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

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

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

```
/*allocate array*/
shared [1] double dst[THREADS*64];
//allocate array
upc_team_broadcast(dst, sizeof(double)*64, even_team);
upc_team_broadcast(dst, sizeof(double)*64, odd_team);
```

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

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?

### Advantages to each approach

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

### 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

---

**Rajesh Nishtala, George Almasi, and Calin Căscaval**

IBM Research