Writing Libraries in MPI*

Anthony Skjellum† Nathan E. Doss Purushotham V. Bangalore†

Computer Science Department† &
NSF Engineering Research Center for Computational Field Simulation
Mississippi State University
PO Box 6176
Mississippi State, MS 39762

Abstract

MPI is the new de facto standard for multicomputer and cluster message passing. This short paper describes the issues underlying the development of parallel libraries with MPI. We motivate the need for parallel libraries, and explain why common message-passing systems do not support them. We describe added features needed to support libraries, and what these features are called in MPI. We mention additional features of MPI that support libraries, but which are beyond the scope of this initial paper on this topic.

We provide guidelines in terms of "do's" and "don'ts" for would-be MPI library writers. We present a linear algebra library in skeletal form on a two-dimensional virtual topology, including code fragments. We summarize by mentioning related work, and comment on future code development of parallel libraries under MPI, as well as porting efforts of existing codes to the system.

1 Introduction

MPI is the new de facto standard for multicomputer and cluster message passing [1]. Besides portability and higher performance, one of the primary motivations for MPI is to support parallel libraries, and, consequently, to support the creation of large-scale multicomputer software.

We describe basic issues in the development of parallel libraries using MPI. We motivate the need for parallel libraries as such, and explain why common message-passing systems do not support them well. We describe the kind of message-passing features that are needed to support libraries, what MPI calls such features, and how they are related to one another. We mention additional features of MPI that support libraries, but which are beyond the scope of this initial paper on this topic.

We provide guidelines in terms of "do's" and "don'ts" for would-be MPI library writers. Given MPI features for library safety, MPI library writers will be able to expect reliable message-passing behavior without sacrificing performance. Furthermore, MPI provides a higher level of abstraction than common message-passing systems, which will help library writers develop more efficient and understandable codes simultaneously. MPI was designed to support primitives that would map efficiently to the existing and expected high performance message passing hardware of multicomputers and gigabit cluster networks. MPI library writers will be able to access such hardware capabilities in portable code.

To concretize the discussion, we present a linear algebra library in skeletal form on a two-dimensional virtual topology, including code fragments. These examples are simplified from other work we have done in the area of scalable libraries, so they represent realistic implementations. We think that this extended example will prove instructive to those who are planning to develop distributed data structures for mathematical libraries [6].

We summarize by mentioning related work, and comment on future code development of parallel libraries under MPI, as well as porting efforts of existing library codes to the system.

2 Motivation

This section motivates the need for parallel libraries. It sets expectations for what libraries need

*Work supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University. Revision 1.1; March 10, 1994.

to work effectively in a distributed memory environment, and explains why common message-passing systems do not support libraries well. MPI's solutions to these needs are discussed.

2.1 Why Parallel Libraries?

Parallel libraries are needed to encapsulate the ponderous details associated with distributed realizations of key algorithms. The programmer needs:

- portable, reusable, high quality codes,
- easily available, reliable codes,
- higher level portability interfaces,
- the ability to hide coding complexities from other programmers and users.

Libraries ensure consistent quality of such procedures, and provide a "higher level" of portability than MPI itself can or should provide. Libraries prevent each programmer from repeating the basic work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies dependent on size of system, or type of floating point), this too needs to be hidden from the user.

2.2 What's Missing?

The following features are missing from common vendor and portability systems:

- contexts of message passing (isolation),
- collective communication functionality, defined with process-group scope,
- virtual-topology naming of processes.

The subsequent paragraphs elaborate on each of these topics, and why libraries need these features.

2.2.1 Communication Contexts — Isolation

In the sequential Fortran and C environments, it is reasonably easy to create libraries, because the stack-oriented procedural programming model has well-defined conditions about reasonable versus erroneous programs. However, in the distributed-memory, message-passing environment, it is difficult to write libraries with either vendor or common portability systems. This seemingly strong indictment is backed up as follows: *There is no reasonable way for libraries to isolate themselves from the on-going point-to-point message passing present in a running application.*

Message tags, the sole means for designating restrictions on message delivery, are insufficient for isolation. For instance, more than one library (or invocation of the same library) could use the same tags. Systems that lack source selectivity have to use tags to create deterministic collective operations; in such a situation, collective and point-to-point messages can be misqueued unless the system reserves tags for this purpose. Finally, wildcard receipt-selectivity on tags destroys any promise of real protection that tags could otherwise afford.

