SLAPS

Sparse **L**inear **A**lgebra on a **P**artitioned Global Address **S**pace

Greg Meyer

Spring 2018

Statement of original work:

All code in the source tree is original work by me for this project, with the following exception: `catch.hpp` is the header-only unit test framework Catch2, which can be found here.

The source code is free and open source online at:
https://github.com/GregDMeyer/slaps.

1 Introduction

SLAPS is a preliminary implementation of distributed sparse matrix-dense vector linear algebra, making use of the PGAS library UPC++. It is a header-only C++ library which currently defines two base class templates: `Vec`, which implements a distributed dense vector, and `Mat`, which implements a distributed sparse matrix. These classes are described below. All classes in SLAPS are templated, such that arbitrary data types can be used for both the indices and the data.

The most important features of SLAPS are:

- **Implicit** remote memory reads and writes in `Vec` class
- **Efficient** SpMV through one-sided UPC++ communication, in some cases outperforming PETSc's MPI implementation
- **Header-only** library allows for ease of use and templated API

1.1 Background

Linear algebra is a classic application for high-performance computing. Often, the matrices involved in real-world problems are "sparse"—many of the elements are zero. Computations involving such matrices occur in diverse fields, including physics and chemistry simulations, engineering, and machine learning. My own research in the physics department studying quantum many-body dynamics makes extensive use of large sparse matrices, to solve high-dimensional linear algebra problems like eigensolving and computing matrix exponentials. In my

field and others, the matrices and vectors are often so large that they must be distributed across many processors.

One of the most important operations in sparse linear algebra is the product of a sparse matrix and a dense vector (SpMV). This one operation is the basis of a large class of algorithms, which can make use of it to compute eigenvalues and functions of matrices. This work, SLAPS, is a framework for efficient sparse matrix-dense vector multiplication.

The traditional approach to distributed memory SpMV is to distribute the matrix and vector across processors' local memories, and then use the message passing interface standard (MPI) for communication. In general this can be quite efficient, but there are some downsides: in particular, two-sided MPI requires both the sending and receiving ranks to be prepared for that operation. When data access is unpredictable (as it often is in sparse linear algebra), knowing when to check for incoming MPI messages can be a difficult problem.

An alternative approach, the one pursued here, is to use a partitioned global address space (PGAS). PGAS languages and libraries allow direct reading and writing from a shared global memory, at the level of nodes' network cards. This means that any process can read from the memory locally stored on any node, without waiting for that node to check for incoming requests.

2 Implementation

Note: the API is well documented in the class template declarations at the beginning of each of the header files in the source tree. In this document I describe just a subset of the member functions in each class.

2.1 `Vec`: a distributed dense vector

The most obvious way to partition a dense vector across processors without duplication is to split the indices as evenly as possible, and give each rank a contiguous range. This is not the only way to partition a vector, but it is the approach that will be taken here.[1]

2.1.1 Data distribution

UPC++ implements a partitioned global address space through global pointers which affinity to one rank: the data is accessible from any rank through remote put and get operations, but it resides locally in one place. SLAPS thus stores the distributed vector as follows:

1. Each rank computes the distribution of vector indices (using the `partition_array` function in `utils.hpp`).

[1] Perhaps a future version of SLAPS will include other vector partitionings.

2. Each rank then allocates its own chunk of shared global memory, with a size corresponding to the rank's range of indices returned from the partitioning in step 1.

3. All ranks broadcast to all other ranks the global pointers to the shared memory allocated in step 2, which are each saved locally in an array.

Now, since every rank has the global pointers corresponding to every part of the vector, any rank can access any part of the distributed vector through UPC++ remote get and put calls.

The partitioning of the vector is available to the user through several functions. One can simply call `partition_array` directly to get the global partitioning, or one can use `Vec::get_local_range(I &start, I &end)` and related functions to get the current rank's local ownership. This is useful so that one can build a vector with minimal remote writes, reducing unnecessary communication.

2.1.2 Implicit remote memory access

The main way of implicitly accessing remote memory elements is through the `Vec::operator[]`. This method returns an object of type `RData`, which is defined in `proxy.hpp`. The `RData` object simply stores the global pointer to the value that has been requested. Like everything in SLAPS, `RData` is templated with index and data types `I` and `D`. The most important functions defined are:

- `RData::operator= (const D &val)`
  - Set the data represented by this `RData` object. The future for the write operation is chained to a global write future which is a member of the governing `Vec` class (see `Vec::set_wait()` below). These operations are not atomic, and care must be taken by the user to coordinates writes from different ranks.
- `RData::prefetch()`
  - Request the value from UPC++, and store the future internally.
- `RData::get()`
  - Wait on the future stored from `prefetch()` and return the value when it's ready. If it has not been called, `prefetch()` is called automatically.

