Application Composition in Ensemble using Intercommunicators and Process Topologies

Yiannis Cotronis

Dept. of Informatics and Telecommunications, Univ. of Athens, 15784 Athens, Greece
cotronis@di.uoa.gr

Abstract. Ensemble has been proposed for composing MPMD applications maintaining a single code for each component. In this paper we describe application composition using intercommunicators and process topologies. We demonstrate on a climate model configuration.

1. Introduction

Coupling MPI [6] applications requires significant code modification and high S/W engineering effort. We have proposed the Ensemble methodology [1,2] in which MPI components are developed separately and applications are composed into different configurations, maintaining a single code for each component. Ensemble enables the composition of genuine MPMD applications, where *P* indicates "*Source Programs*" and not "*Program Execution Behaviors*". Ensemble is particularly appropriate for composing MPI applications on grids [3], where individual components are developed and maintained by different teams. Applications may be composed in various configurations (SPMD, MPMD, regular, irregular) specified symbolically in a high-level composition language. Composed programs though, are pure MPI programs running directly on MPICH [4] or MPICH-G2 [5].

In the past we have presented Ensemble supporting process groups in intracommunicators and their (point-to-point [2] and collective [1]) communication. In this paper we present process groups in intercommunicators and topologies.

The structure of the paper is as follows: In section 2, we present the composition principles, the structure of the components and their low level composition. In section 3, we outline the implementation of three components (atmospheric, ocean and land models), which we use in section 4 to compose a climate model configuration. In section 5 we present our conclusions.

2. The Composition Principles

If processes are to be coupled together in various configurations they should not have any specific envelope data (which implicitly determine topologies) built into their code, but rather be provided individually to each process dynamically. In Ensemble

components all envelope data in MPI calls (i.e., communicators, ranks, message tags and roots) are specified as "formal communication parameters". The "actual communication parameters" are passed to each process individually in command line arguments (CLA).

Let us first show the feasibility of this approach. The envelope data in MPI calls are bound to three types: communicators, ranks (roots are ranks) and message tags. The first two cannot be passed directly in CLA: the communicator, because it is not a basic type and the rank, although an integer, because it is not known before process spawning. Instead we construct communicators and indirectly determine ranks. We associate with each process a unique integer, namely the Unique Ensemble Rank (UER), which is passed as the first parameter in its CLA. We use UERs in CLA to specify "actual communication parameters" and processes need to determine process ranks from UERs (e.g. in a communication call a UER has to be interpreted as the rank of the process associated with it). To this end, a communicator Ensemble_Comm_World is constructed by "splitting" all processes in MPI_COMM_WORLD with a common color, each using as key its UER. The new communicator reorders processes, with MPI ranks equal to UERs. For constructing other communicators we pass an integer indicating a color and construct the required communicator by splitting Ensemble_Comm_World. To find the rank of a process associated with a UER in a constructed communicator we use MPI_Group_translate_ranks and its rank in Ensemble_Comm_World (=UER).

To develop components and compose programs we need to specify four inter-depended entities. The structure of CLAs, which are the composition directives for each process; a routine, say SetEnvArgs, which interprets CLAs and produces MPI envelope data; a data structure, say EnvArgs, for storing MPI envelope data; and finally using elements of EnvArgs in MPI calls.

2.1 The structure EnvArgs and MPI Envelope Bindings

Envelope data in EnvArgs is organized in contexts. For point-to-point communication ports are introduced, which are abstractions of the envelope pair (rank, message tag). Ports with similar usage form arrays of ports (MultiPorts).

```
Context EnvArgs[NrContexts];/*Context Arguments */
typedef struct /* Envelope in a context*/
{ MPI_Comm IntraComm; /* the intracommunicator */
  MultiPortType MultiPorts[NrMultiPorts];
}Context;
```

Communicating processes must be in the same communicator, whether intra or inter. Each multiport may specify a separate inter or topology communicator. As point-to-point communication is performed by the same send/receive routines irrespective of the type of communicator, envelop data in all point-to-point calls may just refer to ports. The type of the actual communicator (intra, inter or topology) would be specified in the CLA and set by SetEnvArgs together with all other data port consistent with their semantics (e.g. if inter then rank will indicate the rank of a process in the remote intracommunicator).

```
typedef struct   /* A MultiPort*/
{ MPI_Comm Comm; /* intra, inter or topology */
  PortType Ports[NrPorts];
}MultiPortType;
```

```
typedef struct   /* A single Port */
{ int Rank;       /* Rank of Communicating Process */
  int MessageTag; /* Message Tag */
}PortType;
```

As the focus of this paper is intercommunicators and process topologies, we do not present other aspects of EnvArgs such as collective communications. We may use elements of EnvArgs in MPI calls, as for example in

```
MPI_Send(Data, count, type,
          EnvArgs[2].Multiports[3].Ports[1].Rank
          EnvArgs[2].Multiports[3].Ports[1].MessageTag,
          EnvArgs[2].Multiports[3].Comm);
```

For convenience we may define virtual envelope names for communicators and multiports. Instead of using envelope data directly, we use macros, which refer to virtual envelope names. We present virtual envelopes and macros in section 3.

