An MPI Co-ordination Library

Murray Cole

1 Introduction

This document introduces an attempt to make structured parallel programming concepts accessible to everyday parallel programmers, in the form of a library which can be run on top of C/MPI. *I have written this largely for my own convenience, as a repository for my ideas on the library as it evolves. Things will added/removed or amended and I don't promise that what is written here corresponds to what is currently implemented, or even that this document is internally coherent.*

2 Conceptual Overview

This library is best thought of as extension to MPI, providing some new collective operations and auxiliary functions. As in MPI, specific data items are tied to and exclusively accessed by individual processes, and the number of processes is fixed at run time (I am ignoring the dynamic extensions to MPI for now). Library calls can be nested to arbitrary depth. Folk-wisdom suggests that the level of nesting in realistic programs will rarely exceed two and perhaps never exceed three (or perhaps we have only written the easy programs so far?).

3 An Example

Before discussing principles, I believe that it will be helpful to consider an application. The code in figure 1 solves a system of linear equations by Gaussian Elimination and Back-Substitution using the pipelined algorithm presented on pages 183-184 of Kumar, for the simple situation in which the number of rows equals the number of processes, p. The coefficient matrix and vector and result vector are distributed by row across the processes. The structure of the algorithm is as follows

1. Gaussian elimination phase. Each processor behaves as follows:
   (a) for each preceding row (so this is null for P_0), receive an updated copy of the row, pass this on to the next process, and update my own row with respect to the new data;
   (b) make a final adjustment to my row, and pass this on to the next process.
2. Back substitution phase. Each processor behaves as follows:
   (a) for each succeeding row (so this is null for P_{p-1}), receive a final value for that row's variable, pass this to the preceding process, and update my own variable with respect to the new data;
   (b) make a final adjustment to my variable, and pass this on to the preceding process.

Each phase is therefore a pipeline. In the first, data is flowing from process 0 and onwards through the sequence of processors. In the second phase, data flows in the reverse direction. A point to note is that there is no "external" data which is pushed through either pipeline. All data is already stored, or generated, within the stages. Similarly, there are no results emerging from the pipeline. The result is the collective state of the computed `myx`, which are left distributed across the processes.

```
#include "framelib.c"
#define N 8          /* and assumes P=N */
float mya[N+1];
int myrealrank;
float myx;

int main (int argc, char *argv[]) {
  spread_t spreads[N+1];
  MPI_Datatype types[N+1];
  void (*stages[N])(void);
  int i, outmul, myrealrank;

  MPI_Init(&argc, &argv);   SkelLibInit();
  myrealrank = myrank();  /* remember this for later */
  GetCoeffs();  /* Straight MPI to prepare/distribute the data */
  /* Set up the parameters for the GE pipeline then call it */
  for (i=0; i<N; i++) {
    stages[i] = (void (*) (void)) eliminate;
  }
  for (i=0; i<N+1; i++) {
    spreads[i] = SPLOCAL; types[i]   = MPI_FLOAT;
  }
  outmul = 0;
  Pipeline (N, stages, myrank(), spreads, types, NULL, 0, 0, NULL, 0, &outmul, mycomm());

  /* Set up the parameters for the BS pipeline, then call it */
  for (i=0; i<N; i++) {
    stages[i] = (void (*) (void)) backsub;
  }
  /* Note the third parameter which achieves the ``reverse direction'' */
  Pipeline (N,  stages, N-myrank()-1, spreads, types, NULL, 0, 0, NULL, 0, &outmul, mycomm());
  MPI_Finalize();
}

void eliminate (void) {
  int startcol = 0;
  float rowin[N+1];
  int i, ncols=N+1;
  while (Take(rowin, &ncols)) {
    if (myrealrank<N-1) Give(rowin, ncols);
    /* work goes here... including decrement ncols */
  }
  /* more work goes here */
  if (myrealrank<N-1) Give(mya, N+1-myrealrank);
}

void backsub (void) {
  int startcol = N-1;
  float knownx;
  int i, ncols, one=1;
  while (Take(&knownx, &one)) {
    if (myrealrank>0) Give(&knownx, 1);
    /* work goes here */
  }
  /* more work here */
  if (myrealrank>0) Give(&myx, 1);
}
```