The following additional operations are defined in the `Vec` class:

- `Vec::set_wait()`
  - Wait for all writes performed through `operator=` to complete. This is achieved by keeping an internal future in the `Vec` class, and conjoining all futures from calls to `RData::operator=` to it using `upcxx::when_all`.

- `Vec::read_range(I start, I end, D* buf)`
- `Vec::read_range_begin(I start, I end, D* buf)`
- `Vec::read_range_end()`
  - Request a range of contiguous values, even if they do not all reside on the same rank. The second and third functions represent the asynchronous version, in which work can be done between the calls to `begin` and `end`.

Here is an example of remotely getting and setting data in SLAPS:

```
Vec<int, double> v(100);

/* get and set arbitrary values */
v[52] = 3.14;              // set on any process
v.set_wait();              // wait for remote writes to complete

double val = v[52].get();  // get from any process

/* copy range into buf, even if it spans >1 processor */
v.read_range(0, 25, buf);  // copy v[0:25] into buf

/* all of the above can be done asynchronously */
auto remote_val = v[21];
remote_val.prefetch();
/* do other work */
double val2 = remote_val.get();
```

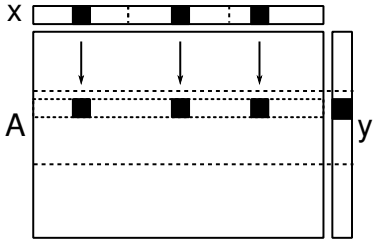
2.1.3 Vector functions

Finally, two mathematical vector functions are implemented:

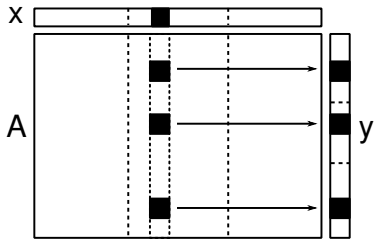
- `Vec::norm()`, a vector 2-norm and
- `Vec::dot(const Vec& b)`, the vector-vector dot product

2.2 `Mat`: a distributed sparse matrix

The other data structure implemented by SLAPS is the distributed sparse matrix. The matrix elements are not stored in globally shared memory, since for the SpMV implementations below, matrix elements do not need to be communicated. Instead the matrix is partitioned across processors, with each processor storing its portion in local memory.



(a) "Row-based" method. Reads from all processes; writes to self.



(b) "Column-based" method. Reads only from self; writes to all processes.

Figure 1: Sparse matrix partitioning schemes for SpMV. These figures graphically represent the operations $\boldsymbol{y} = A\boldsymbol{x}$ (the product of matrix A and vector $\boldsymbol{x}$ is being written into vector $\boldsymbol{y}$). Dotted lines represent partitioning across ranks ($p = 3$ in this example).

2.2.1 SpMV: Mechanics

There are several possibilities for parallel sparse matrix data storage formats and SpMV implementations. The vector partioning described above suggests two implementations for partitioning and SpMV: a row-based approach and a column-based approach (see Figure 1).

For an MPI-based implementation of SpMV, the row-based formulation has some subtleties. Communication must be done in two steps that require coordination: first, send the indices to the process that owns them, and second, receive the data. Both steps require the participation of both ranks. The column-based method would seem superior, since one can read from one's local array, then send computed values to another rank in one step. The disadvantage of MPI in general, however, is that the other rank must do extra work and coordination. It needs to know when to receive the values and how many, and then needs to sum the received values onto its array.

In our one-sided PGAS implementation the situation is quite different. We can do the remote reads required for the row-based method efficiently, and then we only need to write to our local array, which is extremely efficient. In fact, the column-based method is *worse* in a PGAS implementation, because several processes might attempt to write to the same remote value at once—requiring atomic operations. UPC++ doesn't even define atomic operations for floating point data types yet. It could be implemented using Remote Procedure Calls (RPC's), but that defeats the benefit of having one-sided communication.

So, it is clear that a row-based approach (read from all, write to self) is optimal for our PGAS implementation. With the partitioning set we still have some freedom to implement the actual SpMV operation in various ways, and currently four different methods are implemented in SLAPS. These are accessible as derived classes of the base `Mat` class, and are described below.

2.2.2 `Mat`: the base matrix class

The base class `Mat` defines operations and member data useful for all matrix types. It determines the partitioning of rows, and defines the user-visible functionality for setting nonzero matrix elements.

Row partitioning

The partitioning of rows is done in the same way as for the `Vec` class: using `partition_array` from `utils.hpp`.

