Building Grid MPI Applications from Modular Components

Yiannis Cotronis

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

Abstract

Coupling grid applications developed by different teams requires code modification and high S/W engineering effort. In the Ensemble methodology message passing components are developed separately as independent modules, and applications, whether regular, irregular, SPMD or MPMD, are composed from these modular components, without any code modification. We demonstrate by developing two downsized atmospheric and ocean components, which may run on their own or coupled together (climate model) in any configuration depending on geography or other design issues.

1 Introduction

In the Ensemble methodology [2,3,4] 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) for specifying composition directives for MP applications and c) for actually composing applications from components.

The Ensemble tools have been developed on top of the most popular MP APIs, PVM [9] and MPI [12] (MPICH [11]). 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 communication. In this paper we concentrate on composing MPI applications, but the interested reader may refer to [5] for composing PVM applications.

The contribution of Ensemble is that it reduces software engineering (SE) costs when compared to implementing applications directly on MPI, except possibly in the case of regular SPMD applications. Of course SPMD is presently the most popular programming style, mainly due to its simplicity. However, Grids [7] 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.

Ensemble aims to reduce the SE costs incurring in composing application configurations and in maintaining a single code for each of the components involved. Applications may either be SPMD or MPMD and either regular or irregular. Components are developed separately as independent MP programs specifying local and global communication abstractly (like "formal communication parameters"). Modular processes spawned from any of the modular components may communicate (by point-to-point or collectively) with other modular processes, not necessarily spawned from the same component. Applications are composed from modular processes specifying for each process its actual local and collective communications (like "actual communication parameters"). 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 present the requirements for downsized atmospheric, ocean and climate models in a Grid environment; in section 3 we discuss the SE costs when applications are directly programmed in MPI; in section 4 we present Ensemble modular MPI components, demonstrating the principles by the downsized atmospheric and ocean models; in 5 we outline the composition of applications; and finally in section 6 we present our conclusions and plans for future work.

2. Downsized atmospheric, ocean and climate models

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

1. The atm and ocn models may be run on their own.
2. 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).
3. 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.
4. 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.
5. 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 (they execute as if model executes on its own).
    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.

Figure 1 depicts a possible design configuration of coupling atm and ocn processes together. In 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.

3. 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 atm and ocn processes were to be coupled together, their code would be rewritten as procedures in a new program, say AtmOcn. The role of each process (atm or ocn) spawned would be determined by its rank by

```
if (MyRank<=LastAtm) then atm else ocn;
```

as the main program, specifying that processes with ranks 0 to LastAtm behave like atms and the rest like ocns.

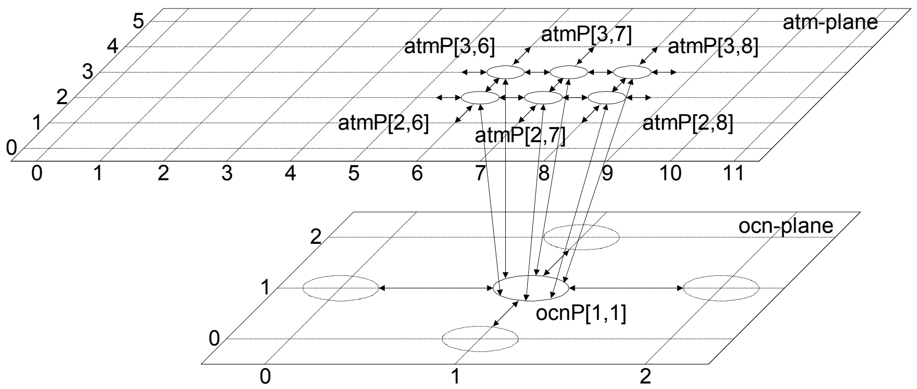

Fig. 1. The coupled climate model

This method works well if the atm and ocn topologies were regular and there were always a direct mapping between atm and ocn processes. The reason is that process communication needs to be explicitly coded. A 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.

4. Modular MPI components

Ensemble components look like MPI programs, but all envelope data related to the origin and the destination of messages in MPI calls (i.e., contexts, ranks, message tags and roots) are specified as “formal communication parameters”. The “actual communication parameters” corresponding to the “formal” ones are passed to each process as command line parameters as they are spawned.

The principle is simple. If processes are to be coupled together in a number of 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 given individually and dynamically to each process as it is spawned. One possibility is via their command line arguments (CLA). The envelope data in MPI calls is bound to a communicator, a rank and a message tag types (roots are ranks). Obviously, 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 its spawning. But there are indirect ways of passing them.

