Collective Communication in PGAS Languages

Collective Communication:

- An operation called by all processes together to perform globally coordinated communication
  - May involve a modest amount of computation, e.g. to combine values as they are communicate
  - Can be extended to teams (or communicators) in which they operate on a predefined subset of the processes

Open Research Questions:

- How does global address space impact design of the collective interface?
- What about the one-sided communication model?
- How do these features affect the synchronization model?
- What is the potential for non-blocking collectives?

Teams:

- Many applications require collectives to be performed across teams (*i.e.* subsets) of the processors
- Currently no interface in UPC
- How do we construct these teams?
  - Thread-Centric: Programmer explicitly specifies the threads that take part in the collective through a language level team construction API
  - Data-Centric: Programmer only specifies the data for the collective. Runtime system then figures out where the data resides and performs the collective

Advantages to each approach

Thread Centric	Data Centric
• Cost of team construction exposed to programmer • Runtime system can spend more time to potentially build better infrastructure for collectives • Teams can be explicitly reused • Simpler transition for MPI programmers	• Collectives focus on operating on shared data rather than threads • Programmer does not need to worry about potentially complex logic to constructing and using a team • Opens up a much richer collective interface – ex: exchange data from even processors into odd processors

Ex: Broadcast A into even slots and B into odd slots of dst

Thread Centric

```
/*allocate array*/
shared [1] double dst[THREADS*64];
shared [64] double *temp_dst;
shared double A,B;
upc_team_t odd_team,even_team;

even_team = /* logic to construct team of all even threads*/
odd_team = /*logic to construct team of all odd threads*/

/* recast into a fully blocked array*/
temp_dst = (shared [64] double*) dst;

/*broadcast only into the slots of the array specified by the team
argument*/
upc_team_broadcast(temp_dst, A, sizeof(double)*64, even_team);
upc_team_broadcast(temp_dst, B, sizeof(double)*64, odd_team);
```

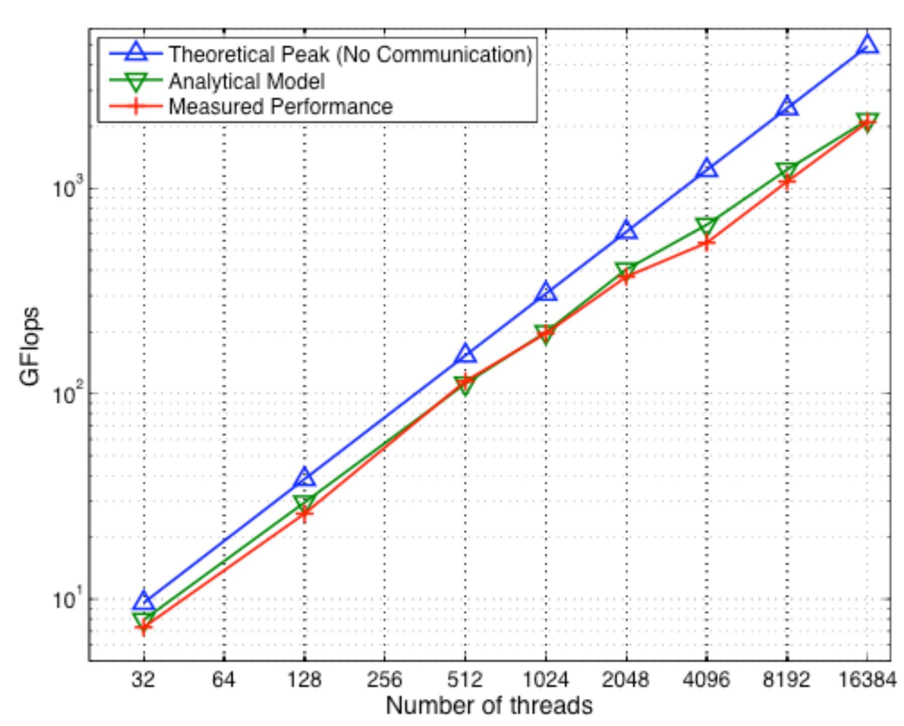
Data Centric

```
/*allocate array*/
shared [1] double dst[THREADS*64];
shared double A,B;

/* let underlying runtime system take care of figuring out where the
data is mapped*/

upc_stride_broadcast(dst<0:2:63>, A,   sizeof(double));
upc_stride_broadcast(dst<1:2:63>, B, sizeof(double));
```