Figure 1: Gaussian elimination as a sequence of pipelines

3.1 Skeleton Model

Each library skeleton has many parameters. Conceptually, subsets of these control

- the operational detail of the skeleton (e.g. code to be run, grouping of processes to pipeline stages or similar)
- the data to be processed (location, type, size and so on for both inputs and results)
- a context within which this all happens (an MPI communicator)

As far as possible, we have tried to follow MPI conventions and practices in selection of parameters, using the concepts of “colors” and of “pointer, type, length” triples to identify data items.

We make one addition in the area of communicators. This supports our intention of allowing the computational nodes of a skeleton to be internally parallel, where appropriate (e.g. in a pipeline stage). When such a node is defined by a call to a skeleton, an associated communicator is also created, encompassing the processes which have been assigned to the node. This communicator, together with its size and a process' rank within it, can be accessed at any time by calling the library functions `mycomm()`, `mycommsize()` and `myrank()`. At initialisation these simply refer to `MY_COMM_WORLD`.

The programmer remains free to create and destroy communicators arbitrarily, but the natural and recommended programming style will be to use a nested sequence of skeleton calls to create a corresponding hierarchy of `mycomms`, with further ad-hoc partitioning restricted to activities within a particular node (e.g. within a worker node in a task farm).

As a concrete example, suppose that we are working within an internally parallel pipeline stage, and wish to broadcast (within the stage) an integer from the highest (locally) numbered process. The required call will be

```
MPI_Bcast(&thing, 1, MPI_INT, mycommsize()-1, mycomm());
```

3.2 Operational Model

Following MPI, there is an initialisation function which must be called by all processes to set up internal data structures required by the run-time support. The computation then proceeds in conventional MPI fashion, until a call to one of the skeletons is made. As with MPI collective communications, this call must be made by all processes in the given communicator or else deadlock ensues. In traditional skeletal style, the run-time system then assumes responsibility for setting up new communicators, calling computational functions and arranging communications according to the requirements of the given skeleton. To the programmer, the task is to simply to write argument functions which operate in the context provided (e.g. a stage or a worker).

3.3 Implicit and Explicit Interactions

An often stated aim of the skeletal approach is to abstract from details of interaction (which for MPI means communication). However, there is a tension between this admirable goal and the pragmatic need for flexibility. Consider pipelines. In some circumstances it may be appropriate to require that there is exactly one input for each output at each stage. This allows specification of a pipeline skeleton which abstracts away all communication. In other circumstances it may be natural to allow a stage to consume more items than it produces (or vice-versa). This suggests a more flexible skeleton in which the programmer has some control over the implied interaction with its neighbouring stages.

In our design, we intend to allow for (and provide) skeletons which exhibit various degrees of flexibility in this respect. To achieve this generically, we introduce the concept of *explicit interaction with the skeleton* (as opposed to with other processes), by means of a pair of auxilliary library functions `Give` and `Take`, handling return and acquisition of new data items respectively, for use in situations where these activities are not controlled implicitly by the currently active skeleton. The key concept for the programmer is to think of these as interacting *with the skeleton* rather than with other processes. To emphasise this, the functions have interfaces which are independent of the currently active skeleton. The skeleton "knows what to expect", while the coordinated function is simply indicating its readiness to receive or return the next piece of data, a concept which is independent of the wider skeletal scheme within which that data exists. As a concrete example, consider a flexible task farm in which one task may produce many results. Upon producing a new result, a worker group will call `Give` to return this. This is quite independent of whether this will actually correspond to some communication with a "farmer" process or with some other storage arrangement. These are details for the implementation.

3.4 Data Model

MPI's approach to specifying data for library operations uses (pointer, length, type) triples both to identify items to be sent and buffers into which to receive. Our library inherits this convention.

