $\alpha_1 + \gamma + \beta_2 = 22$.

Consider the subspace of length 24 formed by restricting the vectors to their first 24 columns. We call these restricted rows $row_i(24a)$, where $1 \leq i \leq 24$. We know that $row_1(24a), \ldots, row_{\alpha_1+2}(24a)$ are linearly independent since $row_1, \ldots, row_{\alpha_1+2}$ must be linearly independent. Also $row_1(24a), \ldots, row_{\alpha_1+\gamma+2}(24a)$ are linearly independent since if they were not then some linear combination of $row_1, \ldots, row_{\alpha_1+\gamma+2}$ would be a length-48 vector with zeros in the first 24 positions; however, all such vectors are in D_3. Let W_1 be the subspace generated by $row_1(24a), \ldots, row_{\alpha_1+2}(24a)$. We have $row_i(24a) \cdot row_j(24a) = 0$ for $i = 1, \ldots, \alpha_1 + 2$ and $j = 1, \ldots, \alpha_1 + \gamma + 2$. So

$$(2 + \alpha_1) + (2 + \alpha_1 + \gamma) \leq \dim(W_1) + \dim(W_1^{\perp}) = 24. \tag{3.3}$$

Next consider the subspace formed by restricting the vectors to their last 24 columns. Call these restricted rows $row_i(24b)$, where $1 \leq i \leq 24$. We know that $row_25 - \beta_2(24b), \ldots, row_{24}(24b)$ are linearly independent since $row_{25} - \beta_2, \ldots, row_{24}$ must be linearly independent. Also $row_25 - \gamma - \beta_2(24b), \ldots, row_{24}(24b)$ must be linearly independent since if they were not then some linear combination of $row_25 - \gamma - \beta_2, \ldots, row_{24}$ would be a length-48 vector with zeros in the last 24 positions; however, all such vectors are in D_2. Let W_2 be the subspace generated by the last β_2 restricted rows. We have $row_i(24b) \cdot row_j(24b) = 0$ for $i = 25 - \beta_2, \ldots, 24$ and $j = 25 - \gamma - \beta_2, \ldots, 24$. So

$$\beta_2 + (\beta_2 + \gamma) \leq \dim(W_2) + \dim(W_2^{\perp}) = 24. \tag{3.4}$$

Adding (3.3) and (3.4), we get the inequality

$$2\alpha_1 + 2\gamma + 2\beta_2 \leq 44.$$

However, we know that $2 + \alpha_1 + \gamma + \beta_2 = 24$ so the inequality is an equality.

The subcode E formed by the last β_2 rows and last 24 columns has weight enumerator $W_E(x, y) = x^0y^{24} + (2^{\beta_2} - 2)x^{12}y^{12} + x^{24}y^0$. We expand the weight enumerator

of the dual, $W_{E^\perp}(x,y) = \frac{1}{2^{\beta_2}} W_E(x+y, x-y)$, and look at the coefficients obtained. Since the coefficients must be integer and positive, $\beta_2 \leq 5$. The list of possibilities for α_1, β_2 and γ is thus as follows:

α_1	γ	β_2
0	20	2
1	18	3
2	16	4
3	14	5

We shall show by construction that the maximum number of linearly-independent words in E is just four. This construction is shown in Fig.(3.3).

Figure 3.3: Construction of a generator matrix for E

	A		B		C		D	
row_1	111	111	111	111	111	111	111	111
row_2	111	111	111	111	000	000	000	000
row_3	111	111	000	000	111	111	000	000
row_4	111	000	111	000	111	000	111	000
row_5	a	b	c	d	e	f	g	h

We can choose any codeword for the first row — we choose the length-24 vector whose components are all 1. Only weight-12 vectors remain. For the second row, we choose the vector which has 12 ones at the start. To find the third row, we use Eq.(2.4) as follows:

$$\begin{aligned} wt(row_2 + row_3) = \; & wt(row_2) + \; wt(row_3) - \; 2(row_2 * row_3) \\ 12 = \; & 12 + \; 12 - \; 2(row_2 * row_3) \end{aligned}$$

So $row_2 * row_3 = 6$, and so for row_3, we place 6 ones in each of sections A and C. Similarly, $row_2 * row_4 = 6$ and $row_3 * row_4 = 6$. Using these facts, we find that row_4 must contain 3 ones and 3 zeros in each of sections A, B, C and D. We then try to find a fifth row. Using the same procedure, we find that any row_5 must contain 3/2 ones and 3/2 zeros in each of sections a, ..., h. This is impossible, so β_2 is at most 4.

Hence there are just three cases to consider: namely, when $\beta_2 = 2$, 3 and 4. In this thesis, I consider only the case $\beta_2 = 4$. Thus E is a $(24, 4, 12)$ code. It can be shown that all such codes are equivalent to the code RM shown in Fig. 2.4. The

subcode formed by the first 4 rows and first 24 columns of D is also a $(24, 4, 12)$ code and equivalent to the one generated by the matrix in Fig. 2.4. Thus for this case, eight rows are known — the first four and the last four. The other two cases will involve larger searches since fewer rows are known — specifically, four rows for the case $\beta_2 = 2$ and six rows for the case $\beta_2 = 3$.

3.2 The Structure of the Remaining Rows

Although we still have to construct the remaining 16 rows, we can determine some information about their form by considering how these rows interact with the known rows.

In these 16 rows, the last 24 columns of each possible linear combination must be codewords of $RM^{\perp}$ because of the intersection with the bottom copy of RM. We know from the weight enumerator that there are 24 codewords of weight 2 in $RM^{\perp}$. If w is such a codeword then for D to be doubly-even w must intersect each of the linear combinations of RM in either zero or two places. Thus w has both of its ones in the same block. All 24 codewords may be generated in 16 rows as follows:

	2 5	2 8	3 1	...	4 5	4 8
row 5	110 101					
		110 101				
			⋱			
row 20						110 101

where a blank space denotes portions filled entirely by zeroes.

Since these 16 rows are linearly independent, together with the last 4 rows they generate $RM^{\perp}$. Therefore we can assume without loss of generality that each of these 16 rows has weight 2 in the last 24 columns. We can determine the possible weights for the first 24 columns by looking at the intersection of each row with the top copy of RM. Let x be any codeword that appears in one of these rows, where

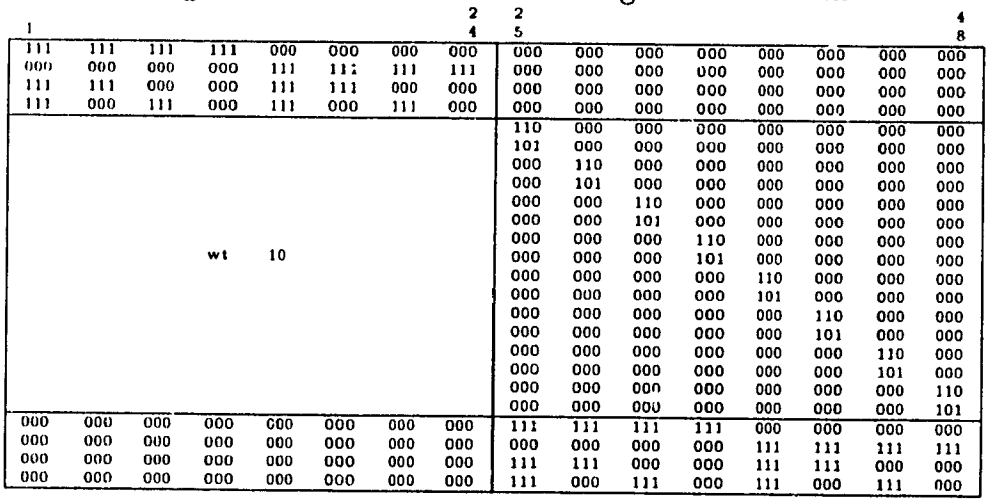
the first 24 columns of x are denoted by x_a and the last 24 columns by x_b. Obviously $wt(x_a) \geq 10$, or $wt(x)$ is less than the minimum weight of D.

The codeword obtained by adding row_1 and row_2 is

$$row_1 + row_2 = 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

Each possible x must also satisfy the condition $wt(x + row_1 + row_2) \geq 12$. Since $wt((x + row_1 + row_2)_b) = 2$, $wt((x + row_1 + row_2)_a) \geq 10$. Thus $wt(x_a) \leq 14$. Since each codeword must be doubly-even, x_a can have a weight of only 10 or 14. Furthermore, if $wt(x_a) = 14$, then $wt((x + row_1 + row_2)_a) = 10$. Therefore we need only consider the case $wt(x_a) = 10$, and the matrix has the form shown in Fig. 3.4.

Figure 3.4: Basic structure of the generator matrix



Each of the eight blocks in the first 24 columns of the generator matrix can contain either 0, 1, 2 or 3 ones. Only certain combinations of block-weights are possible. Define b_0, b_1, b_2, and b_3 to be the number of blocks containing 0, 1, 2, and 3 ones respectively. Then we have $b_0 + b_1 + b_2 + b_3 = 8$. Since the weight in the first 24 columns must be 10, we have $b_1 + 2b_2 + 3b_3 = 10$.

If $b_3 \geq 2$ for a given codeword w, then using the triply-transitive property on column blocks, we can permute the blocks so that in the first two positions we have two blocks of 3 ones and in the third position we have a block of $s > 0$ ones. In the

fourth position there is a block of t ones. We have

$$\begin{array}{rcll} wt(row_1 + w) & = & wt(row_1) & +wt(w) \quad -2(row_1 * w) \\ & \leq & 12 & +12 \quad -2 * 7 \\ & \leq & 10. & \end{array}$$

which is too small. So $b_3 \leq 1$ for any of the unknown rows.

If $b_3 = 0$ then we have

$$\begin{array}{rcl} b_0 + b_1 + b_2 & = & 8 \\ b_1 + 2b_2 & = & 10. \end{array}$$

We have the following cases to consider:

case	b_0	b_1	b_2	b_3
0.1	3	0	5	0
0.2	2	2	4	0
0.3	1	4	3	0
0.4	0	6	2	0

For case 0.1, we can permute the blocks so that the last three positions each contain blocks of 0 ones. Adding row_1 we obtain a vector of weight 8. For case 0.2, we can permute the blocks so that the first two positions each contain blocks of 0 ones and the third contains a block of 1 one. Adding row_2 we obtain a vector of weight ≤ 10. For case 0.4, we can permute the blocks so that the first two positions each contain blocks of 2 ones. Adding row_4 we obtain a codeword of weight 14. Thus the only possibility is case 0.3.

If $b_3 = 1$ then we have

$$\begin{array}{rcl} b_0 + b_1 + b_2 & = & 7 \\ b_1 + 2b_2 & = & 7. \end{array}$$

So $b_0 = b_2$ and we have the following cases to consider:

case	b_0	b_1	b_2	b_3
1.1	0	7	0	1
1.2	1	5	1	1
1.3	2	3	2	1
1.4	3	1	3	1

For case 1.2, we can permute the blocks so that the first position contains the block of 3 ones, the second the block of 2 ones and the third the block of no ones. Adding row_3 we obtain a vector of weight 10. For cases 1.3 and 1.4, we can permute the blocks so that the first position contains the block of 3 ones and the second and

third positions contain blocks of 2 ones. Adding row_1 we obtain a vector of weight ≤ 8.

Hence we need only consider cases 0.3 and 1.1. From here on, I refer to case 1.1 as *a solution of type* $3 \cdot 1^7$, and case 0.3 as *a solution of type* $2^3 \cdot 0 \cdot 1^4$. In general, given a codeword x, if b_0, b_1, b_2 and b_3 are the number of blocks in x containing 0, 1, 2, or 3 ones respectively, then we say that x is of type $0^{b_0} \cdot 1^{b_1} \cdot 2^{b_2} \cdot 3^{b_3}$.