From "Performance without Pain = Productivity: Data Layout and Collective Communication in UPC" by Rajesh Nishtala, George Almasi, and Calin Cascaval, PPoPP 2008 (to appear)

Rajesh Nishtala, George Almasi, and Calin Cascaval
IBM Research

Application Examples w/ Data Centric Collectives on BG/L

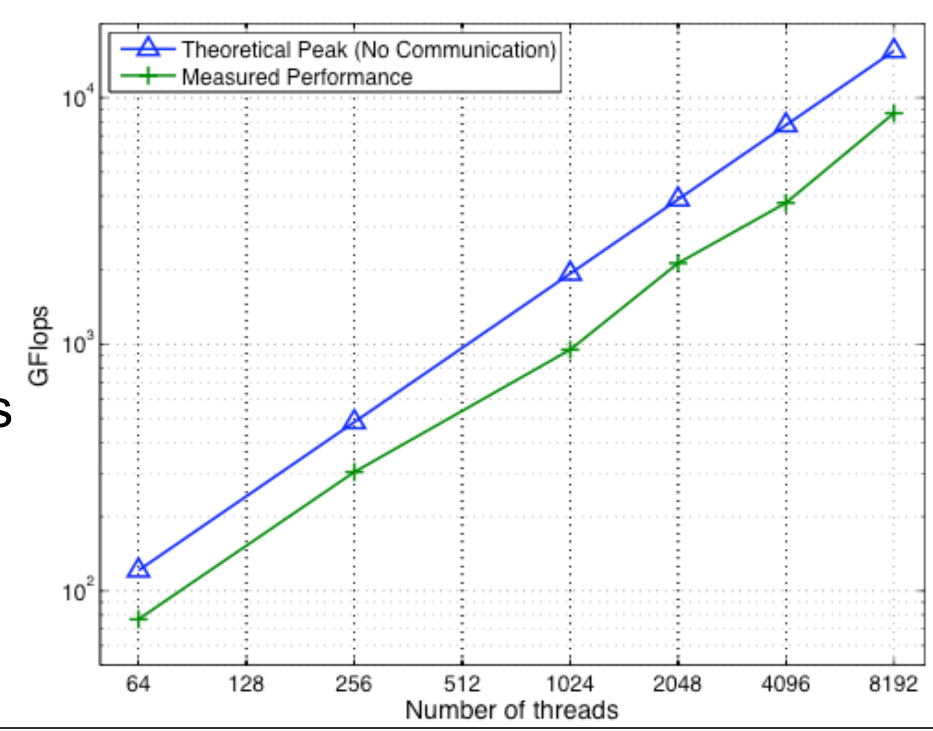


Example 1: 3D FFT

- NX x NY x NZ rectangular domain
- 2D Processor decomposition
- Requires two exchanges
  - Each processor is part of two teams
  - Each exchange happens over different teams
- Bandwidth limited problem
- Analytic model shows performance limits due to network performance
- Can express any long 1D FFT as a 3D FFT

