Towards a GraphBLAS Library in Chapel

Ariful Azad & Aydin Buluç
Lawrence Berkeley National Laboratory (LBNL)

CHIUW, IPDPS 2017

Overview

- High-level research objective:
  - Enable productive and high-performance graph analytics
  - We used GraphBLAS and Chapel to achieve this goal

GraphBLAS

Building blocks for graph algorithms in the language of sparse linear algebra

Chapel

An emerging parallel language designed for productive parallel computing at scale

Both promise: Productivity + Performance

- Scope of this paper: A GraphBLAS library in Chapel

Outline

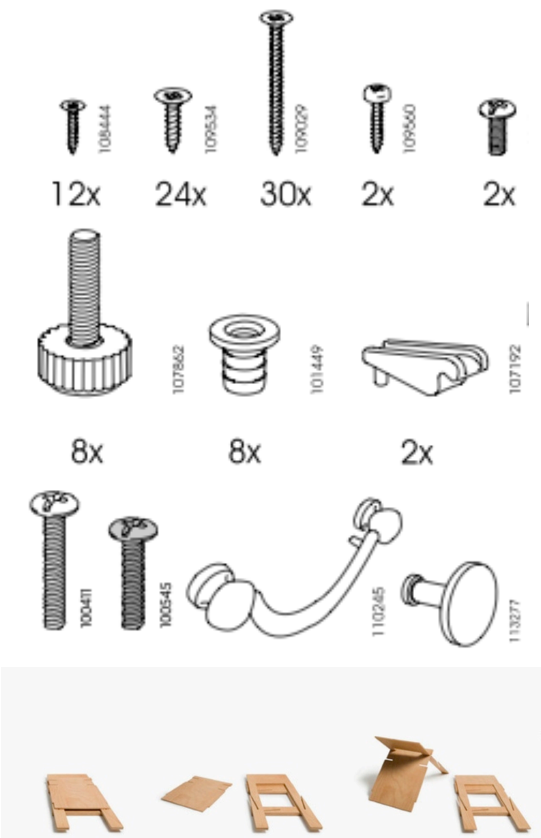
1. Overview of GraphBLAS primitives
2. Implementation of a subset of GraphBLAS primitives in Chapel with experimental results

Warning: this is just an early evaluation as Chapel's sparse matrix support is actively under development. All experiments were conducted on **Chapel 1.13.1.** The performance numbers are expected to improve significantly in future releases of Chapel.

Part 1. GraphBLAS overview

GraphBLAS analogy
A ready-to-assemble furniture shop (Ikea)

Building blocks



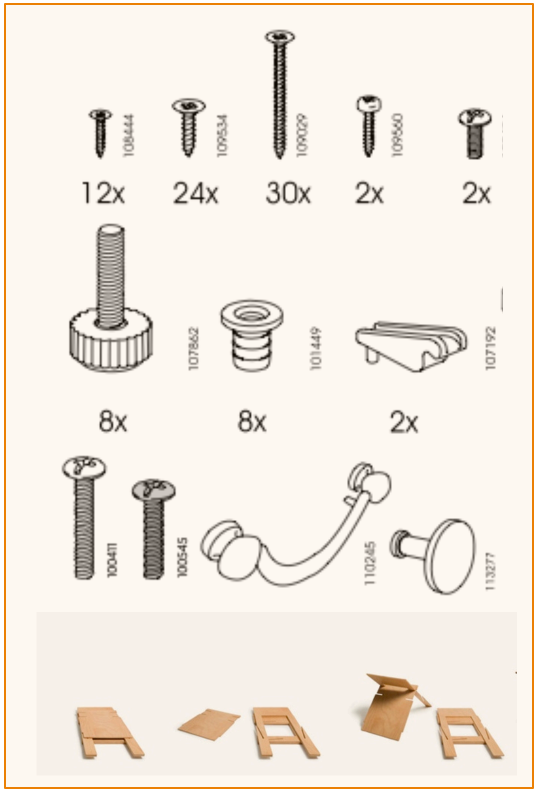
Objects (Algorithms)

Final product (Applications)

Graph algorithm building blocks

- GraphBLAS (http://graphblas.org)
  - Standard building blocks for graph algorithms in the language of sparse linear algebra
  - Inspired by the Basic Linear Algebra Subprograms (BLAS)
  - Participants from industry, academia and national labs
  - C API is available in the website

(*Design of the GraphBLAS API for C*, A Buluç, T Mattson, S McMillan, J Moreira, C Yang, IPDPS Workshops 2017)



GraphBLAS as algorithm building blocks

- Employs graph-matrix duality
  - Graphs => sparse matrix
  - A subset of vertex/edges => sparse/dense vector
- Benefits
  - Standard set of operations
  - Learn from the rich history of numerical linear algebra
  - Offers structured and regular memory accesses and communications (as opposed to irregular memory accesses in tradition graph algorithm)
  - Opportunity for communication avoiding algorithms