Setting matrix elements

The setting of matrix elements occurs in two steps, only the first of which is performed by the base `Mat` class:

1. When users call `Mat::set_element(I row, I col, D val)`, the triple `(row, col, val)` is appended to a list of elements stored by the `Mat` class. This storage corresponds to what is referred to as COO (coordinate) format in the literature: just a 1-D array of coordinates and values. This is good because it allows quick matrix building, with no requirements that the users set elements in any sorted order.

2. Derived classes define a `DerivedMat::setup()` function, which takes the list of elements in COO format and translates it into the storage format defined by the derived class (see below).

Methods defined by derived classes

Derived classes are expected to define the `setup()` function just described, and also the SpMV implementations:

- `DerivedMat::dot(Vec& x, Vec& y)`, which performs the operation $\boldsymbol{y} = A\boldsymbol{x}$
- `DerivedMat::plusdot(Vec& x, Vec& y)`, which performs the operation $\boldsymbol{y} = \boldsymbol{y} + A\boldsymbol{x}$.

2.2.3 `CSRMat`: a base class for matrices using CSR storage format

Derives from: `Mat`

CSR (compressed sparse row) format is one of the most ubiquitous sparse matrix storage formats. It stores a jagged 2D array, for which the first dimension is equal to the number of local rows. The second (jagged) dimension contains each of the nonzero matrix elements in the corresponding row, stored as a `(col, val)` pair. Here `col` is the index of the column where the matrix element `val` sits.

The `CSRMat` class defines the function `CSRMat::setup()` described earlier, which translates the raw COO format matrix elements from calls to `Mat::set_element` into CSR format. It stores elements corresponding to local and remote reads from $\boldsymbol{x}$ separately for efficiency.

2.2.4 `NaiveCSRMat`: SpMV the naive way

Derives from: `CSRMat`

This class defines the `dot` operations in the naive way: just iterate through the rows in the CSR format, requesting remote values when we need them using the implicit remote gets defined in the `Vec` class. This implementation is not expected to be very efficient because it does not hide the latency inherent in the remote gets.

2.2.5 `SingleCSRMat`: Naive SpMV, but with prefetching

Derives from: `CSRMat`

This class defines effectively the same implementation as `NaiveCSRMat`, except that it explicitly prefetches values of `x` that will be needed soon. It only uses x-values once before throwing them away, possibly requiring the same value to be fetched many times. Thus it may be inefficient in general, but efficient on very sparse matrices in which only a few values are needed from `x` and they are not reused.

2.2.6 `BlockCSRMat`: Remote values prefetched in blocks

Derives from: `CSRMat`

This class prefetches entire portions of the `x` array, using the asynchronous version of the `Vec::read_range()` method, and multiplies these values agains all relevant matrix elements at once. These values are read regardless whether they need to be used (i.e. the read is not packed, since that would require extra communication). So, for denser matrices, this class is expect to outperform the `SingleCSRMat`, since it makes good reuse of values in the `x` array. However, for very sparse matrices, a lot of extra values are communicated, so at some level of sparsity we expect `SingleCSRMat` to be faster.

2.2.7 `RCMat`: a custom storage format for PGAS SpMV

Derives from: `Mat`

This is a custom format I implemented for this library, which I haven't seen in any literature (though perhaps it has been implemented before and I just haven't heard of it). I named it RC format for "row-partitioned columns."[2] It combines the benefits of both the communication efficiency of "`Single`" and value reuse of "`Block`" implementations above.

The data structure is visually represented in Figure 2. It is contained as a single array (implemented using a `std::vector`) of index-pointer pairs. The

[2] If anyone knows a name and previous implementation of this format, I would love to hear it!

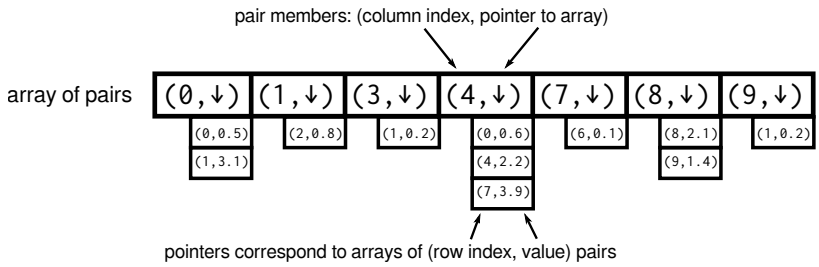


Figure 2: The `RCMat` data structure.

indices correspond to column numbers, and the pointers point to an array of row index-value pairs. The data structure is "row-partitioned" because the matrix elements contained in it still only correspond to those that would be stored by the process in a normal block-row format.

