Implementing a model-based collective communication operation with the MPIBlib/CPM framework

Kiril Dichev

May 23, 2011

Outline

Outline

Why should we use models for collective operations

- MPI implements various collectives
- The standard implementations ignore the characteristics of the underlying communication network
- Communication performance models describe these characteristics
- By using models, we can optimize a collective operation to use this knowledge

Why should we use models for collective operations

Efficient collective communication is implemented with

- tree data structures
- tree-based algorithms

We can use communication performance models for collectives by:

- switching between existing algorithms
- mapping processes to the nodes of the communication tree
- dynamically generating a communication tree

What sort of operation do we implement?

In this tutorial when we say model-based collective communication operation we mean that we

What sort of operation do we implement?

In this tutorial when we say model-based collective communication operation we mean that we

- dynamically generate communication trees

What sort of operation do we implement?

In this tutorial when we say model-based collective communication operation we mean that we

- dynamically generate communication trees
- for some collective operations

What sort of operation do we implement?

In this tutorial when we say model-based collective communication operation we mean that we

- dynamically generate communication trees
- for some collective operations
- with the help of model predictions

What sort of operation do we implement?

In this tutorial when we say model-based collective communication operation we mean that we

- dynamically generate communication trees
- for some collective operations
- with the help of model predictions
- and do the communication over these trees

Outline

MPIBlib

Overview

MPIBlib provides a library with benchmarking functionality

- Suitable for inserting benchmarks into applications
- The library can be used by
  - any MPI applications
  - a set of provided tools and tests
- implements a number of existing collective algorithms (Example: binomial tree algorithm)
- provides a command line tool ('colective') for fast development and testing

MPIBlib

Trees

- We want to avoid code rewriting
- For collectives, we introduce orthogonal concepts which can be combined:
  - Tree algorithms (not depending on the tree)
  - Communication trees (not depending on the algorithm)

MPIBlib

Tree builders

- Tree builders - encapsulate into an object the logic of constructing a tree
- A tree builder generates a communication tree
- The communication tree is used in the tree algorithm

CPM

- Implements heterogeneous communication performance models
- Estimates the model parameters on a cluster
- Uses MPIBlib for benchmarking
- Provides a number of collective operation using model predictions

CPM

Model-based collectives

- In order to implement a model-based collective, we

CPM

Model-based collectives

- In order to implement a model-based collective, we
  - do not reimplement communication algorithms

CPM

Model-based collectives

- In order to implement a model-based collective, we
  - do not reimplement communication algorithms
  - implement tree builders

CPM

Model-based collectives

- In order to implement a model-based collective, we
  - do not reimplement communication algorithms
  - implement tree builders
- A model-based tree builder encapsulates the logic of constructing a model-based communication tree

CPM

Model-based collectives

- In order to implement a model-based collective, we
  - do not reimplement communication algorithms
  - implement tree builders
- A model-based tree builder encapsulates the logic of constructing a model-based communication tree
- Implementing a model-based tree builder is the main part when implementing a model-based collective operation

