Towards a GraphBLAS Library in Chapel

Ariful Azad, Aydın Buluç
{azad,abuluc}@lbl.gov
Computational Research Division
Lawrence Berkeley National Laboratory

***Abstract*—The adoption of a programming language is positively influenced by the breadth of its software libraries. Chapel is a modern and relatively young parallel programming language. Consequently, not many domain-specific software libraries exist that are written for Chapel. Graph processing is an important domain with many applications in cyber security, energy, social networking, and health. Implementing graph algorithms in the language of linear algebra enables many advantages including rapid development, flexibility, high-performance, and scalability. The GraphBLAS initiative aims to standardize an interface for linear-algebraic primitives for graph computations. This paper presents initial experiences and findings of implementing a subset of important GraphBLAS operations in Chapel. We analyzed the bottlenecks in both shared and distributed memory. We also provided alternative implementations whenever the default implementation lacked performance or scaling.**

I. Introduction

Chapel is a high-performance programming language developed by Cray [1]. Supporting a multithreaded execution model, Chapel provides a different model of programming than the Single Program, Multiple Data (SPMD) paradigm that is prevalent in many of the HPC languages and libraries.

GraphBLAS [2] is a community effort to standardize linear-algebraic building blocks for graph computations (hence the suffix -BLAS in its name). In GraphBLAS, the graph itself is represented as a matrix, which is often sparse, and the operations on graphs are expressed in basic linear algebra operations such as matrix-vector multiplication or generalized matrix indexing [3]. Chapel provides support for index sets as first class citizens, hence making it an interesting and potentially productive language to implement distributed sparse matrices.

In this work, we report on our early experiences in implementing a sizable set of GraphBLAS operations in Chapel. Our experience so far suggests that built-in implementations of many sparse matrix operations are not scalable enough to be used for large scale distributed computing. For some of these operations, we provide alternative implementations that improve the scalability substantially.

In many cases, we have found that adhering to a stricter SPMD programming style provides better performance than relying on the recommended Chapel multithreaded programming style. According to our preliminary analysis, this is due to the thread creation and communication costs involved in spawning threads in distributed memory, especially when the data size is not large enough to create work that would amortize the parallelization overheads. This problem, often called *burdened parallelism* in literature [4], is not specific to Chapel and it manifests in many parallel programming platforms such as OpenMP. However, the problem is exacerbated in distributed memory due to increased thread creation and communication costs. The primary goal in implementing a GraphBLAS-compliant library is performance. Consequently, we believe that divergence from the recommended programming style is justified as the library backend is rarely inspected by users and might not even be available for inspection.

II. Background

A. Matrix and vector notations

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is said to be sparse when it is computationally advantageous to treat it differently from a dense matrix. In our experiments, we only use square matrices and denote the number of rows/columns of the matrix by n. The *capacity* of a vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$ is the number of entries it can store. The $nnz()$ function computes the number of nonzeros in its input, e.g., $nnz(\mathbf{x})$ returns the number of nonzeros in $\mathbf{x}$. For a sparse vector $\mathbf{x}$, $nnz(\mathbf{x})$ is less than or equal to capacity($\mathbf{x}$).

Our algorithms work for all inputs with different sparsity structures of the matrices and vectors. However, for simplicity, we only experimented with randomly generated matrices and vectors. Randomly generated matrices give us precise control over the nonzero distribution. Therefore, they are very useful in evaluating our prototype library. In the Erdős-Rényi random graph model $G(n, p)$, each edge is present with probability p independently from each other. For $p = d/m$ where $d \ll m$, in expectation d nonzeros are uniformly distributed in each column. We use f as shorthand of $nnz(\mathbf{x})/\text{capacity}(\mathbf{x})$, which is the *density* of a sparse vector.

In this paper we only considered the Compressed Sparse Rows (CSR) format to store a sparse matrix because this is supported in Chapel. CSR has three arrays: $rowptrs$ is an integer array of length $n+1$ that effectively stores pointers to the start and end positions of the nonzeros for each row, $colids$ is an integer array of length nnz that stores the col ids for nonzeros, and $values$ is an array of length nnz that stores the numerical values for nonzeros. CSR supports random access to the start of a row in constant time. In Chapel, CSR matrices keep the column ids of nonzeros within each row sorted. In Chapel, the indices of sparse vectors are kept sorted and stored in an array. This format is space efficient, requiring only $O(nnz)$ space.

Listing 1: Creating a block-distributed sparse array

```
var n = 6
var D = {0..#n, 0..#n}  dmapped Block({0..#n
    ,0..#n}, sparseLayoutType=CSR));
var spD: sparse subdomain(D); // sparse
    domain
spD = ((0,0), (2,3)); // adding indices
var A = [spD] int; // sparse array
```

Listing 2: **apply()** - version 1

```
// Implementing apply() using forall loop
 proc Apply1(spArr, unaryOp)
 {
   forall a in spArr do
              a = unaryOp(a);
 }
```

Listing 3: **apply()** - version 2

```
// Implementing apply() with local arrays
  proc Apply2(spArr, unaryOp){
      var locArrs =  spArr._value.locArr;
      coforall locArr in locArrs do
      on locArr {
         forall a in locArr.myElems do
             a = unaryOp(a);
      }
  }
```

B. Chapel notations

A *locale* is a Chapel abstraction for a piece of a target architecture that has processing and storage capabilities. Therefore, a locale is often used to represent a node of a distributed-memory system.