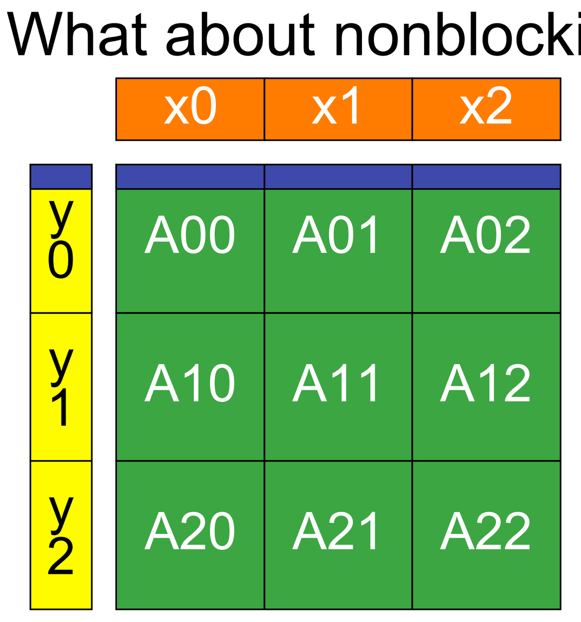
Example 2: Dense Cholesky Factorization

- Uses standard checkerboard layout for distributing the matrix
- Column broadcasts for rank-1 update implemented using data-centric collectives
- UPC implementation takes 25 lines
- Uses ESSL for serial computation



Potential for Non-Blocking Collectives:

- Our previous work has shown that nonblocking point-to-point communication has large performance benefits
- What about nonblocking collectives?



Example: Sparse Matrix Vector Multiplication

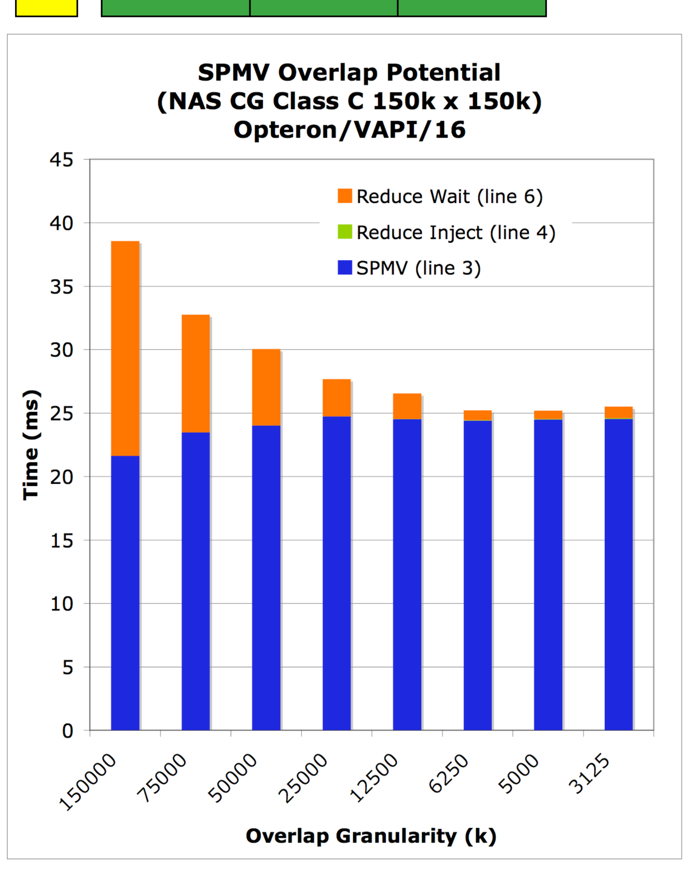
NxN Matrix distributed across 2D processor grid

Each processor needs final value of y for its row of processors

Observation

Why wait to finish SPMV on all rows?