Instead of passing a communicator to a group of processes we may pass an integer indicating the color, which may be used to split MPI_WORLD_COMM and obtain 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 may be interpreted in an MPI_Send call to send a message to the process having UER 3 with message tag 4. Processes may determine associations of UER to MPI Ranks by calling Allgather for each communicator they belong and use MPI ranks thereafter as usual. In our example the UER 3 will be replaced by the associated MPI rank. However, there is a practical problem, as each process must in principle have its own CLA arguments; at least its UER. In MPICH, MPMD applications may be spawned by using a procgroup file, but the same CLA are passed to all processes. This problem is overcome in MPICH-G [8,13] as processes may have their own CLA in RSL [10] scripts.

Having shown the feasibility of passing directly or indirectly envelope data via CLA we may outline the structure of Ensemble Components depicted in figure 2. A process is spawned from executable P with CLA enclosed in square brackets. The CLA are comprised of the UER of the process (2), the color for constructing the communicator (1), and finally the UER (3) and the Message Tag (4) of a communicating process. The CLA are parsed by a routine SetEnvArgs, which performs the necessary operations and stores MPI envelope data in structure EnvArgs. In the code all MPI envelop data refer to structure EnvArgs having the appropriate MPI bindings, but of course at compile time has no values.

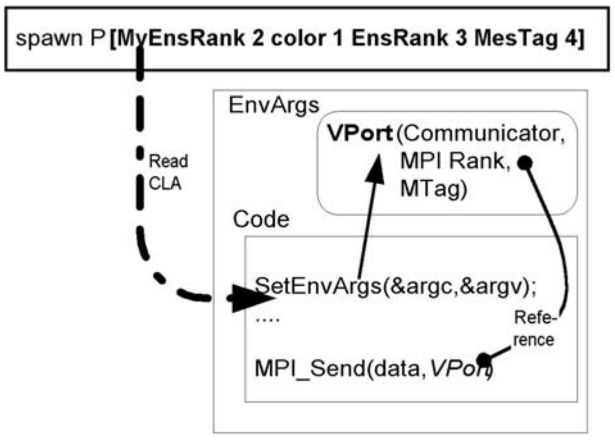
spawn P [MyEnsRank 2 color 1 EnsRank 3 MesTag 4]
Read CLA
EnvArgs
VPort (Communicator, MPI Rank, MTag)
Code
SetEnvArgs(&argc,&argv);
....
Refe-rence
MPI_Send(data, VPort)

Fig. 2. From CLA to MPI bindings

Each component needs its own CLA, EnvArgs structure and SetEnvArgs routine. However, for programming convenience we have defined a generic EnvArgs (which is appropriately shaped for each component) and a universal SetEnvArgs routine. For symbolic tracing and debugging we also pass in CLA and store in EnvArgs symbolic names for processes and contexts. In table 1 we present (top-down) the structures EnvArgs and ProcessNames.

In EnvArgs we keep for each context (struct context) envelope data for point-to-point and reduce operations (roots), which are proper MPI bindings (e.g. ActualComm). There are also fields related to symbolic names of contexts and array bounds (e.g. NrRoots). The values of all symbolic names, array bounds and some of MPI envelope bindings (e.g. message tags) are taken directly from CLA, but the rest are computed (e.g. MyRank and ActualComm), indicated by a D or C in comments.

For point-to-point communication within a context, ports are introduced, which are abstractions of the envelope pair (rank, message tag). Ports with similar semantics are treated as an array of ports (MultiPort), dynamically scaled for individual processes.

Table 1: Declaration of Structures ProcessNames and EnvArgs

```
ProcessId ProcessNames;      /* Process's Ids */
Context EnvArgs[NrContexts]; /* Array for Contexts and Envelope Arguments*/
```

```
typedef struct      /* Symbolic Ids of processes*/;
  char *ProcessName               /* D – Process name, e.g. Atm[3,4] */
  int UER;                        /* D – its Unique Ensemble Rank (UER)*/
}ProcessId;
```

```
typedef struct      /* Envelope Arguments of a single context */
{ char* SymbolicContext;          /* D - The symbolic group name, e.g. atm-plane */
  MPI_Comm ActualComm;            /* C - The constructed communicator /
  int* TopologyParameters;        /* D – size of topology */
  int MyRank;                     /* C - My rank in this constructed communicator */
  int NrMultiports;               /* D – actual number of MultiPorts */
  MultiPortType MultiPorts[GlNrMultiPorts+1];  /* List of MultiPorts */
  int NrRoots                     /* D – actual number of Roots */
  RootType Roots[GlNrRoots+1];    /* List of Roots */
}Context;
```

```
typedef struct      /* A MultiPort*/
{ int NrPorts;                    /* D-actual number of ports in a MultiPort */
  PortType Ports[GlNrPorts+1];    /* List of Ports */
}MultiPortType;
```

```
typedef struct      /* A single Port */
{ ProcessId ProcessNames;         /* D – the symbolic names of communicating proc */
  int Rank;                       /* C - MPI Rank*/
  int MessageTag;                 /* D -the message tag of the port */
}PortType;
```

