Composition of MPI Applications over MPICH-G2

Yiannis Cotronis

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

Abstract

Coupling grid applications requires code modification and high S/W engineering effort. We propose the Ensemble methodology in which message passing components are developed separately and applications, whether regular, irregular, SPMD or MPMD, are composed without component modification. Composed applications are pure MPI programs running on MPICH-G . We demonstrate our approach by developing two simplified atmospheric and ocean components, which may run on their own or coupled together (climate model) in any required configuration depending on geography or other design issues.

1. Introduction

The most popular parallel programming style is presently SPMD, mainly due to its simplicity. SPMD programs are constructed even when "composing" MPMD applications. However, Grids [8] impose new requirements concerning program modularity, since applications (possibly regular SPMD) may need to be coupled with other applications developed by different teams. An application may still be required to run independently, possibly as an SPMD (e.g. atmospheric model), or to be coupled with other applications (e.g. ocean model) running together as MPMD (e.g. climate model). Even regular SPMD applications need substantial code modification to be coupled with other applications. Usually, different code modifications are necessary for different application configurations. For example, the climate model may be used to model the global earth climate or the more local El-Ninio phenomenon (different geography). We may also need to couple only the atmospheric and ocean model and later add land and hydrology models. Code modifications required in each case make single code maintenance of individual applications (e.g. atmospheric, ocean) a difficult task.

We have proposed the Ensemble methodology [3,4] in which message passing (MP) applications are designed and built by composing modular MP components. We have developed tools for a) designing and implementing MP components, b) specifying composition directives and finally c) composing applications. The Ensemble tools have been developed on top of the most popular MP APIs, PVM [9] and MPICH [11] (implementation of MPI [14]). The composed MP applications are pure PVM or MPI programs, relying only on the APIs themselves and do not use any external environment for process communication. Consequently, Ensemble does not modify the capabilities of MP APIs and does not interfere with application behavior. In this paper we concentrate on MPI applications, but the interested reader may refer to [4] for PVM applications.

Ensemble reduces the software engineering costs incurring in composing application configurations by maintaining a single code for each of the components involved and reusing the components in different application configurations. Applications may either be SPMD or MPMD and either regular or irregular. Components in Ensemble are developed separately as independent MP programs specifying point-to-point or collective communication abstractly (like "formal communication parameters"). A processes spawned from a modular component requires actual point-to-point and collective communication directives (like "actual communication parameters"). Applications are composed by spawning modular processes specifying appropriate communication directives for each process. Modular processes may communicate with other modular processes, not necessarily spawned from the same component. In Ensemble, irregular MPMD applications are naturally supported and regular SPMD applications are just a special case.

The structure of the paper is as follows: In section 2, we discuss the software engineering costs when applications are composed as SPMD programs and review other compositional approaches. In section 3, we present the principles of Ensemble, namely the modular components and their composition. In section 4, we demonstrate the principles on a familiar, yet simplified example, the atmospheric, ocean and climate models: we present the composition requirements of the models and outline the implementation of atmospheric and ocean components. In section 5, we present the composition of applications. In section 6 we discuss some MPI issues, which are not covered in this paper; Ensemble under MPICH devices and performance issues. Finally in section 7, we present our conclusions and future work.

2. Software engineering costs of direct MPI implementations

The mainstream practice for "composing" applications is to construct an SPMD application according to the required configuration. For example, if atmospheric (atm) and ocean (ocn) processes were to be coupled together, their code would be rewritten as procedures in a new program. The role of each process spawned (atm or ocn) would be determined by its rank using a statement like `if(MyRank<=LastAtm)then atm else ocn;` in the main program, specifying that processes with ranks 0 to LastAtm behave like atms and the rest like ocns.

This method works well only if atm and ocn topologies were regular and there were always a direct mapping between atm and ocn processes, as process communication needs to be explicitly coded. An MP program designer and implementer has to code in a program P the symmetric interactions of all processes, which will be spawned from the executable of P, in all possible positions in the topology and for any size of the topology. Message passing communication requires process identifiers, which are specified either directly (as process ranks) or indirectly by functions (determining ranks), which presuppose a specific (usually regular) topology. If the topology is not regular, but only partially regular, extensive code modifications are needed within atm, ocn and land code for each configuration.

Other composition approaches aiming to simplify the composition of high performance applications have been proposed by defining extensions to existing languages or by defining composition frameworks using OO techniques.

In the first category are Fortran M [7] and Compositional C++ [2], which provide extensions to Fortran and C++, respectively. In Fortran M there is explicit support of tasks and channels. In CC++ there is explicit control over locality, concurrency, communication and mapping, and can be used to implement tasks and channels.

In the second category, the composition frameworks, is the Component Composition Architecture (CCA) [17], which deals with the broader issue of composition of heterogeneous components. The CCA framework manages "componentized" programs as well as their communication. It does not support MPI, but aims to define a new high performance framework. Charisma [1] combines message passing and shared memory applications; MPI programs are componentized in Charm++ [1] and their communication is managed within the framework.

We aim to be as close as possible to MPI both syntactically and semantically, so that we may maintain all its advantages (e.g. wide support, grid aware implementations), use existing tools (libraries, analysis, visualization, etc.) and reuse existing applications. MPI has become the de-facto standard for high performance applications and it seems that it will also be the standard for grid enabled applications and environments.

3. The Structure of Modular Components and their Composition

Ensemble components look like MPI programs, except that all envelope data related to the sender and the receiver of messages in MPI calls (i.e., communicators, ranks, message tags and roots) are specified as "formal communication parameters". The "actual communication

parameters" corresponding to the "formal" ones will be passed to each process individually in command line arguments (CLA) dynamically. The rational is that if processes are to be coupled together in various configurations (e.g. atm and ocn, atm and land, ocn and land) they should not have any specific envelope data built into their code, but rather provide this data individually to each process as it is spawned.

One possibility is via their CLA. The envelope data in MPI calls are bound to communicator, ranks (roots are ranks) and message tag types; 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, according to the MPI standard, we cannot assume the rank of a process before it is spawned. But there are indirect ways of passing them.