Some GraphBLAS basic primitives

Function	Parameters	Returns	Matlab notation
MxM (SpGEMM)	- sparse matrices A and B - optional unary functs	sparse matrix	C = A * B
MxV (SpM{Sp}V)	- sparse matrix A - sparse/dense vector x	sparse/dense vector	y = A * x
EwiseMult, Add, … (SpEWiseX)	- sparse matrices or vectors - binary funct, optional unarys	in place or sparse matrix/vector	C = A .* B C = A + B
Reduce (Reduce)	- sparse matrix A and funct	dense vector	y = sum(A, op)
Extract (SpRef)	- sparse matrix A - index vectors p and q	sparse matrix	B = A(p, q)
Assign (SpAsgn)	- sparse matrices A and B - index vectors p and q	none	A(p, q) = B
BuildMatrix (Sparse)	- list of edges/triples (i, j, v)	sparse matrix	A = sparse(i, j, v, m, n)
ExtractTuples (Find)	- sparse matrix A	edge list	[i, j, v] = find(A)

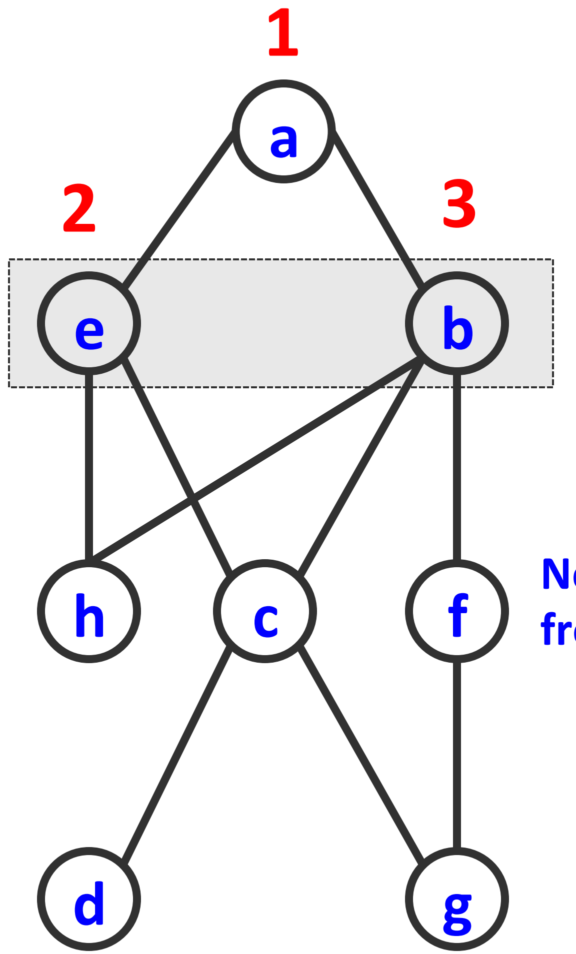
General purpose operations via semirings (overloading addition and multiplication operations)

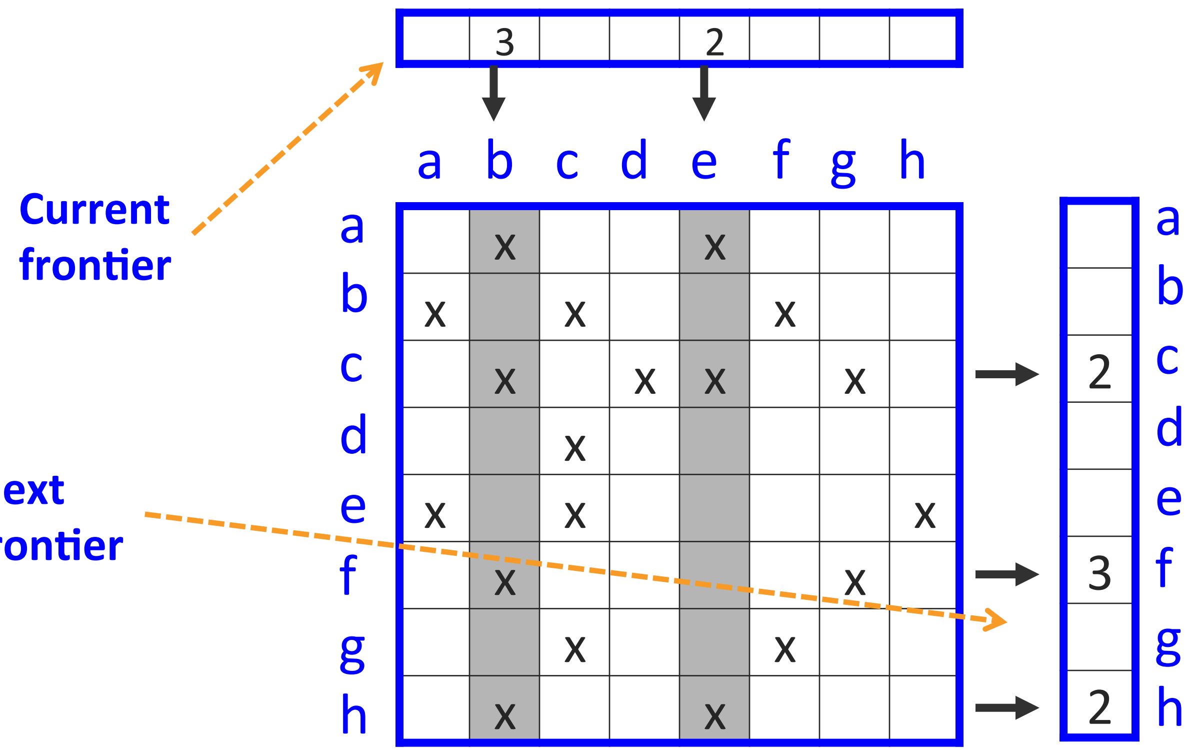
Real field: **(R, +, x)**	Classical numerical linear algebra
Boolean algebra: **({0 1}, |, &)**	Graph traversal
Tropical semiring: **(R U {∞}, min, +)**	Shortest paths
(S, select, select)	Select subgraph, or contract nodes to form quotient graph
(edge/vertex attributes, vertex data aggregation, edge data processing)	Schema for user-specified computation at vertices and edges
(R, max, +)	Graph matching &network alignment
(R, min, times)	Maximal independent set