The library adds two concepts. It is important to note that these only have concrete syntactic presence as arguments to library call statements. There are no new data types and no new access syntax.

The first concept, **spread**, concerns whether each contribution passed by a process is to be considered to be complete in itself or part of some larger logically collective, but physically distributed, whole.

The second concept, **multiplicity** concerns how many conceptual items are being provided (as distinct from the information provided by the MPI length parameter). For example, the input to a pipeline will be a sequence of items, each of which is captured by a conventional MPI triple.

3.4.1 The Standard Data Interpretation

All skeletons in our model share the characteristic of accepting and returning data sets specified by the usual MPI triples together with information on spread and multiplicity. For most skeletons, it will also be necessary to define an ordering on data where the multiplicity is greater than one. For example, in a pipeline with local spread and variable multiplicity, in what order will the items be presented to the first stage? Similarly, how will the output (which may differ in every aspect of its structure and multiplicity) be distributed? To answer such questions we define a *standard data interpretation*.

1. for a global spread data item, the constituent items are implicitly ordered by `myrank()` of the contributing processes;

2. for local spread data the constituent items are similarly ordered by `myrank()` within the group and by the stored order within a process (in other words, all contributions from P_0 in order, then all from P_1 in order, and so on).

None of the above implies that any implicit load balancing is performed by the system. For example, we can imagine a one-for-one farm skeleton in which each result is stored by the process which contributed the corresponding task. Processes could contribute tasks with widely varying multiplicity and will receive a correspondingly imbalanced number of outputs. Equally, implicit load balancing is not prohibited and a different farm skeleton could provide implicit balancing of output distribution. Each skeleton will be required to specify its policy. This issue is entirely orthogonal to the question of distributing individual items to accommodate varying **spread**, as discussed below.

Distributions have no persistence across calls. In other words, as noted above, there is no concept here of persistent distributed data objects in the sense of HPF. Instead, contributions from participating processes are interpreted by the rules above purely for the duration of each call, in accordance with the given parameters.

3.4.2 Spread

MPI programmers are already familiar with the idea of building data structures which are distributed across many processors (e.g. chunks of an image) but still have some conceptually unified existence. MPI provides no support for this. Programmers will usually "implement" it by using the same name for the variable contributed by each process, in SPMD style. Meanwhile, in other applications, data items may be defined completely by their local extent.

The **spread** parameter allows the programmer to distinguish these situations. As an example, consider a farm with many processes in a worker. In one context we may want to work with "global" data items, such as images. In a different context we may want to consider the data returned by each process within a worker to be independent items. Both kinds of data will be specified by a local (pointer, size, type) triple. Our additional **spread** parameter tells the run time system how these should be treated collectively.

To summarise, in a p process context, the contributions passed to a skeleton from each process for a **global** data item will be treated as single logical entity, whereas those passed as **local** will be treated as individual entities.

This raises an interesting question. What happens when a global item is passed from a group of size p to a group of size p'? In such situations the (conceptual) global item will be repartitioned for balanced distribution across the receiving group. More precisely, we view the p "sending" contributions (each of a length known from its MPI triple) as a single sequence of these combined lengths, which is then block partitioned (with rounding) across the p' "receiving" processes. Each receiving process will be informed of the size of chunk it has received (as is already familiar from conventional MPI receives, via the status structure).

3.4.3 Multiplicity

The decision to use MPI triples for data results in an overloading of the concept of "length". Consider the task of specifying the collected input to a task farm. Each task will be described by a triple. In addition, we need to know how many such tasks constitute the complete input. We refer to this as the input's "multiplicity" (similarly for output).

There is some interaction between the multiplicity and spread. Specifically, for an input with global spread, the multiplicities provided by each process must be identical. Note that this is distinct from requiring the contributed lengths to be identical: a process can make an empty contribution. In contrast, for data with local spread, contributed multiplicities can vary. Think of the difference between an input consisting of a sequence of ten distributed images (in which each process must provide a contribution, possibly empty, to each image), and an input consisting of a sequence of ten images, each previously generated independently by different processes, with one process contributing eight images, two others one each and perhaps further processes with nothing to contribute.