For communicators we pass an integer indicating the color, which may be used to split MPI_COMM_WORLD constructing the communicator. For ranks the solution we adopted is to associate each process with a unique integer, named Unique Ensemble Rank (UER). The UER of each process is passed in its CLA. We use UERs in the "actual communication parameters" in CLA. For example, the pair (3,4) in CLA is interpreted in an MPI_Send call as sending a message to the process associated with UER 3 using message tag 4. Processes interpret UERs by constructing communicator Ensemble_Comm_World, in which UERs are the same as MPI ranks (by "splitting" MPI_COMM_WORLD with a common color and key=UER); to find process ranks in other constructed communicators we use MPI_Group_translate_ranks and its UER (or rank) in of Ensemble_Comm_World. Each process requires, in principle, its own CLA arguments, a requirement directly supported in `globus2` [10,11] and RSL scripts. Having shown the feasibility of passing directly or indirectly envelope data via CLA we may outline the structure of Ensemble components and their composition, as depicted in figure 1.

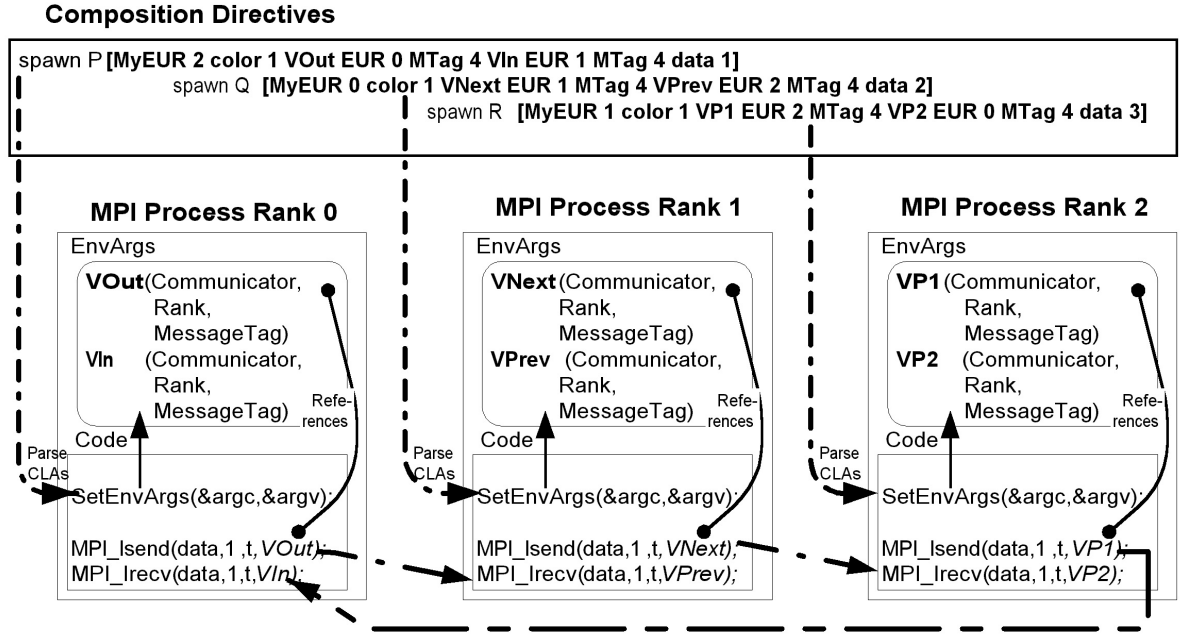

Fig. 1. From CLA to MPI bindings

Three process are spawned from executables P, Q and R, respectively, each having one input and one output port. Their separate CLAs, enclosed in the square brackets, specify that the three processes in the MPMD application are connected in a ring. The CLAs are comprised of the UER of each process (2, 0 and 1 respectively), the color (1) for constructing the common communicator and envelope data for connecting their input and output ports. For process with UER=2 its output port, named VOut is connected with UER=0 with Message Tag=4; its input port, named VIn is connected with UER=1 with Massage Tag=4. Similarly for processes

identified with UER=0 and 1. Note that the virtual names identifying input and output ports in P, Q and R are distinct. The CLAs are parsed by routine SetEnvArgs, which constructs Ensemble_Comm_World, translates UERs to ranks and stores MPI envelope data in structure EnvArgs. In components' code all MPI envelope data refer to EnvArgs ports, which at compile time did not have any values Thus process topology is not built into the code, but is specified externally in the composition directives (process spawning and separate CLAs). The three executables P, Q and R may be reused without any change to built other applications by other composition directives

The code resembles the task/channel model [6], but in a two stage manner. Stage one is within component code (task to port) and stage two is specified in the composition (port-to-port, communicator and root binding). In a way we have extended the task/channel model to deal with communicators and collective communications.

Each component requires its own EnvArgs structure and SetEnvArgs routine. However, for programming convenience we have defined a generic EnvArgs (which is parametrically shaped for each component and possibly each process) and a universal SetEnvArgs routine. For symbolic "`printf`" debugging we also pass in CLA and store in EnvArgs symbolic names for processes and actual communicators. A process is symbolically identified by `ProcessId`.

```
ProcessId ProcessNames;    /* Process's Ids */
typedef struct     /* Symbolic and UER Ids of processes*/;
{ char *ProcessName        /* D- Symbolic Name (SN) of Process */
  int UER;                 /* D- its Unique Ensemble Rank (UER) */
}ProcessId;
```

The values of all symbolic names (strings), array bounds (integers) and message tags are taken directly from CLA, but the rest are computed (e.g. using color to construct by splitting ActualComm), indicated by a D or C, respectively in comments. In EnvArgs we keep for each context (struct Context) envelope data for point-to-point and reduce operations (roots) and commmunicators (e.g. ActualComm in IntraComm). There are also fields related to symbolic names and array bounds (e.g. NrRoots).