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

Let us comment on EnvArgs. NrContexts is the specific number of contexts a process belongs to. Each context element keeps envelope data for the Communicator and process Rank, as well as for Roots and MultiPorts in two arrays. Although the number of Roots and MultiPorts are in general different in each Context, we have used an array (bounded by the highest value of Roots and MultiPorts, resp. in all contexts) rather than a dynamic structure, which would not "waste" memory space. The reason for not using a dynamic structure is that during execution each communication call would need one or two indirect memory accesses, which would reduce application performance. Using arrays all envelope-related data are bound at compile time. Anyway, the "wasted" space is insignificant. The actual number of Roots and MultiPorts in each Context and the actual number of Ports in each MultiPort are stored in NrRoots, NrMultiPorts and NrPorts respectively.

We now outline two components atm and ocn, each solving a finite difference problem, assumed to be "down-scaled" versions of atm and the ocn models respectively.

4.1 The Ensemble atm component

An Ensemble component has two parts, a virtual component envelope and code, in which all envelope data in MPI calls refer to virtual envelopes by macros. The macros bind envelope data to EnvArgs elements transparently. All other arguments have the usual bindings. Pure MPI code is generated (by expanding the macros) and compiled, as all parameters have proper bindings.

The virtual envelope of atm (table 2) requires two contexts. Within a virtual envelope, roots for reductions and ports for point-to-point interactions are defined. 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 (in the code of table 2 not precise, as boundary positions are not checked) to determine N, S, E and W neighbors, as in mainstream SPMD programming. In this case we need to specify the size of the regular mesh NxM (topology arguments). In the send calls we use the macro EnvRank(Atm) to refer to the process rank. This would mean of course that processes in Atm context will always execute as a regular SPMD application (cf. ocn component in the next section). Macros shown in boxed italics.

An Atm process in context X may also communicate, if X is not NULL, with some other process (e.g. ocn or land) via its single port Down[1]. The actual value of context and port 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 I/O files and the threshold. 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;
     Topology Arguments 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), &SendReq[0]);
 MPI_ISend(SRowData, n, MPI_Float, ENVRank(Atm)-N, 1, EnvComm(Atm), &SendReq[1]);
 MPI_ISend(ERowData, m, MPI_Float, ENVRank(Atm)+1, 1, EnvComm(Atm), &SendReq[2]);
 MPI_ISend(WRowData, m, MPI_Float, ENVRank(Atm)-1, 1, EnvComm(Atm), &SendReq[3]);

 MPI_IRecv(NRowData, n, MPI_Float, ENVRank(Atm)+N, 1, EnvComm(Atm), &RecvReq[0]);
 MPI_IRecv(SRowData, m, MPI_Float, ENVRank(Atm)-N, 1, EnvComm(Atm), &RecvReq[1]);
 MPI_IRecv(ERowData, m, MPI_Float, ENVRank(Atm)+1, 1, EnvComm(Atm), &RecvReq[2]);
 MPI_IRecv(WRowData, m, MPI_Float, ENVRank(Atm)-1, 1, EnvComm(Atm), &RecvReq[3]);
 MPI_Waitall(4,&RecvReq,&RecvStatus);

 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, &SendReq, &SendStatus);

