Star forests as a parallel communication model

Jed Brown

2011-12-24

Abstract

Many useful communication patterns can be represented as communication of data between roots and leaves of a star forest. Such communication graphs are easy for a user to specify (by naming the root node for each leaf) and map naturally to one-sided communication primitives provided by the MPI-2 standard and due to be enhanced for MPI-3. If a two-sided representation of the graph is desired, it can easily be constructed. In this note, we describe the communication model, implementation in PETSc, and demonstrate algorithms using star forests.

1 Introduction and specification

Stars are the simplest nontrivial tree, consisting of one root vertex connected to zero or more leaves. Figure 1 shows some example stars. A union of disjoint stars is called a *star forest*.

Star forests are typically partitioned across multiple processes; The graph is specified by calling `PetscSFSetGraph()`, which has prototype

```
typedef struct {PetscInt rank,offset;} PetscSFNode;

PetscErrorCode PetscSFSetGraph(PetscSF sf,PetscInt nroots,PetscInt nleaves,
                const PetscInt *local,PetscCopyMode localmode,
                const PetscSFNode *remote,PetscCopyMode remotemode);
```

If `NULL` is passed for the `local` array, it is assumed that all leafs are part of nontrivial stars. Figure 2 shows an example star forest with example data defined at vertices, and arguments `local` and `remote` to `PetscSFSetGraph()`. The edges of a star forest are entirely specified by the process owning the leaves, therefore a given root vertex is merely a candidate for "incoming" edges. This one-sided specification is important so as to encode graphs containing roots with very high degree, such as globally coupled constraints, in a scalable way. Some algorithms will internally determine the degree of each root vertex and perhaps the process ranks that have leaves of stars with root onwned by the current process. For some implementations (though not the natural MPI one-sided implementation), it may be necessary to construct the full two-sided graph internally.

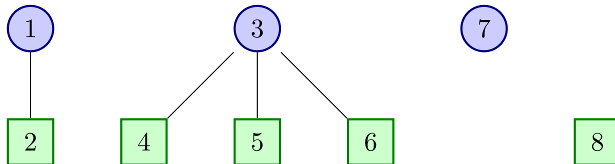

Figure 1: Examples of stars, the union of which forms a star forest. Root vertices are identified with circles and leaves with rectangles. Note that stars with no leaves are allowed as well as solitary leaves with no root. This figure shows unique data associated with each vertex, we will demonstrate operations that move data between roots and leaves.