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). Point-to-point communication between processes is encapsulated in ports, by specifying the same message tag and each the rank of the other.

```
Context EnvArgs[NrContexts];  /* Contexts and Envelope Arguments */
typedef struct     /* Communicator Identification */
{ char* SymbolicName;          /* D - Symbolic Name of Communicator*/
  MPI_Comm ActualComm;         /* C - The constructed communicator */
}CommunicatorId;
typedef struct     /* Envelope and Size Arguments in a single context */
{ CommunicatorId IntraComm;    /* C - the default intracommunicator */
  MultiPortType MultiPorts[NrMultiPorts]; /* List of MultiPorts */
  RootType Roots[NrRoots];                /* List of Roots */
  int SizeParameters[NrSizePars];         /* List of Size Parameters */
}Context;
```

Communicating processes must be in the same communicator, whether intra or inter. Each multiport may specify a separate inter or even 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 simply refer to ports. The type of the actual communicator (intra, inter or topology) would be specified in the composition directives and set by SetEnvArgs together with all other data port according to their semantics (e.g. if inter then rank will indicate the rank of a process in the remote communicator). Separate communicators in each multiport satisfy code generality for point-to-point communication.

The communicator for any collective communication, as well as, the default communicator for point-to-point communication is the intracommunicator IntraComm in Context.

```
typedef struct     /* A MultiPort*/
{ CommunicatorId AnyTypeComm; /* D - possible separate communication*/
  PortType Ports[NrPorts];    /* List of Ports */
}MultiPortType;
```

```
typedef struct     /* A single Port */
{ ProcessId CommProcNames;    /* D - Names of Source or Dest process */
  int Rank;                   /* C - MPI Rank of Source or Dest Process */
  int MessageTag;             /* D -the message tag of the communication */
}PortType;
```

Finally, we define the roots for reduce operations.

```
typedef struct     /* A single Root */
{ ProcessId RootNames;        /* the symbolic names of the root process*/
  int Root;                   /* C - the Rank of the root */
}RootType;
```

We have provided two abstractions, which do not contribute to any composition functionality, but reduce software engineering effort. The first is that envelope data in MPI calls refer to EnvArgs fields (communicators, roots and ports) indirectly using macros and virtual envelope names. Direct MPI bindings to the actual fields of EnvArgs are generated by expanding macros. The second abstraction is the symbolic high-level composition, eliminating tedious and error prone low level composition directives (as used in this section for presenting the basic Ensemble principles). We will present the use of macros and the high level composition on a simplified classical atmosphere/ocean model. In the next section we outline the atmosphere and ocean Ensemble components and in section 5 the high level composition of a climate model.

4. Simplified atmospheric, ocean and climate models

We set the following requirements for the simplified Atmospheric (atm), Ocean (ocn) and Climate Models.

1. The atm and ocn models may be run on their own. The two models may be also coupled with each other or even with others (e.g. Land, Hydrology). Any coupling configuration should be possible according to actual geography (over ocean or land).
2. In the atm the data is just one three-dimensional array A(na,ma,la).
   a. Process Topology is a regular two-dimensional mesh of processes obtained from the decomposition of array A in the x, y dimensions (no reason for irregular topology, as there are no natural boundaries in atmosphere).
   b. Processes exchange halo rows and columns with N, S and E, W neighbors respectively; they compute new values; repeat until convergence.
3. In the ocn model the data is also one three-dimensional array S(ns,ms,ls) in 2-dimensional domain decomposition in x, y dimensions.
   a. Process Topology is a two-dimensional (possibly ragged) mesh, depending on geography (same is true for Land model; Ocean and Land processes may be interlaced, according to the modeled earth surface). Each process operates on a rectangular partition of array S in the x and y dimensions.
   b. Processes exchange halo rows and columns with N, S and E, W neighbors respectively; they compute new values; repeat until convergence.
4. In case atm and ocn are coupled together
   a. The x, y planes of arrays A (lowest plane of atmosphere) and S (highest plane of ocean) are exchanged replacing corresponding values in A and S.
   b. Depending on geography some atm processes may not be coupled with ocn processes.

c. One atm process may be coupled with a number of Ocn processes for load balancing, as atm computations are more demanding than ocn computations. The actual number of ocn processes is not fixed, as it depends on a number of parameters.
d. The simulation stops when both models converge. Atm and ocn processes calculate their own condition for convergence and communicate the result.

Figure 2 depicts a possible design configuration of coupling atm and ocn processes together. In the two planes we depict the two distinct SPMD applications. Only Atm processes in the eastern part of the atm-plane are coupled with ocn processes, under the assumption that the western region does not correspond to ocean, but land. We also note that in the region where atm and ocn processes are coupled there is a one to six correspondence: one ocn corresponds to six atm processes. The rational is to maintain load balancing, as the atm computations are more demanding than the ocn ones. The optimal domain decomposition of the atm model may not in general be the same as that of the ocn model. Such configurations may be programmed directly in MPI, but as we will outline in the next section each requires code modification. Here we have simplified the problem, as in general the optimal coupling may not be an Mx1 problem, but an MxN [18].

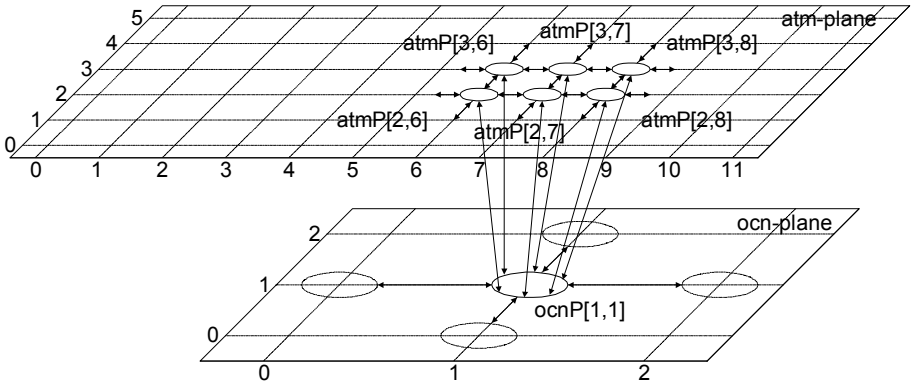

Fig. 2. A configuration for the coupled climate model

We next outline two components atm and ocn, each solving a finite difference problem, assumed to be simplified versions of atm and the ocn models respectively.

