

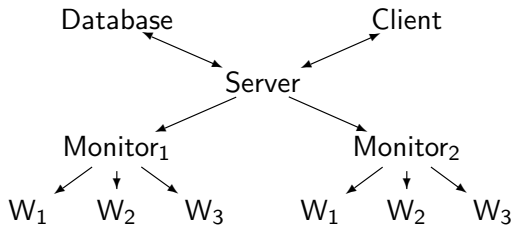
Doubly Even Codes

Robert L. Miller

University of Washington, Seattle

June 18, 2007

Distributed Sage



Distributed Sage is an environment for computations that easily factor into small parts, or jobs. The server, client, and each monitor can be located on different machines. The client generates new jobs and processes completed jobs, and each worker performs the jobs.

Issues arising during a distributed computation

Memory The client should carry the bulk of the memory. The less data you need to pass to each worker, the better. Jobs which carry a large memory load can clog up the database, and slow down the workers.

Errors Errors which occur while a worker is processing a job are not automatically passed back to the client. This is potentially very useful information.

Efficiency For example, at first the d. e. codes computation created each job as soon as it was possible, and passed it along to the server. However, each job can simply be stored as a few integers until it is necessary to create the object. The result: 2.5gb \longrightarrow 20mb on the server.

Machine With many processors, your jobs will stress the machine. For example, at one point we were writing and reading many codes to and from file, and `sage.math` became practically unusable.

Implementing a distributed computation

The idea is pretty simple. You must create a `DistributedFunction` class that does the following:

- Generates new jobs: Jobs are classes themselves. You give each job a segment of code, and you can attach objects (such as lists of matrices). When a worker receives one, it first loads the objects, then executes the code. An object called `DSAGE_RESULT` is saved at the end, and this is passed back to the client by the server.
- Processes finished jobs: The class accomplishes this by instantiating a `process_result` function, which is passed the `DSAGE_RESULT` from the worker.

The Easy Stuff

- All dimension 1 codes are computed by the client.
- All codes generated by weight 4 vectors are known, so the client processes these.
- Zero column codes are created by the client as soon as all codes of smaller length and the same dimension are known. If the contribution of the $[n, k]$ -code is c , and we are creating an $[m, k]$ -code with $m > n$, the contribution is $c \binom{m}{n}$.
- Direct sum codes are created when all the proper sizes are available. If $C = C_1^{p_1} \oplus \dots \oplus C_r^{p_r}$, each C_i an $[n_i, k_i]$ -code with contribution c_i , then the contribution of C is

$$\frac{(\sum_{i=1}^r n_i p_i)!}{\prod_{i=1}^r \binom{n_i}{c_i}^{p_i} p_i!}$$

Indecomposables

This leaves us with discovering all *indecomposable* codes of dimension ≥ 2 . We will automatically have the following cases saturated by doing the easy stuff:

	$k = 1$	$k = 2$	$k = 3$	$k = 4$
$n = 4$	×			
$n = 5$	×			
$n = 6$	×	×		
$n = 7$	×	×	×	
$n = 8$	×	×	×	×
$n = 9$	×	×	×	×
$n = 10$	×	!	×	×

The missing code is

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

Glue

All of the remaining cases can be discovered by gluing vectors onto smaller codes (or gluing codes onto codes). These are the jobs that are passed to the workers. A worker is given several $[n, k - 1]$ -codes to attempt gluing onto, and all known $[n, k]$ -codes. Then it is instructed to do the following:

- Express the code in blocks $[B_1 \ B_2 \ \dots \ B_r]$ where each block B_i is the repetition of a single column some number of times.
- Iterate over vectors (v_1, v_2, \dots, v_r) of integers representing a new vector, and see if this new vector will give a doubly even code. If so, check for equivalence against $[n, k]$ -codes.

Checking glue vectors

Iterate over vectors (v_1, v_2, \dots, v_r) of integers whose sum is divisible by four, using the following

Lemma

- I. If a generator matrix has doubly even rows which are pairwise orthogonal, then the resulting code is doubly even.*
- II. If a generator matrix has doubly even rows whose pairwise sums are doubly even, then the resulting code is doubly even.*

Proof.