Can perform all-reduce after k rows are done

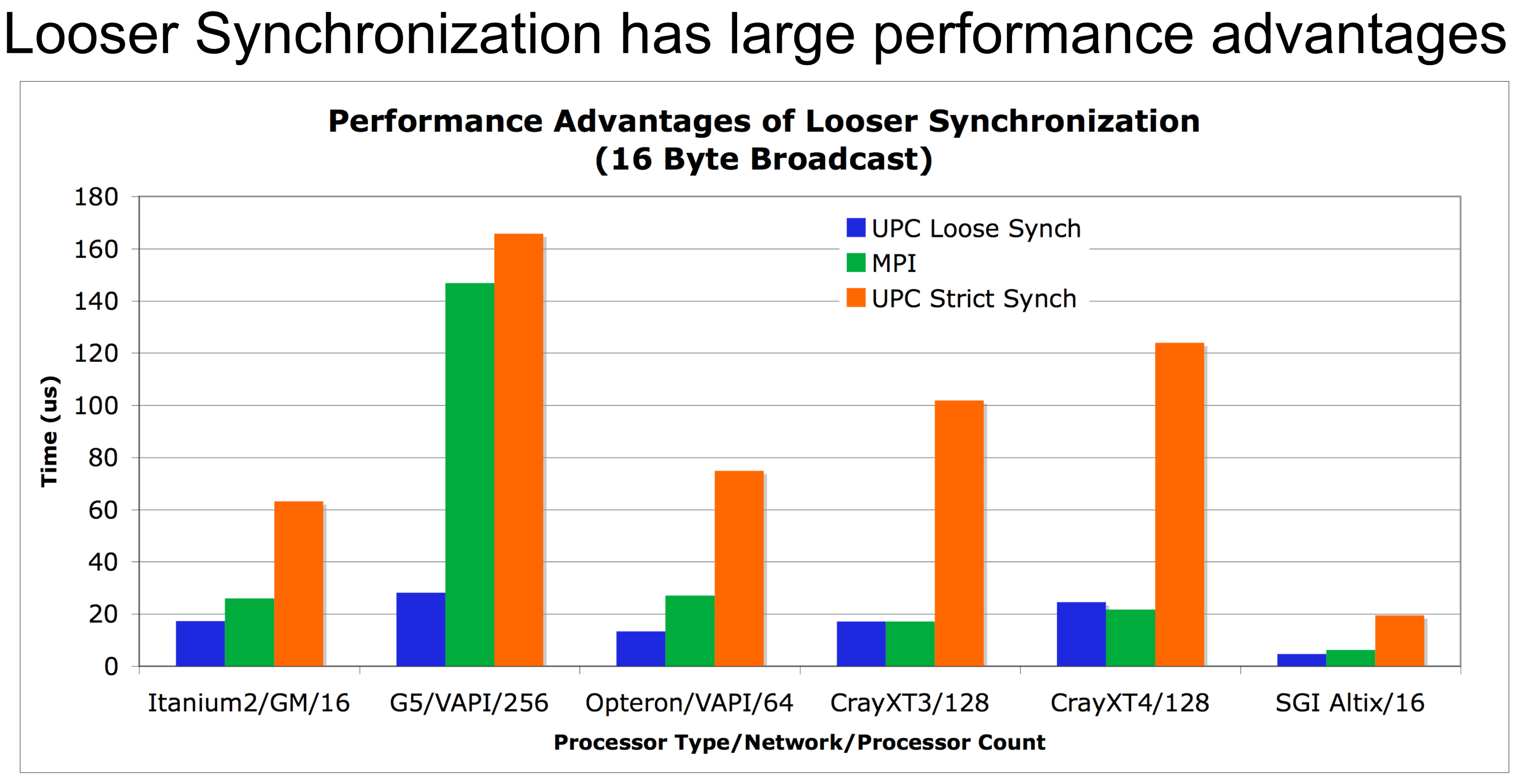


Algorithm:

1. Let segs = N/k
2. For i=0:segs,
   3. Y[(i)*segs,(i+1)*segs) = SPMV on rows [(i)*segs,(i+1)*segs)
   4. Inject Allreduce of k doubles
5. End For
6. Wait for every Allreduce to finish

Synchronization Modes:

- One-sided semantics in PGAS languages allow remote data to be modified before collective is done
  - There is no way of knowing whether the collective is complete on a remote thread without querying it
  - Adding a full barrier for collective over-synchronizes the problem.
- No need to over synchronize a collective if the data is not needed in the current barrier phase
  - UPC exposes the looser synchronization to the programmer through a rich set of synchronization modes
  - Aggregate synchronization by using one barrier to synchronize *all* the collectives
- Looser Synchronization has large performance advantages



Rajesh Nishtala, Paul Hargrove, Dan Bonachea, and Kathy Yelick
Berkeley UPC

BERKELEY LAB