2.2 The CLA structure and SetEnvArgs

The CLAs for each process are envelope data, but effectively they are composition directives, relevant to each process. The structure of CLA is:

- UER
- Number of splits of MPI_COMM_WOLRD. For each split:
  - Its color. If color >= 0 (process belongs in the intracommunicator):
    - Ctxindex
    - Number of multiports in intra; for each multiport:
      - Its MPindex and number of ports; for each port: UER and Msg Tag
    - Number of Topologies from intra; for each topology
      - Orderkey; Cartesian or graph
      - If Cartesian: ndims, dims array, period array
      - If Graph: nnodes, index array, edges array
      - Number of multiports in topology; for each multiport:
        - Its MPindex and Number of ports; for each port: UER and Msg Tag
    - Number of InterCommunicators using intra as local; for each intercomm
      - UER of local and remote leaders; MsgTag
      - Number of multiports in intercommunicator; for each multiport:
        - Its MPindex and Number of ports; for each port: UER and Msg Tag

SetEnvArgs is called after MPI_Init and its actions are directed by CLAs. It first reads UER and the number of splits of MPI_COMM_WORLD; enters a split loop. For each split it reads its color and constructs a new communicator (if negative no communicator is constructed). It reads its index in EnvArgs (Ctxindex). The first split

must be for Ensemble_Comm_World reordering ranks (all processes use a common color). The new communicator is stored in EnvArgs[Ctxindex].IntraComm.

It then reads the number of multiports using the constructed intracommunicator and enters a multiport loop. For each multiport it reads its index in the context (MPindex), sets MultiPorts[MPindex].Comm to IntraComm and reads the number of ports; enters a port loop. For each port it reads UER and MsgTag; translates UER to rank and stores rank and MsgTag.

It then reads the number of topologies to be constructed from IntraComm (superposition of topologies is supported) and enters a topology loop. For each topology a temporary communicator is constructed, reordering processes in IntraComm according to the orderkey given in the CLAs (the orderkey should be consistent with row major numbering in cartesian topologies and the index and edges arrays for graph topologies). From this temporary communicator cartesian or graph topology communicators are constructed by passing appropriate arguments. For Cartesian: ndims, dims array and period array; reorder=false as we would like to maintain orderkey ordering. For graph: nnodes, index array, edges array; again reorder=false. It then reads the number of multiports using the topology; enters a multiport loop, as in the case of the intracommunicator.

It then reads the number of intercommunicators, which use IntraComm as the local communicator. It reads and interprets the UERs of the local and remote leaders, reads MsgTag and constructs the intercommunicator, using Ensemble_Comm_World as bridge communicator. It then reads the number of multiports using this intercommunicator and enters a multiport loop, as in the case of intracommunicators.

The CLAs are tedious to construct and error prone. Furthermore, they cannot be parameterized. For this purpose, we are developing a symbolic and parametric high-level composition language, from which low-level CLA composition directives are generated; the language is outlined by example in section 4.

3. The Component Structure

In the previous section we presented the principles and the low level composition related to point-to-point communication in intra, inter and topology communicators. We will demonstrate these principles using three components. The example application is inspired from the climate model involving Atmospheric, Ocean and Land models. In this section, we outline the three components and in the next section we compose the climate model.

Atm processes communicate within the model with atm neighbors in a two-dimensional mesh, exchanging halo rows and columns of data internal to the atmospheric model. In the virtual envelope of the Atm component (table 1) we specify context Internal, having multiports N, S, E, W, within which all such communication is performed. Atm processes may also communicate with other processes (e.g. ocean or land) in the vertical boundary. For this we specify context VerticalBoundary, having only multiport Down. For simplicity, we have specified at most one port, indicating one-to-one correspondence of atm and surface processes.

The code structure of Atm component is just sketched; the two MPI_Send calls show the use of macros in the envelope data (boxed italics).

Table 1. **The Virtual Envelope and PseudoCode of Atm**

```
  Context Internal;
    Ports N, S, E, W[0..1];
  Context VerticalBoundary;
    Ports Down[0..1];
repeat
  Model Computations
  Communication in context Internal
  MPI_Send(HaloDA,HN,HTA,ENVport(N,1,Internal));
  Communication in context VerticalBoundary
  MPI_Send(VD,VN,VT,ENVport(Down,1,VerticalBoundary));
until termination;
```

Ocean component requires three contexts. One for communicating halo data internal to the model, a second for communicating surface boundary data to land processes and a third for communicating vertical boundary data to atm processes. The first two have N, S, E, W and respectively BN, BS, BE, BW multiports. The third context has just one multiport Up.

