LARC: Linear Algebra via Recursive Compression

Steve Cuccaro (CCS), John Daly (LPS),
John Gilbert (UCSB, CCR-Adjunct), and Jenny Zito (CCS)

September 2017

About our Research Group

Cuccaro and Zito work at Center for Computing Sciences (CCS), which is a Federally Funded Research and Development Center. Daly works at the Laboratory for Physical Sciences at UMD (LPS) and Prof. Gilbert is an adjoint at CCS from UC Santa Barbara.

About our Research Group

Cuccaro and Zito work at Center for Computing Sciences (CCS), which is a Federally Funded Research and Development Center. Daly works at the Laboratory for Physical Sciences at UMD (LPS) and Prof. Gilbert is an adjoint at CCS from UC Santa Barbara.

CCS works on tough algorithmic and computer problems vital to the nation's security, in areas such as high-performance computing, cryptology, signals processing, data analysis, and network security. We also help foster a lively interaction between researchers in government, academia and industry. It is a fun, collaborative environment and several of our colleagues have helped us out at various times.

Computation and Matrix Mathematics

Many applied problems involve matrix math, such as solving systems of linear equations, and carrying out Fourier transforms.

For large matrices, matrix operations consume a large amount of computational time and memory.

There is a long history of clever methods to speed calculations and save memory while carrying out various matrix operations.

Recursion and Matrix Mathematics

Some of these methods rely on subdividing a square matrix into four quadrants.

$$M = \left[\begin{array}{cc|cc} 3 & 2 & -1 & 0 \\ 0 & -2 & 0 & 0 \\ \hline 2 & 0 & -1 & 7 \\ 5 & 8 & -6 & 0 \end{array}\right] = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \quad \textit{where}$$

$$A = \begin{bmatrix} 3 & 2 \\ 0 & -2 \end{bmatrix} \qquad B = \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 2 & 0 \\ 5 & 8 \end{bmatrix} \qquad D = \begin{bmatrix} -1 & 7 \\ -6 & 0 \end{bmatrix}$$

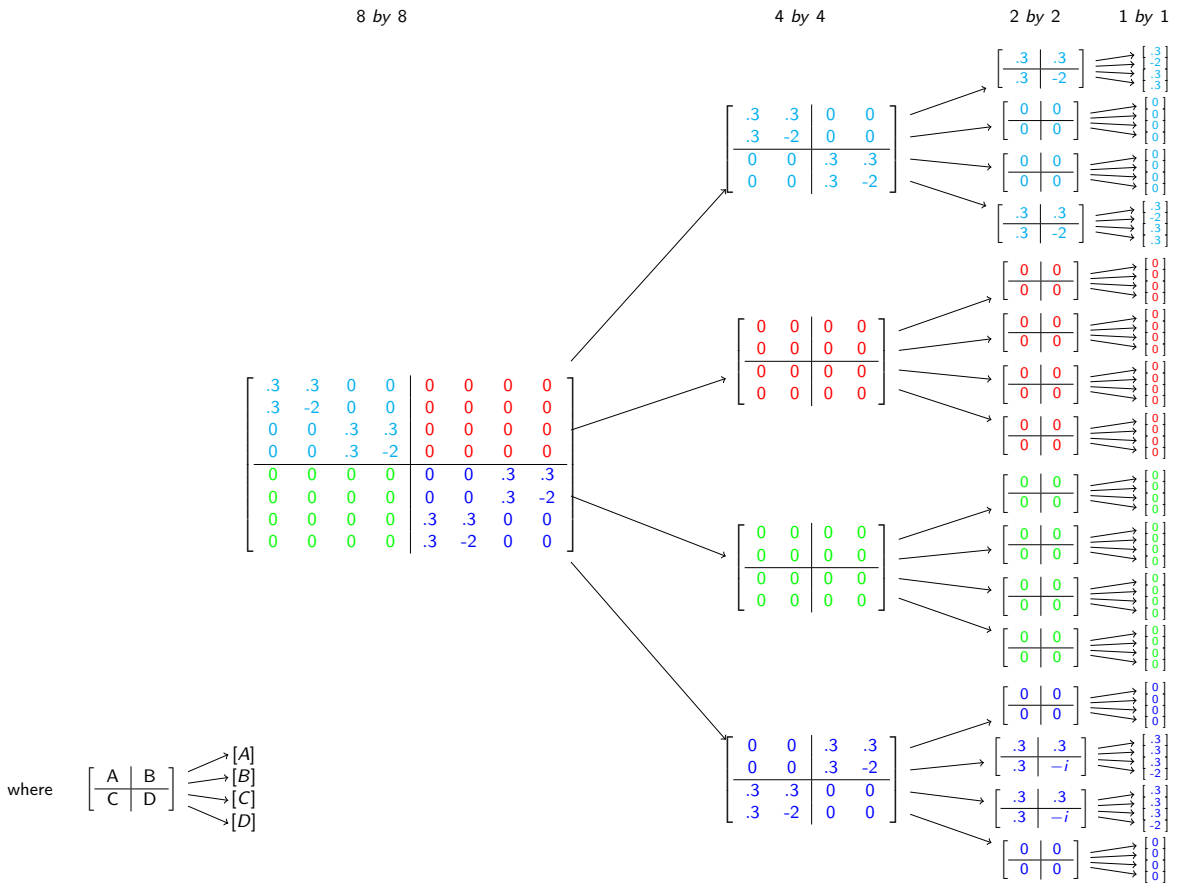
then carrying out operations on the smaller matrices to get the result of an operation on the larger matrices.