Each y, a weight-12 linear combination of RM, has four blocks containing 3 ones and four containing 3 zeros. If a is a solution of type $2^3 \cdot 0 \cdot 1^4$, we can map the three blocks containing 2 ones into the first three blocks; in addition the block containing 0 ones will be mapped to the fourth block because $y * a$ must be even. Thus there is one y whose four blocks containing 3 zeros occur in the same positions as the four blocks of a containing 1 one. So $a + y$ is a solution of type $3 \cdot 1^7$.

We can show by counting that there are fewer possibilities of type $3 \cdot 1^7$ than of type $2^3 \cdot 0 \cdot 1^4$. First consider vectors of type $3 \cdot 1^7$. There are eight blocks in which we can place the block of three ones; in addition, there are three columns in which we can place the 1 in each of the seven remaining blocks. Thus the total number of possibilities is $8 * 3^7 = 17496$. Now consider vectors of type $2^3 \cdot 0 \cdot 1^4$. There are $\binom{8}{3}$ ways to place the three blocks containing 2 ones. Because the number of intersections between any two vectors must be even, the block with only zeros is also fixed. There are three ways to place the ones in each of the seven blocks containing either 1 or 2 ones. This gives a total of $8 * 7 * 3^7 = 122472$ possibilities.

Since a vector of type $2^3 \cdot 0 \cdot 1^4$ can be transformed into a vector of type $3 \cdot 1^7$, to fill in rows 5 to 20 of the generator matrix in the form of Fig. 3.4, we need only consider vectors which are of type $3 \cdot 1^7$ in the first 24 columns.

Chapter 4

Filling rows 5 to 8

This chapter describes how we generate candidates for rows 5 to 8. Each of these rows is of type $3 \cdot 1^7$ in the first 24 columns and has 2 ones in the same block of the last 24 columns.

4.1 Candidates for Rows 5 and 6

Due to the allowed column permutations, there is just one possibility of the required type for the fifth row of the generator matrix, namely

$$row_5 = 111\ 100\ 100\ 100\ 100\ 100\ 100\ 100\ 110\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

For the sixth row, there are two cases to consider after the allowed column permutations, namely:

$$row_{6a} = 111\ 010\ 010\ 010\ 010\ 010\ 010\ 010\ 101\ 000\ 000\ 000\ 000\ 000\ 000\ 000$$

and

$$row_{6b} = 100\ 111\ 100\ 010\ 010\ 010\ 010\ 010\ 101\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

If we choose row_{6a} for the sixth row, then rows 5 and 6 of the generator matrix both have 3 ones in the same block. From here on, I refer to this case as *type a*. In the case where these rows do not have a block of 3 ones in common, row_{6b} is indeed the only possibility we need to consider since we can move the block in which the

sixth row has 3 ones to any position excepting that in which row_5 has 3 ones. From here on, this case is referred to as *type b*.

In order to generate the candidate lists for later rows, we take the list of 17 496 vectors of type $3 \cdot 1^7$. For each candidate x, we determine if $x + combo$ has acceptable weight for each linear combination *combo* of rows 1 to 6. If this is the case, then we add x to the new list. We have to do this for each of the two possible sixth rows, creating two lists *type_a_list* and *type_b_list*.

Type_a_list contains all candidates for row 7 onwards, given that the sixth row is row_{6a}; this contains a total of 3654 candidates. *Type_b_list* is the corresponding list for the case that the sixth row is row_{6b}, and contains 3678 candidates.

4.2 Candidates for Rows 7 and 8

A "simple" algorithm to do a complete search starting at row 7 is as follows:

```
curr_row := 6;
For each matrix completed to curr_row do
begin
    if  matrix is type a then
        candlist := type_a_list;
    else
        candlist := type_b_list;
    compute all linear combinations of the first curr_row rows;
    do_row(curr_row+1, candlist);
end.

Procedure do_row(rownum, candlist)
begin
    for each candidate c in candlist do
    begin
        if all linear combinations of c with rows 1..rownum-1
```

```
            are doubly-even and have weight ≥ 12 and ≤ 36
        then
            store c in candlist_rownum;
    end;
    for each candidate c in candlist_rownum do
    begin
        rownum'th row of generator matrix := c;
        if rownum = 20 then print matrix;
        else if rownum = 8 then
            do_row(rownum+1,candlist_7);
        else if rownum is divisible by 2 then
            do_row(rownum+1, candlist);
        else
            do_row(rownum+1, candlist_rownum);
    end;
end;
```

Note that *candlist* does not change with every row. This is because of the 2 ones that appear in the last 24 columns of all the candidates. Within an (odd, even) row pair (row_i, row_{i+1}), where $7 \leq i \leq 19$, the 2 ones appear in the same 3-column-wide block. Thus $row_i + row_{i+1}$ has a weight in the last 24 columns of 2, and a weight in the first 24 columns of either 10 or 14. However, when we consider the next odd row, namely row_{i+2}, the 2 ones in the last 24 columns appear in a different block than those of the two previous rows. So $row_i + row_{i+2}$ has a weight in the last 24 columns of 4, and a weight in the first 24 columns of either 8, 12 or 16. If we were to check the list of candidates for row_{i+1} to fill row_{i+2} then we would find no candidates that would be acceptable. Thus we use the list of candidates for row_i.

We can modify this algorithm a little in order to make it more efficient. We shall now discuss several things that can be done in order to trim the search tree and thereby save time.

First, if the candidate list is ordered (say lexicographically) then we can insist on certain orderings between rows. Note that within any (odd, even)-row pair (row_i, row_{i+1}), where $7 \leq i \leq 19$, we can insist that row_i is smaller lexicographically than row_{i+1} because it must satisfy the same constraints, namely:

- all the linear combinations of it with rows $1 \ldots i-1$ must have acceptable weights, and
- both the even-row and the odd-row have 2 ones in the same 3-column-wide block of the last 24 columns of the matrix.

Due to the above, and the fact that (row_i, row_{i+1}) must be mutually compatible, we can see that if x is acceptable as odd row i and y as even row $i+1$, then x would be equally acceptable as even row $i+1$ were y to be odd row i. However, these two cases are isomorphic, and so we need only consider the case where y is further down the lexicographically-ordered list of candidates than x.

A second ordering relates to information obtained about the matrix as of the sixth row. Recall that at row 6 there are two non-isomorphic cases, namely *type a*, in which the block of 3 ones in row_5 matches to that of row_6, and *type b*, in which they do not. If we are working on a matrix of type a, then we can insist that all subsequent (odd, even)-row pairs (row_i, row_{i+1}) have blocks of 3 ones that match up. This is the case because if (row_i, row_{i+1}) did not have blocks of 3 ones that matched, then we could exchange this pair with (row_5, row_6) (thereby "turning it into" a type b matrix).

Note: even if we are dealing with a type b matrix, if a later pair of rows have blocks of 3 ones that match, we still may be able to insist that all later pairs have blocks of 3 ones that match, depending on which row this occurs at. However, we don't actually use this in order to simplify programming!

We do not follow the above algorithm exactly because after row 8 it becomes very time-consuming. We use it only up to row 8, using the above modifications, and in addition stopping at each row to run an isomorphism check of the code generated. There are a total of 63 non-isomorphic matrices complete to row 7 — 5 of *type a* and

58 of *type b*. For each of the above matrices, we run one level of the algorithm in order to find the matrices complete to row 8, then check for isomorphic cases. Of a total of 38124 matrices, 1607 are non-isomorphic 9 of *type a* and 1598 of *type b*.

Chapter 5

Estimates

In order to design an algorithm to determine the remainder of the generator matrix, we need to have a good idea of the number of candidates that will need to be examined at each level of the search tree. In this case, a *level* is a particular row of the generator matrix. For further information on the estimation process, refer to [3, 4].

5.1 Size and Shape of the Search Tree

There are 1607 starting matrices where the entries are filled in up to row 8 (of course, rows 21 to 24 are also known). We could use the "simple algorithm" mentioned in the previous chapter, with curr_row = 8. However, knowing nothing about the size of the candidate list at each level after row 8, it is a very good idea to do an estimate by selecting only a few of the good candidates at each level.

Suppose that at level i, we read a candidate list $candlist_i$, generating the survivor list σ_i. Instead of taking for row i, in turn, all $|\sigma_i|$ survivors and continuing the search, suppose that we take only k of them. For each of these k, we determine $\sigma_{i+1,j}$, the list of survivors for level $i+1$ given the particular j'th choice of rows $1 \ldots i$. Then we estimate the total number of survivors at level $i+1$ as

$$\frac{|\sigma_i|}{k} * (|\sigma_{i+1,1}| + \cdots + |\sigma_{i+1,k}|).$$

The estimate of the number of tests required to generate the candidate lists for row $i+2$ is $numtests$ = (total number of survivors at level $i+1$) * (average size of

candidate lists at level $i+1$). For this specific case,

$$numtests = \begin{cases} \frac{|\sigma_i|}{k^2} * & (|\sigma_{i+1,1}| + \cdots + |\sigma_{i+1,k}|) * (|\sigma_{i+1,1}| + \cdots + |\sigma_{i+1,k}|) \\ & \text{if } i \text{ is even.} \\ \frac{|\sigma_i|}{k} * & (|\sigma_{i+1,1}| + \cdots + |\sigma_{i+1,k}|) * (|\sigma_i|) \\ & \text{if } i \text{ is odd.} \end{cases}$$

If we continue the estimation for several consecutive levels, choosing k candidates at every level from i to $i+j$, then the estimate of the total number of survivors at level $i+j$ is:

$$\frac{|\sigma_i|}{k^j} * \begin{pmatrix} |\sigma_{i+1,1}|(\cdots|\sigma_{i+1,1,\ldots,1}| & (|\sigma_{i+j,1,\ldots,1}| + \cdots + |\sigma_{i+j,1,\ldots,k}|) + \cdots + \\ \qquad |\sigma_{i+1,1,\ldots,k}| & (|\sigma_{i+j,1,\ldots,1}| + \cdots + |\sigma_{i+j,1,\ldots,k}|))\cdots) + \\ \qquad \vdots & \\ |\sigma_{i+1,k}|(\cdots|\sigma_{i+1,k,\ldots,1}| & (|\sigma_{i+j,k,\ldots,1}| + \cdots + |\sigma_{i+j,k,\ldots,k}|) + \cdots + \\ \qquad |\sigma_{i+1,1,\ldots,k}| & (|\sigma_{i+j,k,\ldots,1}| + \cdots + |\sigma_{i+j,k,\ldots,k}|))\cdots) \end{pmatrix}.$$

5.2 Estimates Obtained

Using the above, and doing a search using only three survivors at each level, we estimated the number of surviving candidates at each level, and the number of tests required to generate the candidate list for this level. The results are shown in Fig. 5.1.

Figure 5.1: Initial Estimates

row#	#survivors, this test	#survivors, total	#tests req'd
9	$\simeq$ 700	1.1 E6	-
10	5-100	3.2 E7	7.8 E8
11	$\simeq$ 140-160	5.0 E10	2.2 E10
12	0-20	3.8 E10	7.6 E11

As well as computing the linear combinations of the first eight rows of the generator matrix, we can also use the last four rows. If, in fact, we use just *one* of the last four rows (namely row_{21}) then we see a noticeable difference in the number of survivors seen, beginning at level 11. This is because some linear combinations of the first eleven rows will have a weight of 8 in the last 24 columns. Adding these combinations to row_{21}, the weight in the last 24 columns is only 4, and thus the overall weight may be less than 12. Note, however, that this means that we cannot use the list of survivors from level 11 as candidates for row_{13}, since adding row_{21} to a

linear combination of row_{13} with the first ten rows cannot cause its overall weight to become lower. Thus if we are using row_{21} as an extra check, at level 13 we must use the list of survivors from level 9. Fig. 5.2 shows the results of the same estimation process, except also computing the linear combinations involving row_{21}.