};/* while not Done */
```

Few simple macros in MPI envelope arguments using virtual envelope names generate by expansion proper MPI bindings. All envelop arguments of point-to-point communication (rank, message tag, communicator) refer to virtual envelope ports by macros e.g. ENVPort (Down, 1, X). This macro refers to port 1 of multiport Down within the X context. Macro ENVComm(Atm) refers to the actual communicator corresponding to virtual context Atm. A third macro ENVRoot(Vcomm,Vroot), which is not used here, refers to roots.

Thus all communication is expressed in the code without any actual information about the receiver or the sender of messages. The code resembles the task/channel model [5], but in a two stage manner. Stage one is within component code (task to port) and stage 2 specified in the composition (port-port, context and root binding). In a way we have extended the task/channel model to deal with contexts and collective communications.

4.2 The Ensemble ocn component

The ocn component also requires two contexts, ocn for ocn calculations and Y for possible coupling with corresponding 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 on the plane. Any topology may be constructed by appropriate port bindings. In context Y of Ocn multiport Up may have up to N ports, the number of corresponding Atm processes. ENVportN(Up,Y) refers to the number of ports in Multiport Up.

Finally, we specify application arguments for I/O and the convergence threshold, as for the atm component.

The Ensemble components are transformed to pure MPI code by expanding the macros, which generate appropriate MPI bindings (fields of EnvArgs). For example the macro expression ENVPort (Up, i, Y) expands to

EnvArgs[2].MultiPort[0].Port[i].Rank,
EnvArgs[2].MultiPort[0].Port[i].MessageTag,
EnvArgs[2].ActualComm

as context Y is stored in the second element of EnvArgs (after MPI_WORLD_COMM and ocn) and Up is its first multiport. The expansion complies with MPI bindings.

We have developed another class of macros for printing symbolic names of processes, contexts, roots, etc for aiding debugging. For example it would be possible to print tracing lines such as

```
ocn[1,1] sends to atm[2,6] via port
Up[1] within context of Vertical[3]
```

5. The composition of applications

The composition of applications is specified in two levels: by a High Level Composition Tool (HLCT) in which the designer puts the components together using symbolic names for processes, roots, groups, etc. At this level, we specify the number processes and the scaling of their multiports (possibly parametrically), the contexts they belong in, their point-to-point and collective communication.

The HLCT generates Low Level Composition Directives (LLCD), which are MPICH-G globus RSL scripts. For each process an RSL request is generated having its own CLA (argv), which are composition directives related to the specific process. Executing the RSL scripts the application is composed. SetEnvArgs sets envelope data dynamically for each process. The structure of argv for each process is:

- Its Ensemble Rank (UER) and Symbolic Name (SN)
- The number of splits (NS) the process participates followed by NS groups of data specifying the splits
- For each split its color. If color >= 0 (a real context)
  - SN of context and index of context in EnvArgs
  - Number of multiports and roots
  - For each multiport
    - Number of ports
    - For each port
      - a UER and SN, Message Tag
  - For each root
    - a UER and SN
- Application arguments

For an actual example of RSL scripts refer to [4].

In the special case where processes spawned from the same component are allocated on the same parallel machine or cluster (where a local scheduler exists) only one RSL request may be generated, having as CLA a range of UER. All data that would appear in CLA are put in a text file, one line per process. In this case an extra step is needed: processes are put in a temporary group, and depending on their MPI rank in this group, pick an UER, directing them to the appropriate line of the file.

The communicators each process belongs to will be determined at run time by its CLA. Processes spawned either from the same component (SPMD) or from different components (MPMD) may be given the same color for a split and will be in the same communicator. In either case, processes involved will refer to the same actual context. In the former case (SPMD) all have the same virtual context, but in the latter (MPMD) only processes spawned from the same component will have the same virtual context. Processes spawned from the same component may also be organized into different contexts. In this case the virtual context of processes will correspond to a number of actual contexts. From each process's point of view however its virtual context will correspond to exactly one actual context.

In the climate configuration of figure 1 we have two actual contexts atm-plane and ocn-plane corresponding to virtual contexts of atm and ocn respectively. Also groups of six atm processes and one ocn process are in a context of their own, associated with some color C. The X virtual context of the six atm processes and the Y of the ocn process will refer to the same actual context created by color C. If the context of an atm process is NULL it indicates that there it is not coupled with any ocn process. A process cannot (and does not need to) determine if this is because the atm model runs independently or because of the physical constraints.

Table 3. 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
     Ports Up[1..N];

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), &SendReq[0]);
 MPI_ISend(SouthData, n, MPI_Float, ENVPort(South,1,Ocn), &SendReq[1]);
 MPI_ISend(EastData,  m, MPI_Float, ENVPort(East,1,Ocn),  &SendReq[2]);
 MPI_ISend(WestData,  m, MPI_Float, ENVPort(West,1,Ocn),  &SendReq[3]);

 MPI_IRecv(NorthData, n, MPI_Float, ENVPort(North,1,Ocn), &RecvReq[0]);
 MPI_IRecv(SouthData, n, MPI_Float, ENVPort(South,1,Ocn), &RecvReq[1]);
 MPI_IRecv(EastData,  m, MPI_Float, ENVPort(East,1,Ocn),  &RecvReq[2]);
 MPI_IRecv(WestData,  m, MPI_Float, ENVPort(West,1,Ocn),  &RecvReq[3]);
 MPI_Waitall(4, &RecvReq, &RecvStatus);

 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,&SendReq,&SendStatus);
};/* while not done */
```

6 Conclusions

We presented modular MPI components, which may be combined in various configurations (SPMD, MPMD, regular, irregular). However, their code and particular their communication must be compatible to guarantee correct behavior. In that sense modules cannot be developed altogether independently. Ensemble provides the architecture for developing modular components or modifying existing programs with the desired generality for composition. Components may then be composed in various configurations without further modifications. Other compatible components (e.g. land model) may be also coupled with already existing ones (e.g. atm and ocn).