Examples: Algorithms Divide Matrix into Quarters

Here are some examples of algorithms which divide matrices into quarters in order to carry out computation.

- **Strassen's Matrix Multiplication Algorithm** (1969)
- **Quadtree Representation of Matrices for Parallel Processing** (1984 David Wise, multiplication, division, trace)
- **Block Matrix Methods for Discrete Fourier Transform** (1960s on, see Van Loan, "Computational Frameworks for the Fast Fourier Transform")

"Quadtree" Representation (David Wise)



Our Matrix Math Package LARC

LARC stands for **Linear Algebra via Recursive Compression**; we stores matrices and carries out matrix operations using subdivision of matrices into four quadrant submatrices recursively.

$$\left[\begin{array}{cccc|cccc} .3 & .3 & 0 & 0 & 0 & 0 & 0 & 0 \\ .3 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .3 & .3 & 0 & 0 & 0 & 0 \\ 0 & 0 & .3 & -2 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & .3 & .3 \\ 0 & 0 & 0 & 0 & 0 & 0 & .3 & -2 \\ 0 & 0 & 0 & 0 & .3 & .3 & 0 & 0 \\ 0 & 0 & 0 & 0 & .3 & -2 & 0 & 0 \end{array}\right] \qquad M = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \longrightarrow (A, B, C, D).$$

$$\left[\begin{array}{cc|cc} .3 & .3 & 0 & 0 \\ .3 & -2 & 0 & 0 \\ \hline 0 & 0 & .3 & .3 \\ 0 & 0 & .3 & -2 \end{array}\right] \quad \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \quad \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \quad \left[\begin{array}{cc|cc} 0 & 0 & .3 & .3 \\ 0 & 0 & .3 & -2 \\ \hline .3 & .3 & 0 & 0 \\ .3 & -2 & 0 & 0 \end{array}\right]$$

$$\left[\begin{array}{c|c} .3 & .3 \\ \hline .3 & -2 \end{array}\right] \quad \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right] \quad \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right] \quad \left[\begin{array}{c|c} .3 & .3 \\ \hline .3 & -2 \end{array}\right] \quad \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right] \quad \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 0 \end{array}\right] \quad \left[\begin{array}{c|c} 0 \\ \hline 0 \end{array}\right.$$

$$\left[\begin{array}{c} .3 \end{array}\right] \quad \left[\begin{array}{c} .3 \end{array}\right] \quad \left[\begin{array}{c} .3 \end{array}\right] \quad \left[\begin{array}{c} -2 \end{array}\right] \quad \left[\begin{array}{c} 0 \end{array}\right] \quad \left[\begin{array}{c} 0 \end{array}\right] \quad \left[\begin{array}{c} 0 \end{array}\right] \quad \left[\begin{array}{c} 0 \end{array}\right] \quad \left[\begin{array}{c} 0 \end{array}\right.$$

IDA

LARC Recursive Matrix Storage Table

A matrix will be stored recursively, with each unique submatrix having its own record.

$$\left[\begin{array}{cccc|cccc} .3 & .3 & 0 & 0 & 0 & 0 & 0 & 0 \\ .3 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .3 & .3 & 0 & 0 & 0 & 0 \\ 0 & 0 & .3 & -2 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & .3 & .3 \\ 0 & 0 & 0 & 0 & 0 & 0 & .3 & -2 \\ 0 & 0 & 0 & 0 & .3 & .3 & 0 & 0 \\ 0 & 0 & 0 & 0 & .3 & -2 & 0 & 0 \end{array}\right]$$

Each submatrix has a unique Matrix_ID (in our example, this is a letter).
The Matrix_Value for a 1 by 1 matrix is the scalar value. For larger matrices, the Matrix_Value is a list of the four Matrix_IDs corresponding to the matrix's four quadrant submatrices.