Figure 5.2: Estimates using row_{21}

row#	#survivors, this test	#survivors, total	#tests req'd
9	$\simeq$ 700	1.1 E6	-
10	5-100	3.2 E7	7.8 E8
11	$\simeq$ 30	8.3 E8	2.2 E10
12	0-2	1.3 E7	2.2 E10

5.3 Trimming the Search Tree

We can again use the fact that the candidate lists are in lexicographical order to further cut down the search. For every (odd, even)-row pair (row_i, row_{i+1}), we can cut out another 2/3 of the possibilities by using a further ordering. Recall that we are restricting our choice of candidates to those of type $3 \cdot 1^7$, so both row_i and row_{i+1} are of this type. Recall that

$$row_1 + row_2 = 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$$

We shall now show that the codeword $row_i + row_{i+1} + row_1 + row_2$, denoted as $(row_i + row_{i+1})'$, is also a vector of type $3 \cdot 1^7$.

We know that in the first 24 columns, $row_i + row_{i+1}$ must have a weight of 10 or 14 in order for it to be acceptable. If row_i and row_{i+1} have 3 ones in the same block, then $row_i + row_{i+1}$ is either of type $0^3 \cdot 2^5$ or $0 \cdot 2^7$. If row_i and row_{i+1} do not have 3 ones in the same block, then in $row_i + row_{i+1}$, there are two blocks of 2 ones obtained by matching up a block containing 1 one in one vector against a block containing 3 ones in the other. The other blocks will contain either 2 ones or none, again resulting in a vector of type $0^3 \cdot 2^5$ or $0 \cdot 2^7$. If $row_i + row_{i+1}$ is of type $0^3 \cdot 2^5$, then we can permute the blocks so that the 3 blocks of zeros are in the first three positions; adding it to the second row of the generator matrix we obtain a vector of

low weight. So $row_i + row_{i+1}$ must be of type $0 \cdot 2^7$, and $(row_i + row_{i+1})'$ is of type $3 \cdot 1^7$.

Because $(row_i + row_{i+1})'$ is of type $3 \cdot 1^7$ and because it must satisfy the same constraints, it must therefore be in the list of candidates for row_1. Since the candidate list is ordered, we can insist that $(row_i + row_{i+1})'$ is further down the list than row_{i+1}, thereby throwing out 2/3 of the possibilities for this pair.

Chapter 6

Connection Machine Algorithm

This chapter describes the method used to extend each of the 1607 matrices completed to row 8. Even with the shortcuts, the previously-mentioned algorithm is still very slow - extending only one of the starting matrices takes around 1 day to run on a DEC 5000/200 machine. For this reason, we decided to use the CM-2 and use a different kind of algorithm to take advantage of the different architecture. As a result, we completed the entire search on the CM-2 in about 12 CPU hours.

6.1 Architecture of the Connection Machine

The Connection Machine is a Single Instruction, Multiple Data-Stream, or *SIMD* computer - one instruction is carried out in parallel across all "active" processors. The version that I used, a CM-2, has a *back end* consisting of 8K processors each of which have their own memory, and a *front end* which is a SPARC-2. A processor is active if a particular bit (the *context bit*) in that processor is on; this may be controlled by the user. Despite the fact that there are only 8K *physical* processors, we can allocate *virtual* processors; although all the work will not be done concurrently in this case, it will "appear" so to the programmer. The language that I used to write programs for the CM is C/Paris. Paris is a very low-level language for programming the back end.

6.2 Background Information

As previously mentioned, there are several ways to speed up the search. Most of these use the fact that at each level, the candidate list is ordered, then insist upon a particular order between rows of the generator matrix. To generate rows 9 to 20, we consider together sets of four rows (*quads*) $(row_{4i+1}, row_{4i+2}, row_{4i+3}, row_{4i+4})$, insisting upon a certain ordering within the rows of each quad and between quads themselves.

Each vector in the candidate list for row 9 is also a candidate for rows 10 to 20. Of these rows, row_9 intersects only row_{10} in the last 24 columns. This is true in general for any (odd, even)-row pair (row_i, row_{i+1}) between rows 5 and 20. For this reason, for any odd i, $5 \leq i \leq 19$, the list of compatible pairs (row_i, row_{i+1}) is different from the list of compatible pairs (row_i, row_j), where $i < j < 20$ and $j \neq i + 1$.

Now consider the list of compatible (row_9, row_{11}) pairs. We could say that row_9 and row_{11} are compatible if every linear combination *combo* of the first eight rows and row_9 and row_{11} has acceptable weight. But we also know the last four rows of the generator matrix. Until row 11, adding a linear combination of the last four rows to *combo* cannot give a vector of lower weight. At row 11, however, it is possible that *combo* has a weight in the last 24 columns of 8; adding row_{21} will reduce this to 4 and may give a vector of overall low weight.

Assume that we know rows 1 to 8 and are trying to determine the list of compatible (row_9, row_{13}) pairs. In this case, adding a linear combination of the last four rows cannot make a difference since there is no weight-12 linear combination of these rows that has ones in columns 25 to 33 and 37 to 39.

We use two *compatibility matrices*, $C_{9,11}$ and $C_{9,13}$. For two specific row 9 candidates x and y, $C_{9,11}[x][y]$ is TRUE if all linear combinations of the first eight rows, row_{21} and x and y have acceptable weights. Note that this also implies that $C_{9,11}[y][x]$ is TRUE. Similarly, $C_{9,13}[x][y]$ is TRUE if all linear combinations of the first eight rows and x (treated as row 9) and y (treated as row 13) have acceptable weights.

To generate the list of $(row_9, row_{10}, row_{11}, row_{12})$ quads, we first find the list of possible row9-row10 pairs, then use $C_{9,11}$ to determine the compatibility of pairs

against each other. Every element in the list is also a candidate for the quads $(row_{13}, row_{14}, row_{15}, row_{16})$ and $(row_{17}, row_{18}, row_{19}, row_{20})$. We use row_{21} as an extra check for the quad $(row_9, row_{10}, row_{11}, row_{12})$; for the other quads, this is translated to another combination of the last four rows, namely row_{23} for the quad $(row_{13}, row_{14}, row_{15}, row_{16})$ and $(row_{22} + row_{23})$ for $(row_{17}, row_{18}, row_{19}, row_{20})$. We use $C_{9,13}$ to determine the compatibility of quads against each other.

We insist upon the following orderings:

- Within any (odd-even)-row pairs $(row_i, row_{i+1}), i \geq 9$, the order within the candidate list must be row_i, row_{i+1}, $(row_i + row_{i+1})'$, where x' denotes the vector x complemented on the first 24 columns.

- The pair (row_{4k+1}, row_{4k+2}) must come before the pair (row_{4k+3}, row_{4k+4}) in the list of possible pairs for any $k \geq 2$.

- The quad $(row_{4k+1}, row_{4k+2}, row_{4k+3}, row_{4k+4})$ must appear before the quad $(row_{4k+5}, row_{4k+6}, row_{4k+7}, row_{4k+8})$ in the list of possible quads for any $k \geq 2$.

Using the compatibility matrices in the described manner, we do not check all possible combinations of the rows of the generator matrix. In particular, we are missing:

- any linear combinations involving the last four rows, excepting those explicitly mentioned above

- any linear combinations involving vectors from more than two (odd, even)- row pairs - for example, $row_9 + row_{11} + row_{13}$.

Hence even if the CM algorithm returns what it considers to be a "good" matrix there may still be some combinations of the rows which have unacceptable weights; we must, therefore still check the weight enumerator of all the matrices returned by the CM algorithm.

Since the last 24 columns, or "right-hand side", of rows 9 to 20 is already known, we only need to consider vectors of length 24 representing the first 24 columns of these

rows. In addition, each of these rows must be of type $3 \cdot 1^7$, so for each 3-column-wide block there are only four possibilities - one in which the block has 3 ones, and three in which the block has 1 one. Since there are eight such blocks, we can use a table of size 2^{16} to store information about each possible vector; we use the following mapping between length-24 and length-16 vectors:

$$\begin{aligned} 111 &\mapsto 00 \\ 100 &\mapsto 01 \\ 010 &\mapsto 10 \\ 001 &\mapsto 11. \end{aligned}$$

Note that for two length-16 vectors x_{16} and y_{16} and their corresponding length 24 vectors x_{24} and y_{24}, $x_{16} + y_{16}$ maps to $(x_{24} + y_{24})'$.

To build the compatibility matrices, we consider all length-16 vectors v in the table which, when converted to length-24, are doubly-even. These are the possible length-16 vectors which, when mapped to length-24, can result from the addition of a row 9 candidate $r9$ with either a row 11 candidate $r11$ or with a row 13 candidate $r13$; thus we look at the length-16 mapping of $(r9 + r11)'$ or $(r9 + r13)'$. We have to check that each linear combination of v with the first eight rows and, if necessary, row_{21}, has acceptable weight. The complement of the length-24 vector x is denoted as x' and is accomplished by adding rows 1 and 2 to x, so it is one of the combinations that we check.

6.3 Description of Algorithm

A description of, and pseudocode for, the algorithm used on the Connection Machine follows. Note: a † after a particular command indicates that the instruction is carried out across all processors in parallel. In the following type of situation:

if B †
then
    S_1
else

S_2

the condition B is checked in all processors in parallel. The command(s) in S_1 are carried out in parallel across all processors for which B is true; the command(s) in S_2 are carried out in parallel across all processors for which B is false.

For each matrix complete to row 8, we first compute the list of possible row 9's, storing them in R9list:

```
compute all linear combinations of rows 1 to 8 and store each one in a
    separate processor;
R9list := null;
for each candidate c in candlist do
begin
    broadcast c across the set of linear combinations;
    if c + combo has acceptable weight for each combination combo
    then
        add c to end of R9list;
end;
```

To build the compatibility matrices, we need to find which codewords are valid as either $(row_9 + row_{11})I$ or $(row_9 + row_{13})I$. We do this in parallel, as follows:

```
for each doubly-even vector x of length 24, such that each block
    of x contains either 3 ones or 1 one do
begin
    compute LHSwt; {weight in 1st 24 columns of each combination} †
    R11RHSwt := 0; †
    R13RHSwt := 4; †
    if combination includes row 5 or row 6 then †
    begin
        R11RHSwt := R11RHSwt + 2;
        R13RHSwt := R13RHSwt + 2;
```

if combination includes row 7 or row 8 *then* †

begin

R11RHSwt := R11RHSwt + 2;

R13RHSwt := R13RHSwt + 2;

end;

if R11RHSwt < 4 †

then

R11RHSwt := R11RHSwt + 4;

if LHSwt + R11RHSwt ≥ 12 and is divisible by 4 for all combinations

then

$x.R9\_R11$:= TRUE;

else

$x.R9\_R11$:= FALSE;

if LHSwt + R13RHSwt ≥ 12 and is divisible by 4 for all combinations

then

$x.R9\_R13$:= TRUE;

else

$x.R9\_R13$:= FALSE;

end;

We use this information to build the compatibility matrices:

for i := 1 *to* size of R9list *do*

begin

c_1 := i'th candidate in R9list;

for j := i *to* size of R9list *do*

begin

c_2 := j'th candidate in R9list;

$C_{9,11}[i][j] := C_{9,11}[j][i] := (c_1 + c_2).R9\_R11$;

$C_{9,13}[i][j] := C_{9,13}[j][i] := (c_1 + c_2).R9\_R13$;

end;

end;

When using the compatibility matrices, we need the list of row9/row10 tuples. We store these tuples in triple_list.

triple_list := null;
for each candidate $r9$ in R9list *do*
  for each candidate $r10$ in R9list *do*
    if $r10$ comes further down R9list than $r9$ *and*
      $(r9 + r10)'$ is in R9list *and*
      $(r9 + r10)'$ comes further down R9list than $r10$
    then
      add $(r9, r10, (r9 + r10)')$ to end of triple_list;

The procedure *makelist* uses the compatibility matrix *FEcompat* to determine which tuples, of size *startsize1*, in *startlist1* are compatible with which tuples, of size *startsize2*, in *startlist2*. The resulting tuples, of size *newsize* are then stored in *newlist*. It is assumed that startsize2 $\leq$ startsize1. Furthermore, if startsize2 < startsize1 then the last startsize2 components of startlist1 must be elements of startlist2.