2.2.2 Collective Operations

Collective operations, such as reduce, broadcast, and so on, are not defined in all common systems, and when defined are most often only available over the full set of processors (or processes) allocated to the user. Libraries need to work on different collections of processes, and to have available collective operations that only impact the processes that care about the collective operation; uninvolved processes want to remain uninvolved. Furthermore, the collective operations have to work reliably, and without race conditions.

2.2.3 Virtual-Topology Naming

Libraries don't want to describe point-to-point communication in terms of hardware-dependent names; in fact, many algorithms are more natural if described in terms of point-to-point calls relative to a virtual-topology naming scheme. Virtual-topology naming might reflect row or column parallelism in matrix operations, for instance [5].

So, we have established three key areas that will support the creation of libraries:

- Safe communication space (isolation),
- Support for collective operation, not just point-to-point; group scope is needed for collective operations,
- Abstract names for processes are based on virtual topologies, or at least rank-in-group names.

2.3 MPI's Support for Libraries

In MPI, the features that implement the important concepts discussed in the previous section are as follows:

- **Process Groups** define an ordered collection of processes, each with a rank. Process groups define the low-level names for inter-process communication. For point-to-point message passing, process groups define the rank of sender and recipient. For collective communication, process groups define the set of participants.

- **Virtual Topologies** are the final link in the naming process for libraries, providing an alternative to simple rank naming for libraries. Cartesian and general graph topologies are supported. The user can also define his or her own topology strategies.

- **Contexts** provide the ability to have separate safe "universes" of message-passing for process groups in MPI. (A context is a concept made up of a process group and other information, it does not specify an implementation.) We talk about contexts in MPI, but they are actually implemented with communicators.

- **Communicators** encapsulate contexts, groups, and virtual topologies in an object that provides the appropriate scope for all communication operations in MPI. Communicators bind process groups and contexts together to form a safe communication space within the group.

In summary, the use of separate communication contexts by distinct libraries (or distinct library invocations) will insulate communication internal to the library execution from external communication (group safety). This allows the invocation of the library even if there are pending communications, and avoids the need to synchronize each entry into and exit from library code.

2.4 Additional MPI features for libraries

MPI also provides several other features which are useful for building scalable libraries:

- Duplication of communicators allows libraries to get additional communication space easily. (Good implementations may/will cache such additional communication space, resulting in significant performance benefits.)

- Systematic, aligned subsetting of communicators allows communicating SPMD processes to define hierarchies of communicating sub-groups.

- Caching of attributes on communicators allows users to define new operations that have the same conceptual properties as built-in MPI collective operations. This is important for extending services not anticipated by MPI.

3 Guidelines for Library Writers

In this section, we bulletize important ideas for library writers based on our previous experience with the *Multicomputer Toolbox* and *Zipcode* [8]. Programmers who have tried to write libraries with common systems will appreciate the simplicity of writing message-passing code with MPI.

3.1 Do's

- Define persistent distributed structures, based on communicators,

- Duplicate communicators for safe communication space when entering calls to parallel libraries,

- Use MPI-defined virtual topology information to make algorithms easier to understand,

- Augment virtual topology technology as needed (*e.g.*, topology hierarchies, as in section 4),

- Use attribute caching when adding additional collective operations to a communicator.

3.2 Don'ts

- Publish how they use the message resources (*e.g.*, tag allotments),

- Synchronize excessively at the entry and exit of parallel calls,

- Guarantee that communicators be "quiescent" before they can do collective operations,

- Restrict the library programming model to single invocation per process technology,

- Restrict the library programming model to non-overlapping process group assumptions,

- Write libraries that work only over "all" processors (processes) of a user's allocation.

4 A 2D-Grid Vector Class Library

In this section we develop a "vector" library for linear algebra on a two-dimensional logical topology and demonstrate two variants of "dot product" for aligned, distributed vectors.

4.1 Vector Layout

In a 2D logical grid, vectors may either be column or row vectors. We assume that each row (resp, column) vector is replicated over each row (resp, column) of a logical grid. Figure 1 shows how two row vectors might be distributed in a grid. Each vector is broken into two pieces and distributed along the columns of the grid. Each process row has a copy of both vectors.