Matrix Storage Table

Matrix ID	Matrix Value	Matrix Size
A	.3	1 by 1
B	-2	1 by 1
C	(A,A,A,B)	2 by 2
D	0	1 by 1
E	(D,D,D,D)	2 by 2
F	(C,E,E,C)	4 by 4
G	(E,E,E,E)	4 by 4
H	(E,C,C,E)	4 by 4
I	(F,G,G,H)	8 by 8
⋮	⋮	⋮

Block Recursive Matrix Math

$$A = \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right] \quad \text{and} \quad B = \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right]$$

Addition:

$$C = A + B = \begin{bmatrix} (A_1 + B_1) & (A_2 + B_2) \\ (A_3 + B_3) & (A_4 + B_4) \end{bmatrix}$$

Block Recursive Matrix Math (cont.)

Scalar-Matrix Multiplication:

$$C = a \times B = \begin{bmatrix} (a \times B_1) & (a \times B_2) \\ (a \times B_3) & (a \times B_4) \end{bmatrix}$$

Matrix-Matrix Multiplication:

$$C = A \times B = \begin{bmatrix} (A_1 \times B_1) + (A_2 \times B_3) & (A_1 \times B_2) + (A_2 \times B_4) \\ (A_3 \times B_1) + (A_4 \times B_3) & (A_3 \times B_2) + (A_4 \times B_4) \end{bmatrix}$$

Block Recursive Matrix Math (cont.)

Adjoint:

$$C = A^{\dagger} = \begin{bmatrix} {A_1}^{\dagger} & {A_3}^{\dagger} \\ {A_2}^{\dagger} & {A_4}^{\dagger} \end{bmatrix}$$

Kronecker Product:

$$C = A \otimes B = \begin{bmatrix} (A_1 \otimes B) & (A_2 \otimes B) \\ (A_3 \otimes B) & (A_4 \otimes B) \end{bmatrix}$$

LARC Operations Act on Matrix_IDs

Matrix Sum : $\texttt{ID\_C} = \texttt{matrix\_sum}(\texttt{ID\_A}, \texttt{ID\_B})$

$$
\begin{aligned}
&\texttt{if } (\texttt{scalar}(\texttt{ID\_A})) \quad \{\texttt{return}(\texttt{scalar\_sum}(\texttt{ID\_A}, \texttt{ID\_B}));\} \\
&\texttt{else } \{\ \texttt{ID\_C}_1 = \texttt{matrix\_sum}(\texttt{ID\_A}_1, \texttt{ID\_B}_1); \\
&\qquad \texttt{ID\_C}_2 = \texttt{matrix\_sum}(\texttt{ID\_A}_2, \texttt{ID\_B}_2); \\
&\qquad \texttt{ID\_C}_3 = \texttt{matrix\_sum}(\texttt{ID\_A}_3, \texttt{ID\_B}_3); \\
&\qquad \texttt{ID\_C}_4 = \texttt{matrix\_sum}(\texttt{ID\_A}_4, \texttt{ID\_B}_4); \\
&\qquad \texttt{ID\_C} = \texttt{find\_or\_insert\_matrix}([\texttt{ID\_C}_1, \texttt{ID\_C}_2, \texttt{ID\_C}_3, \texttt{ID\_C}_4]); \\
&\qquad \texttt{return}(\texttt{ID\_C}); \\
&\qquad \}
\end{aligned}
$$

Math Identities Shorten Calculations

In our example of matrix addition, there are two math identities which are easy to take advantage of:

- Additive Identity: $A + 0 = A$ (for any matrix A and same sized all-zero matrix 0).
- Commutativity: $A + B = B + A$ (for any matrix A and B of the same size).

So we test to see if either matrix is a zero matrix, and sort the Matrix_IDs when sending them to subroutines, such as the following ...

Saving Time by Not Repeating Operations

For each operation type, such as SUM, PRODUCT, ... we have a table that stores the output of an operation indexed by the inputs.