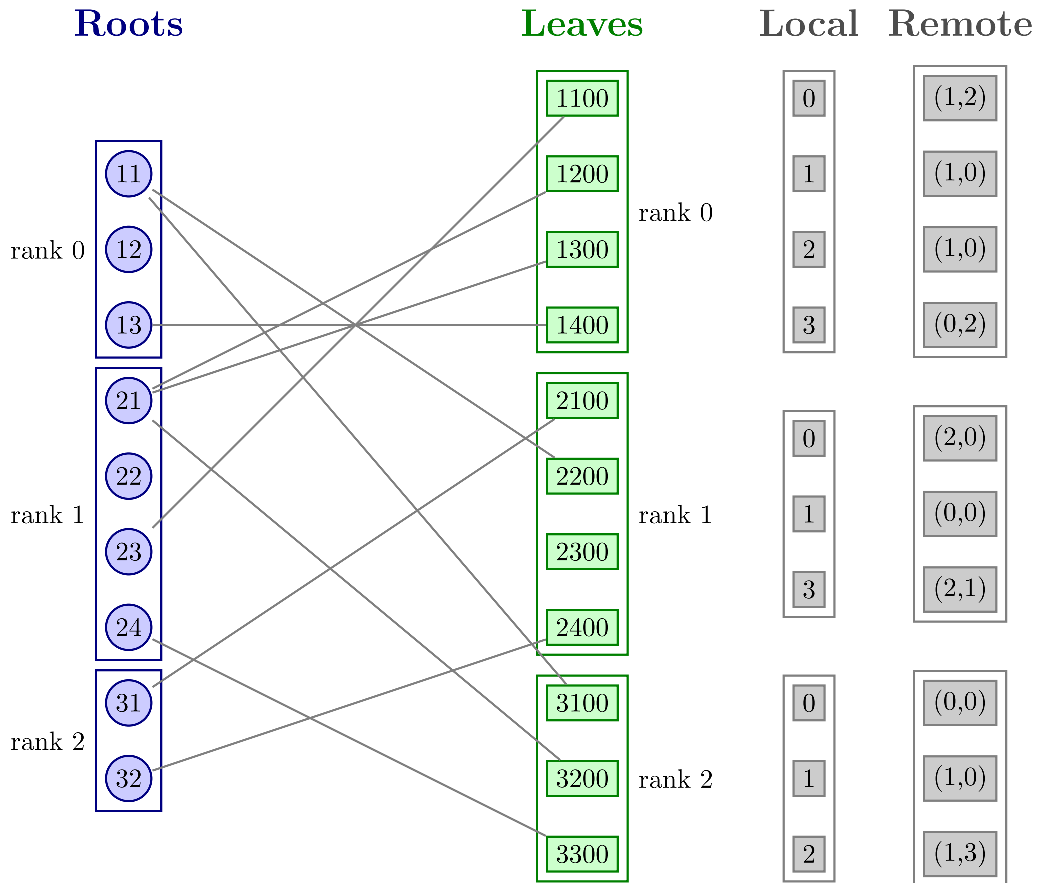

Figure 2: A distributed star forest with associated data partitioned across three processes. The graph is specified by defining the number of roots and leaves on each process and providing arrays containing the number of local leaves that are part of a nontrivial star and the remote address (rank,offset) of the root. The Local and Remote arrays defining this star forest are shown in grey to the right. If `NULL` is passed for the Local array, it is assumed that all leafs are part of nontrivial stars.

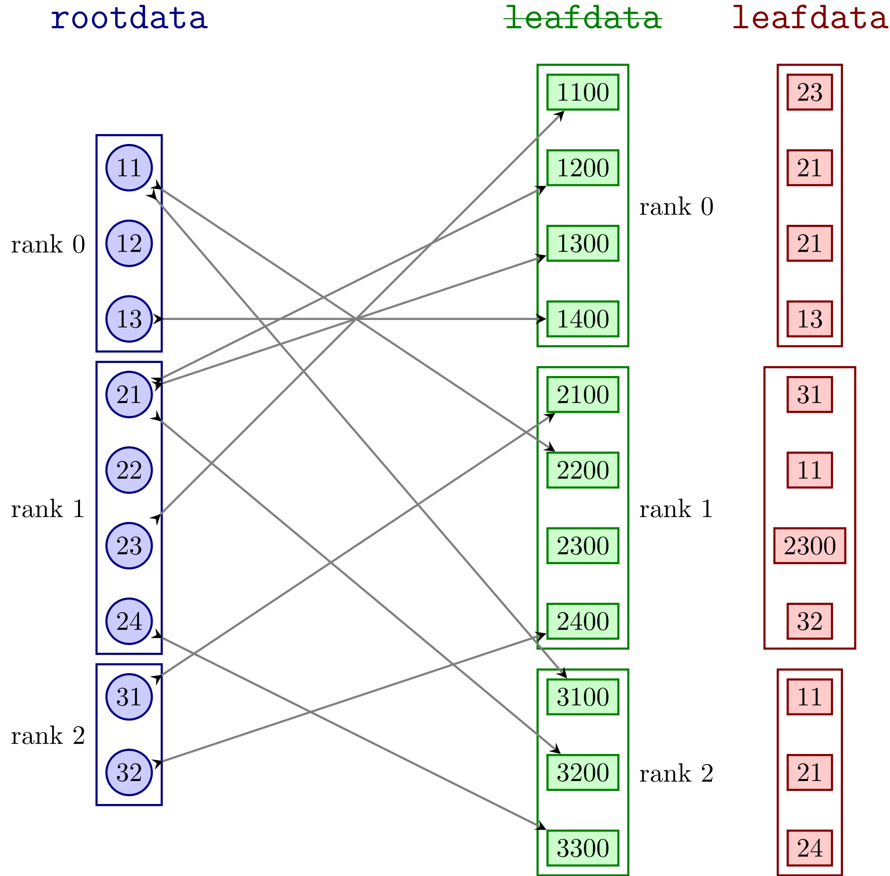

Figure 3: Illustration of the Broadcast operation which updates the provided `leafdata` array (green) in-place, resulting in the red `leafdata`.