Outline

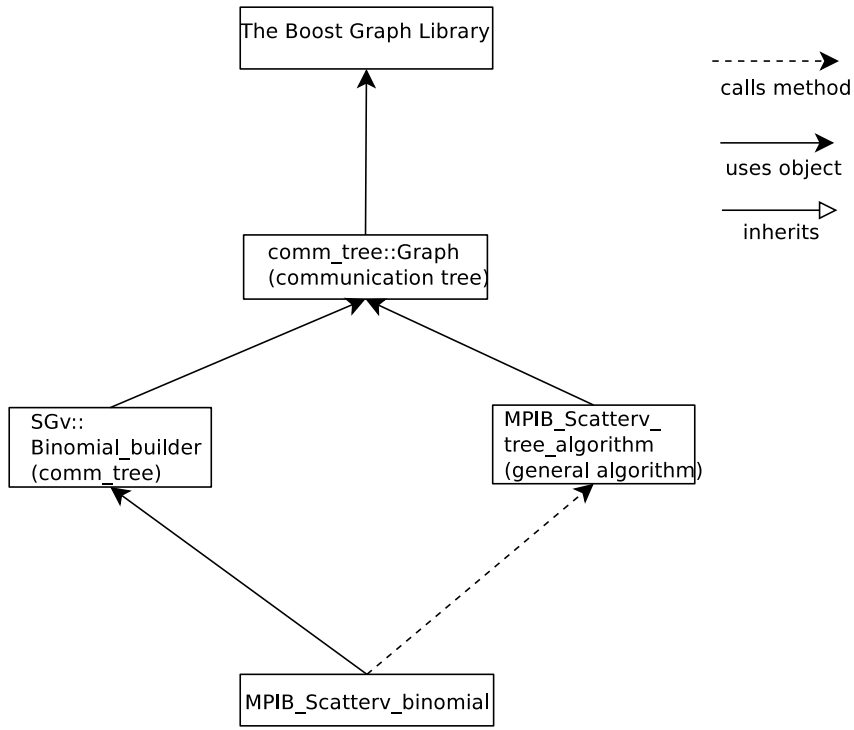
Simple collectives

In MPIBlib, the available collectives follow the naming convention `MPIB_X_Y` with

- X - a communication operation
- Y - a tree algorithm

MPIBlib collectives

Example based on Scatterv operation



How to implement model-based collectives in CPM

A model-based collective operation belongs in CPM

- follows the naming convention `M_X_Y` with
- M - a communication model
- X - a communication operation
- Y - a tree algorithm

How to implement model-based collectives in CPM

A model-based collective operation belongs in CPM

- follows the naming convention `M_X_Y` with
- M - a communication model
- X - a communication operation
- Y - a tree algorithm

The model

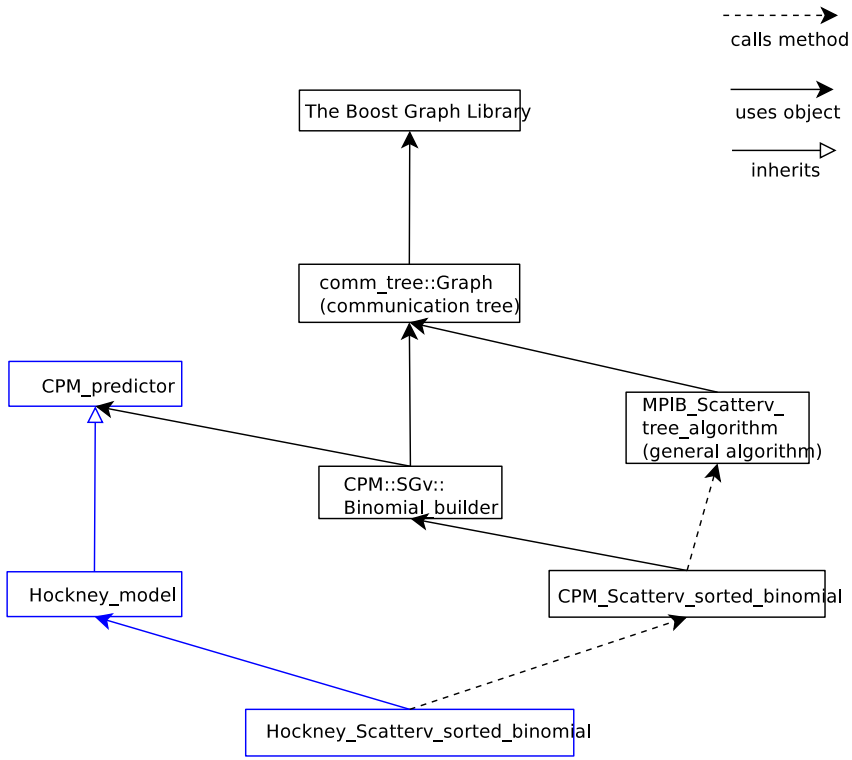
A model-based collective operation can be

- generic
  - depends on the predicted execution time of a communication
  - prediction can be provided by any model
  - Example: predict_p2p returns the predicted execution time for a point-to-point communication
- model-specific - depends on certain communication performance models using parameters specific for these models only