In this paper we only used 2-D block-distributed partitions of sparse matrices and vectors [5], since they have been shown to be more scalable than 1-D block distributions of matrices and vectors. In 2-D block-distribution, locales are organized in a two dimensional grid and array indices are are partitioned "evenly" across the target locales. An example of creating a 2-D block-distributed sparse matrix is shown in Listing 1. A 2-D block-distributed array relies on four classes: (a) `SparseBlockDom`, (b) `LocSparseBlockDom`, (c) `SparseBlockArr` and (d) `LocSparseBlockArr`. `SparseBlockDom` and `SparseBlockArr` describe the distributed domains and arrays, respectively. `LocSparseBlockDom` and `LocSparseBlockArr` describe non-distributed domain and arrays placed on individual locales. `SparseBlockDom` class defines `locDoms`: a non-distributed array of local domain classes. Similarly, `SparseBlockArr` class defines `locArr`: a non-distributed array of local array classes. For efficiency, we directly manipulate local domains and arrays via `_value` field of classes. The actual local domains and arrays in `SparseBlockDom` and `SparseBlockArr` classes can be accessed by `mySparseBlock` and `myElems`, respectively.

C. Experimental platform

We evaluate the performance of our implementations on Edison, a Cray XC30 supercomputer at NERSC. In Edison, nodes are interconnected with the Cray Aries network using a Dragonfly topology. Each compute node is equipped with 64 GB RAM and two 12-core 2.4 GHz Intel Ivy Bridge processors, each with 30 MB L3 cache. We built Chapel version 1.14.0 from source using gcc 6.1.0. We built Chapel from source because the Cray-provided compiler on Edison is much older and does not have several latest sparse array functionalities. We used aries conduit for GASNet and slurm-srun launcher. Finally, qthreads threading package [6] from Sandia National Labs was used for threading.

III. GraphBLAS operations

The upcoming GraphBLAS specification and the C language API contains approximately ten distinct functions, not accounting for overloads for different objects [7]. The API does not differentiate matrices as sparse or dense. Instead, it leaves it to the runtime to fetch the most appropriate implementation. Consequently, it also does not differentiate operations based on the sparsity of its operands. For example, the MXV operation can be used to multiply a dense matrix with a dense vector, a sparse matrix with a sparse vector, or a sparse matrix with a dense vector. Efficient backend implementations, however, has to specialize their implementations based on sparsity for optimal performance.

In this work, we target a sizable subset of the GraphBLAS specification. Our operations are chosen such that they can be composed to implement an efficient breadth-first search algorithm, which is often the "hello world" example of GraphBLAS. Since we are illustrating an efficient backend, we also specialize our operations based on the sparsity of their operands. Below is the list of operations we focus in this paper:

- **Apply** operation applies a unary operator to only the nonzeros of a matrix or a vector.
- **Assign** operation assigns a matrix (vector) to a subset of indices of a another matrix (vector).
- **eWiseMult** can be used to perform element-wise multiplication of two matrices (vectors).
- **SpMSpV** multiplies a sparse matrix with a sparse vector on a semiring.

A powerful aspect of GraphBLAS is its ability to work on arbitrary semirings, monoids, and functions. In layman's terms, a *GraphBLAS semiring* allows overloading the scalar multiplication and addition with user defined binary operators. A semiring also has to contain an additive identity element. A *GraphBLAS monoid* is a semiring with only one binary operator and an identity element. Finally, a *GraphBLAS function* is simply a binary operator and is allowed in operations that do not require an identify element (e.g. eWiseMult).

Listing 4: **Assign()** - version 1

```
proc Assign1(A: [?DA], B: [?DB]) {
{
   //------ Assign domain -------
   DA.clear(); // destroy A
   DA += DB;
   // ------- Assign array -------
   forall i in DA do
      A[i] = B[i];
}
```

Listing 5: **Assign()** - version 2

```
proc Assign2(A: [?DA], B: [?DB]) {
  DA.clear(); // destroy A
  if(DB.size == 0) then return;
  //------ Assign domain -------
  var locDAs =  DA._value.locDoms;
  var locDBs =  DB._value.locDoms;
  coforall (locDA,locDB) in zip(locDAs, locD
      Bs) do
      on locDA {
      locDA.mySparseBlock += locDB.mySparseB
          lock;
      }
  // update global nnz of DA

  // ------- Assign array -------
  var locAs =  A._value.locArr;
  var locBs =  B._value.locArr;
  coforall (locA, locB) in zip(locAs, locBs)
       do
  on locA {
     forall (a,b) in zip(locA.myElems._value
         .data, locB.myElems._value.data) do
             a = b;
  }
}
```

A. ***Apply:*** *Applying a unary operator to a matrix or vector*

Apply takes a unary operator and a matrix (or a vector) as its input. It applies the unary operator to every nonzero of the matrix (vector). The computation complexity of **Apply** is $O(nnz)$ and it does not require any communication.

Chapel implementation. Listing 2 and Listing 3 provide two implementations of **Apply** operating on a sparse array. The function Apply1 in Listing 2 uses a data parallel **forall** loop to iterate over the nonzero entries of the array and applies the supplied unary operation on each entry. By contrast, the function Apply2 in Listing 3 follows an explicit SIMD model where one task is created and run on each locale using a **coforall** loop. Within each locale, local array entries are updated without incurring any communication.

Performance of Apply. Figure 1 shows the shared-memory and distributed-memory performance of two implementations of **Apply** on Edison. Both Apply1 and Apply2 show near-perfect scaling (20× speedup on 24 cores) on a single node of Edison, as expected. However, Apply1 does not perform well on the distributed-memory setting. Even though it is expected that a **forall** loop over a block-distributed domain or array executes each iteration on the locale where that iteration's index is mapped to, it is not implemented for sparse arrays yet. Hence, Apply1 requires lots of fine-grained communication, translating into the poor performance of Apply1. By contrast, Apply2 operates directly on the local array in each locale and shows good scaling as we increase the number of nodes in the right subfigure of Figure 1.