4.1 The Ensemble atm component

An Ensemble component has two parts, a virtual envelope and code (table 1). The virtual envelope of atm requires two contexts. The first is Atm for processes involved in the atm calculations. As the process topology is always a regular mesh, we may not specify Atm ports explicitly, but use functions to determine N, S, E and W neighbors, as in mainstream SPMD programming (in the code of table 2 they are not precise, as boundary positions are not checked). In this case we need to specify the size of the regular mesh MxN (size parameters).

In the code all envelope data in MPI calls refer to the virtual envelope by macros. All other arguments have the usual bindings. In the send calls of atm we use the macro EnvRank(Atm) to refer to the process rank. This would mean of course that processes in the Atm context would always execute as a regular SPMD application (cf. ocn component in the next section). The component atm may be viewed as an example of a "legacy" SPMD program transformed into an Ensemble component. Macros are shown in boxed italics.

Within a virtual context, roots for reductions and ports for point-to-point interactions may be defined. An Atm process in context X may communicate, if X is not NULL, with some other process (e.g. ocn or land) via its single port Down[1]. The actual context and communication channel will depend on the application configuration.

Following the virtual envelope we specify application arguments needed in the calculations. In the case of Atm component we specify the convergence threshold and two I/O files. We note that for the coupled program not to deadlock, it is not sufficient to pass the same threshold value to all atm and ocn processes, as their convergence speed may vary.

Table 2. The Virtual Envelope and Code of Atm

```
Virtual Envelope
 Context Atm;
   Size Parameters N, M;
 Context X;
   Ports Down [1..1];
 Application Arguments Threshold; InputFile; OutputFile
```

```
Code  /* Declarations omitted */
MPI_Init(&argc, &argv);
SetEnvArgs(&argc, &argv);

Done=0
while (!Done) {
 MPI_Isend(NRowData,n,MPI_Float,ENVRank(Atm)+N,1,EnvComm(Atm),&SndReq[0]);
     /* similar Isends to other three neighbors */
 MPI_Irecv(NRowData,n,MPI_Float,ENVRank(Atm)+N,1,EnvComm(Atm),&RcvReq[0]);
     /* similar Irecvs from other three neighbors */
 MPI_Waitall(4,&RcvReq,&RcvStatus);

 AtmComputations(&LocalError);
 MPI_AllReduce(&MaxError,&LocalError,1,MPI_Float,MPI_MAX,ENVComm(Atm));
 if (MaxError < threshold) Done=1;

   /* Possible interactions with X (e.g. Ocean, Land) */
 if (ENVComm(X) != MPI_NULL){
  MPI_Send(&Done,      1, MPI_INT, ENVport(Down,1,X));
  MPI_Recv(&OtherDone, 1, MPI_INT, ENVport(Down,1,X), &st);

  Done = Done && OtherDone;
  if (!Done){
   MPI_ISend(BottomData, L, MPI_FLOAT, ENVport(Down,1,X), &SendDown);
   MPI_IRecv(BottomData, L, MPI_FLOAT, ENVport(Down,1,X), &RecvDown);
   MPI_Wait(&RecvDown, &RecvDownStatus);
   MPI_Wait(&SendDown, &SendDownStatus);
  };/*end if not all Done */
 };/* End of Interactions with X */
 MPI_Waitall(4, &SndReq, &SndStatus);
};/* while not Done */
```

The macros in MPI envelope arguments generate proper MPI bindings. All envelope arguments of point-to-point communication (rank, message tag, communicator) refer to virtual envelope ports by ENVPort macro. For example, ENVPort (Down,1,X) refers to port 1 of multiport Down in communicator X and expands to

```
EnvArgs[2].MultiPort[0].Port[1].Rank,
EnvArgs[2].MultiPort[0].Port[1].MessageTag,
EnvArgs[2].MultiPort[0].AnyTypeComm.ActualComm
```

as context X is stored in the second element of EnvArgs (following Ensemble_Comm_World and atm) and Down is its first multiport. Macro ENVComm(Atm) refers to the actual intra communicator corresponding to virtual context Atm. Other macros (not used in atm or ocn) refer to roots ENVRoot(Vcomm,Vroot) and to the communicator of a specific Multiport ENVComm(Vcomm,Multiport).

4.2 The Ensemble ocn component

The ocn component (table 2) also requires two contexts, Ocn for ocn calculations and Y for possible coupling with atm processes. In the Ocn context there are four multiports. As the process topology of ocn is not necessarily regular, we leave the N, S, E and W neighbors unspecified (cf. atm component). Each multiport may have none or one port depending on its position. Any topology may be constructed by appropriate port bindings. In context Y of Ocn the multiport Up may have up to N ports, the number of corresponding Atm processes. Macro ENVportN(Up,Y) refers to the number of ports in Multiport Up. Finally, we specify application arguments convergence threshold and the files for I/O, as for the atm component.

