Locality hashing to mimic symbolic computation for recursively compressed matrix math

Jenny Zito (speaker), Steve Cuccaro, Mark Pleszkoch

Center for Computing Sciences
(LARC Team: larc@super.org)

May 2021
SIAM Linear Algebra Conference 2021

The LARC team (Jenny, Steve, and Mark) works on large matrix problems using recursive methods.

LARC

Linear Algebra via Recursive Compression

Center for Computing Sciences (IDA/CCS)

Federally Funded Research & Development Center.

`https://www.ida.org/ida-ffrdcs/center-for-communications-and-computing`

Our matrices are stored recursively by expressing each 2^n by 2^n matrix in terms of its four quadrant submatrices.

$$M = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \qquad \left[\begin{array}{cc|cc} 6 & 0 & 2 & 1 \\ 0 & 5 & 1 & 0 \\ \hline 0 & 2 & 6 & 0 \\ 4 & 1 & 0 & 5 \end{array}\right]$$

$$\left[\begin{array}{c|c} 6 & 0 \\ \hline 0 & 5 \end{array}\right] \quad \left[\begin{array}{c|c} 2 & 1 \\ \hline 1 & 0 \end{array}\right] \quad \left[\begin{array}{c|c} 0 & 2 \\ \hline 4 & 1 \end{array}\right] \quad \left[\begin{array}{c|c} 6 & 0 \\ \hline 0 & 5 \end{array}\right]$$

$$[\,6\,][\,0\,][\,0\,][\,5\,] \quad [\,2\,][\,1\,][\,1\,][\,0\,] \quad [\,0\,][\,2\,][\,4\,][\,1\,] \quad [\,6\,][\,0\,][\,0\,][\,5\,]$$

Assign an integer MatrixID to each unique submatrix. Specify a matrix by a list of the MatrixIDs of its quadrant submatrices.

Record for M: M_ID; [A_ID, B_ID, C_ID, A_ID]

Repetition of submatrices leads to compression.

Problem: numerical precision was spawning many nearly equal scalars.

We built a $2^{25} \times 2^{25}$ matrix using one of our applications packages. We were running out of memory, with nearly half a million distinct scalars. Sorting, we saw many near duplicates, differing by amounts around numerical precision:

$$\begin{aligned}
&\ldots \\
&1.3425612028604422246839 68e-17 \\
&1.3425612028604426869071 55e-17 \\
&1.4073865513297407057319 30e-17 \\
&1.4753419811653206417925 24e-17 \\
&1.4753419811653215662388 98e-17 \\
&\ldots
\end{aligned}$$

and checking the math we believed these were symbolically equivalent.

Near duplicate matrices were even a larger problem

The spawning of nearly identical scalars produces a many fold increase in the number of distinct matrices and submatrices that we were storing.

$$\begin{bmatrix} 6 & 0 & 2 & 1 \\ 0 & 5 & 1 & 0 \\ 0 & 2 & 6 & 0 \\ 4 & 1 & 0 & 5 \end{bmatrix} \qquad \begin{bmatrix} 6 & 0 & 2 & 1 \\ 0 & 5 & 1 & 0 \\ 0 & 2 & 6 & 0 \\ 4 & 1 & 0 & 5+\epsilon \end{bmatrix}$$

Gak!

So we would like to collapse scalars together that are almost identical.

However, there can be millions of scalars in our ScalarStore. We need an efficient way to find a previously stored scalar that is close to a newly calculated value.

GOAL: Find a nearby previously stored scalar (quickly), then "snap" to that value.

We divide scalar space into tiny tiles, | tile A | tile B |

with a tile_label function that is easy to compute from any scalar in the tile. tile_label_A tile_label_B

E.g., Shift the scalar left 80 bits and truncate the part after the decimal point. Each tiny tile of width $1/2^{80}$ has a unique integer tile_label.

scalar value s	tile_label(s)
$1.3425612028604422246839 68e-17$	13425612
$1.3425612028604426869071 55e-17$	13425612
$1.4073865513297407057319 30e-17$	14073865

GOAL: Find a nearby previously stored scalar (quickly), then "snap" to that value.