B. ***Assign:*** *Assigning a matrix/vector into another*

Assign operation can be used to assign a matrix (vector) to a submatrix (subvector) of another matrix (vector). For example, this can be accomplished with the Matlab notation $A(I, J) = B$ where I and J signifies row and column indices (respectively) of A into which the function assigns B. In general, assign is a very powerful primitive that can require $O((nnz(A)+nnz(B))/\sqrt{p})$ communication [8]. In this work, we implement a restrictive version of **Assign** that requires the domains of A and B to match. The computation complexity of this simplified **Assign** is $O(nnz(A))$ and it does not require any communication.

Chapel implementation. Listing 4 and Listing 5 provide two implementations of **Assign**. Here we assume that both A and B use the same 2-D distribution (i.e., the same index from their domains is always mapped to the same locale). The function Assign1 in Listing 4 clears the domain of A and then adds the indices from the domain of B. Next, Assign1 uses a data parallel **forall** loop to iterate over the nonzero entries of B and copy them to the corresponding locations of A. Here, we have to iterate over a domain because two sparse arrays are not allowed to iterate together (i.e., zipper iteration is not implemented for sparse arrays yet). By contrast, the function Assign2 in Listing 5 follows an explicit SIMD model where one task is created and run on each locale using a **coforall** loop. Within each locale, local domain and array entries are copied without incurring any communication. Notice that dense arrays stored in each locale can be zippered as shown in Line 18 of Listing 5.

Performance of Assign. Figure 2 shows the shared-memory and distributed-memory performance of two implementations of **Assign** on Edison. Here, Apply2 is an order of magnitude faster than Apply1. This is due to the fact that accessing the ith entry $A[i]$ of the sparse array A requires logarithmic time to find the entry from a compact representation of A. In Apply 2, we iterate over all nonzero entries which eliminates searching for each entry individually. Both Assign1 and Assign2 show reasonable scaling (5-8× speedup on 24 cores) on a single node of Edison. Similar to **Apply**, Assign1 does not perform well on distributed-memory. The reason is again the fine grained communication needed to access array entries.

C. ***eWiseMult:*** *Element-wise multiplication*

eWiseMult performs an element-wise multiplication of two matrices or vectors. The input matrices (vectors) should have the same domain or the output is not well defined. The

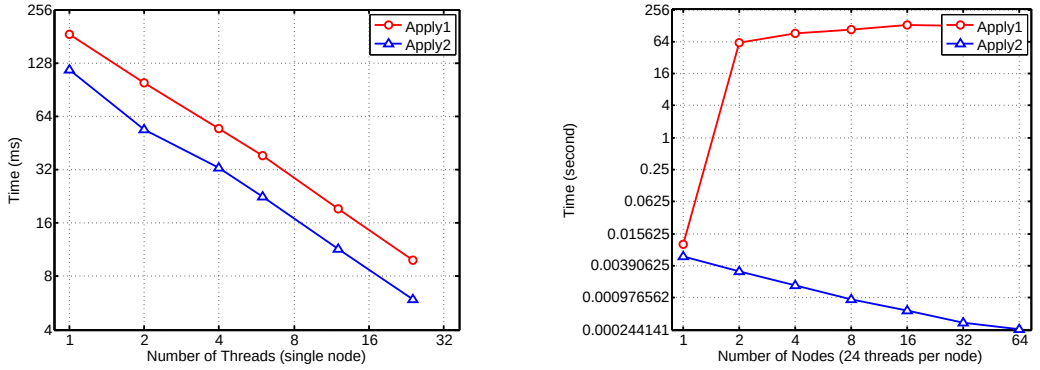

Fig. 1: Shared-memory (left subfigure) and distributed-memory (right subfigure) performance of two implementations of **Apply** on Edison. Input sparse vectors are randomly generated with 10M nonzeros.

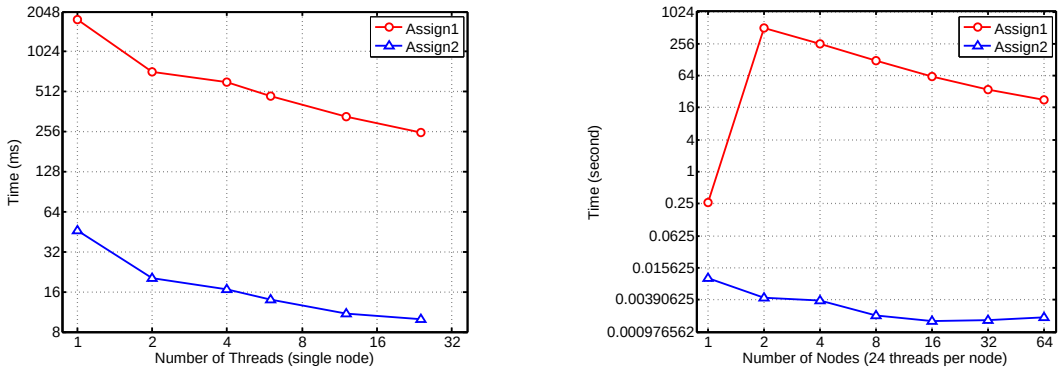

Fig. 2: Shared-memory (left subfigure) and distributed-memory (right subfigure) performance of two implementations of **Assign** on Edison. Input sparse vectors are randomly generated with 1M nonzeros.