Compatibility in general is a dynamic property and is not restricted to the static compatibility of channel binding. We are developing lightweight formal methods used in synergy with program execution to test module compatibility and debug application composition [14].

Our aim is to be as close as possible to MPI both syntactically and semantically, so that we may use all its capabilities and tools (analysis, visualization, etc.). Composed programs are pure MPI programs; we do not use any external environment for gluing components together. Other approaches dealing with a broader problem, that of the composition of heterogeneous components, develop component architectures (CCA [15], Charisma [1]). They manage "componentized" programs (mainly using OO techniques) as well as their communication. Although component architectures are successful in many respects, each has its own limitations in supporting MPI programs.

Application composition, in general, requires a composition environment, in which composition directives are specified outside the modules themselves. This is against SPMD practice, by which all programming is expressed in one source code (computations, topology generation, load balancing, etc.). In Ensemble we have two such environments. The first is the Low Level composition with RSL scripts, not convenient to use, leaving a lot of responsibilities and space for errors to the programmer. The other, the High Level Composition Tool, which uses virtual envelope variables and symbolic names for processes, roots and contexts (virtual and symbolic). We have experimented with a number of them (supporting grammatical, graphical, GUI based directives), each having its own advantages and disadvantages. We currently develop a new HLCT, in which composition directives are specified grammatically and as close as possible to a pseudo-SPMD style of programming. This new composition tool will also manage components, executables, etc as grid resources using web-services.

Performance is in the core of parallel programming. No execution overhead is introduced by Ensemble, as envelope bindings are done at pre-processing and compile time. The only cause for overhead is the execution of SetEnvArgs, which mainly computes that would have been coded anyway (construct communicators, find process rank, etc). Actually, we eliminate function evaluations determining neighboring processes each time a send/recv is invoked. The only overhead is computing the associations of UER and MPI rank (e.g allgather call, internal table creation).

We plan to re-engineer SPMD programs as modular components; to extend the component communication interface with associations of data sub-domains to ports to deal with the coupling of M atm to N ocn processes [16]. Our objective is to define libraries for managing unstructured communication of coupled processes conveniently. Finally we plan to address dynamically configurable applications by modifying EnvArgs at run time, redirecting ports and decoupling/recoupling processes.

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] 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.

[3] Cotronis, J.Y. (1998) Developing Message Passing Applications on MPICH under Ensemble, in Proc. of PVM/MPI'98, LNCS 1497, 145-152, Springer.

[4] Cotronis, J.Y, (2002) Modular MPI Components and the Composition of Grid Applications, Proc. PDP 2002, IEEE Press pp 154-161.

[5] Cotronis, J.Y., Tsiatsoulis Z., Modular MPI and PVM components, PVM/MPI'02, LNCS 2474, pp. 252-259, Springer.

[6] Foster, I. (1995) Designing and Building Parallel Programs, Addison-Wesley Publishing Company, ISBN 0-201-57594-9.

[7] Foster, I., Kesselman, C (eds.) The Grid, Blueprint for the New Computing Infrastructure, Morgan Kaufmann, 1999.

[8] Foster, I., Geisler, J, Gropp, W, Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke S.: Wide-Area Implementation of the Message Passing Interface, Parallel Computing, 24(12):1735-1749, 1998.

[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. www.globus.org

[11] Gropp, W. and Lusk, E. (1999) User's Guide for mpich, a Portable Implementation of MPI, ANL/MCS-TM-ANL-96/6 Rev B

[12] Message Passing Interface Forum (1994) MPI: A Message Passing Interface Standard.

[13] MPICH-G2, http://www.hpclab.niu.edu/mpi/g2_body.html

[14] Tsiatsoulis Z., Cotronis J.Y.: Testing and Debugging Message Passing Programs in Synergy with their Specifications, Fundamenta Informaticae 41, No 3 (February 2000) pp. 341-366.

[15] http://www.csm.ornl.gov/cca/

[16] http://www.csm.ornl.gov/cca/mxn/