Store the first scalar s in a tile as the **representative** of the tile. Place its record in the ScalarStore hash table at the location **index = hash(tile_label(s))**.

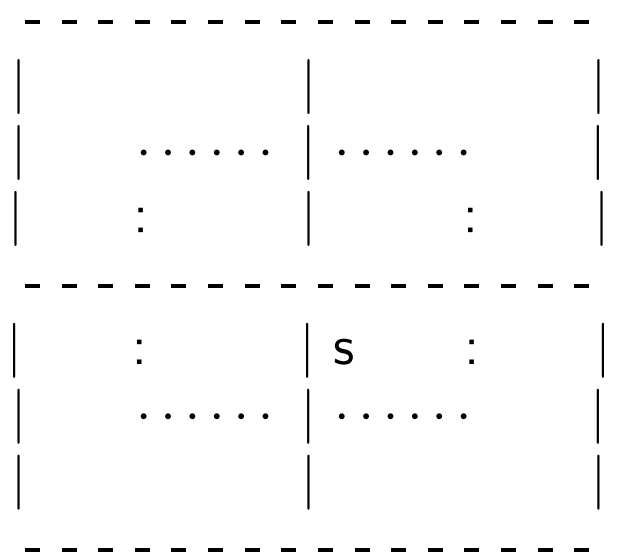
e.g., $s = 1.3425612028604426869071 55e-17$ is stored as the representative of tile 13425612 in the ScalarStore at index: index = hash(tile_label(s)) = hash(13425612)

Any subsequent scalars in that tile will be **snapped** to s.

e.g. $t = 1.3425612028604422246839 68e-17$ is not stored, since we find s in the ScalarStore at index: index = hash(tile_label(t)) = hash(13425612). We will return the MatrixID(s) and use s instead of t.

Region scheme guarantees successful snapping.

We create multi-tile regions designed so that the regional representative is some place in the center. This method is guaranteed to find a close-by previously stored value, if one exists within a quarter of the region width.



multi-tile region in complex scalar space

Goal: Mimic Symbolic / Exact Computation

We want LARC operations to mimic the behavior of certain symbolic equations. Say it is important in our application that

$$\sqrt{2} \times \sqrt{2} = 2.$$

In the LARC ScalarStore, we store the scalars $\{2, s\}$, where the scalar s approximates $\sqrt{2}$ with finite precision. We want a LARC calculation of the product of s with itself to "snap to" 2.

$$\texttt{product(MatrixID(}s\texttt{),MatrixID(}s\texttt{))} \overset{\textit{LARC returns}}{\longrightarrow} \texttt{MatrixID(2).}$$

LARC, operating with Regional Representative Retrieval, mimics symbolic computation as long as we preload important scalars (such as $\sqrt{2}$ in our example) with sufficiently high precision.

The first preloads are always the additive and multiplicative identities.

LARC always preloads the scalars 0 and 1 first. As well as the larger zero and identity matrices:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ldots.$$

This ensures the preservation of mathematical identities for a variety of matrix operations.

We also speed up implementation of the math identities by setting flags `is_zero` and `is_one` in their MatrixRecords.

Summary of Regional Representative Retrieval:

- Use a **locality-sensitive hash** to take scalars in the same **tile** to the same index in the storage table.
- The **region** for a stored scalar consists of multiple tiles that have been claimed in order to place this **representative** scalar (the first one stored) in the central portion of the region.
- The locality sensitive hash is used to check whether the tile for a new scalar has already been claimed by a region. If so, then the new scalar is not stored, but is **snapped** to that region's **retrieved** representative scalar.
- Consider **preloading** a set of scalars to high precision that are most important for mimicking symbolic identities for specific applications.

LARC (Linear Algebra via Recursive Compression) is available at `https://github.com/LARCmath`.

There is also a tutorial and sample applications package called MyPyLARC which uses LARC as its math package, and contains:

- a sparse recursive discrete Fourier transform,
- a lowest energy eigenvalue calculation,
- a triangle counter for graphs, and
- a quantum density matrix construction for a small part of the Google Sycamore circuit.

