LARC's Locality-Sensitive Hash and Initial Thoughts on Implementation of a Locality-Preserving Hash (ROUGH DRAFT)

Steve Cuccaro (IDA/CCS)
Mark Pleszkoch (IDA/CCS)
Jenny Zito (IDA/CCS)

June 2020

Contents

1 Locality-Sensitive Hashing: Numerical Precision and Identities (section primarily from paper MyPyLARC/doc/LARCandMyPyLARC.tex)

Finite precision can lead to math identities going wrong, such as $1-(1-a) \neq a$. In LARC, when we have two or more scalars that differ only by some small amount due to numerical precision issues, it becomes a large problem. Not only do we need to store multiple copies of what should be the same scalar, we also need to store many matrices which are essentially equal, but for these small errors.

We have developed a technique involving locality-sensitive hashing [3]. Locality-sensitive hashing is an algorithmic technique that hashes similar input items into the same hash buckets with high probability. In our case 'similar' means that two scalars are close to equal. The locality-sensitive hash ensures that scalars that would be in the same region hash to the same value, making it fast to find a previously stored representative of that region or to determine that no such representative exists.

When attempting to store a scalar value in LARC's MatrixStore, we first want to determine whether a sufficiently nearby scalar has already been stored, that we should use instead. We detect this with high probability by dividing the space of scalars into small regions, defined by a user-controlled parameter, and hashing all scalars that would be in the same region to the same hash bucket. The first scalar found in a particular region is stored in the MatrixStore and we call this scalar the *representative* of that region. Attempting to store any new value that would lie in an already occupied region will result in LARC returning the MatrixID of the original representative and not storing the new value.

This technique could clearly cause additional issues with failing identities when substitute scalars are used in mathematical operations. For example, if we have already stored the number 22 in LARC, and we try to store the scalar $s = 7\pi \approx 21.9911485751$, and we have set the size of regions to be large e.g. 1/32, then LARC will return the MatrixID for 22 instead of storing s. This can lead to further errors (see examples of how choices of initial parameters can affect this in MyPyLARC/Tutorial/preloading_mults_pi.py).

We have an additional technique called *preloading* which we use to guarantee that preselected identities are satisfied. For example, since zero is particularly important to mathematical identities, it is the first thing that we store when initializing LARC. This ensures that zero is the representative of its region, and any scalar within the region containing zero will be treated as zero. Depending on the application, we may also choose to preload a select set of scalars such as the n^{th}-roots of unity, to ensure they become the representatives of their regions and that mathematical identities with these scalars hold (see examples of this preloading to ensure identities in MyPyLARC/Tutorial/examples_roots_unity.py and fft_paramFile_init.py).

Because of the importance of identities involving zero, LARC gives the user the choice to make the region around zero larger than the other regions used by the locality-sensitive hash. We give the details for input parameters and region size determination in Section 2.

2 Locality-Sensitive Hashing in LARC (will be appendix in next version of paper LARCandMyPyLARC)

As described in the section above, LARC handles issues of numerical precision with a locality-sensitive hash (LSH) technique. We divide the space of scalars into regions and store at most one *representative* scalar for each region. When a new scalar is calculated, if there is already a scalar that has the same or a nearly identical value, LARC would ideally not store the new scalar and instead use the previously stored scalar. This version of LARC settles for solving this problem with high probability.

For speed of computation, the C code inside of LARC usually identifies a MatrixRecord by a pointer. From the Python interface we do not allow the user direct access to the memory, so we always refer to MatrixRecords by their MatrixIDs. In the following discussion we will say we are returning MatrixIDs, but internally a layer exists translating between MatrixRecord pointers and MatrixIDs.

When the user has a new scalar s and wants to determine whether the MatrixStore hash table already contains that scalar or another scalar t which is sufficiently close to s to use instead, LARC does the following.

1. Compute the hash function `LSH`(s), described below, to determine which hash chain to search for s.
2. Traverse the MatrixRecords in the hash chain and whenever the stored item is a scalar t check to see if s and t lie in the same region:
    - If t and s are in the same region, then do not store s and instead return the MatrixID(t) so that from now on t will be used as though it were s.
    - If t and s are not in the same region, continue traversing the hash chain.
3. If the end of the hash chain is reached without finding a scalar t in the same region as s, then create a new MatrixRecord for s, add it to this hash chain, and return MatrixID(s). The scalar s is now the unique representative of its region.

Because scalars that lie in the same region hash to the same hash chain, LARC only needs to search a single hash chain of the MatrixStore hash table to see if there is already a stored representative of the region or determine that no scalar from that region has been stored.