This type of strategy is called “memoization”, and means you exchange storage space for not doing the same work twice.

Operation Storage (Memoization)

For example:

If we are computing the sum of matrices A and B, then we pass the pair of Matrix_IDs sort(ID_A,ID_B) to a function that checks the SUM operations store.

If we have computed the sum before, it returns the result ID_C.

If not, we calculate the sum and store the result in the SUM table, indexed by the pair sort(ID_A,ID_B).

Using Math Identities and Storing Operations

Matrix Sum : $\texttt{ID\_C} = \texttt{matrix\_sum}(\texttt{ID\_A}, \texttt{ID\_B})$

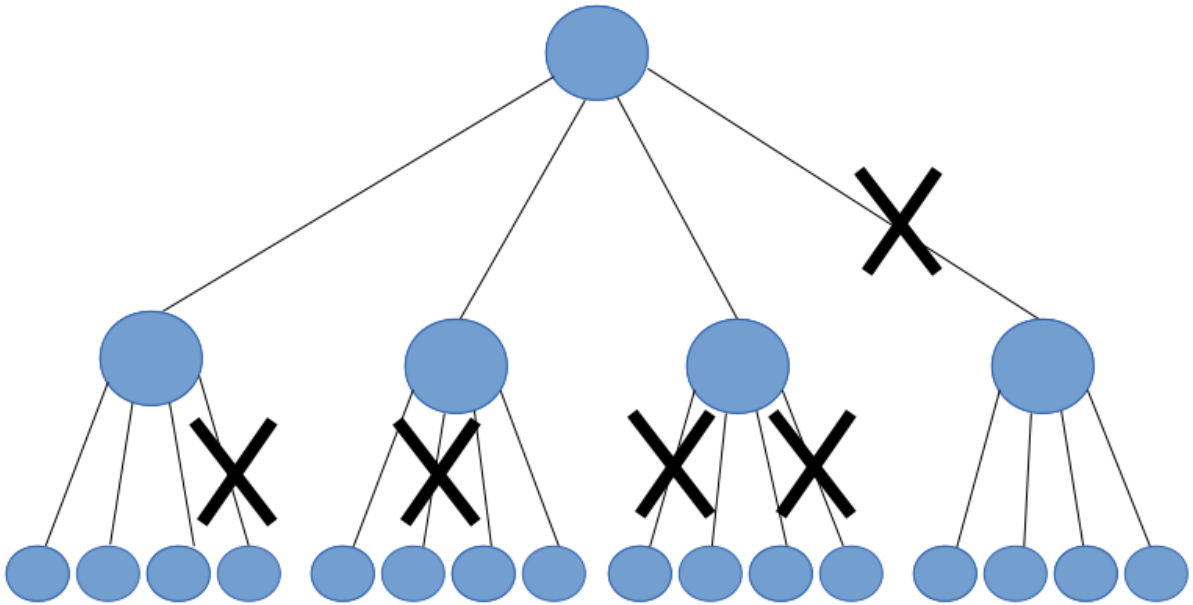
```
if (stored_sum(sort(ID_A, ID_B)))  {return(stored_result(ID_C));}
else if (is_zero_matrix(ID_A))  {return(ID_B);}
else if (is_zero_matrix(ID_B))  {return(ID_A);}
else if (scalar(ID_A))  {return(scalar_sum(ID_A, ID_B));}

else { ID_C1 = matrix_sum(ID_A1, ID_B1);
       ID_C2 = matrix_sum(ID_A2, ID_B2);
       ID_C3 = matrix_sum(ID_A3, ID_B3);
       ID_C4 = matrix_sum(ID_A4, ID_B4);

       ID_C = find_or_insert_matrix([ID_C1, ID_C2, ID_C3, ID_C4]);
       store_sum(sort(ID_A, ID_B), ID_C);
       return(ID_C);
      }
```

Eliminating Unnecessary Computation

Using the math identities and checking to see if a computation has already been stored can eliminate large chunks of the recursive computation.