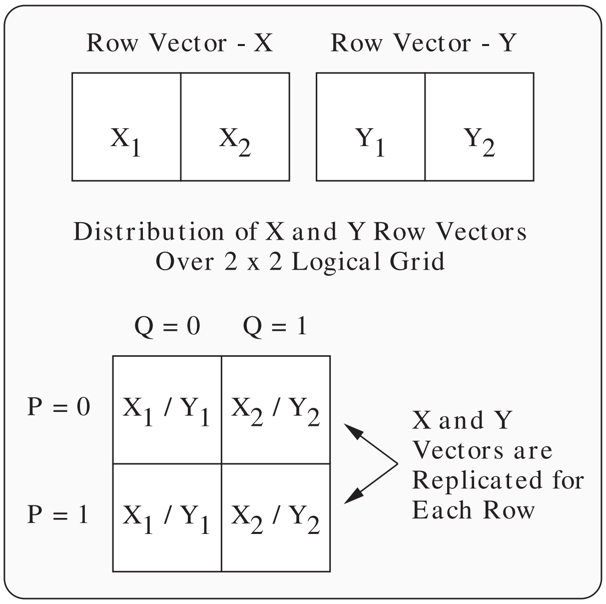

Figure 1: This figure shows the layout of two row vectors over a 2 x 2 logical process grid. Column vectors are laid out in an analogous, fashion, orthogonal to the row vectors.

4.2 Creating a Logical Grid

MPI provides a set of routines which can be used to create virtual topologies. In this first example, we show how a library writer might create a logical 2D grid structure without using the topology routines provided by MPI.

Here is the persistent data structure needed to create a logical grid along with a simple function to initialize a grid.

```
/* from tbx_library.h */
typedef struct tbx_2d_grid
{
  int P, Q;  /* global shape */
  int p, q;  /* position */

  MPI_Comm grid_comm;
  MPI_Comm row_comm;
  MPI_comm col_comm;
} TBX_2D_GRID;

/* from tbx_grid.c */
TBX_GRID *tbx_build_grid(MPI_Comm comm,
                         int P, Q)
{
  TBX_2D_GRID *grid;
  int       my_rank;
  int       p, q;

  /* Make new grid */
  grid = NEW_CLEAR_ITEM(TBX_2D_GRID, 1);
  MPI_Comm_dup(comm, &(grid->grid_comm));

  /* Determine row and column position */
  MPI_Comm_rank(grid->grid_comm, &my_rank);
  p = my_rank / Q;
  q = my_rank % Q;

  /* Split comm into row and col comms */
  MPI_Comm_split(grid->grid_comm, p, q,
                 &(grid->row));
  MPI_Comm_split(grid->grid_comm, q, p,
                 &(grid->col));

  /* Fill in my position and grid size */
  grid->P        = P;
  grid->Q        = Q;
  grid->p        = p;
  grid->q        = q;

  /* Return the newly built grid */
  return (grid);
}
```

4.3 Vector Structure and Creation

Vectors are defined by the following data structure. As a distributed object, they contain local information including global and local lengths, local real data storage, and a type field that indicates if they are row- or column-distributed. In addition, the data structure is based on a logical grid topology data structure, and a data distribution topology. The former encapsulates communicators, the latter encapsulates functions that map global indices to local indices and vice versa. See Figure 2.

```
/* from tbx_library.h */

#define TBX_ROW 0
#define TBX_COL 1

typedef struct tbx_dvector
{
  int length;    /* full length of vector */
  int local_length;/* local vector length */
  int type;         /* row or column type */

  double *data;     /* vector data */

  TBX_2D_GRID *grid;  /* logical grid */
  TBX_DISTRIB *dis;   /* how to map data */

  /* functions working on vectors */
  double (*dot) (struct tbx_dvector *x, *y);
  /* ... */

} TBX_DVECTOR;
```

The following function is the constructor for a distributed vector.

```
/* from tbx_dvector.c */

/*
 * length = total length of vector
 * type   = row or column vector?
 * grid   = grid which vector belongs to
 * dis    = distribution functions
 *          (how coefficients map)
 */
TBX_DVECTOR
 *new_dvector(int length, type,
       TBX_GRID *grid, TBX_DISTRIB *dis)
{
  TBX_DVECTOR *vec;
  int          local_length;

  /* Create a new vector */
  vec = NEW_CLEAR_ITEM(TBX_DVECTOR,1);

  /* figure the local length - i.e., how */
  /* many coefficients (and which ones)  */
  /* map to this process                 */
  local_length = (*dis -> local_length)(
                grid, dis, length, type);

  /* Fill in the vector structure */
  vec->length       = length;
  vec->local_length = local_length;
  vec->data         =
     NEW_CLEAR_ITEM(double,local_length);
  vec->type         = type;
  vec->grid         = grid;
  vec->dis          = dis;

  /* assign values to default fns */
  vec->dot = ddot_strided;
  /* ... */
}
```