Table 2. The Virtual Envelope and Code of Ocn

```
Virtual Envelope
 Context Ocn;
   Ports North[0..1]; South[0..1]; East[0..1];West[0..1];
 Context Y;
   Size Parameters RowMap, ColMap;
   Ports Up[1..RowMap*ColMap];
Application Arguments Threshold; InputFile; OutputFile;
```

```
Code  /* Declarations omitted */
MPI_Init(&argc, &argv);
SetEnvArgs(&argc, &argv);
Done=0;
while (!Done) {
 /*Internal Ocean Communications */
 MPI_ISend(NorthData, n, MPI_Float, ENVPort(North,1,Ocn), &SndReq[0]);
 MPI_ISend(SouthData, n, MPI_Float, ENVPort(South,1,Ocn), &SndReq[1]);
 MPI_ISend(EastData,  m, MPI_Float, ENVPort(East,1,Ocn),  &SndReq[2]);
 MPI_ISend(WestData,  m, MPI_Float, ENVPort(West,1,Ocn),  &SndReq[3]);
 MPI_IRecv(NorthData, n, MPI_Float, ENVPort(North,1,Ocn), &RcvReq[0]);
 MPI_IRecv(SouthData, n, MPI_Float, ENVPort(South,1,Ocn), &RcvReq[1]);
 MPI_IRecv(EastData,  m, MPI_Float, ENVPort(East,1,Ocn),  &RcvReq[2]);
 MPI_IRecv(WestData,  m, MPI_Float, ENVPort(West,1,Ocn),  &RcvReq[3]);
 MPI_Waitall(4, &RcvReq, &RcvStatus);

 OcnComputations(&LocalError);
 MPI_AllReduce(&MaxError,&LocalError,1,MPI_Float,MPI_MAX,ENVComm(Ocn));
 if (MaxError < threshold) Done=1;

/* Possible interactions with Y (e.g.Atm) */
 if (ENVComm(Y)!=MPI_NULL){
  for (i=1; i < ENVportN(Up,Y); i++){
   MPI_Send(&Done,     1,MPI_INT,ENVport(Up,i,Y));
   MPI_Recv(&OtherDone,1,MPI_INT,ENVport(Up,i,Y),&st);
   Done = Done && OtherDone;
  };
  if (!Done){
   for (i=1; i < ENVportN(Up,Y); i++){
    MPI_ISend(Top[i], L, MPI_FLOAT, ENVport(Up,i,Y), &UpSend[i-1]);
    MPI_IRecv(Top[i], L, MPI_FLOAT, ENVport(Up,i,Y), &UpRecv[i-1);
   }; /* end for all Up ports*/
   MPI_Waitall(ENVportN(Up,Y), &UpSend, &UpSendStatus);
   MPI_Waitall(ENVportN(Up,Y), &UpRecv, &UpRecvStatus);
  }; /* end if not all Done*/
 };/* End of Interactions with Y */
 MPI_Waitall(4,&SndReq,&SndStatus);
};/* while not done */
```

5. The Composition of Applications

The composition of applications is specified in three levels (figure 3), 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 grouped into symbolic communicators. In each communicator point-to-point communication channels between process ports are defined. For reduction operations roots are associated with symbolic process names. RHLC specifies reconfigurable applications, in two aspects: a) different configurations may be obtained, depending on the values and the generality of design parameters, and b) applications at this level are independent of any execution environment resources (machines, files, etc).

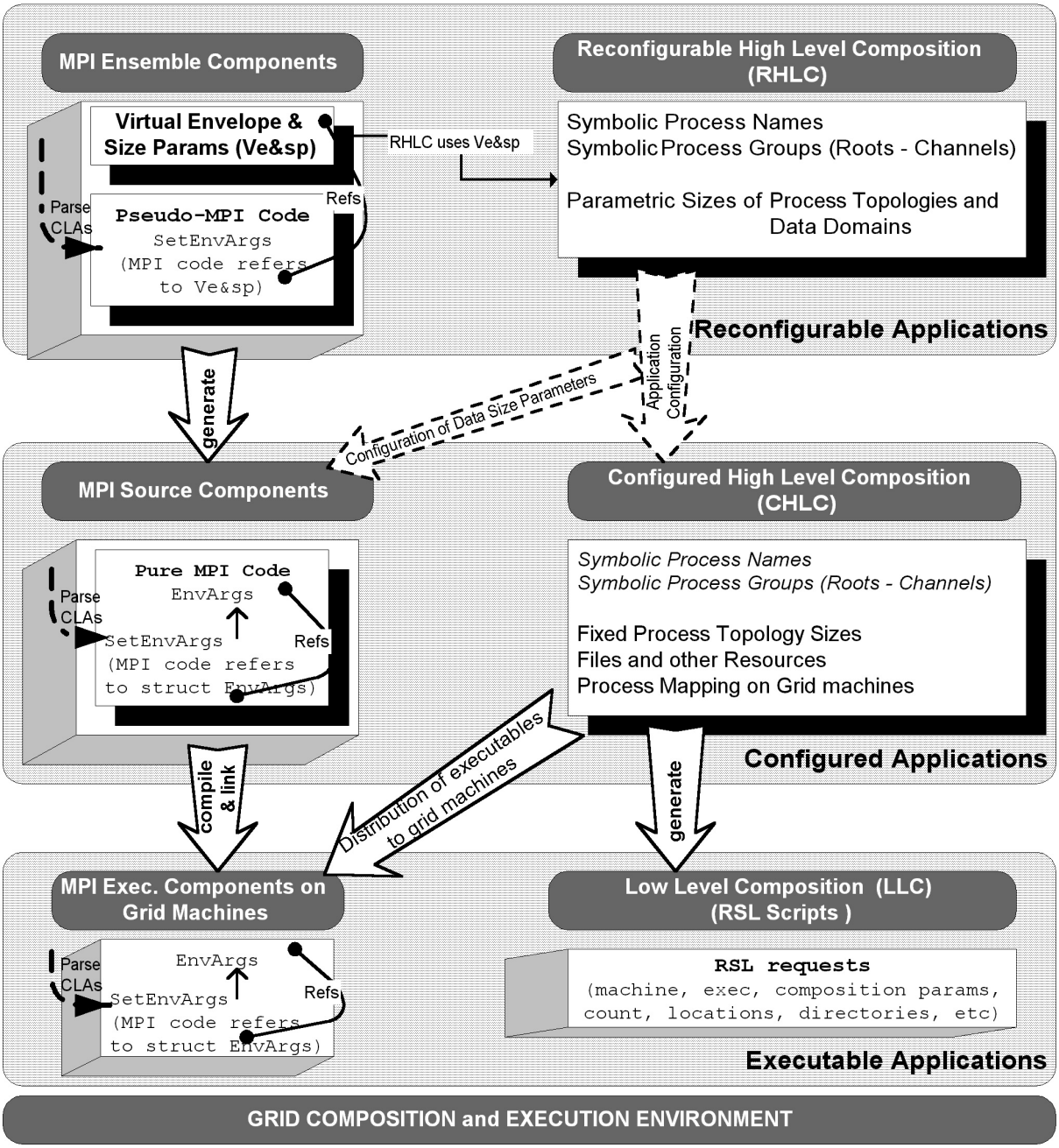

Figure 3: An Overview of Ensemble

The RHLC is then configured by setting values to parameters obtaining the Configured High Level Composition (CHLC). Some parameters may determine sizes of data domains in component code. In CHLC execution resources are also specified. The actual configuration