The locality-sensitive hash function of a scalar s is defined as:

$$\texttt{LSH}(s) = \texttt{Hash(LSH\_region\_center}(s))$$

where our Hash function is a multiplicative Fibonacci hash that we have implemented in order to get the maximally even distribution into hash buckets [2] [1]. The `LSH_region_center`() function returns the center of the region containing the scalar. Two scalars s and t are in the same hash chain if

$$\texttt{LSH}(s) = \texttt{LSH}(t).$$

There may be other scalars that randomly end up in the same hash chain so LARC must still check if

$$\texttt{LSH\_region\_center}(s) = \texttt{LSH\_region\_center}(t).$$

to determine whether s and t are in the same region.

The location and widths of the LSH regions depend on two parameters that are passed at initialization of LARC, `regionbitparam` and `zeroregionbitparam`.

Most regions are of width

$$w = 1/2^{\texttt{regionbitparam}}$$

and are centered about multiples of w. Since zero is particularly important to mathematical identities we want it to be the representive of its region, so it is *pre-stored*, meaning placed in the MatrixStore immediately after the initialization. Users have the option to make the LSH region containing zero larger than the standard regions, by setting the parameter `zeroregionbitparam` to be smaller than `regionbitparam`. When this is the case, the width of the region containing zero is given by the formula

$$w_z = 1/2^{\texttt{zeroregionbitparam}} - 1/2^{\texttt{regionbitparam}}.$$

This value approaches $1/2^{\texttt{zeroregionbitparam}}$ when `zeroregionbitparam` is very much smaller than `regionbitparam`. The reason for the slightly-odd formula is to make the region boundaries of the different-sized regions line up, and to maintain the property that all regions away from zero have the same size.

Here are a few examples setting $k =$ `regionbitparam`$-$`zeroregionbitparam`. We see that for $k =$ either 0 or 1, $w_z = w$; for $k = 2$, $w_z = 3w$; for $k = 3$, $w_z = 7w$; and for $k = 4$, $w_z = 15w$. In general for $k > 0$ the formula is $w_z = (2^k - 1)w$. We have assumed that it never makes sense to have the regions around zero be smaller than other regions, so if the user (accidentally?) enters a initialization parameters with the value of `zeroregionbitparam` set greater or equal to `regionbitparam`, then LARC ignores the `zeroregionbitparam` parameter and uses `regionbitparam` to calculate the width of all regions.

When scalars are any of the complex types, the calculation of region occurs independently for real and imaginary parts. This means that regions along the axes that are away from the origin are non-square rectangles when $w_z > w$; the rectangles are w wide along the axis and w_z wide perpendicular to the axis. The user can enter -1 for either or both of the LSH parameters to get a default value. The default value for `zeroregionbitparam` is to set it equal to `regionbitparam`, making all regions equal in size. The default for `regionbitparam` is set to two bits less than machine precision for C `long double`.

For non-complex spaces the region about zero is an interval `( , )` which does not include the boundaries. For regions on the positive side of zero we write the region `[ , )`, for regions on the negative side of zero we write the region `( , ]` to indicate that these regions include the boundary closer to zero and do not include the boundary further from zero. For complex scalarTypes, the region boundaries are marking off areas of the complex plane, with the region around the origin including none of its boundaries, any region along either axis including only the boundary closer to the origin, and other regions away from both axes including the two boundaries closer to the axes and excluding the other two boundaries. See Figure 1 below.

As described in the first section of this document, special identities can be preserved by pre-storing the scalars involved in the identities so that they become the unique representatives of their regions. This works most of the time, but there are cases when this is tricky. The routines in MyPyLARC/Tutorial involving multiples of π or roots-of-unity have examples that can help the user understand the techniques and pitfalls of using pre-storing to preserve identities.

```
...........  ..........  ...............   .........  .........
.          | .         | .               . |        . |        .
. -3w+3wI  | . -2w+3wI | .    +0w+3wI    . | 2w+3wI . | 3w+3wI .
.          | .         | .               . |        . |        .
.__________| ._________| ._______________. |________. |________.
...........  ..........  ...............   .........  .........
.          | .         | .               . |        . |        .
. -3w+2wI  | . -2w+2wI | .    +0w+2wI    . | 2w+2wI . | 3w+2wI .
.          | .         | .               . |        . |        .
.__________| ._________| ._______________. |________. |________.
...........  ..........  ...............   .........  .........
.          | .         | .               . |        . |        .
.          | .         | .               . |        . |        .
.          | .         | .               . |        . |        .
. -3w+0wI  | . -2w+0wI | .    +0w+0wI    . | 2w+0wI . | 3w+0wI .
.          | .         | .               . |        . |        .
.          | .         | .               . |        . |        .
.          | .         | .               . |        . |        .
...........| ..........| ................  |.........  |.........
.__________  ._________  ________________   ________.   ________.
.          | .         | .               . |        . |        .
. -3w-2wI  | . -2w-2wI | .    +0w-2wI    . | 2w-2wI . | 3w-2wI .
.          | .         | .               . |        . |        .
...........| ..........| ................  |.........  |.........
____________ ___________ ________________  _________.  _________.
.          | .         | .               . |        . |        .
. -3w-3wI  | . -2w-3wI | .    +0w-3wI    . | 2w-3wI . | 3w-3wI .
.          | .         | .               . |        . |        .
...........| ..........| ................  |.........  |.........
```