Table 2. **The Virtual Envelope and PseudoCode of Ocean**

```
  Context Internal;
    Ports N, S, E, W[0..1];
  Context SurfaceBoundary;
    Ports BN, BS, BE, BW[0..1];
  Context VerticalBoundary;
    Ports Up[0..1];
repeat
  Model Computations
  Communication in context Internal
  MPI_Send(HD,HN,HTO,ENVport(N,1,Internal));
  Communication in context SurfaceBoundary
  MPI_Send(SBD,SBN,SBT,ENVport(BN,1,SurfaceBoundary));
  Communication in context VerticalBoundary
  MPI_Send(VD,V,VT,ENVport(Up,1,VerticalBoundary));
until termination
```

The virtual envelope of the Land component is similar to Ocean.

Table 3. **The Virtual Envelope of Land**

```
  Context Internal;
    Ports N,S,E,W[0..1];
  Context SurfaceBoundary;
    Ports N,S,E,W[0..1];
  Context VerticalBoundary;
    Ports Up[0..1];
```

4. The High Level Composition

The composition of applications is specified in three levels each representing an abstraction of application composition. The top level is the Reconfigurable High Level Composition (RHLC) in which symbolic names for processes are defined as component instances, their number is parametrically specified and they are grouped into symbolic communicators (intra, inter and topology). In each communicator point-to-point communication channels between process ports are defined. RHLC specifies reconfigurable applications, in two aspects: a) different configurations may be obtained, depending on the values and the generality of parameters, and b) they are independent of any execution environment resources (machines, files, etc). The RHLC is then configured obtaining the Configured High Level Composition (CHLC) by setting values to parameters and specifying execution resources. The actual configuration decisions are outside the scope of Ensemble. From CHLC Low Level Composition (LLC) directives (cf section 2) are generated.

We demonstrate the RHLC of a climate model (fig. 1). Atm processes form a cartesian topology (AtmPlane) exchanging data internal to the model. Ocean and Land (depicted as grey) processes together also form a Cartesian topology (SurfacePlane). Here however, there are two kinds of communications: internal to each model (ocean to ocean and land to land) or boundary (ocean to land). Finally, there is a vertical exchange of data, between the two planes, for which we will use an intercommunicator. Node that in the components' code there is no indication of the communicator type used in the point-to-point communication. This is considered a design decision for the RHLC.

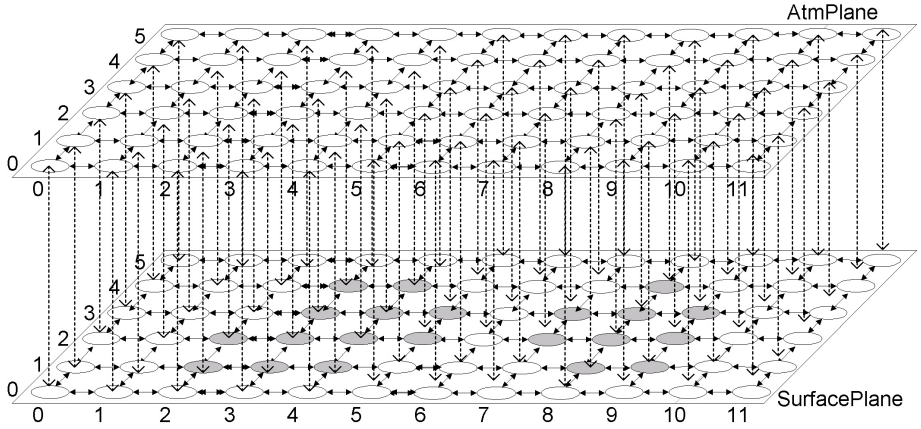

Fig. 1. **The Climate Model**

In the Parameter section of RHLC (table 4), integer identifiers are declared, which parameterize the design: AtmRows, AtmCols and SurfRows, SurfCols respectively, determine the sizes of the two process meshes; elements of array OLmask determine the covering of SurfacePlane by Ocean (=0) or Land (=1).

Next the processes in the application (in Ensemble_Comm_World) are specified as instances of components. All AtmP(,) processes are instances of Atm, but SurfP(,) are

either instances of Ocean or Land determined by OLmask. Processes are then grouped into communicators (intra, inter, topology) and within each communicator its point-to-point communications are specified.