decisions are outside the scope of Ensemble, which may be based on experience or obtained from external tools or environments, such as GradSoft [13]. Configuration information may be used to distribute executables to grid machines. Low Level Composition (LLC) directives are generated from the CHLC, which are `globus2` RSL scripts, encapsulating the actual application composition in MPICH-G2, as outlined in section 3.

We demonstrate the three composition levels of a climate model. For reasons of simplicity we have chosen a regular mesh topology for ocn processes, leaving a number of other issues general. In the Parameter section of RHLC (table 3), integer identifiers may be declared, which parameterize the design: AtmRows, AtmCols and OcnRows, OcnCols respectively, determine the sizes of AtmP and OcnP process meshes. Some AtmP processes are coupled with OcnP ones. As both atms and ocn form meshes we only need to specify that OcnP(0,0) is coupled with AtmP(FirstCouplRow,FirstCouplCol). Finally, we allow for smaller meshes (RowStep by ColStep) of AtmP processes to be coupled with a single OcnP process.

Table 3. A Reconfigurable High Level Composition of a Climate model

```
Parameter
  AtmRows, AtmCols, OcnRows, OcnCols, FirstCouplRow, FirstCouplCol,
  RowStep, ColStep;
IntraComm ENSEMBLE_COMM_WORLD;
  component Atm: processes AtmP(AtmRows,AtmCols);
  component Ocn: processes OcnP(OcnRows,OcnCols);
IntraComm AtmPlane;
  processes AtmP(*,*)in Atm{N is AtmRows; M is AtmCols};
IntraComm OcnPlane;
  processes OcnP(*,*) in Ocn;
  channels
  #i:0(1)OcnRows
   [#j:0(1)OcnCols
     [ (i<OcnRows):OcnP(i,j).Ocn.South[1] <-> OcnP(i+1,j).Ocn.North[1];
       (j<OcnCols):OcnP(i,j).Ocn.East[1]  <-> OcnP(i,j+1).Ocn.West[1];];
IntraComm AtmX;
  processes AtmP(FirstCouplRow..AtmRows,FirstCouplCol..AtmCols)in X;
IntraComm OcnY;
  processes OcnP(*,*) in Y{RowMap is RowStep; ColMap is ColStep};
InterComm Coupling;
  IntraComm1 AtmX; MultiPorts Down; leader AtmP(2,6);
  IntraComm2 OcnY; MultiPorts Up;   leader OcnP(0,0);
  channels
  #i:0(1)AtmRows-FirstCouplRow
   [#j:0(1)AtmCols-FirstCouplCol
     [((i/RowStep) <= OcnRows)and((j/ColStep) <= OcnCols):
      AtmP(i+FirstCouplRow,j+FirstCouplCol).X.Down[1] <->
      OcnP(i/RowStep,j/ColStep).Y.Up[(i%RowStep*ColStep)+j%ColStep+1]]
```

Next the processes in the application (in Ensemble_Comm_World) are specified as instances of components atm and ocn, respectively, by symbolic names, e.g. AtmP(,) and OcnP(,). Processes are then grouped into communicators (inter or intra communicators). The intra-communicators AtmPlane and OcnPlane consist of entirely AtmP and OcnP processes respectively, in their atm and ocn contexts, respectively. In AtmPlane, N and M parameters correspond to AtmRows and AtmCols respectively; no other specification is required, as the communication is already built into the atm code. This is not the case with OcnPlane, where channels need to be explicitly specified. For each OcnP process its connection of the North, South, East, and West ports is specified by channel-patterns. A channel-pattern is a guarded point-to-point pattern of the form

`Guard:Process.Context.MultiPort[expr]<->Process.Context.MultiPort[expr].`

Only point-to-point patterns with guards=true are evaluated. `Process` is the symbolic name of a process, `Context` is the symbolic context within which communication takes place and `Mul-`

`tiPort[index]` is a communication port. A channel-pattern may also be a for-structure controlling one or more channel-patterns `#index:from(step)to[{channel-pattern[expr]}+]`

Some AtmP and all OcnP processes participate in the coupling context; atm processes in their X virtual context and ocn processes in their Y virtual context. There are two design options: specify their communication either within a intracommunicator, or by an intercommunicator between two disjoined intracommunicators. In this composition example we demonstrate the latter. Two intracommunicators AtmX and OcnY, are specified for AtmP and for OcnP processes, respectively, from which the intercommunicator Coupling is specified. Coupling is based on intracommunicators AtmX and OcnY and involves Multiports Down and Up, respectively; leader processes are explicitly specified and implicitly the bridge communicator is Ensemble_Comm_World. The channels in Coupling define connections of AtmP with OcnP ports. The complex channel pattern specifies coupling of the AtmP meshes (RowStep by ColStep) with a single OcnP. The guard ensures OcnP processes are defined.