- **Shortened semiring notation: (Set, Add, Multiply)**. Both identities omitted.
- **Add:** Traverses edges, **Multiply:** Combines edges/paths at a vertex

Example: Exploring the next-level vertices via SpMSpV

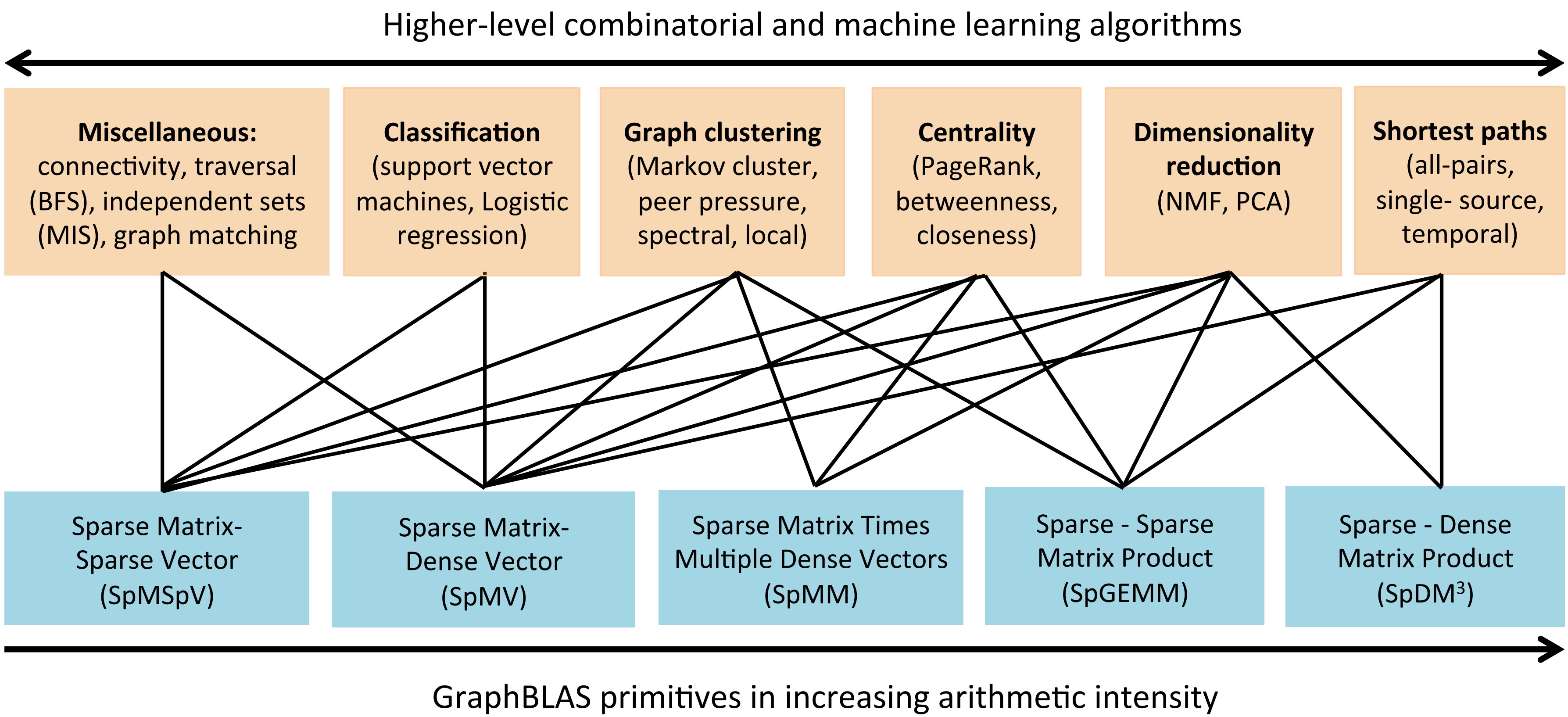
Overload (multiply,add) with (select2nd, min)





Adjacency matrix

Algorithmic coverage



- Develop high-performance algorithms for 10-12 primitives.
- Use them in many algorithms (boost productivity).