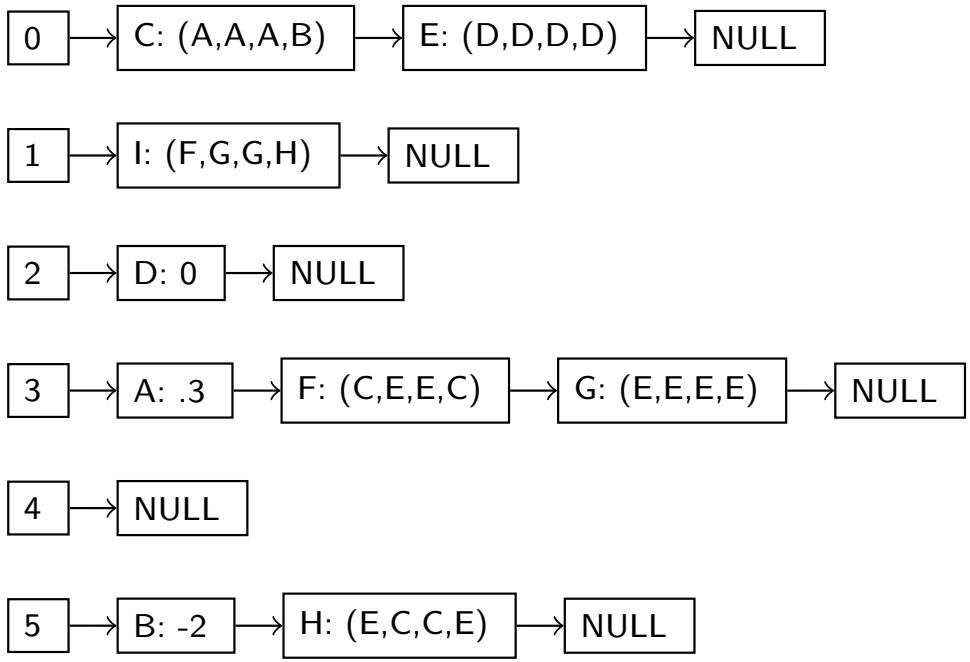
Summary of LARC Features

- Every matrix and submatrix is assigned a unique integer matrix_ID and stored only once.
- This allows compact recursive storage of the matrices in a matrix store.
- The matrix_IDs also allow compact storage of matrix operations (memoization) in an operations store.
- All matrix algorithms are depth-first recursive.
- We use a hashing scheme for fast retrieval and storage.

Fast Hash Retrieval

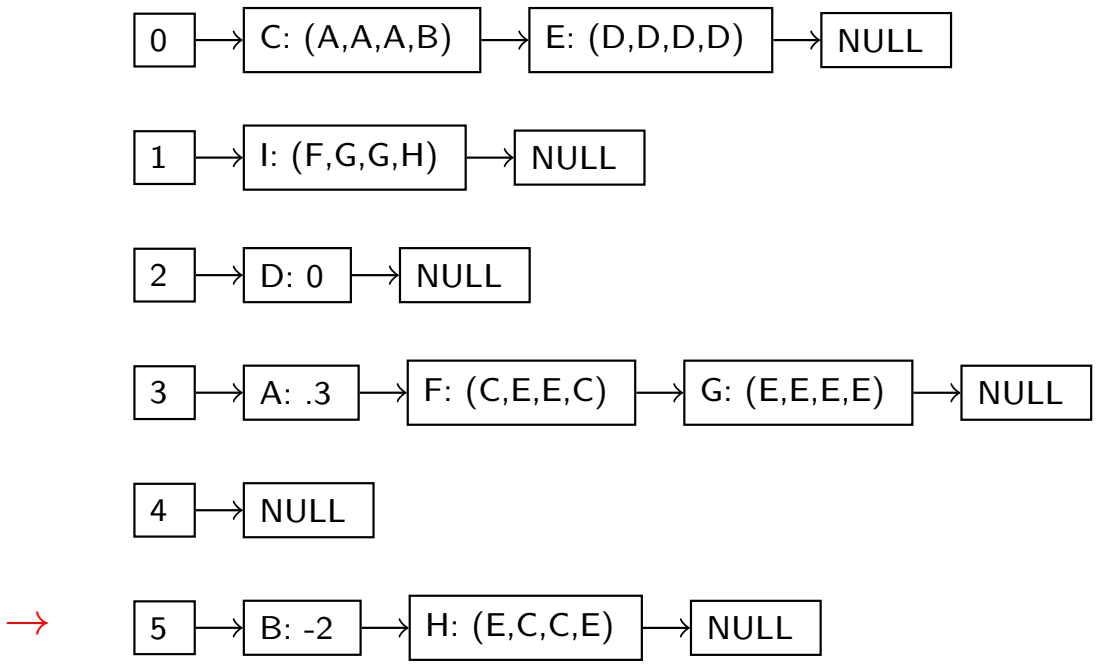
How does fast hash retrieval work? Continuing the matrix storage example, pick a hash function $H()$ and then construct chains:

Matrix ID	Matrix Val	Hash of Mat Val
A	.3	3
B	-2	5
C	(A,A,A,B)	0
D	0	2
E	(D,D,D,D)	0
F	(C,E,E,C)	3
G	(E,E,E,E)	3
H	(E,C,C,E)	5
I	(F,G,G,H)	1
$\vdots$	$\vdots$	$\vdots$



Find or Insert into Hash Table

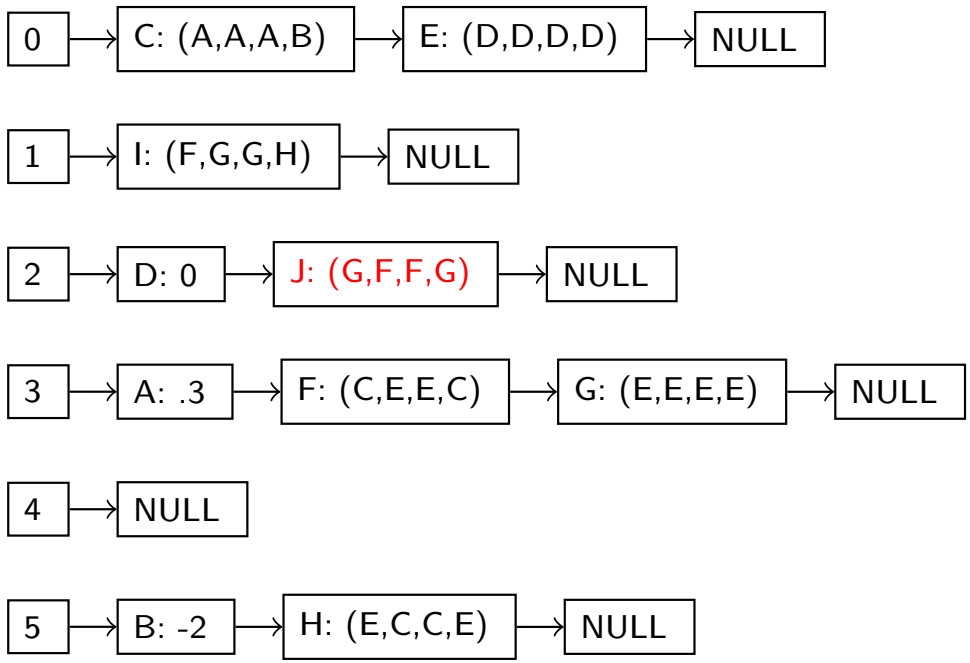
Example: Look for MatrixVal $= (E, C, C, E)$ with Hash=5.
In this case we find (E, C, C, E) and return the MatrixID H.



Find or Insert into Hash Table

Example: Look for MatrixVal = (G, F, F, G) with Hash=2.
In this case we don't find it, so we create a new matrix (J).

Matrix ID	Matrix Val	Hash of Mat Val
A	.3	3
B	-2	5
C	(A,A,A,B)	0
D	0	2
E	(D,D,D,D)	0
F	(C,E,E,C)	3
G	(E,E,E,E)	3
H	(E,C,C,E)	5
I	(F,G,G,H)	1
J	(G,F,F,G)	2
⋮	⋮	⋮



IDA

Locality Preserving Hashes for Scalar Values

There are many schemes that allow searches in a hash table to find an item whose pre-hash value is close to the item you are looking for, with names such as "nearest neighbor search" and "locality-sensitive hashing".
For example, given an address, find a nearby post office (reference: Knuth "Art of Computer Programming").

Locality Preserving Hashes for Scalar Values

There are many schemes that allow searches in a hash table to find an item whose pre-hash value is close to the item you are looking for, with names such as "nearest neighbor search" and "locality-sensitive hashing".
For example, given an address, find a nearby post office (reference: Knuth "Art of Computer Programming").

The idea is to store objects in the hash table in such a way that it is easy to retrieve an object whose pre-hash value is close to your desired value. If we can do this, we don't need to fill up our tables with nearly identical values.

Our Locality Preserving Hash

We divide the space into a grid of the desired fineness, e.g. .1 by .1 boxes. Then we define a Rounding Hash which sends every scalar in the same box to the same storage location in the hash table.