Figure 1: A diagram of the LSH regions near the origin for the case $k = 2$, not quite to scale. Each region is labeled by its center point, in which I is the squareroot of negative 1. The center column along the imaginary axis will be three times the width of the other columns and the center row along the real axis is is three times the width of the other rows. Dotted lines indicate non-inclusive boundaries.

3 Implementation of a Locality-Preserving Hash

The authors have written out an algorithm that would provide a locality-*preserving* hash instead of one that is merely *sensitive*, that is, it would guarantee the retrieval of any previously stored scalar value that was close to a new value, instead of only retrieving it with high probability. The current version can fail to find a nearby scalar if the two scalars lie on opposite sides of a region boundary. We might implement the locality-preserving hash in a future version of LARC if it seems like this would be helpful in solving some class of problems of interest.

3.1 Proposal for Locality-Preserving Hash Implementation

```
PROPOSAL for Experimental Branch of LARC with RegionRecords to
Create a Locality-Preserving Hash instead of a Locality-Sensitive Hash
=============================================================

This is compatible with still using a MatrixRecord to store a scalar.
However, for now let us assume that the term
    ScalarRecord  means matrix_record_t structure containing a scalar level =0
    MatrixRecord means matrix_record_t structure with level > = 1

Hash Table Storage:
* MatrixStore contains MatrixRecords (matrices of level > 0)
* OperationStore contains  OperationRecords w
   concerning both MatrixRecords and ScalarRecords
* RegionStore contains RegionRecords
   and RegionRecords link to ScalarRecords (matrices of level = 0)

Other Option:
* MatrixStore contains MatrixRecords (level > 0) and
                       RegionRecords
    hash chain where I used to put the ScalarRecord  (roughly speaking)

RegionRecord associated with any region in whcih I either have a
```

```
scalr that I have stored, or I have scalars in adjacent regions

that are within epsilon of the border

         ..........................
.           r?              .
        .   --------------------   .
.   |                   |  .
.   |                   |  .
.   |                   |  .
.   |      s S          |  .  t
.   |                   |  .
.   |                   |  .
        .   --------------------   .

         ..........................

These nearby scalars could be in 8 neighboring regions.

 Create a RegionStore  hash table contain RegionRecords

Region Record  for S
   9 fields
     rep region  s
     rep regN   r (only if that rep is within epsilon of the border
     rep regS
     rep regW   (blank because nothing was selected yet in that region)
     rep regE (blank because it was further than epsilon)
     regNW ...

How big is epsilon?
      Set epsilon to be smaller than w/2 say w/4 (really close by)


How to store and retrieve scalars?
```

```
      Whenever we want retrieve/store a scalar we go the the RegionStore
      do the normal LSH hash on the scalar to know which region to go to

      Search the chain, either there exists a single RegionRecord
      that is the correct one,

If there is RegionRecord for the Region
      which case at least one of
      these 9 entries (luckily our choice of region mapping makes num
      neighboring regions always 8 for complex and 2 for real)
          rep Region
          rep of N that is within epsilon of border
  rep of NW that is within epsilon
  ...

        In the RegionRecord go through our criteria to find
a representative that we identify.
    if s is within epsilon of existing rep t of the Region
       return t
    elsif s is within epsilon of any of the over the border guys b_i
       have some criteria to rank which to choose
       return b_best
            elsif there is an existing rep t,
       return t
            else
       create new scalarRecord for s
       add the scalarID of this scalarinto this RegionRecord
          as the region rep
       add the scalarID as region rep in this record
          and add it to any other RegionRecord if it is within epsilon
  of that border
     e.g. if we are within epsilon of N then go find
          the RegionRecord for N and add Srep = scalarID(s)

  (and this step might require creating a new RegionRecord
  that will now only have a single boundary record inside.)

         If we completed the search of the RegionStore hash chain for
```

```
 this region without finding a RegionRecord corresponding to this
 region, then such a record does not exist and
    create a new ScalarRecord for s
    create a new RegionRecord and put s in as the region rep
```