Procedure makelist(FEcompat, newlist, startlist1, startlist2,
    newsize, startsize1, startsize2)
  load each tuple in startlist1 into field *CMtuple* of a separate processor;
  CMcompat := all-1 vector of length *sizeof*(R9list); †
  for each column k of FEcompat *do*
    AND k into CMcompat in each processor referencing k;
  {processor having CMtuple = $(x_1, \ldots, x_{startsize1})$ now has }
  {CMcompat = FEcompat$[x_1] \wedge \ldots \wedge$ FEcompat$[x_{startsize1}]$ }
  newlist := null;
  for each tuple $(y_1, \ldots, y_{startsize2})$in startlist2 *do*
  begin

context_bit := CMcompat[y_1] AND ... CMcompat[$y_{startsize2}$];

for each processor with context_bit = TRUE *do*

if ($y_1, \ldots, y_{startsize2}$) comes further down startlist2

than ($x_{startsize1-startsize2+1}, \ldots, x_{startsize1}$)

then

add ($x_1, \ldots, x_{startsize1}, y_1, \ldots, y_{startsize2}$) to end of newlist;

end;

The complete algorithm follows:

For each matrix completed to row 8 *do*

begin

if matrix is type a *then*

candlist := type_a_list;

else

candlist := type_b_list;

compute list of possible row 9's and store in R9list;

determine which codewords are valid as either $(row9 + row11)'$

or $(row9 + row13)'$;

compute list of R9/R10 tuples and store in triple_list;

build compatibility matrices $C_{9,11}$ and $C_{9,13}$;

makelist($C_{9,11}$, hexlist, triplelist, triplelist, 6, 3, 3);

makelist($C_{9,13}$, dozenlist, hexlist, hexlist, 12, 6, 6);

makelist($C_{9,13}$, answerlist, dozenlist, hexlist, 18, 12, 6);

end.

Chapter 7

Results

This chapter summarizes the results of the search for a $(48, 24, 12)$ doubly- even self-dual code carried out on the Connection Machine, including how we determine if any of these is equivalent to the only known code of these parameters.

7.1 Extended Quadratic Residue Code

Because the only known $(48, 24, 12)$ doubly-even, self-dual code is the Extended Quadratic Residue code, we naturally want to know if we find this code during our search. To do this, we consider the 17296 codewords of weight 12 obtained from the Extended Quadratic Residue code.

There are three generators for the automorphism of the code [5, page 491], namely:

$$\sigma_1 : \left\{ \begin{array}{lcll} \infty & \mapsto & \infty & \\ i & \mapsto & i+1 & \pmod{47} \end{array} \right\}$$

$$\sigma_2 : \left\{ i \mapsto \frac{-1}{i} \pmod{47} \right\}$$

$$\sigma_3 : \left\{ \begin{array}{lcll} \infty & \mapsto & \infty & \\ 0 & \mapsto & 0 & \\ i & \mapsto & ir^2 & \pmod{47} \end{array} \right\}$$

where r is a primitive root mod 47, i.e. r satisfies $r^{46} \equiv 1 \pmod{47}$ and $r^i \not\equiv 1 \pmod{47} \; \forall \, 0 < i < 46$. We use $r = 45$, so $r^2 = 4 \pmod{47}$.

All of the matrices returned by the Connection Machine algorithm have the same basic structure, as shown in Fig. 3.4. All of the codewords in the generator matrices are of weight 12, and furthermore the matrices are each broken up into 3

basic parts. In order to determine if we have generated codes equivalent to the Extended Quadratic Residue code, we consider only the weight-12 words in the Extended Quadratic Residue code, and try to create generator matrices in the same basic form as those we have produced.

To determine if every weight-12 codeword from the Extended Quadratic Residue code matrix can be mapped to every other, we use an "expanding horizon": we choose any codeword x and recursively apply σ_1, σ_2 and σ_3 to it until we obtain a previously seen codeword. At the end, we check if all codewords in the list have been seen. The 17296 codewords are divided into three orbits, one of size 8648 and two of size 4324. This means that there are three possible first rows. We use as orbit representatives codeword #12973, codeword #14877 and codeword #6300 respectively. For each row_1, we find the list *clist* of 630 weight-12 codewords which do not intersect row_1. For the case we are considering, there must be a subcode contained in the last 4 rows which intersects neither row_1 nor row_2; this subcode has 14 weight-12 vectors. Thus a given codeword in *clist* is a possible row_2 if 14 other codewords in *clist* do not intersect it. For each row_1, we break up the list of possible row_2's into orbits under the action of the automorphism group of row_1.

There are a total of fourteen (row_1, row_2) combinations once we break the lists of row_2's into orbits. For row_1 = #12973, the row_2's are in six different orbits. For row_1 = #14877, the row_2's are in five different orbits. For row_1 = #6300, the row_2's are in three different orbits.

To find the rest of the "top" part of the generator matrix for each of the above, we find all codewords in the list of 17296 that intersect only row_1 or row_2. There are a total of 14 rows in the "top" part; these are all the weight-12 linear combinations of the first four rows. There are 192 rows in the "middle" section, 24 of which are of type $3 \cdot 1^7$ and 168 of type $2^3 \cdot 0 \cdot 1^4$. The "bottom" section consists of all weight 12 codewords which intersect neither row_1 nor row_2.

So we now have a total of 14 matrices of size 220×48 formed from the extended quadratic residue code and with the required structure. Of these, there are five non-isomorphic matrices.

7.2 Results from the Connection Machine Algorithm

Of a total of 1607 non-isomorphic matrices complete to row 8, 9 are of *type a* and 1598 are of *type b*. We used the Connection Machine to try to complete the rest of the rows for each of the 1607 cases.

Using the Connection Machine algorithm we found a total of 1326 complete matrices. Of these, 392 are of *type a* and 934 are of *type b*. As previously mentioned, it is possible that some of these have codewords of invalid weight — a weight less than 12 or not doubly-even. When we checked the weight enumerator of each of these matrices, we found that only 20 are acceptable - 11 *type a* and 9 *type b*.

For each of the 1326 cases, we found the 220×48 matrix composed of three parts:

- a "top" part of 14 rows, consisting of all the weight-12 linear combinations of the first four rows of the generator matrix
- a "middle" part of 192 rows, consisting of:
  - the middle 16 rows of the generator matrix
  - $row_i + row_{i+1}$ for each odd-row - even-row pair of the middle 16 rows (row_i, row_{i+1})
  - for each of these (which are all of type $3 \cdot 1^7$) the corresponding 7 vectors of type $2^3 \cdot 0 \cdot 1^4$.
- a "bottom" part of 14 rows, consisting of all the weight-12 linear combinations of the last four rows of the generator matrix

We ran an isomorphism check on the above 1326 matrices, and of course also compared them to the 5 matrices of the same structure which arise from the Extended Quadratic Residue code. We found that 446 are non-isomorphic; 47 of these are of *type a* and 399 are of *type b*. Of the 20 matrices with acceptable weight enumerator, there are 5 non-isomorphic matrices — 1 of *type a* and 4 of *type b*. Each of these 5 is isomorphic to one of the matrices from the Extended Quadratic Residue code.

7.3 Conclusion

The search for extremal $(48, 24, 12)$ doubly-even codes can be divided into three cases. We completed a search assuming one of these cases, finding only codes that are isomorphic to the Extended Quadratic Residue code. The other two cases are still open.

Bibliography

[1] Assmus, E. F. Jr. and Mattson, H. F. Jr., New 5-designs, *J. Combin. Theory*, 6:122-151, 1969.

[2] Berlekamp, E. R., *Algebraic Coding Theory*, McGraw-Hill, 1968.

[3] Knuth, D. E., Estimating the efficiency of backtrack programs, *Mathematics of Computation*, 29:121-136, 1975.

[4] Lam, C. W. H., *Computational Combinatorics - A Maturing Experimental Approach*, course notes, Concordia University, 1992.

[5] MacWilliams, F. J. and Sloane, N. J. A., *The Theory of Error-Correcting Codes*, North-Holland, 1977.

[6] Mallows, C. L. and Sloane, N. J. A., An upper bound for self-dual codes, *Info. and Control*, 22:188-200, 1973.

[7] Pless, V., *Introduction to the Theory of Error-Correcting Codes*, John Wiley and Sons, 1989.

[8] Van Lint, J. H., *Coding Theory*, Springer-Verlag, 1971.

Appendix

This contains the CM programs and header files used to find complete matrices given the first eight rows.

The mainline for the search is in the file *bldcode.c*, which is compiled to *bldcode*. Three parameters are required — the command line
cmattach bldcode main cand_file input_file
will "attach" to the CM, which will then run the program *bldcode* with option *main*. The list of candidates for the remaining rows of the generator matrix should be in the file *cand_file*. Further required information should be in the file *input_file*, which should contain the following information, in order:

- number of starting matrices in this batch
- row 6
- (row 7, row 8) for each of the starting matrices to be tested in this batch