The LARC math package is primarily written in C and has a user-friendly SWIG generated Python wrapper. MyPyLARC is written in Python and illustrates the use of LARC.

The Danielson-Lanczos lemma expresses a DFT in terms of two-half sized DFTs in a block diagonal matrix.

$$F_3 = C_3 \times \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes F_2 \right) \times P_3 =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & w & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & w^2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & w^3 \\ 1 & 0 & 0 & 0 & w^4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & w^5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & w^6 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & w^7 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & i & -1 & -i & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & -i & -1 & i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & i & -1 & -i \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & -i & -1 & i \end{bmatrix} \times$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \left(\begin{array}{c} \text{where } w = e^{2\pi i/n} \\ \text{and } n = 2^3 \end{array} \right)$$

Preloading the roots of unity for the discrete Fourier transform matrix in LARC

We would like LARC to mimic closure under multiplication of roots-of-unity (the "twiddle" factors).

$$w_n^j \times w_n^k = w_n^m \qquad \text{for } m \equiv j + k \pmod{n}$$

where $w_n = e^{2\pi i/n}$ is the primary n-th root of unity.

We can create the analogous behavior in LARC

$$\texttt{product(MatrixID(}s_j\texttt{),MatrixID(}s_k\texttt{))} \overset{\textit{LARC returns}}{\longrightarrow} \texttt{MatrixID(}s_m\texttt{)}$$

under Regional Representative Retrieval, by preloading high precision approximations s_j for all the n-th roots of unity

$$s_j \approx w_n^j = \left(e^{2\pi i/n}\right)^j \qquad \text{for } j = 1, 2, \ldots n-1.$$

Questions?

A explanatory paper on LARC and MyPyLARC is available (along with the code) at `https://github.com/LARCmath`.

We have open problems in the paper. We also have a summer program that might be of interest to advanced undergraduate or graduate students.

The authors may be contacted at larc@super.org.

Additional Slides

How LARC stores scalars and matrices.

Each unique scalar is given a ScalarRecord which is stored in the ScalarStore hash table. Each ScalarRecord which contains a MatrixID and the value of the scalar. The hash function which determines the location of the record for scalar s in the store is `index = hash(tile_label(s))`. When a new scalar is calculated, the ScalarStore is checked to see if that scalar already has a MatrixID, or if there is another nearby scalar that is previously stored and can be used instead.

We discuss how the tile width parameters should be chosen in the paper `https://github.com/LARCmath/MyPyLARC/doc/LARCandMyPyLARC.pdf`

How LARC stores scalars and matrices.

Each unique matrices is given a MatrixRecord which is stored in the MatrixStore hash table. Each MatrixRecord contains a MatrixID for this matrix and a recursive description of the matrix given by the `SubMatList` which is the list of the four MatrixID's of the quadrant submatrices of the matrix. Row and column vectors may be stored with recursive definition in terms of two half-size vectors. The index of a matrix record in the MatrixStore is determined by `index = hash(SubMatList)` so that a record may be recovered if you know its quadrant submatrices.

$$M = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \qquad \text{Record for M:} \quad \text{M\_ID; } [\ \text{A\_ID, B\_ID, C\_ID, D\_ID}\ ]$$

Matrix Storage: LARC stores each unique matrix and submatrix once, using a compact recursive representation.

Each unique matrix M is given a MatrixID M_{ID} and its own record in the MatrixStore. For a matrix M with quadrant submatrices A, B, C, D

$$M = \left[\begin{array}{c|c} \text{A} & \text{B} \\ \hline \text{C} & \text{D} \end{array}\right],$$

its MatrixRecord contains its MatrixID and its recursive definition, listing the four quadrant submatrix MatrixIDs (NW,NE,SW,SE).