To obtain the climate CHLC of figure 2, we set the constants AtmRows=5, AtmCols=11, OcnRows=2, OcnCols=2, FirstCouplRow=3, FirstCouplCol=0, RowStep=3 and ColStep=2. From CHLC the LLC is generated. For each process an RSL request is generated passing its composition directives as CLA (argv). The application is composed by executing the RSL scripts; SetEnvArgs sets envelope data dynamically for each process (as in section 3). The structure of CLA is the following:

- The Unique Ensemble Rank (UER) and Symbolic Name (SN) of the process
- The number of splits of Ensemble_Comm_World. For each split:
- its color. If color >= 0 (process belongs in the intracommunicator):
    - SN of communicator and its index in EnvArgs
    - Number of multiports; for each multipart
        - Its index and indication for Intra or Inter
        - If Inter: its SN; local intra; UER and SN of local and remote leaders
        - Number of ports; for each port: UER and its SN; Message Tag
    - Number of Roots; for each root: UER and its SN
    - Number of size parameters; for each parameter an integer value
- Application arguments

For an actual example of a command-line argument list let us consider process AtmP[2,7] in the configuration of figure 2.

```
31 AtmP[2,7]     /UER (generated from CHLC) and Symbolic Name (SN)
2                /Two Splits of ENSEMBLE_COMM_WORLD
1 AtmPlane 1     /1st Split: Color, Symbolic Name, index in EnvArgs (atm)
0 0 2               /No Multiports, no Roots, two Parameters
11 5                /Values for two parameters N and M
3 AtmX 2         /2nd Split: Color, Symbolic Name, index in EnvArgs (X)
1 0 0               /One Multiport, no Roots, no Size Parameters
0 Inter             /1st Multiport is the first and an InterCommunicator
Coupling AtmX 2          /SN; local group: SN and index in EnvArgs
30 AtmP[2,6]             /UER and SN of local leader
73 OcnP[0,0]             /UER and SN of remote leader
1                        /One Port
77 OcnP[1,1] 99               /1st port: Process UER, Process SN, tag
0.01 InF OutF    /Application args: Threshold, InputFile, OutFile
```

We assume that all communicators in EnvArgs are initialized to MPI_COMM_NULL and all ranks (ports and roots) to MPI_PROC_NULL (this assumption is implemented in SetEnvArgs). If a communicator of a process remains MPI_COMM_NULL it indicates that it is not coupled with any process. An uncoupled process, e.g. atmP(0,0), cannot (and does not need to) determine if this is because the whole atm model runs independently or because this process is not coupled.

All processes participate in the splits, but not all construct communicators (e.g. the second split of AtmP(i,j) for i<FirstCouplRow and j<FirstCouplCol), indicated in their CLA with color=-1. Processes constructing the same actual communicator will be given the same color whether spawned from the same executable (SMPD communicator) or from different ones (MPMD communicator). In the former case, all processes refer to the communicator by their common virtual context (e.g. atm for AtmP processes in context AtmPlane). In the latter, only processes spawned from the same component will refer to the communicator through the same virtual context (e.g. in context Coupling for AtmP is X, but for OcnP is Y). Processes spawned from the same executable may also be grouped into different communicators; for example each ocn process together with its communicating atm processes. In this case, the common virtual context (e.g. Y of OcnP) will refer to different actual communicators.

6. Performance, MPICH devices and MPI support of Ensemble

In the special case where processes are spawned from the same component and allocated on the same parallel machine or cluster (where a local scheduler exists) only one RSL subjob may be generated. This not only a matter of convenience, but of improving performance, as communication within a subjob may use an efficient vendor-supplied MPI and not facilitated over TCP/IP (consequently, MPMD applications in MPICH-G2 cannot avoid using TCP/IP communication). We set the appropriate number for `(count= )` and instead of passing directly CLAs, they are put in a file, one line per CLA set. We provide in argv a color and the filename with the CLAs. Processes construct a temporary communicator by the color given, use their temporary rank to read their "CLA" from the Rank-th line of the CLA file and free the temporary communicator.

In section 3, we required that Unique Ensemble Ranks are passed separately to processes, and for this purpose we relied on `globus2` and RSL scripts, which support separate argument lists. Using an MPICH device such as `ch_p4` and the –p4pg option, atmP and ocnP processes may certainly be spawned, but they all get the same arguments. For Ensemble to be used on such devices we have introduced a simple technique: First, we identify processes spawned from the same component (not necessarily from the same executable); for this we declare within each source code a unique constant identifier (e.g. "atm"). Then, pass as common CLAs to all processes, a list of source identifiers involved in the application. Processes parse the list, find their identifier and record its position in the list. Processes are then grouped into temporary separate SPMD communicators by using the position as the split color. Finally, processes use their temporary rank to read the Ensemble CLAs in the Rank-th line of a file.

Performance is the essence of parallel programming. Structure EnvArgs is designed with performance considerations in mind. Although the number of contexts in each program and the number of MultiPorts and Roots in each context are in general different, we have used arrays bounded by the highest values of Roots, MultiPorts and Ports over all contexts. The alternative were a dynamic structure, not "wasting" memory space, but each communication call would need one or two indirect memory accesses, reducing performance. Using arrays all envelope-related data are bound at compile time. Anyway, the "wasted" space is insignificant. The actual number of MultiPorts and Roots in each context and the actual number of Ports in each MultiPort are stored in each context.

We compared performance of Ensemble implementations with equivalent SPMD programs and, as expected no execution variations were observed. We run tests on a modest Linux cluster of 8 nodes connected with fast Ethernet on MPICH ch_p4 device. Test programs involved point-to-point and collective (data movement and reduction) operations.

The only possible cause of overhead is in SetEnvArgs, which is executed once anyway. Apart from the cost of reading CLAs from files (in the special cases described above), the most significant overhead introduced is the construction of Ensemble_Comm_World, that is a

single collective call of MPI_Comm_split of all processes in MPI_COMM_WORLD. All other operations in SetEnvArgs are either local and scalable (e.g. MPI_Comm_group and MPI_Group_translate_ranks), or would have been computed anyway in the classical SPMD approach (e.g. communicator splits). Tests confirmed the above analysis. We compared execution of SetEnvArgs with the initialization part of classical SPMD programs (communication construction, getting communication size and process rank). We performed two kinds of scalability tests on SetEnvArgs. In the first the number of required splits remained constant and the number of processes in the groups was increased. In the second the number of splits was increased. As expected only a small overhead was observed, due to the extra split.