2 Operations

We define operations that update data on roots and on leaves. All operations are split into matching *begin* and *end* phases. Any number of operations can be in-flight at one time, but the data buffers may not be altered during that time and the values in the result buffer are undefined. Any `MPI_Datatype` can be associated with each root and leaf for a given update. The most common types for this `unit` are integers, scalars (real and complex), and user-defined structures.

2.1 Broadcast

The Broadcast operation injects `rootdata` into `leafdata`.

```
PetscErrorCode PetscSFBroadcastBegin(PetscSF sf,MPI_Datatype unit,
                        const void *rootdata,void *leafdata);
PetscErrorCode PetscSFBroadcastEnd(PetscSF sf,MPI_Datatype unit,
                        const void *rootdata,void *leafdata);
```

The broadcast operation is illustrated in Figure 3 for same graph as in Figure 2.

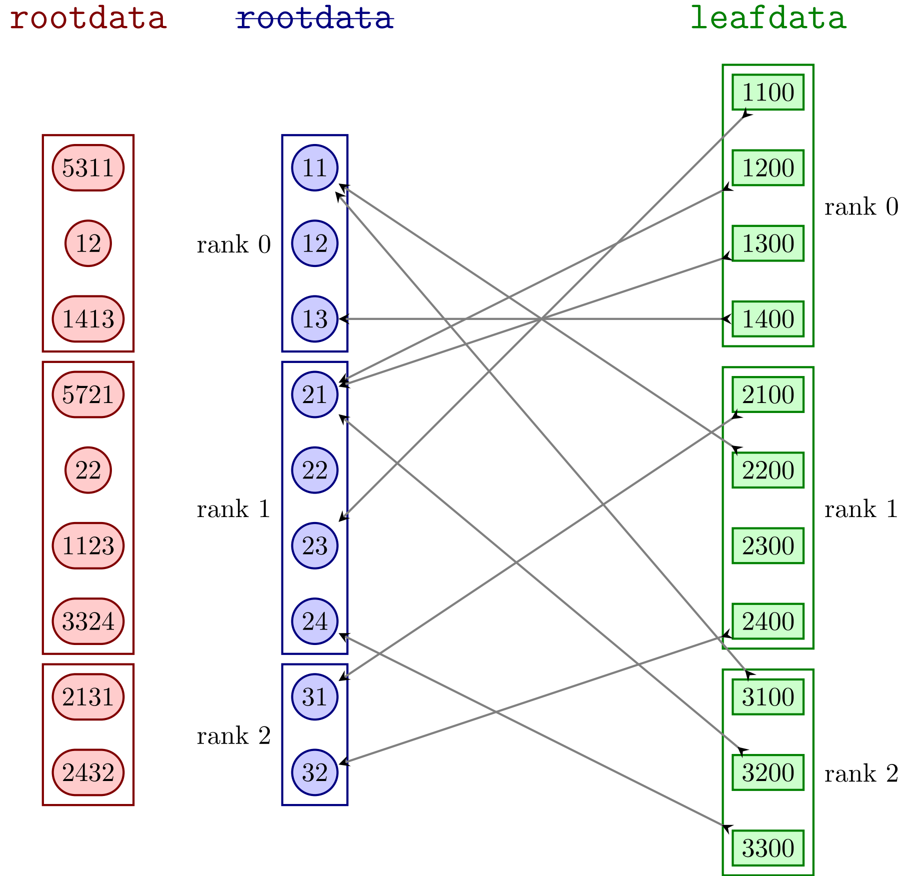

Figure 4: Illustration of Reduce using MPI_SUM, updating the provided `rootdata` array (blue) in-place, resulting in the updated `rootdata` (red).