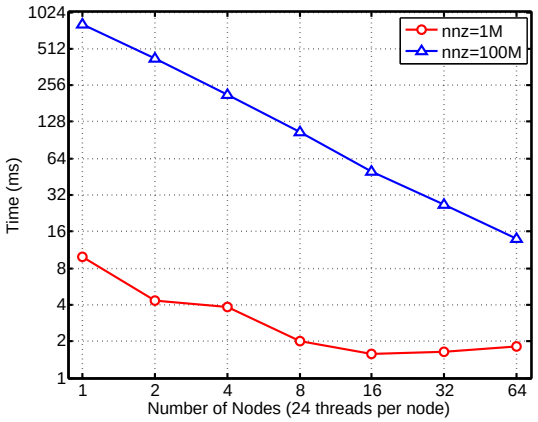

Fig. 3: The performance of the distributed-memory Assign operation (Assign2) on Edison. 24 threads are used on each node. Input sparse vectors are randomly generated 1M and 100M nonzeros.

name **eWiseMult** is a throwback to the mathematical roots of GraphBLAS. In practice, **eWiseMult** returns an object whose indices are the "intersection" of the indices of the inputs. The values in this intersection set are "multiplied" using the binary operator that is passed as a parameter. The computation complexity of **eWiseMult** is $O(nnz(A)+nnz(B))$ and it does not require any communication.

Chapel implementation. In this work, we focus only on the case where the multiplication is between a sparse vector and a dense vector. Listing 6 shows this specialized implementation. Listing 6 takes a sparse vector $\mathbf{x}$, a dense vector $\mathbf{y}$ and a binary operator $op()$ and returns a sparse vector $\mathbf{z}$. The ith entry $\mathbf{x}[i]$ of $\mathbf{x}$ is kept in $\mathbf{z}$ if $f(\mathbf{x}[i], \mathbf{y}[i])$ is satisfied. Similar to our implementation of **Assign** in Listing 5, we directly access local domains and arrays in each locale and create the local domains and arrays of $\mathbf{z}$. In contrast to **Assign** and **Apply**, we do not know the index set of $\mathbf{z}$ in advanced for **eWiseMult**. Hence, we use an atomic variable (Line 17 and 21 of Listing 6) to create a temporary dense array `keepInd` in each locale and then add these indices to the output vector using the "**+=**" operator. In practice, we can avoid the atomic variable by keeping a thread-private array in each thread and merge these thread-private arrays via a prefix sum operation.

Performance of eWiseMult. In our experiments, the dense vector $\mathbf{y}$ is simply a Boolean vector and the binary function $f(\mathbf{x}[i], \mathbf{y}[i])$ returns *true* when $\mathbf{y}[i]$ is *true*. We initialize $\mathbf{y}$ in a way that half the entries in $\mathbf{x}$ are kept in the output vector $\mathbf{z}$. Figure 4 shows the shared-memory performance of **EwiseMult** on a single node of Edison. In this experiment, 10K, 1M and 100M nonzeros of $\mathbf{x}$ are used. Going from 1 thread to 24 threads, we observe 13× speedup when $nnz(\mathbf{x})$ is 100M. This

Listing 6: **eWiseMult** : sparse-dense case.

```
proc eWiseMult(x: [?xDom], y, op)
{
    //Inputs. x: a sparse vector, y: a dense
        vector, op: a binary operator
    // Access local domains and arrays
    var lxDoms = xDom._value.locDoms;
    var lzDoms = zDom._value.locDoms;
    var lxArrs = x._value.locArr;
    var lyArrs = y._value.locArr;
    // Create the new domain
    const pDom = {xDom.low..xDom.high}
      dmapped Block({xDom.low..xDom.high});
    var zDom: sparse subdomain(pDom);
    coforall (lxDom,lzDom,lxArr,lyArr) in
        zip(lxDoms,lzDoms,lxArrs,lyArrs) do
      on lxDom {
        var nnz = lxDom.mySparseBlock.size;
        var keepInd:[0..#nnz] lxDom.idxType;
        var k : atomic int; k.write(0);
        forall (ind,val) in zip(lxDom.myS
            parseBlock,  lxArr.myElems) do
        //keep indices if the binary
            function is satisfied
          if(!op(val, lyArr[ind])) then
                keepInd[k.fetchAdd(1)]=ind;
        keepInd.remove(k.read(), nnz-k.read
            ());
        lzDom.mySparseBlock += keepInd;
   }
   //update global nnz of zDom
  var z: [zDom] x.eltType;
  //update values of z (not shown)
  return z;
  }
```

performance is reasonable and can be further improved by avoiding atomic operations. Figure 5 shows the distributed-memory performance of **eWiseMult** on Edison with (a) 1 thread per node and (b) 24 threads per node. When $nnz(\mathbf{x})$ is 100M, we see more than 16× speedup when we go from 1 node to 32 nodes. We do not see good performance for 1M nonzeros (and beyond 32 nodes for 100M nonzeros) because of insufficient work for each thread ($64 \times 24 = 1536$ threads in the right subfigure of Figure 5).

D. Sparse matrix-sparse vector multiplication (SpMSpV)

Sparse matrix-sparse vector multiplication is the operation $\mathbf{y} \leftarrow \mathbf{xA}$ where a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is multiplied by a sparse vector $\mathbf{x} \in \mathbb{R}^{1 \times m}$ to produce a sparse vector $\mathbf{y} \in \mathbb{R}^{1 \times n}$.