$\text{Storage\_Location}(X) = H(R(X))$

That is, we store the value X, by first applying an approximation function R which rounds it to the closest grid point, and then apply a hash function H.

This function sends every X in a gridbox to the same storage location, but can send things that are physically close but in different gridboxes to different storage locations.

One representative hashing

We want to save space ...

So we only save one value per mesh. Everyone who gets saved later is collapsed to the single grid-representative that was first into the grid.

How does our system work?

Imagine a town council wants to hold a meeting and invites each city block which has grievances to send one representative. The council asks citizens to send all their grievances through their block representative.

You call the town council to ask who your block representative is.

- If they have a block representative listed, then they give you the name,
- otherwise they appoint **you** as your block's representative.

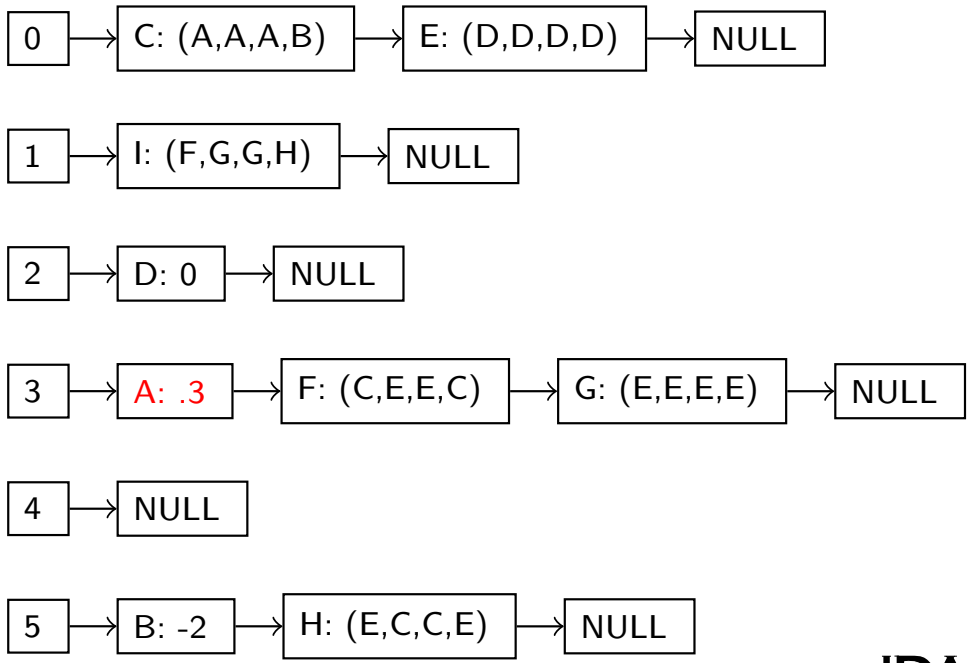
Locality Hashing

Fix an accuracy (the size of the blocks), e.g. .25. Then for any scalar value s instead of hashing to $H(s)$ with a new hash function $H^*(s) = H(R(s))$ where R rounds to the given accuracy.

Locality Hashing

Example: Find MatrixID for .29, or a scalar close to .29. The rounding function gives $R(.29) = .25$ which hashes to $H(.25) = 3$. On hash chain 3, look for scalar s' rounding to $R(s) = R(s') = .25$. We find $s' = .3$ and return MatrixID A.

Matrix ID	Matrix Val	Hash of Mat Val
A	.3	3
B	-2	5
C	(A,A,A,B)	0
D	0	2
E	(D,D,D,D)	0
F	(C,E,E,C)	3
G	(E,E,E,E)	3
H	(E,C,C,E)	5
I	(F,G,G,H)	1
:	:	:



Preloading special numbers to preserve identities

Finite precision can lead to math identities going wrong. Rounding to use the Locality Hash reduces the precision even more.

We can mitigate problems with identities, by preloading the numbers we care about most for our problem.
For example by preloading $\sqrt{2}$ and 2 they become the grid-representatives for each of their gridboxes which will preserve

$$\sqrt{2} * \sqrt{2} = 2$$

LARC works well on Block Recursive Algorithms

Example: Factorization of Discrete Fourier Transform into a recursive description in terms of sparse matrices (from Van Loan Computational Frameworks for the Fast Fourier Transform).

$$F_k = C_k(I_2 \otimes F_{k-1})P_k$$

where F_k is the Fourier Transform matrix of size $2^k \times 2^k$ with (j,k)th entry $e^{jk\frac{2\pi i}{n}}$. Note $w_n = e^{\frac{2\pi i}{n}}$ is the nth root of unity.