2.2 Reduce

The Reduce operation updates `rootdata` using `leafdata` and a specified operation.

```
PetscErrorCode PetscSFReduceBegin(PetscSF sf,MPI_Datatype unit,
                      const void *leafdata,void *rootdata,MPI_Op op);
PetscErrorCode PetscSFReduceEnd(PetscSF sf,MPI_Datatype unit,
                      const void *leafdata,void *rootdata,MPI_Op op);
```

Any builtin `MPI_Op` (including `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_MAXLOC`, etc) can be used. Additionally, `MPI_REPLACE` can be used to overwrite, though this is usually only useful if it is known that the data from all leaves is identical (for example, if there is at most one leaf per root). Depending on the implementation and possible future updates to the MPI standard, it may also be possible to use user-defined reduction operations. Figure 4 illustrates the reduce operation.

2.3 Fetch-and-op

An important synchronization and setup primitive is to atomically fetch a value and update it.

```
PetscErrorCode PetscSFFetchAndOpBegin(PetscSF sf,MPI_Datatype unit,
                      void *rootdata,const void *leafdata,void *leafupdate,MPI_Op op);
```

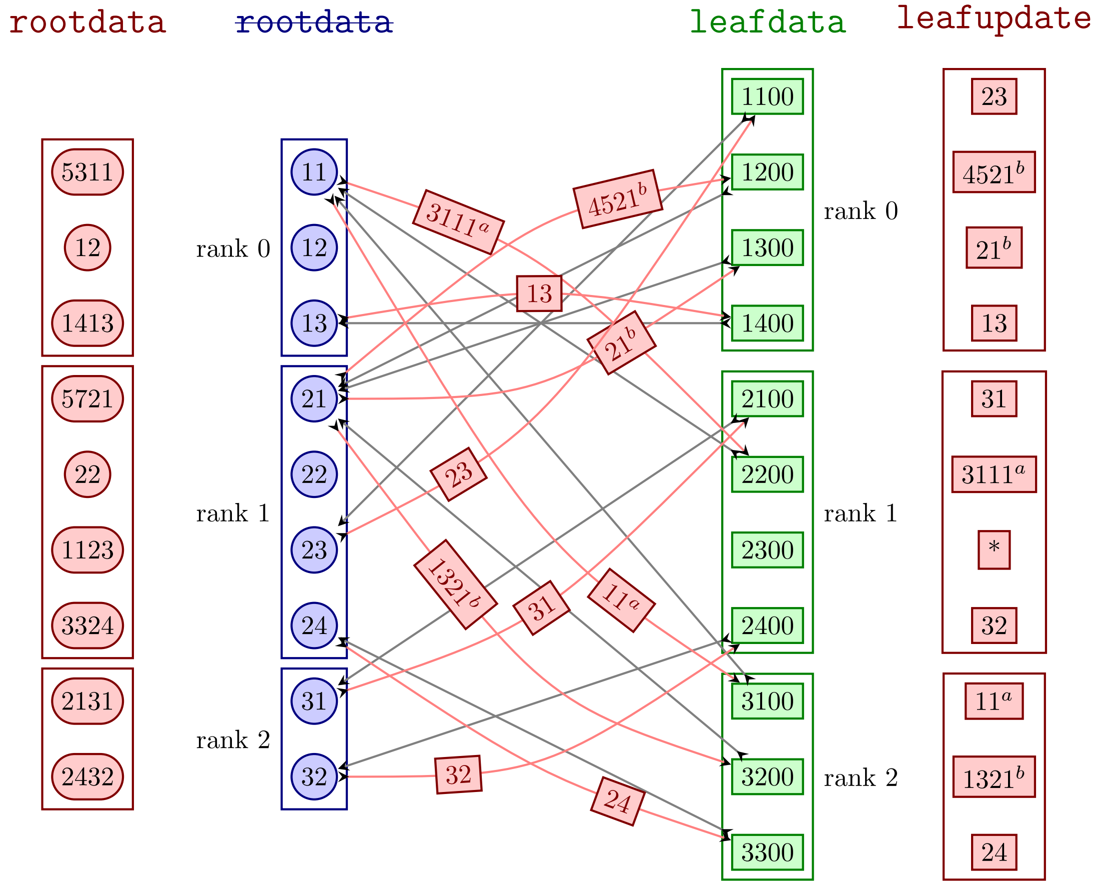

Figure 5: Illustration of FetchAndOp using MPI_SUM, updating the provided `rootdata` array (blue, struck out) in-place, resulting in the updated `rootdata` and `leafupdate` (red). There are two non-deterministic groups indicated by the superscripts a and b. Within these groups, the updates can occur in any order, resulting in different values appearing in `leafupdate`. The asterisk in the `leafupdate` on rank 1, corresponding to the non-participating leaf, indicates that existing value is unmodified. For all participtaing leafs, `leafupdate` is overwritten with the value at the corresponding root at the time of the update.

```
PetscErrorCode PetscSFFetchAndOpEnd(PetscSF sf,MPI_Datatype unit,
                     void *rootdata,const void *leafdata,void *leafupdate,MPI_Op op);
```

Atomicity is only guaranteed with base type granularity (e.g. machine word) and must (currently) be an MPI builtin reduction operation. In our experience, the most commonly used variant is *fetch-and-add* which is used to simultaneously count contributions from leaves (found in final value in `rootdata`) and determine offsets at which to send a second round of communication (found in `leafdata`). An illustration of the FetchAndOp primitive is shown in Figure **??**.

2.4 Gather

[TODO: Gather leafdata to roots]

2.5 Scatter

[TODO: Scatter unique data to each leaf from roots]

2.6 Extensions

A non-uniform variant that can associated a different amount of data (number of units) per star would be desirable to minimize metadata. This could be achieved using an additional argument to `PetscSFSetGraph()` or a different function that used a struct with more elements (instead of `PetscSFNode`).

3 Algorithms