SpMSpV is the most complex operation we have implemented on Chapel for this paper. We provide a simple but reasonably efficient implementation using a sparse accumulator or SPA. The algorithm iterates of the nonzeros of the input vector x and fetches rows of $A[i,:]$ for which $x[i] \neq 0$. The nonzeros in those rows are merged using the SPA, which is a data structure that consists of a dense vector of values of the same type as the output y, a dense vector of Booleans (`isthere`) for marking whether that entry in y has been

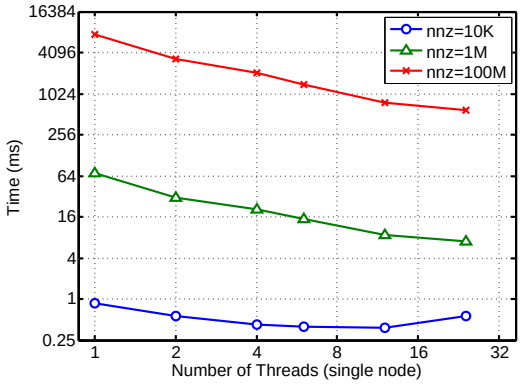

Fig. 4: The performance of shared-memory **eWiseMult** with a sparse and a dense vector on a single node of Edison. Nonzeros in the input sparse vectors are shown in the legend. Input sparse vectors are randomly generated with different number of nonzeros. About half of the nonzero entries are deleted after the **eWiseMult** operation.

initialized, and a list (or vector) of indices (`nzinds`) for which `isthere` has been set to "true". We note that there exists more efficient but complex algorithms for SpMSpV in the literature [9]. Figure 6 shows an example of the sparse matrix-sparse vector multiplication using a sparse accumulator.

Chapel implementation. Considering the complexity of **SpMSpV**, we provide separate shared- and distributed-memory implementations. Function `SpMSpV_shm` in Listing 7 shows the shared-memory implementation of **SpMSpV**. `SpMSpV_shm` has three steps: (a) merge necessary rows of the matrix based on the nonzero entries of the input vector $\mathbf{x}$ via SPA (b) sort indices of the merged entries, and (c) create the output vector $\mathbf{y}$ from SPA. For the SPA-based merging, the Boolean vector `isthere` keeps track of entries visited in the selected rows of the matrix. Since multiple threads can visit the same column in different rows, `isthere` is made atomic. `nzinds` stores the unique columns visited by all threads and `localy` stores the row indices where the stored columnes are discovered. In the second step, we sort unique columns identified by all threads stored in `nzinds`. In the final step, we create the output vector from the sorted indices and other SPA data structures.

Function `SpMSpV_dist` in Listing 8 shows the distributed-memory implementation of **SpMSpV** that uses `SpMSpV_shm` as a subroutine in each locale. We divide `SpMSpV_dist` into three steps: (a) gather parts of $\mathbf{x}$ along the processor row, (b) perform local multiply using `SpMSpV_shm`, and (c) scatter the output vector across processor columns via a shared SPA. The first and third steps require communication, while the second step performs bulk of the computation.

Performance of the shared-memory SpMSpV. Figure 7 demonstrates the performance of the shared-memory SpMSpV algorithm on a single node of Edison. For SpMSpV experiments, we generated Erdős-Rényi matrices with different sparsity patterns. Here n denotes the number of rows/columns

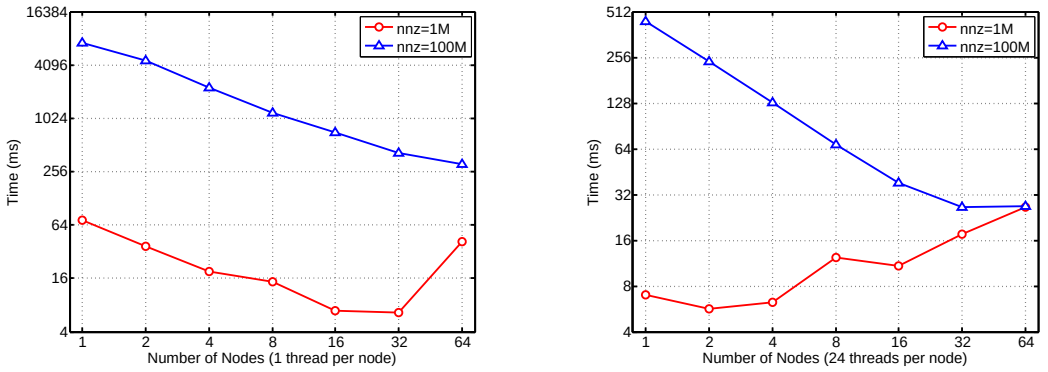

Fig. 5: The performance of distributed-memory **eWiseMult** with a sparse and a dense vector on Edison with (a) 1 thread per node and (b) 24 threads per node. Nonzeros in the input sparse vectors are shown in the legend. Input sparse vectors are randomly generated with different number of nonzeros. About half of the nonzero entries are deleted after the **eWiseMult**operation

Listing 7: **SpMSpV** : shared-memory implementation.

```
proc SpMSpV_shm(A: [?ADom], x: [?xDom])
{
 //Inputs. A: sparse matrix (CSR format), x:
      sparse vector
  var ciLow = ADom.dim(2).low;
  var ciHigh = ADom.dim(2).high;
  var ncol = ADom.dim(2).size;
  //allocate SPA data structures
  var isthere : [ciLow..ciHigh] atomic bool;
  var localy : [ciLow..ciHigh] int;
  var nzinds : [0..#ncol] int;
  var k : atomic int; k.write(0);
  // Step1: SPA
  forall (rid,inval) in zip(xDom,x){
    var rstart = ADom._value.rowStart(rid);
    var rend = ADom._value.rowStop(rid);
    if(rend>=rstart){
    // for each nonzero of the selected row
      for i in rstart..rend {
        var colid = ADom._value.colIdx(i);
        // only keeping the first index
        if(!isthere[colid].read()){
          nzinds[ k.fetchAdd(1)] = colid;
          isthere[colid].write(true);
          // keep row index as value
          localy[colid] = rid;
        }
      }
    }
  }
  // Step2: remove unused entries and sort
  nzinds.remove(k.read(), ncol-k.read());
  mergeSort(nzinds); //parallel merge sort
  //Step3: populate the output vector
  const yParentDom = {ciLow..ciHigh};
  var yDom: sparse subdomain(yParentDom);
  yDom += nzinds; // used specialized code
  var Y: [yDom] int;
  forall (si,sv) in zip(yDom, Y) do
      sv = localy[si];
   return Y;
}
```