Expectation: two-layer productivity

Graph algorithms — user space

use

GraphBLAS operations — library

use

Chapel's productivity features — language

Part 2. Implementing a subset of GraphBLAS operations in Chapel

For Chapel: A subset of GraphBLAS operations

	Parameters	Returns	
Apply	x: sparse matrix/vector f: unary function	None	x[i] = f(x[i])
Assign	x: sparse matrix/vector y: sparse matrix/vector	None	x[i] = y[i]
eWiseMult	x: sparse matrix/vector y: sparse matrix/vector	z: sparse matrix/vector	z[i] = x[i] * y[i]
SpMSpV	A: sparse matrix x: sparse vector	y: sparse vector	y = Ax

Experimental platform

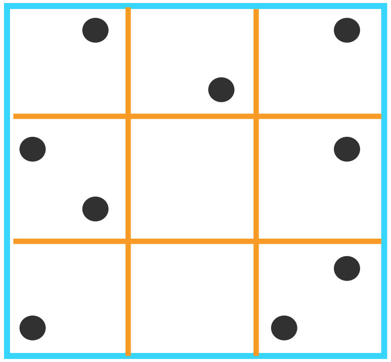
- Chapel details
  - Chapel 1.13.1 (the latest version before the IPDPS deadline)
  - Chapel built from source
  - CHPL_COMM: gasnet/gemini
  - Job launcher: slurm-srun
- Experiment platform: NERSC/Edison
  - Intel Ivy Bridge processor
  - 24 cores on 2 sockets
  - 64 GB memory per node
  - 30-MB L3 Cache

Sparse matrices in Chapel

- Block distributed sparse matrices. The dense container is block distributed.
- We used compressed sparse block (CSR) layout to store local matrices.

```
var n = 6
const D = {0..n-1, 0..n-1}
        dmapped Block(1..3,1..3);
var spD: sparse subdomain(D);
var A = [spD] real;
```

In this example:
#locales = 9



In our results, we did not include time to construct arrays

The simplest GraphBLAS operation: **Apply** (x[i] = f(x[i]))

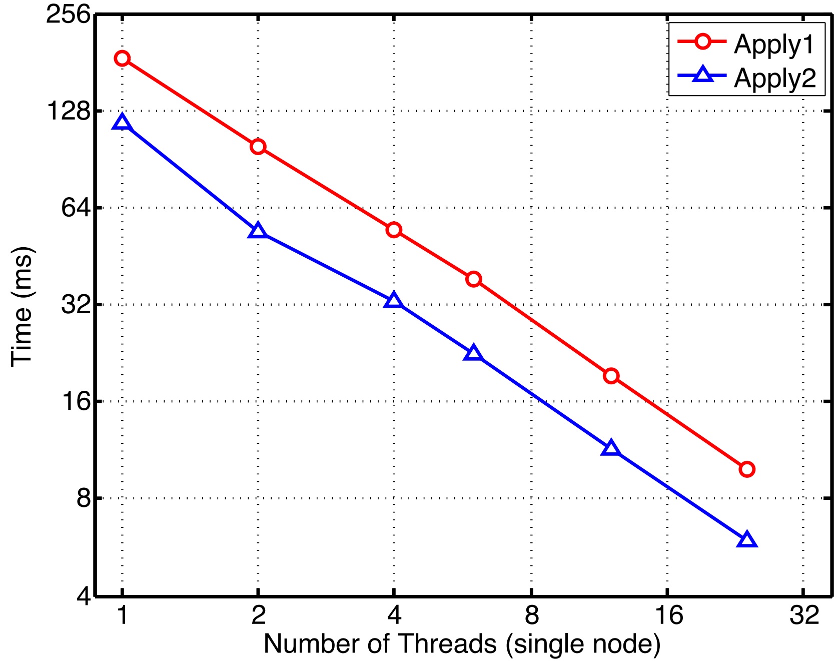
```
// Implementing apply() using forall loop
 proc Apply1(spArr, unaryOp)
 {
   forall a in spArr do
              a = unaryOp(a);
 }
```

Apply1:
high-level
(Chapel style)

```
// 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);
      }
  }
```

Apply2
manipulating
internal arrays
(MPI style)