Space limitations we did not permit us to cover two MPI features. The first is supporting message tags. In this paper we assumed that tags are generated, but this technique is not sufficient for the use of pairs (MPI_ANY_SOURCE, Some-Specific-Tag) in receive calls. Tags cannot safely be built into the code in our composition environment. For this purpose we define tag identifiers in EnvArgs and give them values in composition directives. To receive from a specific process with MPI_ANY_TAG, we have extended the concept of root in EnvArgs to that of special processes. No special support is required, when both wild cards are used.

The second MPI topic, which we did not cover in this paper, is process topologies. Ensemble specifies topologies at RHLC, which have to be associated with MPI topologies. We note here that this is only beneficial for efficient mapping purposes in MPP systems, as Ensemble deals with convenient naming of processes much more generally than MPI itself. The principle for associating Ensemble topologies to MPI is to give processes UERs in an order consistent with Cartesian (e.g. row major) and Graph creation; then split communicators using as key the UERs to maintain the relative order. In RHLC Graph topologies are described by channel patterns, setting the arc order in terms of ports. For Cartesian topologies we do not use channels, but associate ports with neighbors using dimension and periods, as used in MPI_Create_Cart.

7. Conclusions and Future Work

Ensemble methodology and its tools enable the composition of genuine MPMD applications, where *P* indicates "*Source Programs*" and not "*Program Execution Behaviors*", maintaining a single code for each component. Ensemble reduces software engineering costs as components 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 `globus2` or any other device supporting spawning processes from several executables. Consequently, any tool, e.g. for analysis, visualization, performance, or library for MPI programs may be used.

Ensemble also reduces software engineering costs for developing single components, as the programmer need only be concerned with data movement (collective or point-to-point), reduction and synchronization. Furthermore, code is neutral to the communicator type (intra, inter, topology) for point-to-point communication, determined by the composition directives. Consequently, it may be advantageous to use Ensemble even for SPMD applications.

The Virtual Envelope of components provides explicit description of process communication and is a basis for specifying communication semantics and determining component compatibility, a fundamental property for correct program behavior. Compatibility cannot be determined solely on virtual envelopes, as it is a dynamic property, not restricted to the static and local compatibility of channel binding, message types, etc. We are currently developing an environment to test and debug components and composed applications based on formal component specifications, which are used in synergy with program tracing [16].

The language for Reconfigurable High Level Composition is still under development. We have experimented with a number of languages and environments in the past (grammatical, graphical, GUI based) each having its own advantages and disadvantages. Its design has been proven a difficult task. For improving its usage, we try to reflect in HLCL SPMD practice and experience.

Currently, routines for data distribution of coupled MxN processes [18] are under development, dynamically associating data sub-domains to multiports. We also plan to reengineer SPMD applications as components and to integrate Ensemble with functionally complementary environments and tools, such as GradSoft [13], for configuring efficient applications. Ensemble presently supports static process groups as defined in MPI-1 Our long-term plan is supporting design and implementation of dynamic applications [5] under MPI-2.

Acknowledgment. This work has been supported by the Special Account for Research of the University of Athens.

References

[1] Bhandarkar M. A. CHARISMA: A Component Architecture for Parallel Programming, http://www.cs.uiuc.edu/Dienst/UI/2.0/Describe/ncstrl.uiuc_cs/UIUCDCS-R-2002-2274, 2002.
[2] Chandy, K.M. and Kesselman, K. CC++: A declarative concurrent object-oriented programming notation. In Research Directions in Concurrent Object Oriented Programming. MIT Press, 1993.
[3] Cotronis, J.Y. (1996) Efficient Composition and Automatic Initialisation of Arbitrarily Structured PVM Programs, in Proc. of 1st IFIP International Workshop on Parallel and Distributed Software Engineering, pp 74-85, Chapman & Hall.
[4] Cotronis, J.Y., Tsiatsoulis Z., Modular MPI and PVM components, PVM/MPI'02, LNCS 2474, pp. 252-259, Springer.
[5] Darema F. SPMD Model: Past, Present and Future. Invited talk at PVM-MPI 2001.
[6] Foster, I. (1995) Designing and Building Parallel Programs, Addison-Wesley Publishing Company, ISBN 0-201-57594-9.
[7] Foster, I. and Chandy, K.M. Fortran M: A Language for modular parallel programming. J. Parallel and Distributed Computing, 25(1), 1995.
[8] Foster, I. and Kesselman, C (eds.) The Grid, Blueprint for the New Computing Infrastructure, Morgan Kaufmann, 1999.
[9] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User's guide and Reference Manual, ORNL/TM--12187.
[10] Globus Quick Start Guide, Globus Software version 1.1.3 and 1.1.4, February 2001. http://www.globus.org/
[11] Gropp, W. and Lusk, E. Installation and user's guide for mpich, a portable implementation of MPI, ANL-01/x, Argone National Laboratory, 2001.
[12] 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).
[13] Kennedy K., Mazina M., Mellor-Crummey J., Cooper K., Torczon L., Berman F., Chien A., Dail H., Sievert O., Angulo D., Foster I., Gannon D., Johnsson L., Kesselman C., Aydt R., Reed D., Dongarra J., Vadhiyar S., and Wolski R: Toward a Framework for Preparing and Executing Adaptive Grid Programs. April 2002, Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002), Fort Lauderdale, FL.
[14] Message Passing Interface Forum MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4):165-414, 1994.
[15] MPICH-G2, http://www.hpclab.niu.edu/mpi/g2_body.html
[16] Tsiatsoulis Z. and Cotronis J.Y.: Testing and Debugging Message Passing Programs in Synergy with their Specifications, Fundamenta Informaticae 41, No 3 (February 2000) pp. 341-366.
[17] http://www.csm.ornl.gov/cca/
[18] http://www.csm.ornl.gov/cca/mxn/