In this tutorial, we only discuss generic model-based collectives

The model

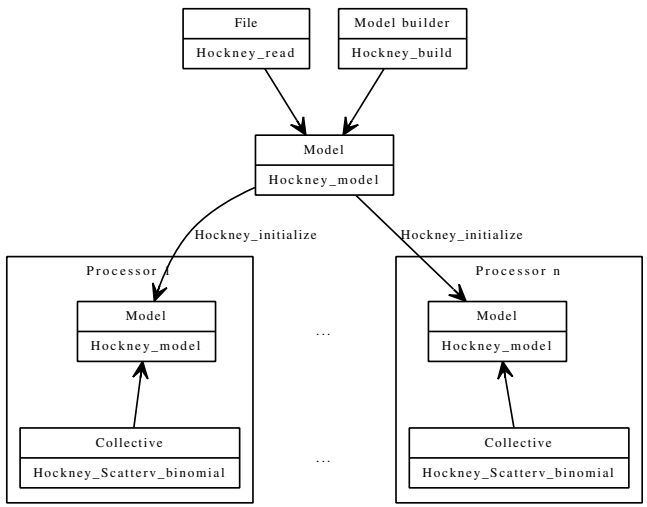
Example based on Scatterv operation



The model

Initialization

- Master node can read a model from a file or
- all nodes can build the model by performing collective benchmarks
- then, model parameters are broadcasted to all nodes



The model

Initialization

Example:

```
if (rank == 0) {
  Hockney_read(stream, &model);
}
Hockney_initialize(comm, model);
if (rank == 0) Hockney_free(model);
```

Analogy to the MPI communicator:

- the model instance has a global scope like the MPI communicator for MPI programs
- it is independent from the collective operation
- similar init and finalize calls

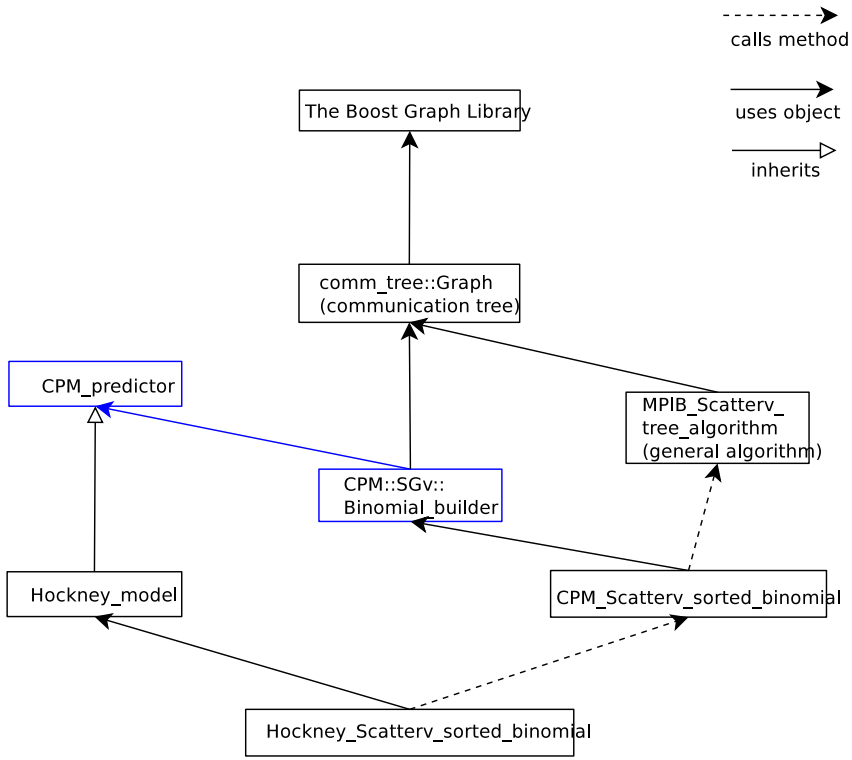
How to implement model-based collectives in CPM

A model-based collective operation belongs in CPM

- follows the naming convention `M_X_Y` with
- M - a communication model
- X - a communication operation
- Y - a tree algorithm