3.5 Run Time Support

Behind the scenes we need to maintain information on the stack of communicators which are in play. Beyond that, each function will handle its own co-ordination code, with the `Give` and `Take` functions handling redistribution of global items according to information on the context stack. A straightforward implementation would code each skeleton independently. We could subsequently embed code to make context sensitive runtime decisions about strategy, based on info on the stack. This would be similar to the kind of things we could do in a compiler for a structured language.

4 Library Components

4.1 SkelLibInit

The initialisation function. Programmer doesn't need to know what this does, but must call it (like `MPI_Init`).

```
void SkelLibInit(void)
```

4.2 mycomm, mycommsize and myrank

Provide user functions which are being coordinated by skeletons with access to the currently active communication context, such as a pipeline stage or a farm worker.

```
MPI_Comm mycomm(void)
```

```
int mycommsize(void)
```

```
int myrank(void)
```

4.3 Give and Take

Skeleton body code interacts with the system, and thereby indirectly with other body components, through calls to functions `Give` and `Take` which respectively return or request a new data item. Some aspects of the data specification (e.g. spread and multiplicity) are obtained implicitly from the dynamic context (i.e. as was passed to the currently active skeleton). The data location (i.e. a pointer) and length must be given explicitly. As with all skeleton functions, `Give` and `Take` are collective operations within the currently active `mycomm()` (e.g. within a worker, or a stage). For a local spread, only process 0 actually receives or returns data, but all processes receive the boolean result of `Take`.

```
Give(void *out, int len)
```

`out` is a pointer to the local portion of the data item to return. It must contain `len` items of the type and spread specified by the dynamically enclosing skeleton.

```
bool_t Take(void *in, int *len)
```

`in` is a pointer to pre-allocated space in memory for receiving up to `*len` data items, with the returned result indicating receipt of a real value and `*len` being set to indicate its actual length. Providing an insufficiently long buffer will result in a runtime error.

4.4 Skeleton Functions

Here are the two skeletons I have specified so far. In what follows the C type `spread_t` is a two valued enumerated type for spreads, indicating local data (**SPLOCAL**) or global data (**SPGLOBAL**).

4.4.1 Farm

This is a flexible task farming construct. The worker code, which may be internally parallel, is invoked once in each worker sub-group and must make its own calls to `Give` and `Take`. The skeleton effects a dynamic load balancing of tasks to processes, and so is most useful when the workload associated with individual applications is expected to vary widely.

Notice that the size of a task is uniform (given by `inlen`) as is the size of a result (given by `outlen`) . A simple variant of the skeleton, `Farmv` could relax this restriction by making these parameters into arrays. The name `Farmv` is chosen to stress the analogy with MPI's own `MPI_Scatter` and `MPI_Scatterv` (and similar) variants. The distribution of outputs will be balanced across the processes in `comm`.

```
void Farm (int nw, void worker(void), int col,
           void *in,  int inlen, int inmul, spread_t inspr, MPI_Datatype inty,
           void *out, int outlen, int *outmul, spread_t outspr, MPI_Datatype outty,
           MPI_Comm comm)
```

where

`int nw`

denotes the number workers to be created

`void worker(void)`

is the worker task code to be run once in each worker, calling `Give` and `Take` as appropriate

`int col`

controls allocation of processes to workers. `col` must be an integer in the range `0..nw-1` and indicates the worker to which the process will be allocated

`void *in`, `int inlen`, `int inmul`, `spread_t inspr`, `MPI_Datatype inty`

describe the collection of input tasks. `in` points to the first item, with subsequent items found by the usual address arithmetic involving `inlen` and `inty`. `inmul` and `spr` give the spread and multiplicity of the input

`void *out`, `int outlen`, `int *outmul`, `spread_t outspr`, `MPI_Datatype outty`

describe the results, following the standard data interpretation. `out` points to the user-allocated buffer for results, which must have the given common outlen. `outmul` has two purposes. It is set by the programmer to tell the system the length of the pre-allocated out buffer (in units of the given outlen), so that buffer overflow can be detected, and is amended by the system to return the actual number of outputs created (if overflow has not occurred). `outspr` and `outty` give the usual spread and type.