Ghost update Given a star forest (SF) representing the "local to global" map for a finite difference, finite element, or finite volume method, the ghost update is simply Broadcast. This is the same update used for sparse matrix-vector multiply when using a row partition.

Finite element assembly from a cell partition Given the same SF as above for the ghost update, finite element residuals (computed using a non-overlapping cell partition) are assembled using Reduce with the SUM operation. This is the same updated needed for sparse matrix-vector multiply using a column partition (or transpose-multiply using a row partition).

Extracting a submatrix from a sparse matrix The difficult part of parallel submatrix extraction is determining which columns are needed (by any processes) from the local block of rows. Given two SFs, `sfA` mapping reduced local column indices (leaves) to global columns (roots) of the original matrix A, and `sfB` mapping "owned" columns (leaves) of the submatrix B to global columns (roots) of A, we determine the local columns to retain using

1. Reduce using `sfB`, resulting in a distributed array indicating the columns of B each column of A will be replicated into.
2. Broadcast using `sfA` to get these values in the reduced local column space.

The algorithm proceeds by preparing the requested rows of A into non-split form and distributing them to new owners (which can be done using a row-based SF).

Ownership discovery and transfer In a common scenario, the old owner of a point knows which process should be the new onwner, but not vice-versa. A simple, scalable way to move the points is

1. Create an SF with one root per process and one leaf per rank that we should send to.
2. FetchAndOp to SUM the number of points. This results in root data containing the total number of incoming points and leafs knowing the offset at which to place their outbound points.
3. Create a new SF with one root per incoming point and one leaf per outbound point (using the offset determined above). Note that in this SF, each root has exactly one edge.
4. Reduce with REPLACE to obtain the new data.

Graph distribution Suppose a parallel graph is stored using an SF `sfVertex` mapping local vertices (leaves) to their non-redundant global counterparts (roots) and a local indexed array containing neighbor vertices for each owned vertex. **[TODO: be more concrete about the data structure]** A partitioner marks each vertex with its new owner and we must migrate it to the new owner.

1. Discover the new vertices as desribed above, resulting in `sfNew` mapping old vertices (leaves) to new vertices (roots).
2. Broadcast the new owner over `sfVertex`.
3. Reduce the number of neighboring vertices to the new owners (REPLACE).
4. Create a new SF to communicate neighboring vertices (now referenced to the new owners) from old owners (leaves) to new owners (roots) and send with Reduce (REPLACE).