$$M_{ID} : [A_{ID}, B_{ID}, C_{ID}, D_{ID}].$$

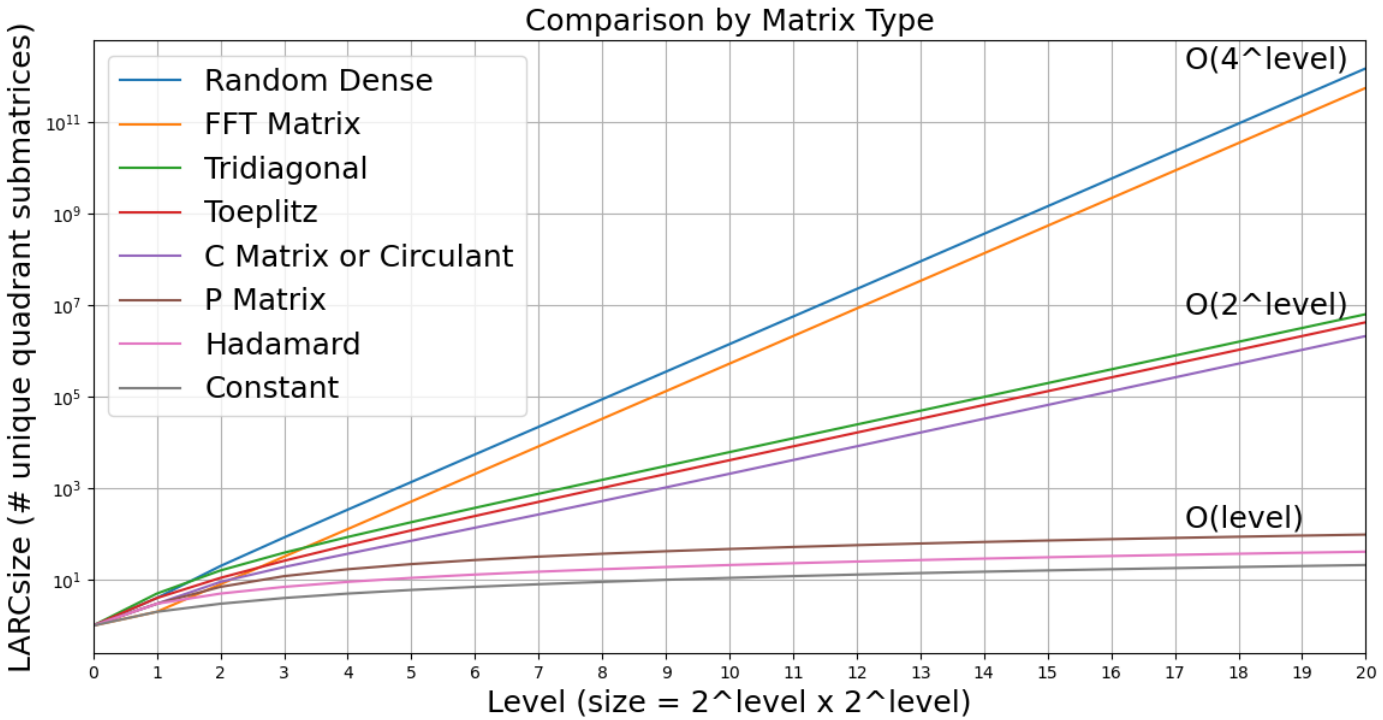
When M is 1×1, its MatrixRecord contains M_{ID} and the value of the scalar.

Example: How LARC stores the 2^3 by 2^3 matrix C_3.

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & w & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & w^2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & w^3 \\ \hline 1 & 0 & 0 & 0 & w^4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & w^5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & w^6 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & w^7 \end{array}\right]$$

MatrixID	Value of scalar or Recursive definition
A_{ID}	1
B_{ID}	0
C_{ID}	$e^{(2\pi i/8)}$
D_{ID}	$e^{2(2\pi i/8)}$
$\vdots$	$\vdots$
I_{ID}	$e^{7(2\pi i/8)}$
J_{ID}	$[A_{ID}, B_{ID}, B_{ID}, A_{ID}]$
K_{ID}	$[B_{ID}, B_{ID}, B_{ID}, B_{ID}]$
L_{ID}	$[A_{ID}, B_{ID}, B_{ID}, C_{ID}]$
M_{ID}	$[D_{ID}, B_{ID}, B_{ID}, E_{ID}]$
N_{ID}	$[F_{ID}, B_{ID}, B_{ID}, G_{ID}]$
O_{ID}	$[H_{ID}, B_{ID}, B_{ID}, I_{ID}]$
P_{ID}	$[J_{ID}, K_{ID}, K_{ID}, J_{ID}]$
Q_{ID}	$[L_{ID}, K_{ID}, K_{ID}, M_{ID}]$
R_{ID}	$[N_{ID}, K_{ID}, K_{ID}, O_{ID}]$
S_{ID}	$[P_{ID}, Q_{ID}, P_{ID}, R_{ID}]$