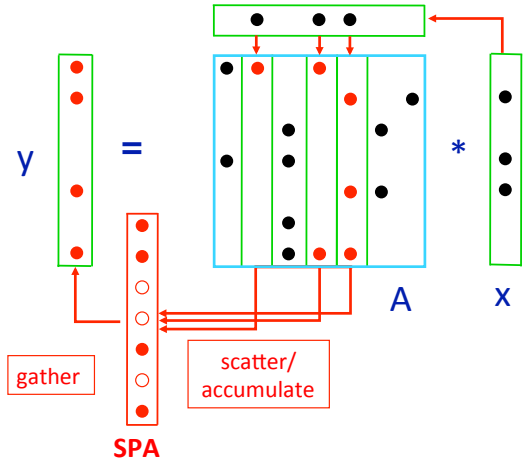

Fig. 6: Sparse matrix-sparse vector multiplication using a sparse accumulator (SPA) [10]. Our actual Chapel implementation is row-wise but we chose to draw the figure column-wise for better visualization. Neither the algorithm nor its complexity is affected by the use of row-wise vs. column-wise representation.

of the matrix and d denotes the number of nonzeros in each row of the matrix. We also randomly created the input vector that is f percent full meaning that it has nf nonzeros.

To better understand the performance, we showed three components of the shared-memory **SpMSpV** (as described in Listing 7) separately in Figure 7. Overall, `SpMSpV_shm` achieves 9-11$\times$ speedups when we go from 1 thread to 24 threads on Edison. In Figure 7, we observed that sorting is the most expensive step in shared-memory SpMSpV implementation. In our code, we use parallel merge sort available in Chapel. Since SpMSpV requires sorting of integer indices, a less expensive integer sorting algorithm (e.g., radix sort) is expected to reduce the sorting cost down, as was observed in our prior work [9].

Performance of the distributed-memory SpMSpV. Fig-

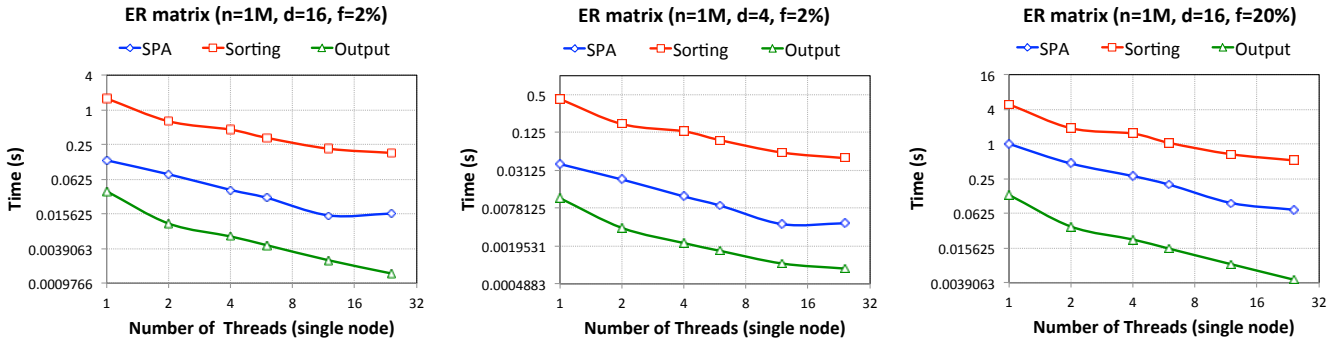

Fig. 7: The performance of the shared-memory SpMSpV algorithm on a single node of Edison. The scalability of each component of the algorithm is shown separately. Erdős-Rényi matrices with different parameters are used. The number of nonzeros in the input vector is nf.

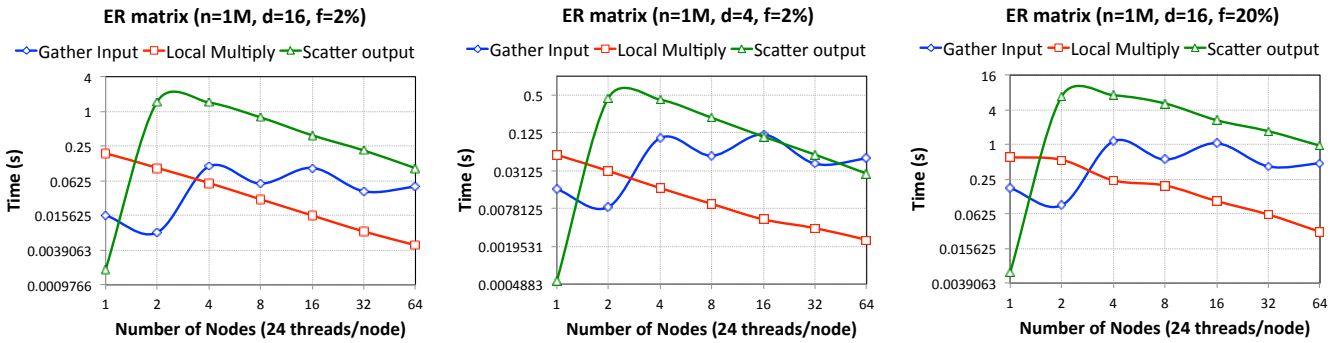

Fig. 8: The performance of the distributed-memory SpMSpV algorithm on Edison. The scalability of each component of the algorithm is shown separately. Erdős-Rényi matrices with 1 million rows and columns are used. The number of nonzeros in the input vector is nf.