Rows 1 to 5 are the same for all input matrices, but the sixth row is different for *type a* and *type b* matrices, so the searches for *type a* and *type b* matrices must be run in separate batches.

```
/*************************************************************************
******/
/*
/* bldextended.h
/*
/* */

CM_geometry_id_t curr_geom; /* current geometry */
CM_vp_set_id_t curr_vp; /* current vp set */

#define c8_id_fldlen 12
/* length of CM field containing the C8id of tuple-members.
/* Note: 700 is the expected C8 size, but 12 bits are allowed
/* just in case.
/*   */
#define maxids 12 /* max #ids contained in any tuple */
#define max_tuples 15000 /* maximum size of tuple array */
typedef unsigned short tuple[maxids][max_tuples];
  /* A particular vector of candidate ids in the C8 set.
  /* There are 1.5*N ids per tuple (N currently 2, 4, or 8)
  /* The k'th tuple, for a tuple composed of n ids, is stored in
  /* tuple[0][k]...tuple[n-1][k]; tuple[n]...tuple[maxids-1] are
  /* all-zero vectors.
  /* */
  .

int build_extended_pairs(
  /* If a tuple represents 2 or more candidates which are known to
  /* be mutually compatible, then a necessary (but not sufficient)
  /* condition that 2 tuples are compatible is that each row in one
```

```
  /* of the tuples is compatible with all of the rows in the other
  /* tuple.  This routine does this test.
  /* On entry each VP represents a tuple and contains a column of
  /* the multicompat containing a 1 for each candidate compatible
  /* with all rows of that tuple. The column is in field cmcompat.
  /* This function then passes through the list of test tuples,
  /* testing each against each VP, and builds a list of
  /* extended_pairs which contains the sequence number of each pair
  /* of compatable tuples.  The return value is the number of
  /* extended pairs found.
  /* Assumption: cm_test_tuple is contained in fe_test_tuple.
  /* Then, due to ordering, only look at processors containing
  /* tuples further down the list than the corresponding part
  /* of fe_test_tuple[][ttndx].
  /* The VP set must be correct on entry.
  /* */
#ifdef SMARTC
  tuple fe_test_tuple;          /* set of FE tuples to test */
  int fe_tuple_size;            /* #candidates in each FE tuple */
  CM_field_id_t cm_test_tuple; /* set of tuples in CM */
  int cm_tuple_size;            /* #candidates in each CM tuple */
  tuple extended_pairs; /* set of extended pairs found */
  int max_extended_pairs; /* Size of extended_pair vector */
    /* Stop and return -Ntt if we find too many to fit, where Ntt
    /* = number of test_tuples whose results are completely
    /* included in extended pairs, else return 0. */
#endif
  );

void
```

```
create_valid_c8_triples(
#ifdef SMARTC
tuple triplevec;
#endif
);
  /* Return the list of valid pairs from globals C8list, which has
  /* C8actual entries in it. Element 0 has the number of tuples in
  /* the list. The rest of the elements have the 16-bit addresses
  /* of the 3 elements associated with each pair.  */

CM_field_id_t send_tuples_to_cm(
#ifdef SMARTC
  tuple tuples,
  int tuplesize
#endif
);
  /* send each of numtuples tuples of size tuplesize stored in
  /* tuples to a separate processor. There are tuples[0][0] tuples
  /* in all. Appropriate geometry should be defined beforehand e.g.
  /* when sending triples, a vp-ratio of 2 is required (12 000
  /* triples expected). */
```

```
/* convert.h */

typedef struct{  /* records potential cands */
int val24; /* 24-bit value of cand */
int val16; /* 16-bit value of cand */
int C8id;  /* mapping into list of good
/* candidates for C8 */
      }candrec;


int read_cands(/* char *filename, candrec *cands, int maxcands */);
  /* read all the candidates from file filename into the
  /* array at cands. maxcands is the size of the array, so
  /* we are not allowed to read more than this.
  /* Return the number of cands read. Return maxcands+1
  /* if there are more in the file than the array can handle.
  /* */

void                    /* convert from 16-long to 24-long format */
l16tol24(
#if 0
  int A[],
  int B[]
#endif
);

void /* convert from 16-bit to 16-long form */
btol16(
#if 0
  int b,
```

```
  int A[]
#endif
);
```

```
/* multicompat.h -
/* Module to distribute the multi-compat matrix to the VPs.
/* A multi-compat is made from a compatability matrix by storing
/* the AND of several columns of the compatability matrix into
/* each VP.  The particular columns which are combined within
/* each VP depends on the values of a set of fields within the
/* individual VPs.
/* */

CM_field_id_t cmcompat;  /* Field in CM to store columns of the
 /* multi compat */


void distribute_multi_compat(
#if 0
  int mask; /* Pick out bit to use from valid */
  int shift; /* Shift count to get bit in low order position */
  CM_field_id cand_flds;  /* Vector of field ids pointing to the
    /* fields to be tested. */
  int nflds; /* Number of fields to be tested. */
  int fld_len; /* length of the test fields. All are the same. */
  int C8_cand_16; /* Vector of candidates to use to form
/* multi-compat */
  int Ncands; /* Number of candidates to process */
  int Ntuples; /* Number of tuples that will be tested; only the first
   Ntuples processors should contain columns of compat
   matrix - the others should have all zeroes */
#endif
  );
```

```
/* paratest.h -
/* Control the testing of a set of candidates against a context in
parallel. */

#define COORDSIZE 10
  /* length of a coord field (codeid and batchid). */

#define NUM_PROCS 8192
#define STRING_LEN 24
#define POPCOUNT_LEN 6
#define SINGLY_EVEN_LEN POPCOUNT_LEN - 1
#define DOUBLY_EVEN_LEN POPCOUNT_LEN - 2
#define RESULT_STR_LEN 4

typedef struct {  /* Define the geometry for the parallel structure
  /* to be set up. */
  int width;  /* of the code space. == 256 at row 8 */
  int Nbatch; /* Number of parallel batches to do at once. Must be
                 power of 2 */
  CM_geometry_id_t geometry;  /* to be used for reading back the
 results. */
  CM_vp_set_id_t build_vp_set;
  CM_field_id_t codeid, batchid;
    /* identify the codeid (= news coord within width, 0 base) and
    /* batchid (= news coord with batches, 0 base) that each
    /* processor is in. Codes index along the zero'th coord, batches
    /* along the first. */
} paratest_rec;

extern paratest_rec *define_paratest(/* int width, int Nbatch */);
```

```
  /* Return a paratest_rec which will allow parallel testing of
  /* candidates in a code space of size width, with Nbatch results
  /* in parallel. */
  /* NOTE: this routine will change the vpset and geometry!!!! */

extern void do_paratest(
/* paratest_rec *ptr, int *cands, int Ncands, char *bits */);
  /* Do a parallel test of the set of Ncands candidates in cands[]
  /* according to the parallel parameters in ptr, which must have
  /* been created by define_paratest. On return, the results of the
  /* tests of the candidates will be stored as bits in bits[], with
  /* 1 byte per candidate.
  /* The bits set will be cand_r9, r9_r11, and r9_r13, as defined
  /* in table64.h.  NOTE: this routine will change the vpset and
  /* geometry!!!! */
  /* NO candidate with a weight > 20 is allowed (length of count
  /* field!!) */

extern void setup_basis(
/* paratest_rec *code, int basis[], int numbasis */);
  /* Build the basis code for testing acceptability of various
  /* strings with paratest. */
```

```
/* table64.h -
/* Describe the record for the table of 64K 16-bit values. */

typedef struct {  /* For each possible 16-bit value */
  int val24; /* 24-bit equivalent of 16-bit value */
  short canid;  /* Candidate id if this value is a candidate
    /* at R6 or -1 */
  short C8id;   /* position in list of C8 candidates if good;
   -1 if not good for C8      */
  short valid;  /* Records the validity of this value in
    /* several contexts, 1 bit per context. Contexts
    /* selected by constants below. */
  short popcount; /* population count of candidate */
}lhscodes;

/* Identify bits in lhscode.valid */

#define cand_r7 1
  /* Bit set if one of 3700 initial candidates */
#define cand_r9 2
  /* bit set if one of 700 candidates in context of row 8 */
#define r9_r11 4
  /* Bit set if this represents the XOR of 2 valid r9 candidates,
  /* and is valid as a r9-r11 pair. */
#define r9_r13 8
  /* Bit set if this represents the XOR of 2 valid r9 candidates,
  /* and is valid as a r9-r13 pair. */

#define tablesize 65536 /* 2^16 = #cand to be converted */
lhscodes table64k[tablesize];
```

```
#define MAX_C6_SIZE 4000
#define C8listsize MAX_C6_SIZE

extern int C8_cand_16[C8listsize];
/* 16-bit vals of remaining cands at R9 */
extern int C8actual; /* Number of items actually there */
extern int C6actual; /* Number of items actually there */

FILE *r7r8file; /* contains all r7-r8 combinations to test */

int
maketable(/* char *filename, candrec cands[], int maxcand */);
/* read candidates from 'filename'; maxcands = max #candiates
/* fill up table64k. table64k[i] has canid = to position of
/* i in 'filename's candidate list (if in list), in which case
/* we also have valid = 1 */
#ifndef DEBUG
#define prism_alloc(len) CM_allocate_heap_field(len)
#else
#define prism_alloc(len) CM_allocate_heap_field(32)
#endif

#define Abort(msg) { fprintf(stderr,"%s\n",msg); exit(1); }

#define assert(e) if (!(e))Abort("assert failed");

#define timing
```

```
/*****************************************************************/
/*   bldcode.c -                                                 */
/*   Mainline for search for generator matrix for 48-24-12 code  */
/*   assuming that there are 4 special rows at the start and end */
/*   of the matrix.                                              */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <cm/paris.h>
#include <cm/cmsr.h>
#include "paratest.h"
#include "table64.h"

#define Abort(msg) { fprintf(stderr,"%s\n",msg); exit(1); }

void main(argc, argv)
  char **argv;
  int  argc;

{
  char *option;

  if (argc < 2)Abort("Option required");
  option = argv[1];

/****************** Init */

  fprintf(stderr,"warm booting..."); fflush(stdout);
```

```
  CM_init();
  fprintf(stderr,"done\n");

  printf("Testing option *****%s*****\n", option);
  if (strcmp(option, "paratest") == 0)
  {
    test_paratest();
  }
  else if (strcmp(option, "main") == 0)
  {
    if(argc < 4) Abort("Filenames required\n");
    build_it(argv[2], argv[3]);
  } else {
    printf("%s unknown!!\n", option);
  }
  printf("Finis\n");
}  /* main */
```

```
/* bldextended.c -
/* Module to test a set of frontend tuples against a set of cm
/* tuples, given that the multicompat is present. */

#include <stdio.h>
#include <math.h>
#include <cm/paris.h>
#include <cm/cmsr.h>
#include "paratest.h"
#include "table64.h"