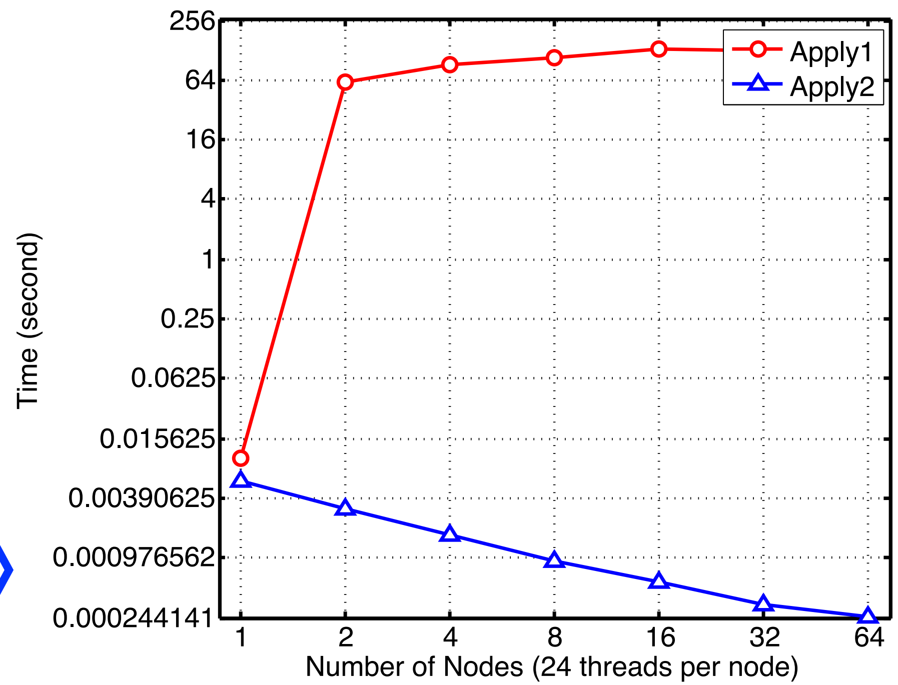
Example, simple case : **Apply** (x[i] = f(x[i]))



Apply1: high-level (Chapel style)
Apply2: manipulating internal arrays (C++ style)
x: 10M nonzeros
Platform: NERSC/Edison

Data parallel loops perform well in shared memory

But do not perform well in distributed memory

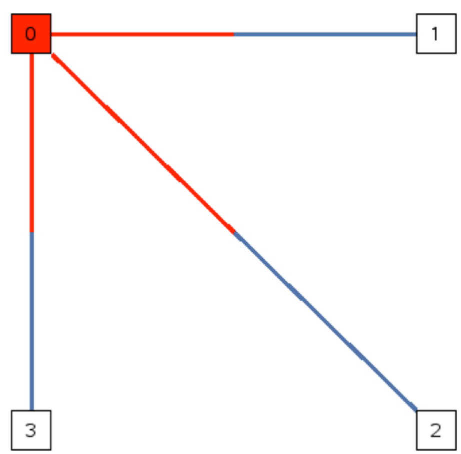


Performance on distributed-memory

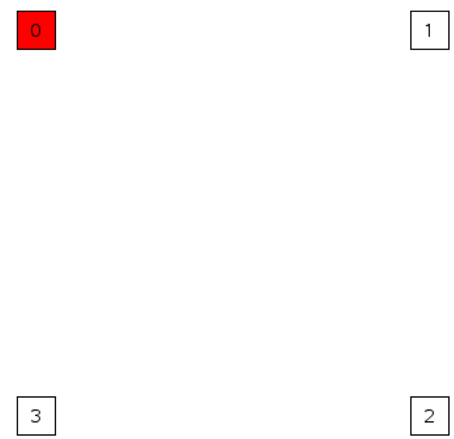
Using chplvis on four locales

Red: data in, blue: data out

Apply 1



Apply 2



All work at locale 0

This issue with sparse arrays has been addressed about a week ago

Assign x[i] = y[i]

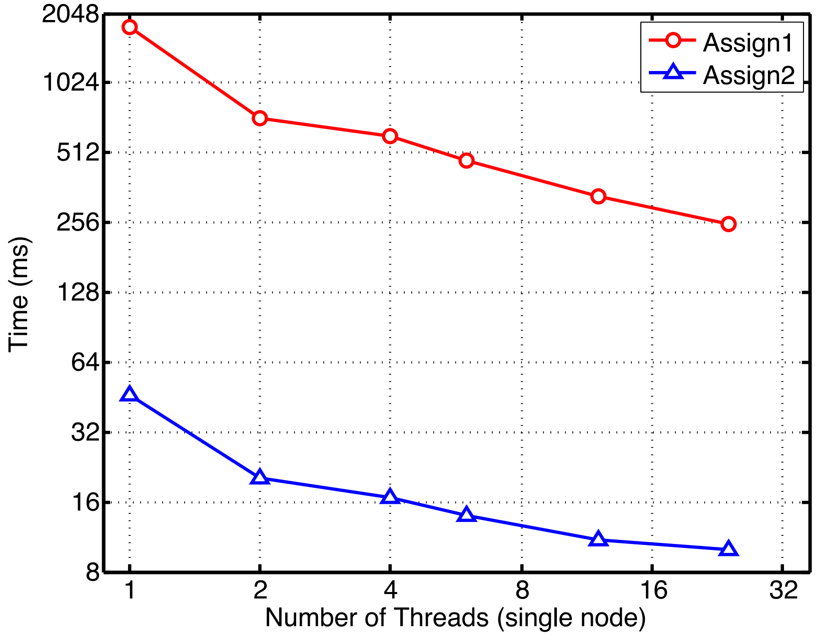
```
proc Assign1(A: [?DA], B: [?DB]) {
{
   //------ Assign domain -------
   DA.clear(); // destroy A
   DA += DB;
   // ------- Assign array -------
   forall i in DA do
      A[i] = B[i];
}
```