3.2 Cleaning Issues

We copied down this general discussion of cleaning, but should prune this down to the parts that bear on the LPH vs. LSH.

```
===========================================
General Discussion on Cleaning: (from April 28th)
------------------------------
All hash tables have the same general structure:

head of hash chain
in hash table          <->        hash node     <->      hash node   <->
indexed by hash value
                             (ptr to prev node)       (ptr to prev node)
                             (ptr to next node)       (ptr to next node)
                               (PTR to Record)          (PTR to Record)

Records are appropriate type for that Hash Table Store.

We always have table that tells us for each MatrixID what is the MatrixPTR
and if we remove that MatrixRecord then we set MatrixPTR to be NULL

For LSH (now):
  MatrixStore: MatrixRecord (now, big matrices and scalars)
  OperationsStore: OperationRecord
  InfoStore: InfoRecord

For LPH
  ScalarStore: RegionRecord (LPH, we could move scalar store).
               and RegionRecords contain one or more MatrixIDs which
       allow us to find the pointer to the associated MatrixRecords
```

```
      (for scalars).
  MatrixStore: MatrixRecord (now, big matrices and scalars)
  OperationsStore: OperationRecord
  InfoStore: InfoRecord

RegionRecord
   contains a number called the RegionCenter
   contains a slot for MatrixID of the RegionRepresentative
   contains 2 slots (for real types) and 8 slots (for complex types)
       which may contain the MatrixIDs that are the RegionRepresentatives
       of Regions that neighbor this region (in the case that the
       associated scalar in that region was sufficiently
       close to the region border).
   at least one slot is not empty, they could all be full.

In the LPH version:
Scalars are stored in the ScalarStore and the
MatrixID for a particular stored scalar could be inside RegionRecords for
at most 2 regions for real, and at most 4 regions for complex
(because the condition "sufficiently close to the border" restricts
the number of regions).

In the LSH version:
Scalars are stored in the MatrixStore and the
MatrixID for a particular stored scalar appears only once
in its own MatrixRecord.

In both versions:
The MatrixID for a particular stored scalar can appear in a huge number
of OperationsRecords.

For cleaning purposes, to delete a Scalar
------------------------
In LSH to delete a scalar from the MatrixStore,
LARC just has to delete a single MatrixRecord for that scalar.

In LPH to delete a scalar from the ScalarStore,
we might have to clean/delete up to 4 RegionRecords.
```

In both versions, when cleaning the OperationsStore it doesn't
make sense to try clean all the records of single scalar.
If are visiting a OperationsRecords during search, then we
check to see if any of the Pointer(MatrixIDs for input or output
are now gone) if they are we delete the record.
Option to clear out the entire OperationsStore.

To accomplish cleaning of the ScalarStore in LPH.
It would be handy to know the location of the up to 4
region records so we could clean/delete them.
So the empty slots for A,B,C,D (that would have been
used for the recursive definition of a matrix)
could become pointers to hash nodes for the
RegionRecords that contain that ScalarID.
PTR to the HashNode for the RegionRecord that contains
that ScalarID.

```
============================================================
(reduce the above general for this LPH treatment)

CLEANING out Scalars from Region Records:
    * since we are saving the matrixID (scalarID)
      if we cleaned, then we discover this by checking to
         see if PTR(scalarID) = NULL  (means scalar was removed)
    * Interesting: we had four extra empty slots in MatrixRecord that
       we were not using when we had a scalar.
         A  PTR to RegionRecord in which s is the rep
 B  PTR to RegionREcord in which s within epsilon
 C
 D  NULL
      There are at most 4 region records in which a particular scalar
      appears.
      Option for cleaning (or other algorithms).
    * Same counter system that exists already.
      We don't clean if any larger matrix refers to this record
```

3.3 Discussion on Hash revision cost/benefit

We have had some discussions on whether it was worth the effort to code the LPH. One issue that is hard to figure out in advance is whether the extra cost of record size being larger (in order to keep information on neighboring regions) is more than balanced out by having less spawning of nearly identical scalars (which leads to not only extra records for scalars, but extra records for nearly identical matrices, and for nearly identical operations).

We have done a few experiments, and hope to brainstorm soon on which experiments would help us decide whether to implement the LPH, or whether we actually need to implement it in order to accurately measure the relative merits of the LPH vs LSH.

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed, MIT Press, 2009.

[2] D. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2nd ed., Addison-Wesley, 1997.

[3] J. Leskovec, A. Rajaraman and J. Ullman, "Mining of Massive Datasets, Chapter 3", http://www.mmds.org.