Details of Van Loan Sparse Block FFT

$$F_k = C_k(I_2 \otimes F_{k-1})P_k$$

$$C_k = \begin{bmatrix} I_{k-1} & D_{k-1} \\ I_{k-1} & -D_{k-1} \end{bmatrix}$$

$$D_k = \text{diag}[1, w_{2^{k+1}}, w_{2^{k+1}}^2, \ldots, w_{2^{k+1}}^{2^k-1}]$$

$$I_k = \text{diag}[1, 1, \ldots, 1].$$

Details of Van Loan Sparse Block FFT (cont.)

The matrix P_k is the $2^k \times 2^k$ inverse shuffle matrix (its transverse is its inverse and would shuffle a column vector).

$$P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad P_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

How LARC Improves Existing Work

In order to carry out a single matrix multiplication addition or trace, Wise use pointers to farm out quadrants of a matrix for computation to different parallel processors. These operations are similar to the way we recursively do math in LARC.
Our decision to assign each matrix a unique matrix ID and then recursively describe matrices in terms of these IDs for storage allows us to: store certain matrices in a small amount of storage (sparse, few distinct scalar values, or have some recursive structure), never have to store the same matrix twice, and memoize matrix operations in compact storage.
We have added algorithms to handle other recursive matrix operations (e.g. Kronecker Product) and algorithms (e.g. Sparse Block Discrete Fourier transform).

How LARC Improves Existing Work (cont.)

Our work on proximity hashing will also be useful to others who would like to send values that are physically close to each other to the same hash-chain for easy retrieval of an item which is close enough. The fact that we use an approximate value to determine where to hash the value, but store a value in full precision, coupled with pre-loading important identities, allows us to preserve important identities such as $\sqrt{2}^2 = 2$.

LARC also works on vectors and non-square, although we didn't show them in the slides. All the math works as long as the matrices are a power of 2 in each dimension. If you are working with matrices that are not 2^n by 2^m, it is possible to pad the matrix out with zeros to still use the package.

Acknowledgments

We are grateful to the many people who helped with us develop LARC by participating in discussions, testing existing code, adding additional capabilities, helping create the python SWIG wrapper, and working on applications which utilize LARC as a math library. These people include:

Matt Calef, Bill Carlson, John Fregeau, Lisa Jones, Larry Hiller, James Howse, JKK, Steve Kratzer, Laurie Law, Van Molino, Mark Motley, Mark Pleszkoch, Michael Schneider, Doug Shors, Michelle Snider, and Andy Wills.

Thank you!

– Steve Cuccaro, John Daly, John Gilbert, and Jenny Zito.

Backup Slides

Recursive Techniques for Matrix Operations

Applications of linear algebra may involve carrying out matrix operations on very large matrices. For many applications, these matrices can be very sparse (lots of zeros) or have a repetitive structure. We will talk about a method to carry out matrix math based on recursion (solving a big problem, by breaking it up into smaller problems). In the process we will talk about some tricks, like using hash tables to store matrices and to store operations. We are able to store very large matrices using limited memory in a recursively compressed format and to carry out matrix operations within this compressed format by using recursive formulations of the matrix operations. Our methods are useful in solving large science and engineering problems like carrying out Fourier transforms.

Most Effective for Sparse or Recursive Matrices

Extreme Example: The n by n identity matrix I_n

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dense Format: Requires n^2 words to store. Thus a 2^{15} by 2^{15} matrix fits in a lab top (8 gigibytes).
Sparse Row Format (list non zero position and value in each row): Requires $2 \times n$ words to store. Thus a 2^{15} by 2^{15} matrix fits in a cheap phone (512 kilobytes).
LARC Format: Requires about $log_4(n)$. Thus a 2^{15} by 2^{15} matrix fits in one printed paragraph (600 bytes).

Most Effective for Sparse or Recursive Matrices

Extreme Example: The n by n Hadamard matrix H_n which show up in Physics and Image Processing applications.

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Dense Format and Sparse Row are the same: Requires n^2 words to store. Thus a 2^{15} by 2^{15} matrix fits in a lab top (8 gigibytes).
LARC Format: Requires about $log_4(n)$. Thus a 2^{15} by 2^{15} matrix fits in one printed paragraph (600 bytes).