Assign1: high-level (Chapel style)

```
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;
      }
```

Assign2: manipulating internal arrays (MPI style)

Shared-memory performance: **Assign** (x[i] = y[i])



Assign1: high-level (Chapel style)

Assign2: manipulating internal arrays (C++ style)

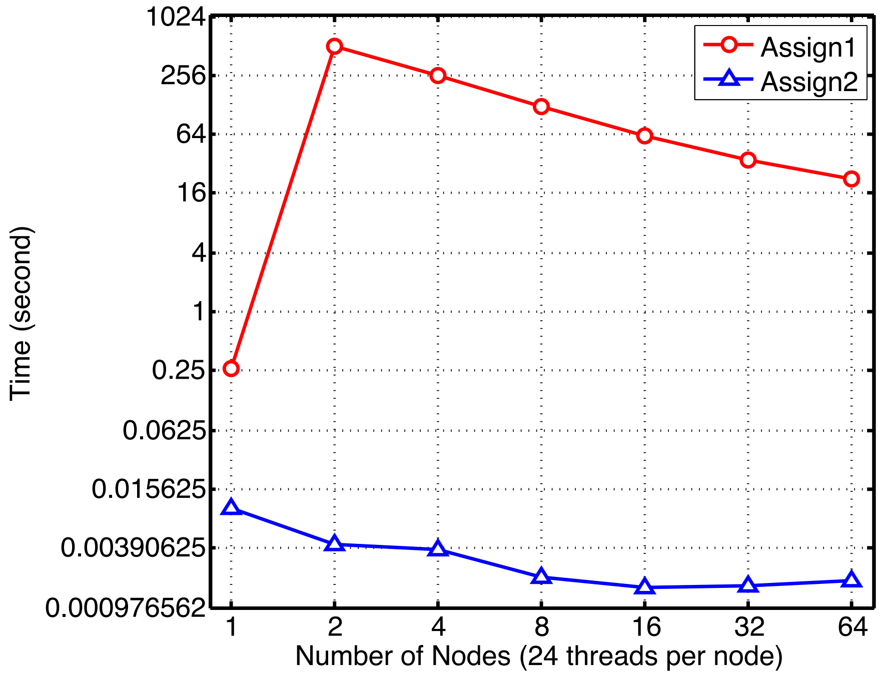
x: 1M nonzeros

Platform: NERSC/Edison

Big performance gap
Even in shared memory

Why?
Indexing a sparse domain uses binary search. For assignment it can be avoided

distributed-memory performance: **Assign** (x[i] = y[i])