The LARCsize of a matrix is the number of unique quadrant submatrices in its quadtree.



A k-level matrix has sized $2^k \times 2^k$.

How LARC carries out matrix operations

Operations in LARC are carried out recursively in terms of the quadrant submatrices. The operations take MatrixID's as arguments and return a MatrixID. To define an operations you must express the operation in terms of the quadrant submatrices.

Operations that have been completed are given an OperationRecord in the OperationStore hash table (they are memoized) so that no operation need be repeated. The OperationRecord contains the MatrixIDs of the inputs and outputs, and the name of the operation. The index into the OperationStore hash table is `index = hash(operation name, MatrixIDs of inputs)`.

Example: How LARC carries out matrix operations recursively using quadrant submatrices.

$$\text{Given matrices } P = \left[\begin{array}{c|c} P_{00} & P_{01} \\ \hline P_{10} & P_{11} \end{array}\right] \text{ and } Q = \left[\begin{array}{c|c} Q_{00} & Q_{01} \\ \hline Q_{10} & Q_{11} \end{array}\right]$$

$$\text{ADD: } P + Q = \left[\begin{array}{c|c} P_{00} + Q_{00} & P_{01} + Q_{01} \\ \hline P_{10} + Q_{10} & P_{11} + Q_{11} \end{array}\right]$$

$$\text{MULT: } P \times Q = \left[\begin{array}{c|c} P_{00} \times Q_{00} + P_{01} \times Q_{10} & P_{00} \times Q_{01} + P_{01} \times Q_{11} \\ \hline P_{10} \times Q_{00} + P_{11} \times Q_{10} & P_{10} \times Q_{01} + P_{11} \times Q_{11} \end{array}\right]$$

LARC matrix operations take MatrixIDs as arguments. Hashtable storage is indexed by functions of MatrixIDs.

For example: $M_{ID} = \text{MULT}(P_{ID}, Q_{ID})$.

OperationStore look up:
Hash("mult" ,P_{ID},Q_{ID},) will determine if the result M_{ID} is already known.

After recursive calculation MatrixStore look up:
Hash($[A_{ID}, B_{ID}, C_{ID}, D_{ID}]$) determines if result M is stored and its MatrixID.

Matrix Operations: Our FFT example uses the operations ADD $+$, MULT $\times$, and tensor product KRON $\otimes$).

$$\text{Given matrices } P = \left[\begin{array}{c|c} P_{00} & P_{01} \\ \hline P_{10} & P_{11} \end{array}\right] \text{ and } Q = \left[\begin{array}{c|c} Q_{00} & Q_{01} \\ \hline Q_{10} & Q_{11} \end{array}\right]$$

$$\text{KRON: } P \otimes Q = \left[\begin{array}{c|c} P_{00} \otimes Q & P_{01} \otimes Q \\ \hline P_{10} \otimes Q & P_{11} \otimes Q \end{array}\right] \text{ if } P \text{ is not a scalar,}$$

$$= \left[\begin{array}{c|c} P \times Q_{00} & P \times Q_{01} \\ \hline P \times Q_{10} & P \times Q_{11} \end{array}\right] \text{ if } P \text{ is a scalar.}$$

LARC also can compress vectors and carry out matrix vector multiplication (which is needed for the FFT).