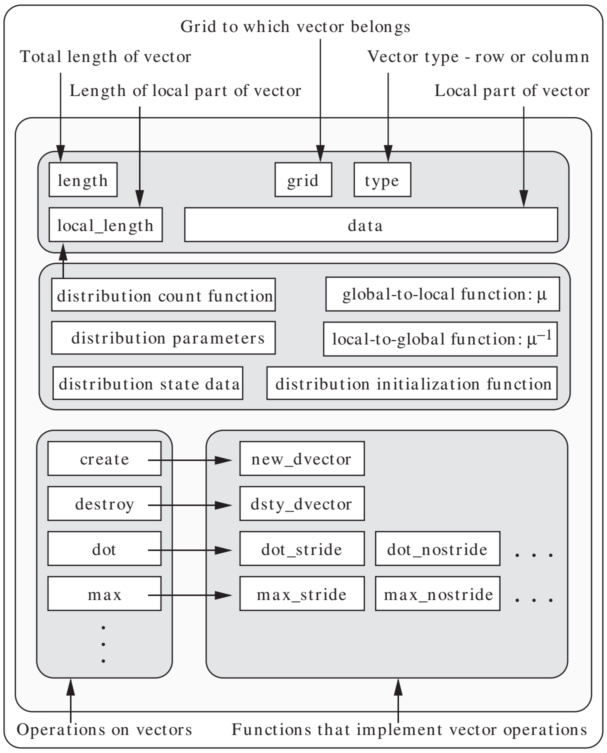

Figure 2: This figure shows the structure of a distributed vector.

4.4 Dot Product Functions

High-quality libraries will have to provide more than one version of a function that would perform efficiently over different conditions (we call such a set of methods a *poly-algorithm*). For example, here we provide two dot product functions (strided and non-strided). The strided version has a reduced computational load at the expense of more communication. Conversely, the non-strided version is computationally more intensive but fewer messages are needed, because the algorithm exploits data redundancy. The most efficient function is determined by comparing issues such as message latency and bandwidth, and floating point performance, all with reference to the size of the vector, and the logical grid dimensions P, Q.

4.4.1 Non-Strided

The non-strided function works independently in process rows (resp, columns) for row-distributed (resp, column-distributed) vectors. First, the local contribution to the dot product is computed in each process, and then, a "combine" function, called `MPI_Allreduce()` sums the partial sums, leaving the dot product in each process. The collective communication occurs independently in rows (resp, columns).

```
/* from tbx_dvector.c */
double ddot_nostride(TBX_DVECTOR *x, *y)
{
  int    i;
  double local_sum = 0.0;
  double sum = 0.0;

  /* Check for compatible types */
  if (x->type != y->type)
  {  /* erroneous */ }

  /* Sum up my portion of the vector */
  for (i = 0; i < x->local_length; i++)
    local_sum += x->data[i] * y->data[i];

  /* Get row or column sum based on type */
  if (x->type == TBX_ROW)
    MPI_Allreduce(&local_sum, &sum, 1,
      MPI_DOUBLE,MPI_SUM,x->grid->row_comm);
  else
    MPI_Allreduce(&local_sum, &sum, 1,
      MPI_DOUBLE,MPI_SUM,x->grid->col_comm);

  /* Return the result */
  return (sum);
}
```

4.4.2 Strided

The strided version of dot product relies on the fact that there is data replication. Hence, only a fraction of each sum need be computed in any process. However, given striding, the communication required to form the global sum requires `MPI_Allreduce()` to work over all processes specified by the logical grid, rather than independently in rows or columns, as above.

```
/* from tbx_dvector.c */
double ddot_stride(TBX_DVECTOR *x, *y)
{
  int    i, stride;
  double local_sum = 0.0;
  double sum = 0.0;

  /* Check for compatible types */
  if (x->type != y->type)
  {    /* erroneous */  }


  /* Sum up the local part */
  if (x -> type == TBX_ROW)
     for (i = x->grid->p; i < x->local_length;
          i += x->grid->P)
        sum += x -> data[i] * y -> data[i];
  else
     for (i = x->grid->q; i < x->local_length;
          i += x->grid->Q)
        sum += x->data[i] * y->data[i];

  /* Get the sum of all parts */
  MPI_Allreduce(&local_sum, &sum, 1,
    MPI_DOUBLE, MPI_SUM, x->grid->grid_comm);

  /* Return result */
  return(sum);
}
```

5 The Example Revisited

In this section we revise the "vector" library for linear algebra on a two-dimensional logical topology and demonstrate two variants of "dot product" for distributed vectors. This time, we use the MPI-defined cartesian topologies to help. As before, in a 2D logical grid, vectors may either be column or row vectors. Everything else described in the first example remains the same.