The model-based tree builder

Example - MPIB_Scatterv_binomial



The model-based tree builder

- The builder must implement a build function which generates a tree
- usage of Boost required
- all model logic is done by calling the model predict_p2p function

The model-based tree builder

Example - Process Mapping

```
class Binomial_builder {
private:
CPM_predictor* predictor;
...
void build(int root, int count,
  Graph& g, Vertex& r, Vertex& u, Vertex& v)
{
...
  //Get the rank of a vertex we are visiting
  Vertex s = <get some vertex with already assigned rank>
  int source = get(vertex_index, g, s);

  //Find the rank from unassigned ranks which has
  // the fastest link to the rank of the current vertex
  for (deque<pair<int, double> >::iterator i = ranks.begin();
  i != ranks.end(); i++) {
    int target = i->first;
    i->second = predictor->predict_p2p
      (predictor, source, target, counts[target]);
  }
  deque<pair<int, double> >::iterator i =
    min_element(ranks.begin(), ranks.end(), second_less()):

  //Create a vertex and edge in the tree to build the fastest
  //possible connection to the current rank
  int target = i->first;
  ranks.erase(i);
  Vertex t = add_vertex(g);
  put(vertex_index, g, t, target);
  add_edge(s, t, g);
...
```

The model-based tree builder

Example - Process Mapping

```
class Binomial_builder {
private:
CPM_predictor* predictor;
...
void build(int root, int count,
  Graph& g, Vertex& r, Vertex& u, Vertex& v)
{
...
  //Get the rank of a vertex we are visiting
  Vertex s =<get some vertex with already assigned rank>
  int source = get(vertex_index, g, s);

  //Find the rank from unassigned ranks which has
  // the fastest link to the rank of the current vertex
  for (deque<pair<int, double> >::iterator i = ranks.begin();
  i != ranks.end(); i++) {
    int target = i->first;
    i->second = predictor->predict_p2p
      (predictor, source, target, counts[target]);
  }
  deque<pair<int, double> >::iterator i =
    min_element(ranks.begin(), ranks.end(), second_less()):

  //Create a vertex and edge in the tree to build the fastest
  //possible connection to the current rank
  int target = i->first;
  ranks.erase(i);
  Vertex t = add_vertex(g);
  put(vertex_index, g, t, target);
  add_edge(s, t, g);
...
```

The model-based tree builder

Example - Process Mapping

```
class Binomial_builder {
private:
CPM_predictor* predictor;
...
void build(int root, int count,
  Graph& g, Vertex& r, Vertex& u, Vertex& v)
{
...
  //Get the rank of a vertex we are visiting
  Vertex s =<get some vertex with already assigned rank>
  int source = get(vertex_index, g, s);

  //Find the rank from unassigned ranks which has
  // the fastest link to the rank of the current vertex
  for (deque<pair<int, double> >::iterator i = ranks.begin();
  i != ranks.end(); i++) {
    int target = i->first;
    i->second = predictor->predict_p2p
      (predictor, source, target, counts[target]);
  }
  deque<pair<int, double> >::iterator i =
    min_element(ranks.begin(), ranks.end(), second_less()):

  //Create a vertex and edge in the tree to build the fastest
  //possible connection to the current rank
  int target = i->first;
  ranks.erase(i);
  Vertex t = add_vertex(g);
  put(vertex_index, g, t, target);
  add_edge(s, t, g);
...
```

How to implement model-based collectives?