#include "bldextended.h"
#include "multicompat.h"


int build_extended_pairs(fe_test_tuple, fe_tuple_size, cm_test_tuple,
 cm_tuple_size, extended_pairs, max_extended_pairs)
  tuple fe_test_tuple; /* set of FE tuples to test */
  int fe_tuple_size; /* #candidates in each FE tuple */
  CM_field_id_t cm_test_tuple; /* set of tuples in CM */
  int cm_tuple_size; /* #candidates in each CM tuple */
  tuple extended_pairs; /* set of extended pairs found */
  int max_extended_pairs; /* Size of extended_pair vector */
    /* Stop and return -Ntt if we find too many to fit, where
    /* Ntt = number of test_tuples whose results are completely
    /* included in extended pairs, else return 0. */

{ int nfound;
  int ttndx, valndx;
  int numtuples; /* #of FE tuples to test */
  int full_count = 0, count;
```

```
  int i, j, firstnews;
  CM_field_id_t news_id;
#define coord_len 14

#ifdef info
  printf(
    "Build_extended_pair(fe_tuple_size = %d, cm_tuple_size = %d)\n",
fe_tuple_size, cm_tuple_size);
#endif

  nfound = 0;
  numtuples = fe_test_tuple[0][0];
  news_id = CM_allocate_heap_field(coord_len);
  CM_set_context();
  CM_my_news_coordinate_1L(news_id, 0, coord_len);

#ifdef timing
  CM_timer_clear(2);
  CM_timer_start(2);
#endif
  for (ttndx = 1; ttndx <= numtuples; ttndx++)
  {
    CM_set_context();
    /* Assumption: cm_test_tuple is contained in fe_test_tuple.
       Then, due to ordering, only look at processors containing
       tuples further down the list than the corresponding part
       of fe_test_tuple[][ttndx] */
    CM_u_ge_constant_1L(
cm_test_tuple,
fe_test_tuple[fe_tuple_size-cm_tuple_size][ttndx], c8_id_fldlen);
```

```
    CM_logand_context_with_test();

    /* AND each bit of cmcompat corresponding to a member of
       the FE tuple */
    for (valndx = 0; valndx < fe_tuple_size; valndx++)
      CM_logand_context(
CM_add_offset_to_field_id(cmcompat, fe_test_tuple[valndx][ttndx]));

    if ((count = CM_global_count_context()) > 0)
    /* valid extended pair(s) found in this iteration */
    {
      if(full_count >= max_extended_pairs) /* too many to fit */
return -full_count;

      /* Pick out results here */
      for(i = full_count; i < full_count + count; i++)
      {
/* return news_id of first processor with context-bit set to FE */
firstnews = CM_global_u_min_1L(news_id, coord_len);
        CM_u_write_to_processor_1L(
CM_fe_make_news_coordinate(curr_geom, 0, firstnews),
CM_context_flag, 0, 1);

for(j = 0; j < fe_tuple_size; j++)
  extended_pairs[j][i+1] = fe_test_tuple[j][ttndx];
for(j = 0; j < cm_tuple_size; j++)
  extended_pairs[fe_tuple_size+j][i+1] = CM_u_read_from_processor(
CM_fe_make_news_coordinate(curr_geom, 0, firstnews),
CM_add_offset_to_field_id(cm_test_tuple, j*c8_id_fldlen),
c8_id_fldlen);
```

```
#ifdef info
if(i < 200)
{
  for(j = 0; j < cm_tuple_size+fe_tuple_size; j++)
    printf("%d\t", extended_pairs[j][i+1]);
  printf("\n");
}
#endif
      }
      nfound++;
      full_count += count;
    }
  }
#ifdef timing
  CM_timer_stop(2);
  printf("Time required to determine #extended pairs + use send to
read results:\n");
  CM_timer_print(2);
  printf("\n");
#endif

  extended_pairs[0][0] = full_count;
#ifdef info
  printf("build_extended_pair: %d live scans\n", nfound);
  printf("\tfull count of good %d-es: %d\n",
fe_tuple_size+cm_tuple_size, full_count);
#endif
  return 0;
}  /* build_extended_pairs */
```

```
void
create_valid_c8_triples(triplevec)
tuple triplevec;
{ int i, j, xor;
  int np;


  np = 0;
  for (i = 0; i < C8actual; i++)
    for (j = i + 1; j < C8actual; j++)
    {
      xor = C8_cand_16[i] ^ C8_cand_16[j];
      if ((xor > C8_cand_16[j]) && (table64k[xor].valid & cand_r9))
      {
/* (C8_cand_16[i], C8_cand_16[j], xor) is a valid tuple; increment np
   by 1 and put C8id's of each member of the tuple in triplevec,
   which is passed back and eventually sent to BE */

        np++;
triplevec[0][np] = i;
triplevec[1][np] = j;
triplevec[2][np] = table64k[xor].C8id;
      }
    }
#ifdef info
  printf(" %d triples created\n", np);
#endif
  triplevec[0][0] = np;
} /* create_valid_r9_r10_pairs */
```

```
CM_field_id_t send_tuples_to_cm(tuples, tuplesize)
tuple tuples;
int tuplesize;
  /* send each of numtuples tuples of size tuplesize stored in
  /* tuples to a separate processor. Appropriate geometry should be
  /* defined beforehand.
  /* */
{ int i, numtuples, temp;
  CM_field_id_t tupleids;
  int offset[1], start[1], end[1], axis[1], dim[1];

#ifdef timing
  printf("send_tuples_to_cm:\n");
  CM_timer_clear(2);
  CM_timer_start(2);
#endif

  tupleids = CM_allocate_heap_field(tuplesize*c8_id_fldlen);
  numtuples = tuples[0][0];

  offset[0] = 1;
  start[0] = 0;
  end[0] = numtuples;
  axis[0] = 0;
  dim[0] = numtuples;
  for(i = 0; i < tuplesize; i++)
    CM_u_write_to_news_array_1L(tuples[i], offset, start, end, axis,
CM_add_offset_to_field_id(tupleids, i*c8_id_fldlen),
c8_id_fldlen, 1, dim, sizeof(unsigned short));
```

```
#ifdef timing
  CM_timer_stop(2);
  CM_timer_print(2);
  printf("\n");
#endif
  return tupleids;
} /* send_tuples_to_cm */
```

```
/*************************************************************************/
/* buildexec.c -
/* Executive for testing (and running) the entire program.
/* Entered via entry point build_it from bldcode.c
/* with option "main".
/*
/*************************************************************************/


#include <stdio.h>
#include <math.h>
#include <cm/paris.h>
#include <cm/cmsr.h>
#include "paratest.h"
#include "table64.h"
#include "convert.h"
#include "bldextended.h"

#define numbasis  8     /* number of basis vectors at current level */
#define width     256   /* = 2^8 = size of codespace at curr level  */
#define numbat    32    /* = 2^13/2^8 = #batches to run at a time */

int basis[numbasis];
int C8_cand_16[C8listsize]; /* remaining candidates at R9 */
int C8actual;                   /* Number of items actually there */
int C6actual;                   /* Number of items actually there */
candrec c6set[MAX_C6_SIZE];
    /* Hold all the candidates as of row 6 (= candidates for row 7) */
```

```
static init(file1, file2)
char *file1, *file2;
{
  C6actual = maketable(file1, c6set, MAX_C6_SIZE);

  r7r8file = fopen(file2, "r");
  if(r7r8file == NULL)
  {
    printf("Error - cannot open %s\n", file2);
    exit(1);
  }
}  /* init */

void
re_init(file1)
char *file1;
{
  C6actual = maketable(file1, c6set, MAX_C6_SIZE);
}

void
define_vp(vp_ratio)
int vp_ratio;
{ int dim[1];
          .

  dim[0] = vp_ratio*NUM_PROCS;
  curr_geom = CM_create_geometry(dim, 1);
  curr_vp = CM_allocate_vp_set(curr_geom);
}
```

```
void
display_matrices(answer_ids, size)
/* display all vectors (size vectors per matrix)
/* found in answer_ids[] */
tuple answer_ids;
int size; /* #vectors in matrix */
{ int i, j, k;
  int vec16[16], vec24[24];

  printf("%d complete matrices found\n", answer_ids[0][0]);
  for(i = 1; i <= answer_ids[0][0]; i++)
  {
    for(j = 0; j < size; j++)
    {
      btol16(C8_cand_16[answer_ids[j][i]], vec16);
      l16tol24(vec16, vec24);
      for(k = 0; k < 24; k+=3)
printf("%d%d%d ", vec24[k], vec24[k+1], vec24[k+2]);
      printf("\n");
    }
    printf("\n");
  }
}

void build_it(file1, file2)
char *file1, *file2;
  /* Try to build a (48,24,12) code using candidate list in file1 and
     f7-r8's in file2. The CM has been warm booted. */
{ int cntr11, cntr13;
  int i, j, t, row7_8_choice, maxtests;
```

```
CM_field_id_t c8_triple_flds, c8_hex_flds, c8_dozen_flds;
tuple c8_triple_ids, c8_hex_ids, c8_dozen_ids, answer_ids;
paratest_rec *code;

CM_timer_clear(5);
CM_timer_start(5);

init(file1, file2);
fscanf(r7r8file, "%d", &maxtests);
for (row7_8_choice = nextr7_8(basis); row7_8_choice < maxtests;
     row7_8_choice = nextr7_8(basis))
{
  /* nextr7_8 should put the required row 7 and row 8 into the
  /* basis vector */
  printf("case %d:\n", row7_8_choice);
  code = define_paratest(width, numbat);
  setup_basis(code, basis, numbasis);

  do_R9(code, C6actual, c6set);
  do_X(code);

  cntr11 = cntr13 = 0;
  for(i = 0; i < tablesize; i++)
  {
    if (table64k[i].valid)
    {
      if (table64k[i].valid & r9_r11)
      {
        if (!(table64k[i].valid & r9_r13))
          printf("R11 but not R13 in valid for table[%d]/n",i);
```

```
            cntr11++;
            if ((table64k[i].valid & (cand_r7 | cand_r9)))
              printf("R9 or R7 and R11 in valid for table[%d]/n",i);
          }
          if (table64k[i].valid & r9_r13)
          {
            cntr13++;
            if ((table64k[i].valid & (cand_r7 | cand_r9)))
              printf("R9 or R7 and R13 in valid for table[%d]/n",i);
          }
        }
      }
#ifdef info
    printf(" %d r9/r11s and %d r9/r13s were found\n", cntr11, cntr13);
#endif
    create_valid_c8_triples(c8_triple_ids);

  /* define and set up geometry for vp-set of size 2 */
    define_vp(2);
    CM_set_vp_set(curr_vp);
    c8_triple_flds = send_tuples_to_cm(c8_triple_ids, 3);
    distribute_multi_compat(r9_r11, offset_from_bit(r9_r11),
c8_triple_flds, 3, c8_id_fldlen, C8_cand_16, C8actual,
c8_triple_ids[0][0]);
    t = build_extended_pairs(c8_triple_ids, 3, c8_triple_flds, 3,
c8_hex_ids, max_tuples);
#ifdef info
    printf("Return from build_extended_pairs: %d\n", t);
#endif
```

```
    define_vp(1);
    CM_set_vp_set(curr_vp);
    c8_hex_flds = send_tuples_to_cm(c8_hex_ids, 6);
    distribute_multi_compat(r9_r13, offset_from_bit(r9_r13),
c8_hex_flds, 6, c8_id_fldlen, C8_cand_16, C8actual,
c8_hex_ids[0][0]);
    t = build_extended_pairs(c8_hex_ids, 6, c8_hex_flds, 6,
c8_dozen_ids, max_tuples);
#ifdef info
    printf("Return from build_extended_pairs: %d\n", t);
#endif

    t = build_extended_pairs(c8_dozen_ids, 12, c8_hex_flds, 6,
answer_ids, max_tuples);

    if(answer_ids[0][0] > 0) /* good complete matrice(s) found */
      display_matrices(answer_ids, 18);
    else printf("zilch\n");

    re_init(file1);
  }
  printf("overall time:\n");
  CM_timer_stop(5);
  CM_timer_print(5);
}  /* build_it */
```

```
/* convert.c */
#include <stdio.h>
#include "convert.h"

#define length 24

int cand16[16]; /* curr. cand. in 16-long form */
int cand24[length]; /* curr. cand. in 24-long form */

FILE *candfile, *outfile;


void /* convert from 24-long to 16-long format */
l24tol16(A, B)
int A[];
int B[];
{ int i, aindex, bindex;

  for(i = 0; i < 8; i++)
  {
    aindex = 3*i;
    bindex = 2*i;
    B[bindex] = (A[aindex] == 0);
    B[bindex+1] = (A[aindex+1] == 0);
  }
}

void /* convert from 16-long to 24-long format */
l16tol24(A, B)
int A[];
```

```
int B[];
{ int i, aindex, bindex;

  for(i = 0; i < 8; i++)
  {
    aindex = 2*i;
    bindex = 3*i;
    B[bindex] = (A[aindex] == 0);
    B[bindex+1] = (A[aindex+1] == 0);
    B[bindex+2] = (A[aindex] == A[aindex+1]);
  }
}