5.1 Creating a Logical Grid

MPI provides a set of routines which can be used to create virtual topologies. The function `MPI_Make_cart()` is used for cartesian topologies. The structure `tbx_2d_grid` is unchanged from before. However, the function `tbx_build_grid()` makes use of the MPI cartesian virtual topology functions, and is somewhat different.

```
#define N_DIMS 2

/* from tbx_grid.c */
TBX_GRID *tbx_build_grid(MPI_Comm comm,
                            int P, Q)
{
  TBX_2D_GRID  *grid;
  int        my_rank;
  int           p, q;
  int   dims[N_DIMS], /* hold dimensions */
       local[N_DIMS], /* local position  */
      period[N_DIMS]; /* aperiodic flags */

  /* Make new grid */
  grid = NEW_CLEAR_ITEM(TBX_2D_GRID, 1);

  /* Generate a new communicator with
     virtual topology added: */
  dims[0] = P; dims[1] = Q;
  period[0] = period[1] = FALSE;
  MPI_Make_cart(comm, N_DIMS, dims, period,
              TRUE, &(grid->grid_comm));

  /* map back to topology coordinates: */
  MPI_Comm_rank(grid->grid_comm, &my_rank);
  MPI_Cart_coords(grid->grid_comm, my_rank,
                N_DIMS, &local);
  p = local[0];
  q = local[1];

  /* Split comm into row and col comms */
  MPI_Comm_split(grid->grid_comm, p, q,
              &(grid->row));
  MPI_Comm_split(grid->grid_comm, q, p,
              &(grid->col));

  /* Fill in my position and grid size */
  grid->P        = P;
  grid->Q        = Q;
  grid->p        = p;
  grid->q        = q;

  /* Return the newly built grid */
  return (grid);
}
```

6 Related Work

To find out more about MPI, see the document [1] available on **netlib**. A quick way to access this is to send a mail message to *netlib@ornl.gov* containing the line `send index from MPI`. Alternatively, one can anonymous ftp the source from **netlib2@cs.utk.edu**.

To learn more about full-scale parallel libraries and general distributed data structures along the lines described above, see one or more of the following references concerning the *Multicomputer Toolbox* [3, 4, 5, 6, 7].

7 Conclusions

We introduced MPI, the new de facto standard for multicomputer and cluster message passing [1]. We described the issues underlying the development of parallel libraries with MPI. In fact, we began by motivating the need for parallel libraries per se, and we explained why common message-passing systems do not support them. We described added features needed to support such libraries, and what these features are called in MPI. We mentioned additional features of MPI that support libraries, but which are beyond the scope of this initial paper on this topic (see [6]).

We provided basic "do's" and "don'ts" for would-be MPI library writers. We presented a linear algebra library in skeletal form on a two-dimensional virtual topology, including code fragments. We summarized by mentioning related work, and comment on future code development of parallel libraries under MPI, as well as porting efforts of existing codes to the system.

References

[1] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report Technical Report No. CS-93-214., University of Tennessee, November 1993. Available on **netlib**.

[2] William Gropp and Ewing Lusk. An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message-Passing Interface. Technical Report MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[3] Anthony Skjellum. The Multicomputer Toolbox: Current and Future Directions. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[4] Anthony Skjellum, Steven F. Ashby, Peter N. Brown, Milo R. Dorr, and Alan C. Hindmarsh. The

Multicomputer Toolbox. In G. L. Struble et al., editors, *Laboratory Directed Research and Development FY91 – LLNL*, pages 24–26. Lawrence Livermore National Laboratory, August 1992. UCRL-53689-91 (Rev 1).

[5] Anthony Skjellum and Chuck H. Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[6] Anthony Skjellum et al. Scalable Libraries Using MPI: Abstraction, Performance, Portability. In preparation, 1994.

[7] Anthony Skjellum, Alvin P. Leung, Charles H. Still Steven G. Smith, Robert D. Falgout, and Chuck H. Baldwin. The Multicomputer Toolbox – First-Generation Scalable Libraries. In *Proceedings of HICSS–27*. IEEE Computer Society Press, 1994. HICSS–27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[8] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 1994. (Invited Paper, to appear in Special Issue on Message Passing).

[9] Steven G. Smith, Robert D. Falgout, Charles H. Still, and Anthony Skjellum. High-level Message-passing Constructs for Zipcode 1.0: Design and Implementation. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, 1993.