This data structure has numerous advantages:

- For each (column index, pointer) pair, a single value from the `x` array is multiplied by all the values in the (row index, value) array. This makes provably optimal reuse of `x` values: each value is fetched at most once, or zero times if it's not needed.
- Since column indices are stored, no unnecessary `x` values will be communicated.
- For prefetching, the up-front storage of column indices makes it trivial to calculate which indices will be needed soon, reducing computational overhead.
- Because matrix elements are still row-partitioned, all writes are done locally.

I expect this data structure to perform well for both very sparse and less sparse matrices: it makes optimal reuse of remote values, but also does not do any unnecessary communication.

3 Performance Results

These performance results were obtained using NERSC's Cori supercomputer. The measurements were performed using 1 process per node, for up to 32 nodes. The processor clock speed for the Haswell processors was limited to 2.3 GHz, to reduce variance from turbo boost. For each plot, all data points were obtained with the same set of nodes on Cori, to keep the effects of topology constant. In addition to testing the performance of SLAPS, the performance of the PETSc library's default SpMV was measured for comparison.

For these preliminary tests, matrices with uniform nonzero density across the row were used. For future testing, it would be interesting to examine performance on matrices which are clustered near the diagonal, or in bands.

For reproducibility, the build parameters for PETSc and SLAPS can be found in the source tree, under the "`benchmark/plots`" directory. The benchmark code itself is in "`benchmark/petsc`" and "`benchmark/slaps`".

The plots are located in the appendix.

3.1 Density scaling

The density scaling plot can be seen in Figure 3. It explores the behavior of the various implementations as the density of nonzeros in the sparse matrix is varied. This is not inherently a "parallel" benchmark, since each point was computed with the same number of nodes, but is important for understanding the scaling behavior.

The x-axis of the plot is nonzero density per row (i.e. number of nonzeros, divided by length of row). The y-axis is a measured of scaled computational time. Since a less dense matrix would trivially run faster, the runtime was divided by the density at each point to get a measure of "run time per unit density," which should be proportional to FLOP/s.[3]

These data points were all collected using 32 nodes, 1 processor per node on Cori Haswell. The matrix dimension was held constant at 10,000.

3.2 Strong scaling

The strong scaling plot can be seen in Figure 4a.

The x-axis is number of processors. Each processor was on a separate node. The y-axis is the "parallel efficiency," scaled to the performance of the PETSc benchmark on one processor. It is

$$\text{parallel efficiency} = \frac{(\text{PETSc } p = 1 \text{ runtime})}{(\text{runtime}) \times (\#\text{ processors})}$$

So, it's proportional to FLOP/s per processor.

For this computation, the matrix dimension was 30,000. It had a density of 10^{-2} (one nonzero per 100 matrix elements), and the computation was iterated 100 times for sampling.

3.3 Weak scaling

The weak scaling plot can be seen in Figure 4b.

The x-axis is the number of processors, and the y-axis is wall time in seconds (not scaled in any way). The matrix dimension was set as $d = 30,000 \times \sqrt{p}$, where p is the number of processors (giving a linear scaling of work with number

[3] In practice, the benchmark was just iterated a number of times equal to the inverse density, to obtain equal computational work.

of processors). The density was 10^{-2}, and the computation was iterated 50 times for sampling.

4 Discussion

First, for density scaling in Figure 3, I had predicted that the `BlockCSR` implementation would perform well for denser matrices, but that the `RC` format would overtake it at low matrix density, because it avoid unnecessary communication of unused x-values. However, it appears that the extra efficiency gained from communicating x-values in blocks completely dominates the costs of communicating unnecessary ones.

In the strong scaling plot, Figure 4a, both the `RC` and especially the `SingleCSR` formats do not appear to scale well. This is probably because the density for this plot was relatively high (10^{-2}), so communication of individual elements was a dominant cost. The SLAPS `BlockCSR` and PETSc implementations both see a jump in performance as the number of processors increases from 1, perhaps due to better cache use. As the number of nodes continues to increase, the efficiency decreases slightly, presumably due to communication cost. Interestingly, SLAPS `BlockCSR` format surpasses PETSc's performance at 32 nodes, and this outperformance was repeatable over several runs. The noisy nature of the `BlockCSR` strong scaling suggests that the block size needs to be tuned depending on the number of local rows. Hopefully that could lead to significantly better performance.

In the weak scaling plot (Figure 4b), we see that the `RC` format does not scale too favorably with problem size. Again, the overhead of individually requesting elements dominates. The `BlockCSR` and PETSc implementations do scale quite well: after an initial increase, their weak scaling behavior becomes flat.