int /* convert from 16-long to 16-bit form */
ltob16(A)
int A[];
{ int v, j;

  v = 0;
  for(j = 0; j < 16; j++)
  {
    v <<= 1;
    v += A[j];
  }
  return v;
}

int /* convert from 24-long to 24-bit form */
ltob24(A)
int A[];
```

```
{ int v, j;

  v = 0;
  for(j = 0; j < 24; j++)
  {
    v <<= 1;
    v += A[j];
  }
  return v;
}

void
btol16(b, A) /* convert from 16-bit to 16-long form */
int b;
int A[];
{ int i;

  for(i = 15; i >= 0; i--)
  {
    A[i] = b & 01;
    b >>= 1;
  }
}

void /* get candidates and convert to 16-bit format */
processcand(cands, index)
candrec *cands;
int index;
{
  l24tol16(cand24, cand16);
```

```
  cands[index].val16 = ltob16(cand16);
  cands[index].val24 = ltob24(cand24);
}

int
read_cands(filename, cands, maxcands)
char *filename;
candrec *cands;
int maxcands;
{ int i, t, f, count;
  char line[48];

  candfile = fopen(filename, "r");
  if(candfile == NULL)
  {
    printf("Error - cannot open %s\n", filename);
    exit(1);
  }
  count = 0;
  while((count < maxcands) &&
((f = (fscanf(candfile, "%s", &line))) != EOF))
  {
    for(i = 0; i < length; i++)
      cand24[i] = (int)(line[i] - '0');
    /* skip last 24 positions */

    processcand(cands, count);
    count++;
  }
  fclose(candfile);
```

```
  return count;
}
```

```
/* evalC8.c */
#include <cm/paris.h>
#include <stdio.h>
#include "convert.h"
#include "paratest.h"
#include "table64.h"

#define length 24
#define pxlistsize 32896 /* total number of the original 2^16 entries
   in table64k[] that must be checked by
   paratestx; this is all doubly-even entries */


void
do_R9(code, C6actual, cands)
paratest_rec *code;
int C6actual;
candrec cands[];
/* check all C6 candidates to see if good for R9 */
{ int i, k, ind;
  int clist[MAX_C6_SIZE];
/* list of R9 cands to pass to paratest_R9 */
  char cbits[MAX_C6_SIZE]; /* cbits[i] stores the 'valid' field (from
      table64k[]) returned by paratest of entry
      i in clist */

  /* move all candidates in 'cands' to clist, to pass to
     paratest_R9 */
  for(i = 0; i < C6actual; i++)
  {
```

```
    clist[i] = cands[i].val24;
    cbits[i] = 15;
  }
  do_paratest_R9(code, clist, C6actual, cbits);
#ifdef info
  printf("C6actual = %d\n", C6actual);
#endif

  k = 0; /* counts current position in list of good C8's */
  for(i = 0; i < C6actual; i++)
  {
    ind = cands[i].val16;
    table64k[ind].valid |= (short)cbits[i];
    if(table64k[ind].valid & cand_r9) /* good C8 candidate */
    {
      C8_cand_16[k] = ind; /* store 16-bit val in C8_cand_16 */
      table64k[ind].C8id = k; /* store position in list */
      k++;
    }
  }
  C8actual = k;
#ifdef info
  printf("number of good R9 candidates found: %d\n", C8actual);
#endif
}

void
do_X(code)
paratest_rec *code;
/* check all doubly-even entries of table64k[]
```

```
    to check if good for R9_R11 or R9_R13 */
{ int i, k;
  int pval16[pxlistsize];
/* pval16[i] stores 16-bit val of entry i in plist */
  int plist[pxlistsize]; /* list of xor cands to pass to paratest_X */
  char pbits[pxlistsize]; /* pbits[i] stores the 'valid' field (from
   table64k) ret by paratest of plist[i] */

  /* move all doubly-even entries of table64k[] to plist, to pass to
     paratest_x - these are potential R9_R11's and R9_R13's     */
  k = 0; /* k marks current index of plist */
  for(i = 0; i < tablesize; i++)
  {
    if((table64k[i].popcount & 03) == 0) /* doubly-even */
    {
      if(k >= pxlistsize)
Abort("pxlistsize too small");
      plist[k] = table64k[i].val24;
      pval16[k] = i;
      pbits[k] = 15;
      k++;
    }
    /* else popcount not doubly even - invalid for R9_R11, R9_13
(so do nothing whatsoever)  */
  }

  do_paratest_X(code, plist, pxlistsize, pbits);
  for(i = 0; i < pxlistsize; i++)
    table64k[pval16[i]].valid |= (short)pbits[i];
}
```

```
int nextr7_8(basis)
  int basis[];
{ int i, j;
  char line[48];
  static int R7R8 = 0;

  if(R7R8 == 0)
  {
     /* first 5 rows are the same for all cases */
    basis[0] = 16773120;
    basis[1] = 4095;
    basis[2] = 16519104;
    basis[3] = 14913080;
    basis[4] = 15878436;

     /* read r6 from file */
    basis[5] = 0;
    fscanf(r7r8file, "%s", &line);
    for(i = 0; i < length; i++)
    {
      basis[5] <<= 1;
      basis[5] = basis[5] ^ (int)(line[i] - '0');
    }
  }
  /* read next r7/r8 combo from file */
  for(j = 6; j < 8; j++)
  {
    basis[j] = 0;
```

```
    fscanf(r7r8file, "%s", &line);
    for(i = 0; i < length; i++)
    {
       basis[j] <<= 1;
       basis[j] = basis[j] ^ (int)(line[i] - '0');
    }
  }
  return R7R8++;
}
```

```
/* inittable.c */
#include <stdio.h>
#include "convert.h"
#include "table64.h"



int
getval24(v, pop)
int v;
int *pop;
/* return the 24-bit conversion of 16-bit val v;
/* pop contains the population count of the 24-bit value */
{ int i, u = 0, bval;

  *pop = 0;
  for(i = 0; i < 8; i++)
  {
    bval = v & 03; /* retrieve 2 rightmost bits of v */
    v >>= 2;
    if(bval == 0) /* 00 (->111) in current block */
    {
      (*pop)+=3;
      u+=(7 << 3*i);
    }
    else
    {
      (*pop)++;
      if(bval == 1) /* 01 (->100) in current block */
u+=(4 << 3*i);
```

```
      else if(bval == 2) /* 10 (->010) in current block */
u+=(2 << 3*i);
      else  /* bval = 3; 11 (->001) in current block */
u+=(1 << 3*i);
    }
  }
  return u;
}

int
maketable(filename, cands, maxcand)
char *filename;
candrec cands[];
int maxcand;
{ int numcands, i, ind;
  int pop, popdistrib[25];

  numcands = read_cands(filename, cands, maxcand);
#ifdef info
  printf("numcands = %d\n", numcands);
#endif

  for(i = 0; i < 25; i++)
    popdistrib[i] = 0;
  for(i = 0; i < tablesize; i++)
  {
    table64k[i].canid = -1;
    table64k[i].C8id = -1;
    table64k[i].valid = 0;
    table64k[i].val24 = getval24(i, &pop);
```

```
    table64k[i].popcount = pop;
    popdistrib[pop]++;
  }
  for(i = 0; i < numcands; i++)
  {
    ind = cands[i].val16;
    table64k[ind].canid = i+1;
    table64k[ind].valid |= cand_r7;
/* good as r7 cand - set corresp. bit */
  }
  return numcands;
}
```

```
/* multicompat.c -
/* Create multi compat in cm from compat info in table64k
/* and a list of candidates. */

#include <stdio.h>
#include <math.h>
#include <cm/paris.h>
#include <cm/cmsr.h>
#include "table64.h"
#include "multicompat.h"
#include "bldextended.h"

#define BITS_PER_WORD 32
#define BITSHIFT 5

static cmcompatlen = 0;     /* Length of cmpompat to date. It is
      grown bigger as needed. */

void select_target(candidate, nflds, field, testlen)
  CM_field_id_t field;
  int candidate;  /* The candidate number to find */
  int nflds;      /* Number of fields to test */
  int testlen;    /* length of each field */

  /* Set the context bit in each vp in current vpset iff the value
  /* candidate is one in field. field is divided up into maxids
  /* sections, each of length testlen; the first nflds of these
  /* each contain one C8id. */
{
  int i;
```

```
  CM_set_context();
  for (i = 0; i < nflds; i++)
  {
    CM_u_ne_constant_1L(CM_add_offset_to_field_id(field, i*testlen),
candidate, testlen);
    CM_logand_context_with_test();
  }
  CM_invert_context();
}  /* select_target */


void longand(target, bitvec, bitlen)
  CM_field_id_t target;  /* Target to AND the bits into */
  int *bitvec;  /* Bit vector to be downloaded to cm */
  int bitlen;   /* Length in bits */

  /* AND *bitvec into target in every VP with its context bit set.
  /* bitlen is the length of the vector bitvec in bits and may be
  /* arbitrarily large. */
{
  int i;

  /* We assume integral number of words. */
  assert((CM_maximum_integer_length & (BITS_PER_WORD-1)) == 0);

  for (i = 0; i < (bitlen - BITS_PER_WORD); i+=BITS_PER_WORD)
  {
    CM_logand_constant_2_1L(target, *bitvec, BITS_PER_WORD);
    target = CM_add_offset_to_field_id(target, BITS_PER_WORD);
```

```
    bitvec += BITS_PER_WORD >> BITSHIFT;
  }
  if (i < bitlen)
    CM_logand_constant_2_1L(target, *bitvec, bitlen - i);
} /* longand */

void distribute_multi_compat(mask, shift, cand_flds, nflds, fld_len,
     C8_cand_16, Ncands, Ntuples)
  int mask; /* Pick out bit to use from valid */
  int shift; /* Shift count to get bit in low order position */
  CM_field_id_t cand_flds;  /* Vector of field ids pointing to the
    /* fields to be tested. */
  int nflds; /* Number of fields to be tested. */
  int fld_len; /* length of the test fields. All are the same. */
  int *C8_cand_16; /* Vector of cands to use to form multi-compat */
  int Ncands; /* Number of cands to process */
  int Ntuples; /* Number of tuples that will be tested; only the
   first Ntuples processors should contain columns of
   compat matrix - the others should have all zeroes */
{
  CM_field_id_t segment, news_coord;
  int i, j, k, bit, xor;
  static int bitveclen = 0;
  static int *bitvec = NULL;
  int count, temp;
#define coord_len 15 /* note: allows for vp-ratio <= 4 only */

#ifdef info
  printf("distribute_multicompat(mask = %d, shift = %d, nflds = %d,
fld_len = %d, Ncands = %d\n", mask, shift, nflds, fld_len, Ncands);
```

```
#endif

#ifdef timing
  CM_timer_clear(1);
  CM_timer_start(1);
#endif

  /* Ensure bitvec big enough to build string in */
  if (bitveclen < ((Ncands + 8) >> 3))
  {
    if (bitvec != NULL) free(bitvec);
    bitveclen = (Ncands+8)>>3;
    bitvec = (int *)malloc(bitveclen);
  }
  /* Ensure that cmcompat is big enough. */
  if (cmcompatlen < Ncands)
  {
    if (cmcompatlen > 0) CM_deallocate_heap_field(cmcompat);
    cmcompat = CM_allocate_heap_field(Ncands);
    cmcompatlen = Ncands;
  }

  /* determine the first Ntuples processors */
  news_coord = CM_allocate_heap_field(coord_len);
  CM_my_news_coordinate_1L(news_coord, 0, coord_len);
  CM_set_context();
  CM_u_lt_constant_1L(news_coord, Ntuples, coord_len);
  CM_logand_context_with_test();

  /* Set target to all ones in these processors */
```

```
  segment = cmcompat;
  for (i = 0; i < (Ncands - CM_maximum_integer_length);
       i+=CM_maximum_integer_length)
  {
    CM_logorc1_2_1L(segment, segment, CM_maximum_integer_length);
    segment = CM_add_offset_to_field_id(segment,
CM_maximum_integer_length);
  }
  CM_logorc1_2_1L(segment, segment, Ncands - i);

  /* ensure target is all zeroes in rest */
  CM_invert_context();
  segment = cmcompat;
  for (i = 0; i < (Ncands - CM_maximum_integer_length);
       i+=CM_maximum_integer_length)
  {
    CM_u_move_zero_1L(segment, CM_maximum_integer_length);
    segment = CM_add_offset_to_field_id(segment,
CM_maximum_integer_length);
  }
  CM_u_move_zero_1L(segment, Ncands - i);
  /* do everything in all processors from now on */
  CM_set_context();

  /* Now build and download the Ncands columns of compat */
  for (i = 0; i < Ncands; i++)
  {
    for (j = 0; j <= (int)(Ncands/BITS_PER_WORD); j++)
    {
      bitvec[j] = 0;
```

```