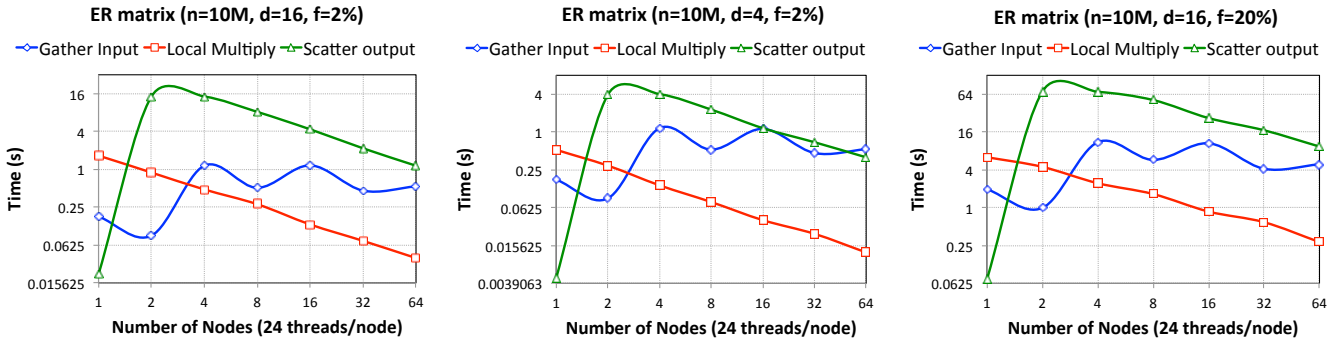

Fig. 9: The performance of the distributed-memory SpMSpV algorithm on Edison. 24 threads are used in each node. The scalability of each component of the algorithm is shown separately. Erdős-Rényi matrices with 10 million rows and columns are used. The number of nonzeros in the input vector is nf.

ure 9 demonstrates the performance of the distributed-memory SpMSpV algorithm on a single node for different sparsity patterns of the matrix and vector. As before, we showed three components of the distributed-memory SpMSpV (as described in Listing 8) separately in Figure 9. The total runtime of `SpMSpV_dist` does not go down as we increase the number of nodes because of expensive inter-node communication in gathering and scattering the vectors. Figure 9 shows that the computation time needed for the local multiplication attains up to 43× speedup when we go from 1 node to 64 nodes on Edison. However the communication time needed to gather the input vector increases by several orders of magnitude and dominates the overall runtime when run on multiple nodes. The communication time needed to scatter the output vector

Listing 8: **SpMSpV** : distributed-memory implementation.

```
proc SpMSpV_dist(A: [?ADom], X: [?xDom])
{
  var distM = ADom._value.dist;
  var n = ADom.dim(2).size;  //#columns of A
  const denseDom = {0..#n} dmapped Block
      ({0..#n});
  //global SPA
  var isthere: [denseDom]  atomic bool;
  coforall l in  distM.targetLocDom do on
      distM.targetLocales[l]  {
      ref lADom=ADom._value.locDoms[l].mySparseBlock;
      ref lArr=A._value.locArr[l].myElems;
      var pc = distM.targetLocDom.dim(2).size;
      // Step1: gather x along the processor row
      var lxDom: sparse subdomain({lADom.dim(1).low..lADom.dim(1).high});
      var rnnz = 0;
      for i in  distM.targetLocDom.dim(2) {
          var remotexDom =  xDom._value.locDoms[(l(1) * pc +  i)].mySparseBlock;
          rnnz += remotexDom.size;
      }
      lxDom._value.nnz = rnnz;
      lxDom._value.nnzDom = {1..rnnz};
      // copy remote parts of x along the processor row
      var rxi = 1;
      for i in  distM.targetLocDom.dim(2)  {
          var rxDom =  xDom._value.locDoms[(l(1) * pc +  i)].mySparseBlock;
          forall (si,di) in zip(rxDom._value.indices(1..rxDom.size), rxi..#rxDom.size) do
            lxDom._value.indices[di] = si;
          rxi += rxDom.size;
      }
      var lx: [lxDom] int;
      // copy value of x (not shown)
      //Step2: Local multiplication
      var ly = SpMSpV_shm(lArr, lx);

      //Step3: Scatter the output vectors across locales (only indices)
      forall (id,inval) in zip(ly.domain,ly) do
        if(!isthere[id].read()) then
          isthere[id].write(true);
  }
   // locally create output from isthere
   var y = denseToSparse(isthere);
   return y;
}
```

oscillates when we increase node counts. Further investigation is required to find the root cause of this oscillating behavior.

IV. Discussion of findings

Our effort is an ongoing work of implementing GraphBLAS operations in partitioned global address space (PGAS) languages including Chapel. Sparse matrix functionalities in Chapel are also under development and expected to perform significantly better in future releases. We will continue to improve the operations presented in this paper and add new functionalities to the GraphBLAS library. Here, we provide a high-level summary of our findings based on the preliminary evaluation of the GraphBLAS library in Chapel. We also provide some insights in language and library extensions that might improve the performance of graph analytics in Chapel.