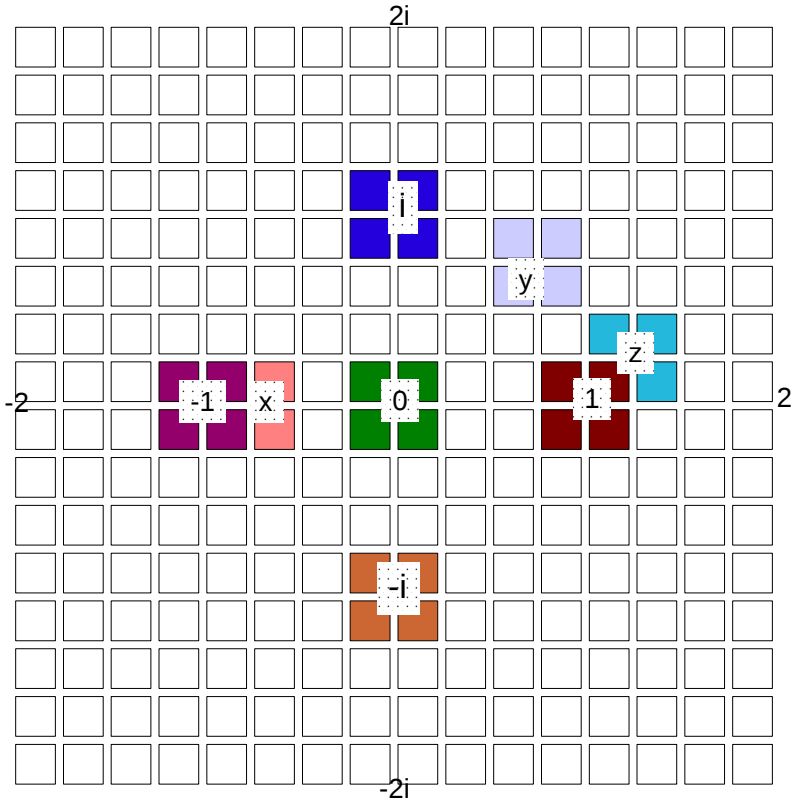
$$\text{Given matrix } P = \left[\begin{array}{c|c} P_{00} & P_{01} \\ \hline P_{10} & P_{11} \end{array}\right] \text{ and column vector} V = \left[\begin{array}{c} V_0 \\ \hline V_1 \end{array}\right]$$

$$\text{Matrix Vector Product: } P \times V = \left[\begin{array}{c} P_{00} \times V_0 + P_{01} \times V_1 \\ \hline P_{10} \times V_0 + P_{11} \times V_1 \end{array}\right]$$

Regions Claiming Tiles

In this example $\{0, 1, -1, i, -i\}$ are preloaded and their regions are able to claim 4 tiles.

We display the tiles claimed by $\{x, y, z\}$.



LARC can load sparse data into LARC compressed format.

LARC can load sparse data into LARC compressed format without ever having data in a dense matrix format.

Code for the loader is used in the triangle counting application in MyPyLARC to read in the benchmark data.

Density Matrix Formalism and Google Sycamore quantum circuit.

An explanation of how to make a quantum circuit into a matrix problem using the density matrix formalism is given in the explanatory paper LARCandMyPyLARC.pdf at

`https://github.com/LARCmath/MyPyLARC/doc`.

MyPyLARC demos this process in the Sycamore_play subdirectory with a portion of the Google Sycamore quantum circuit.

We are developing a bounding scalar type in LARC.

The bounding scalar type is described in the explanatory paper LARCandMyPyLARC.pdf at

`https://github.com/LARCmath/MyPyLARC/doc`

and will be used to bound probabilities in Markov chain computations.

References

S. Cuccaro, J. Daly, J. Gilbert, and J. Zito, LARC: Linear Algebra via Recursive Compression, GitHub:LARCmath LARC/doc, AACC, 2017.

J. Leskovec et al, Kronecker Graphs: An Approach to Modeling Networks, J. of Mach.

V. Strassen, Gaussian elimination is not optimal, Num. Math. 1969.

D. Wise and J. Franco, Costs of Quadtree Representation of Non-dense Matrices, J. Parallel and Distributed Computing, 1990.

G.C. Danielson, C. Lanczos, "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids", *J. Franklin Inst.*, vol. 233, pp. 365-380, 435-452, 1942.