`MPI_Comm comm`

the communicator within which the call is executed. This should normally be `mycomm()`, but an explicit parameter is included for flexibility.

4.4.2 Pipeline

A sequence of stages, in the conventional manner. Each stage function (which may be internally parallel) is invoked once (and so maintains state from one item to the next), and is responsible, through calls to `Give` and `Take` for managing its own data flow. This provides considerable flexibility, allowing, for example, data to be generated or consumed internally to the pipeline, and/or results to be stored internally in the pipeline. More conventionally, a sequence of input data items can be passed to the call from participating processes, according to the standard data interpretation. Similarly, a sequence of outputs can be collected and returned. Their distribution will be balanced across the processes in `comm`. A `Pipelinev` variant could be provided, by analogy with `Farmv`.

```
void Pipeline (int ns, void (*stages[])(void), int col,
               spread_t spr[], MPI_Datatype ty[],
               void *in,  int inlen, int inmul,
               void *out, int outlen, int *outmul,
               MPI_Comm comm)
```

where

`int ns`

is the number of stages.

`void (*stages[]) (void)`

are the processes for the stages, in order, from source to sink.

`int col`

allows control of allocation of processes to stages. `col` should be an integer in the range `0..ns-1`, indicating the stage to which each process should be allocated.

`spread_t spr[], MPI_Datatype ty[]`

Type information for the stage interfaces. These arrays must have `ns+1` elements, one for each stage input and one for the final output. The length and location of transfers will be handled dynamically in `Give` and `Take`.

`void *in, int inlen, int inmul,`

describe the sequence of input tasks, according to the standard data interpretation. `in` points to the first item, with subsequent items found by the usual address arithmetic involving `inlen` and `ty[0]`. `inmul` gives the multiplicity of the input.

`void *out`, `int outlen`, `int *outmul`

describe the results, following the standard data interpretation. `out` points to the user-allocated buffer for results, which must have the given common outlen. `outmul` has two purposes. It is set by the programmer to tell the system the length of the pre-allocated out buffer (in units of the given outlen), so that buffer overflow can be detected, and is amended by the system to return the actual number of outputs created (if overflow has not occurred).

`MPI_Comm comm`

the communicator within which the call is executed. This should normally be `mycomm`, but an explicit parameter is included for flexibility.

4.5 Other Skeletons and Variants

Many other skeletons are possible. At the moment I intend to introduce

- a "Replicate" skeleton, for use as a pipeline stage, allowing a number of identical groups to be created concurrently within a pipeline stage. This will allow stages which are otherwise bottlenecks, and which cannot be further parallelised internally, to be enhanced. The effect will be somewhat like P3L's farm as a stage, but with internal parallelism and without the dynamic load balancing.
- a range of "Divide and Conquer" inspired skeletons, for various restricted forms of D&C (rather than a monolithic D&C skeleton).

Furthermore, I think there is considerable scope for defining families of variants of each skeleton, each making its own restrictions and benefitting or suffering from corresponding implementation changes. I have already noted the possibility of "v" variants of pipeline and farm to allow flexibility in data length. It would also be possible to specify, for example,

- a farm with "one-for-one" creation of results for inputs. Such a skeleton would absorb the calls to `Give` and `Take`, leaving the programmer to code the task;
- similarly, a pipeline with "one-for-one" stages.

Of course, there is a danger in allowing the possibilities to explode. Equally, if undertaken in a principled way, the effect could be similar to way OO languages provided families of related operations (e.g. constructors with varying numbers of arguments), which seems to be both useful and acceptable in practice.