Table 4. **A Reconfigurable High Level Composition of a Climate model**

```
Parameter
  AtmRows, AtmCols, SurfRows, SurfCols;
  OLmask[SurfRows,SurfCols];
IntraComm Ensemble_Comm_World;
  processes AtmP(AtmRows,AtmCols) from component Atm;
  processes SurfP(SurfRows,SurfCols)from component
  {#i:0(1)SurfRows [#j:0(1)SurfCols
    [(OLmask[i,j]=0):Ocean; (OLmask[i,j]=1):Land]]};
IntraComm AtmPlane;
  processes AtmP(*,*)in Internal;
  cartesian on N,S,E,W;
  channels
  #i:0(1)AtmRows [#j:0(1)AtmCols [
    (i<AtmRows): AtmP(i,j).^S[1] <-> AtmP(i+1,j).^N[1];
    (j<AtmCols): AtmP(i,j).^E[1] <-> AtmP(i,j+1).^W[1]]]
IntraComm SurfaceInternal;
  processes SurfP(*,*) in Internal;
  cartesian on N,S,E,W;
  channels
  #i:0(1)SurfRows [#j:0(1)SurfCols [
    (i<SurfRows)&(OLmask[i,j]=OLmask[i+1,j]):
           SurfP(i,j).^S[1] <-> SurfP(i+1,j).^N[1];
    (j<SurfCols)&(OLmask[i,j]=OLmask[i,j+1]):
           SurfP(i,j).^E[1] <-> SurfP(i,j+1).^W[1];]]
IntraComm SurfaceOther;
  processes SurfP(*,*) in SurfaceBoundary;
  cartesian on BN,BS,BE,BW;
  channels
  #i:0(1)SurfRows [#j:0(1)SurfCols [
    (i<SurfRows)&(OLmask[i,j]!=OLmask[i+1,j]):
           SurfP(i,j).^BS[1] <-> SurfP(i+1,j).^BN[1];
    (j<SurfCols)&(OLmask[i,j]!=OLmask[i,j+1]):
           SurfP(i,j).^BE[1] <-> SurfP(i,j+1).^BW[1]]]
IntraComm SurfaceGroup;
  processes SurfP(*,*) in VerticalBoundary;
InterComm Coupling;
  Intra1 AtmPlane;     leader AtmP(0,0);
  Intra2 SurfaceGroup; leader SurfP(0,0);
  channels
  #i:0(1)AtmRows [#j:0(1)AtmCols[
   AtmP(i,j).VerticalBoundary.Down[1] <->
   SurfP(i,j).VerticalBoundary.Up[1]]]
```

The first group is AtmPlane, in which AtmPs participate in their Internal context. They form a Cartesian geometry and the communication channels are explicitly specified by channel-patterns. A basic channel-pattern is a guarded point-to-point pattern of the form `Guard:Proc.Context.Port<->Proc.Context.Port`. Only point-to-point patterns with guards=true are valid. `Process` is the symbolic name of a process, `Context` is the virtual context within which communication takes place and `Port` is a communication port. The symbol ^ indicates the default context. A channel-pattern may also be a for-structure controlling channel-patterns `#index:from(step)to[{channel-pattern[expr]}+]`. Cartesian on N, S, E, W indicates a Cartesian topology and the multiports using it.

For the SurfacePlane we specify two groups: one in the context of internal and a second in SurfaceBoundary. The former connects ports within each model and the latter connects ports in the boundary of the two models. Both groups however specify the same cartesian topology. Finally, AtmPs and SurfPs form an intercommunicator. We first construct SurfaceGroup of all surface processes and use it together with AtmPlane to construct the intercommunicator and specify communication channels.

5. Conclusions

We presented application composition in Ensemble using intercommunicators and topologies, thus fully supporting MPI. The low-level composition aspects (CLA structure, SetEnvArgs, EnvArgs and macros) have been completed and the High Level Composition is under development. Ensemble reduces S/W engineering costs as a single reusable code is maintained for each component. Ensemble also reduces costs for developing single components (code only involves data movement, reduction and synchronization). Consequently, it may be advantageous to use Ensemble even for SPMD applications. Furthermore, components are neutral to the communicator type (intra, inter, topology) for point-to-point communication. Consequently, a process spawned from a component, may be part of a topology or an intercommunicator, as specified by the composition directives.

Acknowledgment. Supported by Special Account for Research of Univ. of Athens.

References

1. Cotronis, J.Y, (2002) Modular MPI Components and the Composition of Grid Applications, Proc. PDP 2002, IEEE Press pp 154-161.
2. Cotronis, J.Y., Tsiatsoulis Z., Modular MPI and PVM components, PVM/MPI'02, LNCS 2474, pp. 252-259, Springer.
3. Foster, I. and Kesselman, C (eds.) The Grid, Blueprint for the New Computing Infrastructure, Morgan Kaufmann, 1999.
4. Gropp, W. and Lusk, E. Installation and user's guide for mpich, a portable implementation of MPI, ANL-01/x, Argone National Laboratory, 2001.
5. Karonis, N., Toonen B., Foster, I.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, preprint ANL/MCS-P942-0402, April 2002 (to appear in JPDC).
6. Message Passing Interface Forum MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4):165-414, 1994.