      for (k = 0; (k < BITS_PER_WORD) &&
   (j*BITS_PER_WORD+k < Ncands); k++)
      {
/* determine compatibility of cand i with cand (j*BITS_PER_WORD+k) */
xor = C8_cand_16[i] ^ C8_cand_16[j*BITS_PER_WORD+k];
        bit = ((table64k[xor].valid & mask) >> shift);
        bitvec[j] |= (bit << k);
      }
    }
    select_target(i, nflds, cand_flds, fld_len);
    longand(cmcompat, bitvec, Ncands);
  }
#ifdef timing
  CM_timer_stop(1);
  CM_timer_print(1);
  printf("\n");
#endif
} /* distribute_multi_compat */
```

```
/* paratest.c -
/* Module to perform a parallel test of a set of candidates against a
/* particular code space. */

#include <stdio.h>
#include <math.h>
#include <cm/paris.h>
#include <cm/cmsr.h>
#include "paratest.h"
#include "table64.h"
#include "testprt.h"

#define NUM_PROCS 8192
#define STRING_LEN 24
#define POPCOUNT_LEN 6
#define SINGLY_EVEN_LEN POPCOUNT_LEN - 1
#define DOUBLY_EVEN_LEN POPCOUNT_LEN - 2
#define RESULT_STR_LEN 4

static int test_allocation = 0;
static CM_field_id_t testcode
/* Combination of value and basecode to be tested. */,
       basecode
/* Code value in the  original space */,
       full_popcount
/* where logcount puts its result. */,
       codetest_result
/* Where the RESULT_STR_LEN bit result gets stored */,
       singly_even
/* address the bit which tests singly vs doubly even counts */,
```

```
        doubly_even
 /* address the count  field with 2 low order bits ignored. */,
        curcnt  /* Compute full count / 2 (left and right side) here,
                /* but divided by 2 (the low order bit always 0). */,
        basecount_R13
/* RHS contribution to a count for r9/r13 test */,
        basecount_R11
/* RHS contribution to a count for r9/r11 test */,
        r9_r11_results
/* bit location  to store this result */,
        r9_r13_results
/* bit location  to store this result */,
        r9_results
/* bit location  to store this result */,
        merged_result_field;
/* reduce results to here for each space */

int offset_from_bit(bitmask)
int bitmask;
  /* return the bit offset that corr to the bit mask */
{
  if (bitmask == 1) return 0;
  if (bitmask == 2) return 1;
  if (bitmask == 4) return 2;
  if (bitmask == 8) return 3;
  Abort("error in offset_from_bit");
}  /* offset_from_bit */

static void init()
{
```

```
  test_allocation = 1;
  testcode = prism_alloc(STRING_LEN);
  basecode = prism_alloc(STRING_LEN);

  full_popcount = prism_alloc(POPCOUNT_LEN);
  merged_result_field = prism_alloc(RESULT_STR_LEN);
  curcnt = prism_alloc(SINGLY_EVEN_LEN);
  basecount_R13 = prism_alloc(SINGLY_EVEN_LEN);
  basecount_R11 = prism_alloc(SINGLY_EVEN_LEN);
  CM_set_context();
  codetest_result = prism_alloc(RESULT_STR_LEN);
  singly_even = CM_add_offset_to_field_id(full_popcount, 1);
  doubly_even = CM_add_offset_to_field_id(full_popcount, 2);
  r9_results = CM_add_offset_to_field_id(codetest_result,
offset_from_bit(cand_r9));
  r9_r11_results = CM_add_offset_to_field_id(codetest_result,
offset_from_bit(r9_r11));
  r9_r13_results = CM_add_offset_to_field_id(codetest_result,
offset_from_bit(r9_r13));
}   /* init */


paratest_rec *define_paratest(width, Nbatch)
  int width;
  int Nbatch;
{
  paratest_rec *ptr;
  int dim[2];
  int i,j,k;
```

```
  ptr = (paratest_rec *)malloc(sizeof(paratest_rec));
  if (ptr == NULL) Abort("malloc fail");
  ptr->width = dim[0] = width;
  ptr->Nbatch = dim[1] = Nbatch;
  ptr->geometry = CM_create_geometry(dim, 2);
  ptr->build_vp_set = CM_allocate_vp_set(ptr->geometry);
  CM_set_vp_set(ptr->build_vp_set);
  if (test_allocation == 0) init();
  ptr->batchid = prism_alloc(COORDSIZE);
  ptr->codeid = prism_alloc(COORDSIZE);
  CM_my_news_coordinate_1L(ptr->batchid, 1, COORDSIZE);
  CM_my_news_coordinate_1L(ptr->codeid, 0, COORDSIZE);
  return ptr;
}  /* define_paratest */

void
setup_basis(code, basis, numbasis)
  /* set up codespace using basis vectors in basis[] */
paratest_rec *code;  /* for this paratest */
int basis[];
int numbasis;  /* Number of basis vectors */
{ int i;
  CM_field_id_t blockor;

  CM_set_context();
  /* store combinations in basecode */
  CM_u_move_zero_1L(basecode, STRING_LEN);
  for(i = 0; i < numbasis; i++)
  {
```

```
    CM_load_context(CM_add_offset_to_field_id(code->codeid, 1));
    CM_logxor_constant_2_1L(basecode, basis[i], STRING_LEN);
  }

  CM_set_context();
  /****** NOTE: basecount_R1x has the low order bit trimmed (it is
  /****** always 0). Thus all arithmetic operations into them have
  /****** been divided by 2!! */
  CM_u_move_zero_1L(basecount_R11, SINGLY_EVEN_LEN);
  CM_u_move_constant_1L(basecount_R13, 2, SINGLY_EVEN_LEN);
  blockor = CM_allocate_stack_field(1);
  for(i = 4; i < numbasis; i+=2)
  {
    /* add 1 (2) to basecount fields in all processors containing at
    /* least a 1 in pid, for each block of 2 rows */
    CM_logior_always_3_1L(blockor,
CM_add_offset_to_field_id(code->codeid, 1),
CM_add_offset_to_field_id(code->codeid, i+1), 1);
    CM_load_context(blockor);
    CM_u_add_constant_2_1L(basecount_R11, 1, SINGLY_EVEN_LEN);
    CM_u_add_constant_2_1L(basecount_R13, 1, SINGLY_EVEN_LEN);
  }
  /* if basecount_R11 < 2 (4), basecount_R11 += 2 (4); if >= 4,
     no changes necessary due to adding a combination of extra rows */
  CM_set_context(); /* do comparison on all processors */
  CM_u_lt_constant_1L(basecount_R11, 2, SINGLY_EVEN_LEN);
  CM_logand_context_with_test();
/* store test result in context bit */
  CM_u_add_constant_2_1L(basecount_R11, 2, SINGLY_EVEN_LEN);
  CM_deallocate_stack_through(blockor);
```

```
} /* setup_basis */

void read_up_results(ptr, bits)
  paratest_rec *ptr;
  char *bits;
  /* Read a batch of paratest results from the back end according to
  /* the geometry in *ptr and store them in *bits.
  /* If bits == NULL, this is a setup call, and no info is obtained
  /* from the backend. */
{
  static int feoffset[2] = { 0, 0}, festart[2] = {0, 0},
feend[2] = {1, 1}, feaxis[2] = {1, 0}, fedim[2] = {8192, 1};
  static int cur_ndx;

  if (bits == NULL) {
    cur_ndx = 0;
    feend[0] = ptr->Nbatch;
    return;
  }
  CM_u_read_from_news_array_1L(bits+cur_ndx, feoffset, festart,
feend, feaxis, merged_result_field, RESULT_STR_LEN,
2, fedim, CM_8_bit);
  cur_ndx += ptr->Nbatch;
} /* read_up_results */


void do_paratest_X(ptr, cands, Ncands, bits)
  paratest_rec *ptr;
  int * cands;
  int Ncands;
```

```
  char * bits;
{
  int cand_i, batch_i, batch_end;
  int bi;

  CM_set_vp_set(ptr->build_vp_set);
  batch_end = ptr->Nbatch;
  read_up_results(ptr, NULL);  /* init indices*/
#ifdef timing
  CM_timer_clear(0); CM_timer_start(0);
#endif
  for (cand_i = 0; cand_i < Ncands; cand_i += ptr->Nbatch) {
    /* Send a test string to each section of cm */
    for (batch_i = 0; batch_i < batch_end; batch_i++) {
      CM_set_context();
      CM_u_eq_constant_1L(ptr->batchid, batch_i, COORDSIZE);
      CM_logand_context_with_test();
      bi = cand_i + batch_i;
      if (bi >= Ncands) {
        CM_u_move_constant_1L(testcode, 0, STRING_LEN);
      } else {
        CM_logxor_constant_3_1L(testcode, basecode, cands[bi],
STRING_LEN);
      }
    }
    /* Now process the batch */
    CM_set_context();
    CM_u_logcount_2_2L(full_popcount, testcode, POPCOUNT_LEN,
STRING_LEN);
    CM_u_move_zero_1L(codetest_result, RESULT_STR_LEN);
```

```
    /* Set flag for r9/r13 */
    CM_u_add_3_1L(
curcnt, singly_even, basecount_R13, SINGLY_EVEN_LEN);
    CM_u_ge_constant_1L(curcnt, 6, SINGLY_EVEN_LEN);
    CM_store_test(r9_r13_results);
    CM_logandc2_2_1L(r9_r13_results, curcnt, 1);
    /* Now for r9/r11 */
    CM_u_add_3_1L(
curcnt, singly_even, basecount_R11, SINGLY_EVEN_LEN);
    CM_u_ge_constant_1L(curcnt, 6, SINGLY_EVEN_LEN);
    CM_store_test(r9_r11_results);
    CM_logandc2_2_1L(r9_r11_results, curcnt, 1);
    /* End of testing. Now lets reduce the results and upload them */
    CM_set_context();
    CM_reduce_with_logand_1L(merged_result_field, codetest_result,
      0, RESULT_STR_LEN, 0);
    read_up_results(ptr, bits);
  }
#ifdef timing
CM_timer_stop(0); printf("do_paratest: ");
CM_timer_print(0); printf("\n"); CM_timer_clear(0);
#endif
}  /* do_paratest_X */

void do_paratest_R9(ptr, cands, Ncands, bits)
  paratest_rec *ptr;
  int * cands;
  int Ncands;
  char * bits;
{
```

```
  int cand_i, batch_i, batch_end;
  int bi;

  CM_set_vp_set(ptr->build_vp_set);
  batch_end = ptr->Nbatch;
  read_up_results(ptr, NULL);  /* init indices*/
  for (cand_i = 0; cand_i < Ncands; cand_i += ptr->Nbatch) {
    /* Send a test string to each section of cm */
    for (batch_i = 0; batch_i < batch_end; batch_i++) {
      CM_set_context();
      CM_u_eq_constant_1L(ptr->batchid, batch_i, COORDSIZE);
      CM_logand_context_with_test();
      bi = cand_i + batch_i;
      if (bi >= Ncands) {
        CM_u_move_constant_1L(testcode, 0, STRING_LEN);
      } else {
        CM_logxor_constant_3_1L(
testcode, basecode, cands[bi], STRING_LEN);
      }
    }
    /* Now process the batch */
    CM_set_context();
    CM_u_logcount_2_2L(
full_popcount, testcode, POPCOUNT_LEN, STRING_LEN);
    CM_u_move_zero_1L(codetest_result, RESULT_STR_LEN);
    CM_u_add_3_1L(
curcnt, singly_even, basecount_R13, SINGLY_EVEN_LEN);
    CM_u_subtract_constant_2_1L(curcnt, 1, SINGLY_EVEN_LEN);
    CM_u_ge_constant_1L(curcnt, 6, SINGLY_EVEN_LEN);
    CM_store_test(r9_results);
```

```
    CM_logandc2_2_1L(r9_results, curcnt, 1);
    /* End of testing. Now lets reduce the results and upload them */
    CM_set_context();
    CM_reduce_with_logand_1L(merged_result_field, codetest_result,
      0, RESULT_STR_LEN, 0);
    read_up_results(ptr, bits);
  }
} /* do_paratest_R9 */


int
get_pop(v) /* return popcount of 24-bit vector v */
int v;
{ int i, pop;

  pop = 0;
  for(i = 0; i < 24; i++)
  {
    pop += (v & 01); /* add rightmost bit into pop */
    v >>= 1;
  }
  return pop;
}


void test_paratest()
{
  paratest_rec *code_test_at_r8;
  int cands[80];
  char results[200];
```

```
  int i,j,k;

  printf("Ready to call define_paratest\n");
  code_test_at_r8 = define_paratest(256, 32);
  printf("Exit from call define_paratest\n");
  for (i = 0; i < 80; i++) {
    results[i] = 15;  /* Not legal response */
    cands[i] = rand() & 0xffffff;
    while(get_pop(cands[i]) & 03) /* not doubly-even */
      cands[i] = rand() & 0xffffff;
  }
  printf("Ready to call do_paratest\n");
  do_paratest_X(code_test_at_r8, cands, 50, results);
  printf("Exit from call do_paratest\n");
  printf("candidates chosen:\n");
  for (j = 0; j < 80; j+=10) {
    for (i = 0; i < 10; i++) {
      printf(" %X", cands[i+j]);
    }
    printf("\n");
  }
  printf("results:\n");
  for (j = 0; j < 80; j+=10) {
    for (i = 0; i < 10; i++) {
      printf(" %2x", results[i+j]);
    }
    printf("\n");
  }
  printf("Exit from call test_paratest\n");
} /* test_paratest */
```