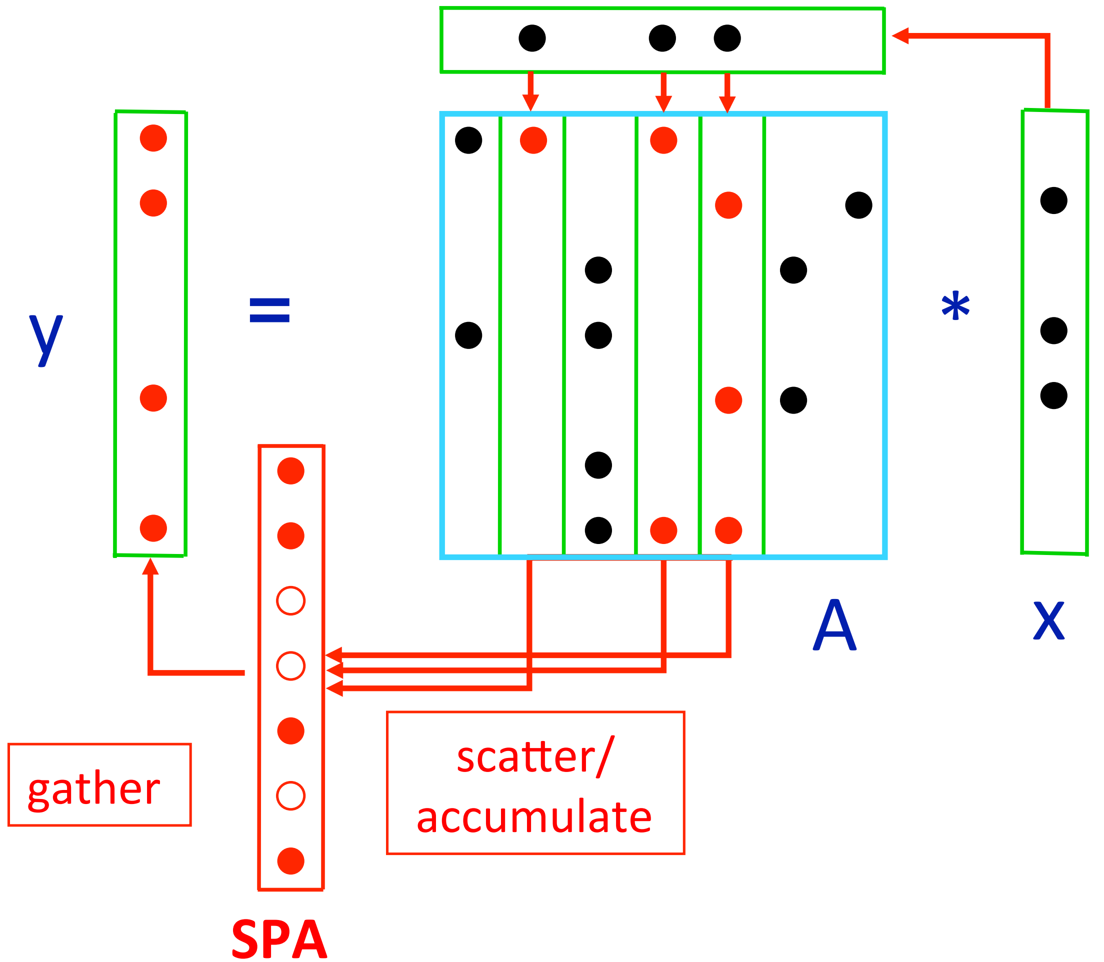


Big performance gap
Even in distributed memory

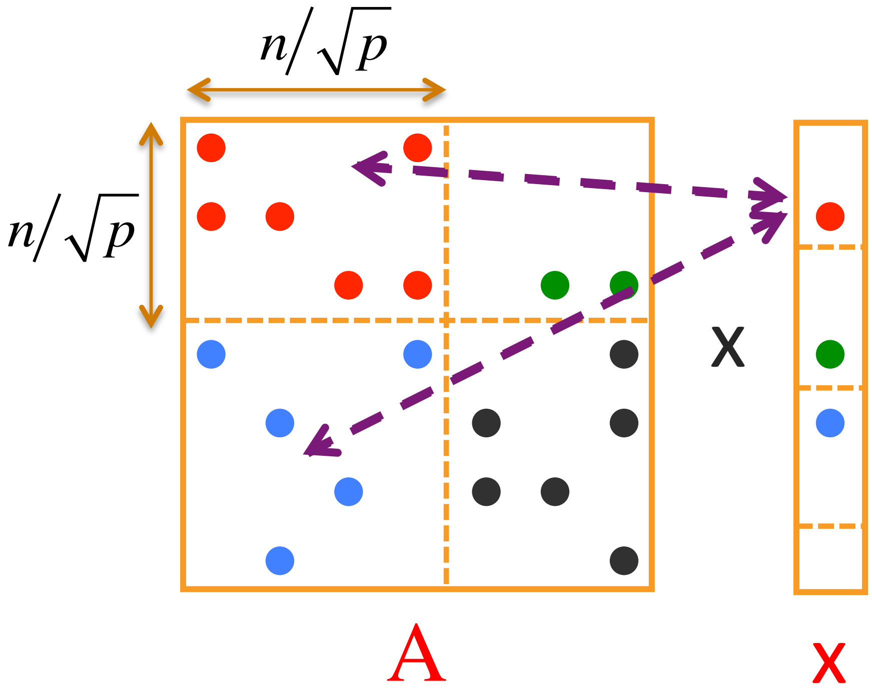
Assign1: high-level (Chapel style)
Assign2: manipulating internal arrays (C++ style)
x: 1M nonzeros
Platform: NERSC/Edison

Example, complex case: SpMSpV (y = Ax)

Algorithm overview



Sparse matrix-sparse vector multiply (SpMSpV)