- **In sparse array computation in Chapel, reasonable performance can be achieved by manipulating low-level data structures.** We have shown in our implementations of GraphBLAS operations that operations on sparse arrays can attain good performance if we directly manipulate low-level data structures. Therefore, we believe that high-level interface to the low-level data manipulations will improve the productivity of sparse array computation and graph analytics.
- **Data parallel accesses to a single sparse domain or array show good performance on a single locale.** On a single locale, data parallel accesses to a single sparse domain or array show good performance. We used this approach to implement **Apply** (Listing 2) that shows excellent scaling on a single node of Edison as shown in the left subfigure in Figure 1.
- **Data parallel accesses to a single sparse domain or array do not perform well on multiple locales.** It is expected that a **forall** loop over a block-distributed domain or array executes each iteration on the locale where that iteration's index is mapped to. However, this functionality is not implemented for sparse arrays yet. Hence, Apply1 defined in Listing 2 requires a large volume of fine-grained communication, translating into the poor performance as shown in the right subfigure of Figure 1.
- **Lack of support for data parallel zipper iteration with different sparse domains or arrays hurts performance.** Data parallel zipper iteration with different sparse domains or arrays is not allowed. Hence, for implementing **Assign** in Listing 4, we needed to iterate over the domain of an array and access elements of arrays by indices. Accessing an element of a sparse array by index requires logarithmic time, impacting the sequential and shared-memory performance as shown in the left subfigure of Figure 2.
- **Specialized implementations of sparse domain/array operation are needed.** The performance of most GraphBLAS operation will be improved if specialized implementations of sparse domain/array operation become available. For example, distributed-memory algorithms

can be simplified if the domain maps of two sparse arrays are exactly equal. Similarly, if a domain is guaranteed to be used by a single array, we can perform certain operations more effeciently. For example, **Assign** can be implemented by clearing the domain and then reinitializing it as described in Listing 4.

- **Bulk-synchronous communication of sparse arrays might improve the performance.** Sparse matrix computations often have very low computational intensity. Hence, a large volume of fine-grained communication negatively impacts the performance GraphBLAS operations. For example, the communication-heavy tasks of the distributed-memory SpMSpV dominate its runtime because we accessed remote entries of the inout and output vectors one element at a time. This performance impact is evident in Figure 9. We can mitigate this effect by using bulk-synchronous execution and batched communication [11]. We would like to mention that bulk-synchronous communication does not necessarily boost the performance of graph operations all the time. For example, traversing a small number of long paths in a bipartite graph matching algorithm benefits from fine-grained asynchronous communication [12].
- **Support for collective communication might improve the productivity and performance.** In our SpMSpV implementation, we gathered parts of the input vector along each processor column and scattered parts of the output vectors along each processor row. MPI provides functions for a number of team collectives. Support for these operation is expected to improve the productivity and performance of graph algorithms.
- **Placing multiple locales on a single compute node does not perform well.** It is often desirable to place multiple locales on a single compute node, especially in the presence of NUMA architecture. For example, Intel Ivy Bridge processor on Edison has two sockets. From our past experience, we observed that placing two or four MPI ranks on a node of Edison gives the best performance. However, the performance of our code degrades significantly when we placed more than one locales on a single node as shown in Figure 10.

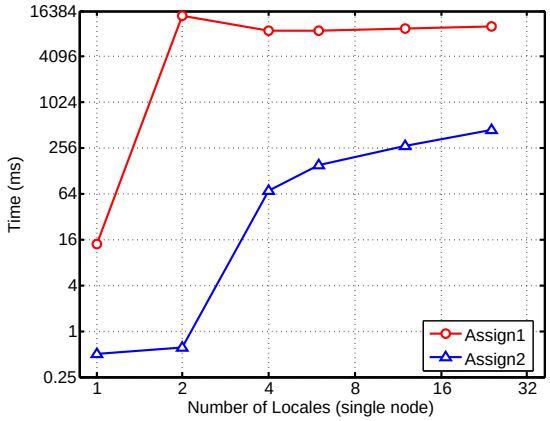

Fig. 10: The performance of two implementations of the distributed-memory Assign operation. All locales are placed on a single node of Edison. 1 thread per locale is used on each node. Input sparse vectors are randomly generated with 10,000 nonzeros.

V. Conclusions and Future Work

In this paper, we presented our approach for implementing a prototype GraphBLAS library. After evaluating built-in operations, in many cases, we opted to implement our own low-level functionality for better scalability and higher performance. We identified many shortcomings with the existing distributed sparse matrix support in Chapel and provided recommendations on how to remedy them. We have also found that Chapel excels in several important aspects regarding productivity and ease of use. In addition to providing the first set of primitives towards a functional graph processing library in Chapel, our work also sheds light on how to better exploit parallelism in Chapel for sparse matrices and vectors in other domains such as scientific computing.

Our future work consists of finishing a complete GraphBLAS-compliant library, ideally in collaboration with the Chapel language developers. While many algorithms for the remaining GraphBLAS primitives exist, efficient implementations of novel concepts in GraphBLAS, such as masks, have not been attempted in distributed memory before. Finally, we plan to implement and evaluate complete graph algorithms written in our GraphBLAS Chapel library.

Acknowledgments

This work is supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

References

[1] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[2] "The GraphBLAS Forum," http://graphblas.org/.

[3] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[4] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 145–156.

[5] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in chapel: Philosophy and framework," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association, 2010.

[6] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *Proceedings of the IPDPS*. IEEE, 2008.

[7] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *IPDPS Workshops*, 2017.

[8] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[9] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Proceedings of the IPDPS*, 2017.

[10] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[11] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011.

[12] A. Azad and A. Buluç, "Distributed-memory algorithms for maximum cardinality matching in bipartite graphs," in *Proceedings of the IPDPS*. IEEE, 2016.