$wt(x) + wt(y) - wt(x + y) \equiv 2\langle x, y \rangle \pmod{4}$, plus details. □

to check whether the resulting code will be doubly even. If it is, then check for equivalence to known codes. If it is new, add it to the output.

The Setup

The game board is the chart of possible $[n, k]$ values for a code, and looks something like

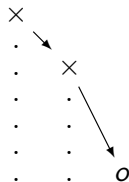
```
0
0
0 0
0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0 0
0 0 0 0 0
⋮
```

Basic Moves

- Adding a zero column: add one to n



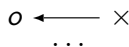
- Taking direct sums (e.g. $[4, 1] \oplus [6, 2]$):
sum all n 's and all k 's



- Glue: add one to k : $x \rightarrow o$

- The above are sufficient

- Take a subspace: subtract one from k



Advanced moves: Stack sums $C_1 \wedge C_2$ (I)

For example, input:

$$C_1 = [1 \ 1 \ 1 \ 1], C_2 = [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1],$$

output:

$$C_1 \wedge_{data} C_2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Criteria: for each pair of rows r_1 in C_1 , r_2 in C_2 , their supports must intersect to give a set with cardinality $c_{1,2}$ such that

$$c_{1,2} \equiv 0 \text{ or } 2 \pmod{4}.$$

Advanced moves: Stack sums $C_1 \wedge C_2$ (II)



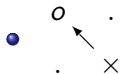
If c is the size of the overlap, then add the n 's and subtract c to get the new n , and add the k 's to get the new k .

- Pick the size c of the overlap.
- Pick c columns from C_1 and c columns from C_2 .
- Check the $\equiv 0$ or $2 \pmod{4}$ condition on each pair of rows.

Advanced moves: Moving off the board



Lengthening: Given a code that does not contain the unit vector (all ones), one can *lengthen* it by adding an all zero coordinate and adding the all ones vector. May not preserve doubly-even condition.



Cross-sectioning: The converse of lengthening. Given a code, you select a coordinate j , remove all codewords with a 1 in coord. j , then delete coordinate j .

Lengthening and cross sectioning, cont'd

- Bilous, R. T. and van Rees, G. H. J. Self-Dual Codes and the $(22,8,4)$ Balanced Incomplete Block Design. J. of Combin Designs. V13 (2005) 363-376.
- Used to go between doubly-even $[33, 16, \geq 4]$ -codes with no zero coord's and self-dual *singly* even $[34, 17, \geq 4]$ -codes.
- Relevant cases: $[17, 8]$ and $[25, 12]$.

Advanced moves: Modification (I)

- Koch, H. On Self-Dual, Doubly Even Codes of Length 32. J. Comb. Thy. A. V51 (1989) 63-76.
- Given $H \subset C$ and D doubly even codes, consider doubly-even isomorphisms (linear isom. preserving weight mod 4)
 $\psi : H^\perp/H \longrightarrow D^\perp/D$, and define the *modification* C_ψ to be the set of words $c_1 + d_2$, where $d_2 \in D^\perp$ such that there is a word $c_1 + c_2 \in C$ with $d_2 + D = \psi(c_2 + H)$.
- C_ψ is doubly even.

Advanced moves: Modification (II)

- Suppose H is an $[l, a]$ -code, D is an $[m, d]$ -code, and C is an $[l + k, a + b]$ -code.
- Constraints: $m - 2d = \dim(D^\perp/D) = \dim(H^\perp/H) = l - 2a$.
- Results: C_ψ is a $[m + k, ?]$ -code.
- Used to reverse engineer the problem of classifying self-dual d.e. codes of length 32.
- Requires subspace information, isomorphism enumeration and automorphism consideration. My opinion: not good for classifying all codes.

Wrapping up...

- `http://www.robertlmiller.com/research.html`
- Computing resources