It is notable that SLAPS outperforms PETSc for a range of nonzero densities in Figure 3. This seems to be attributable to faster communication through UPC++. Interestingly, however, in the weak scaling plot (Figure 4b), we see PETSc outperforming SLAPS on a similar benchmark. In both plots, the performance is consistent across several data points. This appears to imply that both implementations are similar in their efficiency, and small differences, such as the problem size and density, as well as topology of the compute nodes, can cause one to outperform the other.

SLAPS in general shows that UPC++, and PGAS more broadly, has potential to compete in performance with MPI for sparse matrix-dense vector multiplication. SLAPS is a preliminary implementation, and it would not be surprising to see performance significantly increase as the implementation is tuned. Especially helpful would be autotuning the prefetch block size in SLAPS to hide all of the latency from UPC. This is a clear next step in optimizing this library. SLAPS has been enjoyable to build and use, especially since it uses modern C++ (unlike PETSc, written in C). I hope that it is useful as a benchmark of UPC++, and can find applications in my research and others.

5 Appendix: Plots

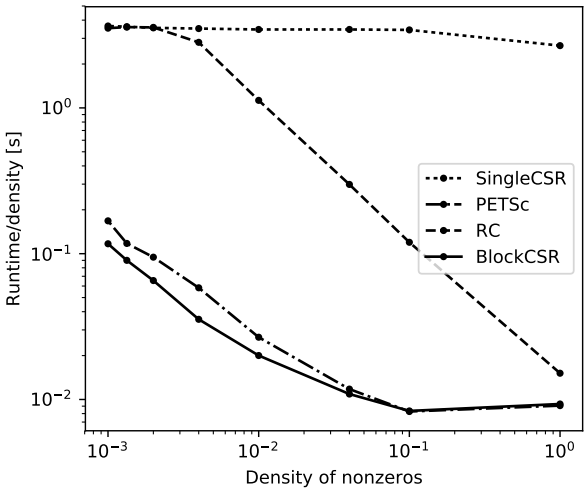
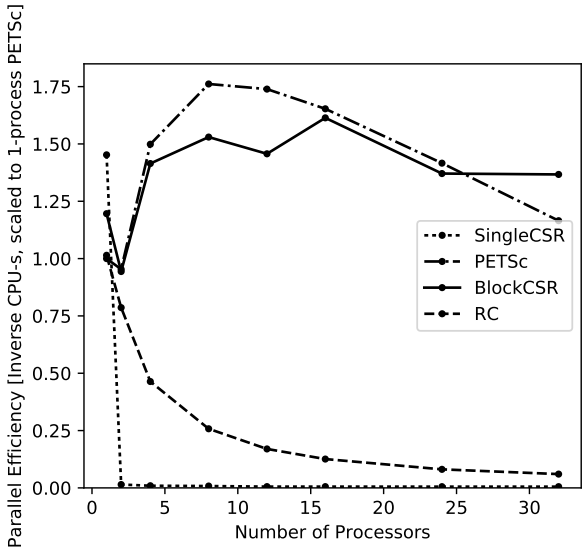
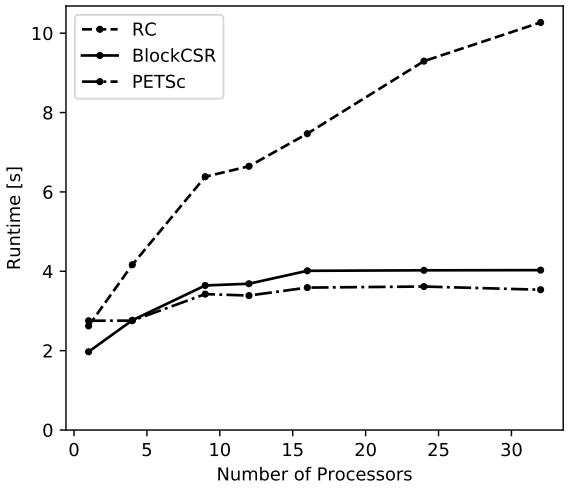


Figure 3: Density scaling behavior. All points were computing on 32 nodes, with 1 process per node. Matrix dimension was 10,000.



(a) Strong scaling behavior. Matrix dimension 30,000, density 10^{-2}. 100 iterations. 1 process per node.



(b) Weak scaling behavior. Density 10^{-2}, 50 iterations. Matrix dimension scaled as $d = 30,000 \times \sqrt{p}$.

Figure 4: Benchmarking plots for SLAPS. All benchmarks were performed on NERSC's Cori Haswell machine, with one process per node. Details are available in the source tree, under the `benchmark` directory.