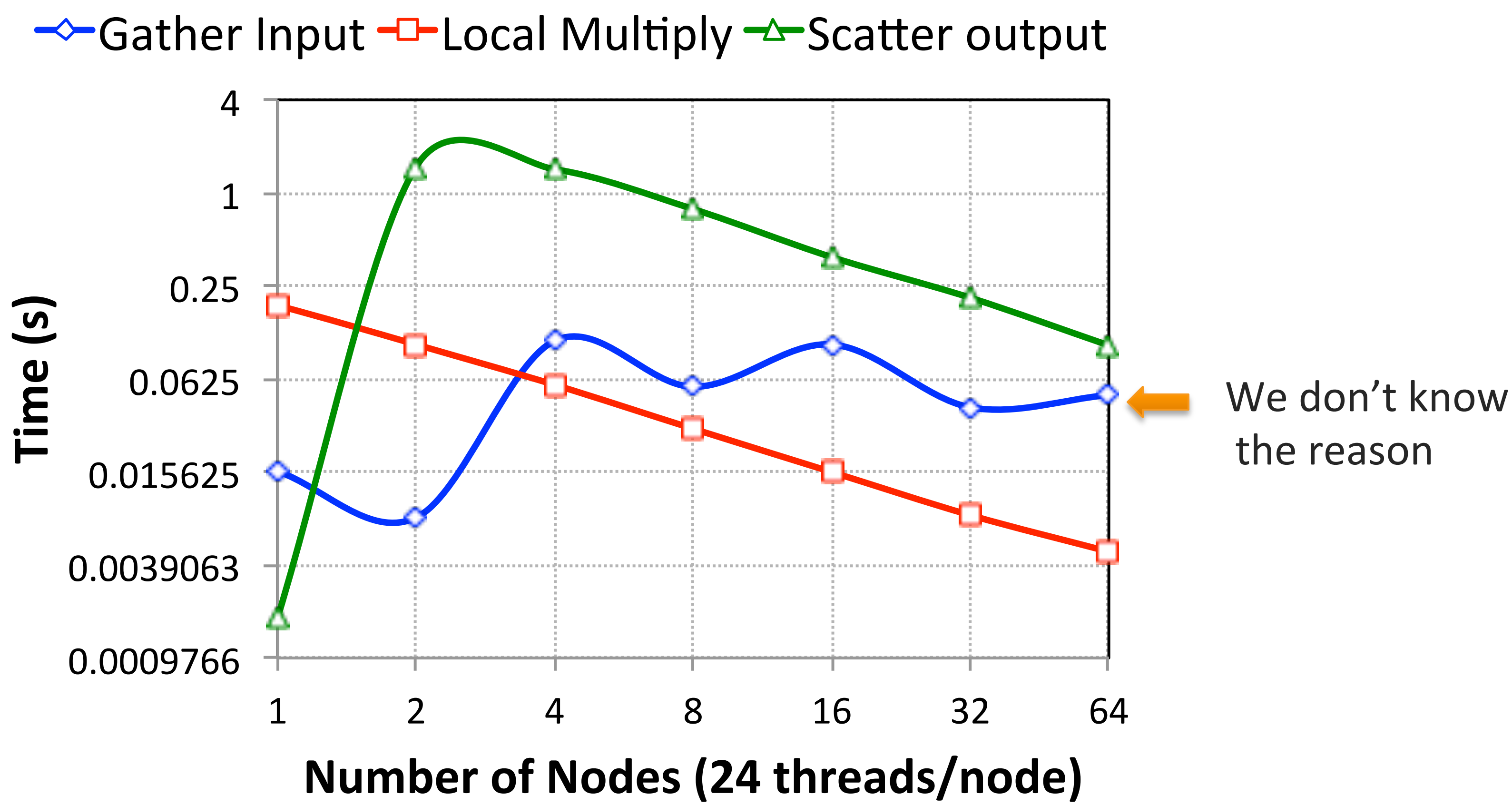
P processors are arranged in $\sqrt{p} \times \sqrt{p}$ Processor grid

Algorithm (MPI Style)
1. Gather vertices in processor column 2. Local multiplication 3. Scatter results in processor row

Algorithm (Chapel Style)
Multiply (access remote data as needed). No collective communication

Distributed-memory performance of SpMSpV on Edison

A: random; 16M nonzeros   x: random; 2000 nonzeros



Remote atomics are expensive in Chapel

Requirements for achieving high performance

- ❑ Exploit available spatial locality in sparse manipulations
  - – Efficient access of nonzeros of sparse matrices/vectors
  - – Chapel is almost there, needs improved parallel iterators

- ❑ Use bulk-synchronous communication whenever possible
  - – Avoid latency-bound communication
  - – Team collectives are useful

Our experience: productivity vs. performance

Productivity (easy to develop a prototype)

Task	Hardness	Why?
Data structure	medium	Manipulating domains and arrays
Functionality	easy	Fewer lines of code with built-in features
Parallelization	easy	No need to think about communication

Performance (hard to achieve performance)

Task	Hardness	Why?
Data structure	hard	Manipulating low level data structures
Shared-memory	medium	Data parallel iterators for sparse data
Distributed-memory	hard	Needs bulk synchronous communication, team collectives, etc.

Summary

- We have implemented a prototype GraphBLAS library in Chapel
  - Implemented breadth-first search as a representative algorithm using these primitives

- Library development in Chapel is easy (relative to C++)

- Chapel's distributed-sparse matrix support is still under development. The distributed-memory performance is expected to improve over time.

Future direction

- Finish a complete GraphBLAS-compliant library in a PGAS language (including Chapel)
  - Achieving high performance is our focus
  - Benchmark our library against other programming models and languages
- Design complex graph algorithms using the library to demonstrate its utility
  - Understand the impact of programming models on graph analytics

Acknowledgement and relevant references

- Funded in part by DOD/ACS and in part by DOE/ASCR
- Acknowledgement: Costin Iancu (LBNL), Brad Chamberlain (Cray), Michael Ferguson (Cray), Engin Kayraklioglu (George Washington University)

- References:

  – A. Azad and A. Buluç, IPDPS Workshops 2017, Towards a GraphBLAS library in Chapel.
  – A. Azad and A. Buluç, IPDPS 2017, A work-efficient algorithm for sparse matrix-sparse vector multiplication algorithm.

Questions?