A model-based collective operation belongs in CPM

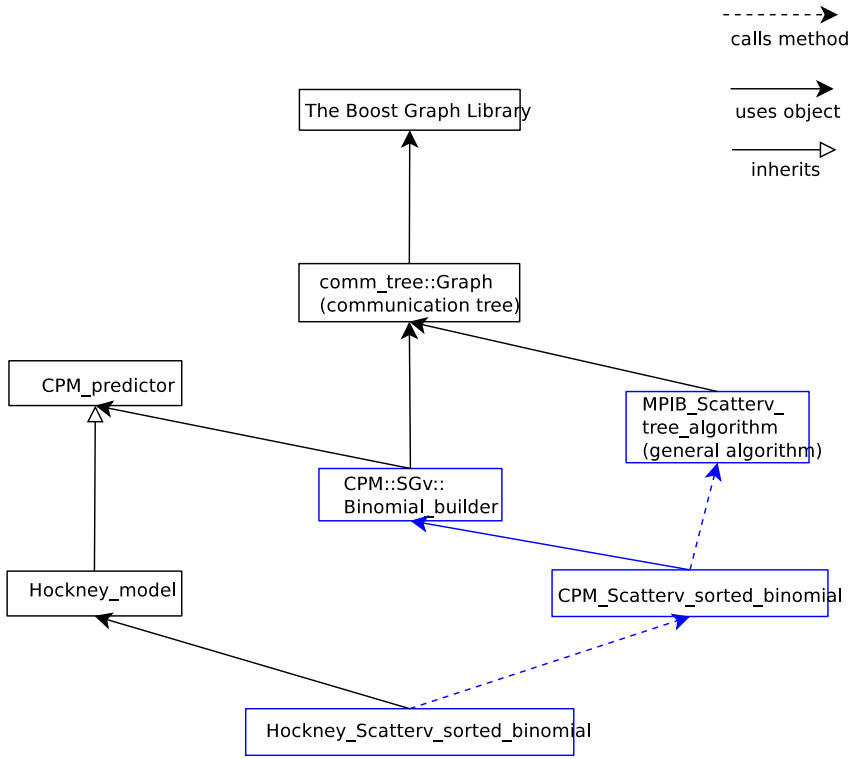
- follows the naming convention `M_X_Y` with
- M - a communication model
- X - a communication operation
- Y - a tree algorithm

Communication operation

- We don't need to worry about the communication operation - MPIBlib provides that
- We only need to integrate the components

Integrating CPM and MPIBlib components into a model-based collective communication

Example based on Scatterv operation



Integrating CPM and MPIBlib components into a model-based collective communication

The integration includes:

Integrating CPM and MPIBlib components into a model-based collective communication

The integration includes:

- passing an initialized predictor according to the model M (example for Hockney model) :

```
Hockney_Scattev_sorted_binomial (...){
CPM_Scatterv_sorted_binomial(&Hockney_model_instance->predictor, ...
```

Integrating CPM and MPIBlib components into a model-based collective communication

The integration includes:

- passing an initialized predictor according to the model M (example for Hockney model) :

```
Hockney_Scattev_sorted_binomial (...){
CPM_Scatterv_sorted_binomial(&Hockney_model_instance->predictor, ...
```

- calling the right tree algorithm (Y) with the right model-based tree builder (X)

```
extern "C" int CPM_Scatterv_sorted_binomial(CPM_predictor* predictor, ... {
return MPIB_Scatterv_tree_algorithm(Binomial_builder(predictor, no), ...
```

Outline

Tools for using the model-based collectives

- Generate a (Hockney) model file - essential !

```
mpirun --machinefile <>  -np <> model -C Hockney -o <model-file>
```

Tools for using the model-based collectives

- Generate factors for Scatterv/Gatherv - only relevant for benchmarks on irregular operations
  - factors determine the message sizes for the processes
  - argument -c or -r for CPU-based or random size distribution

```
mpirun -np 4 --machinefile <>  generate_factors -c > factors.out
```

Tools for using the model-based collectives

- running a benchmark on the new collective operation

```
mpirun -np 4 --machinefile <> collective -l <CPM installation>/lib/libcpm_coll.so  \
-O Hockney_Scatterv_dfs_binomial_min -f factors.out \
-o model=Hockney,file=<generated model file>,sgv=2 > \
 Hockney_Scatterv_dfs_binomial_min.out
```

- MPIBlib 'collective' documentation for generic arguments (all except for -o)

```
mpirun -np 1 collective -h
```

- CPM documentation on "subopt" (-o) arguments
  - e.g. on sgv decides where the communication tree is generated
  - 'verbose' (i.e. -o verbose,...) is useful for debugging the generated tree (tree is output)