Accepted and will be presented in Euro PVM-MPI 2002

Modular MPI and PVM Components

Yiannis Cotronis and Zacharias Tsiatsoulis

Department of Informatics and Telecommunications, Univ. of Athens, 157 84 Athens, GR.
{cotronis, zack}@di.uoa.gr

Abstract. In the Ensemble methodology message passing applications are built from separate modular components. Processes spawned from modular components specify open communication interfaces (point-point or collective), which are bound at run time according to application composition directives. We give an overview of the concepts and tools. We present and compare the design of modular PVM and MPI components.

1 Introduction

In the Ensemble methodology [1,2,3,4] message passing (MP) applications are designed and built by composing modular MP components. We have developed tools for: designing and implementing separate components; designing topologies; specifying allocation resources; generating composition directives. These tools comprise the Ensemble Composition Architecture, which has been developed on top of PVM [8] and MPI [10,11]. Ensemble designs reduce software engineering costs when compared with direct design and implementation on PVM or MPI. One advantage is that Ensemble distinguishes design and implementation issues; another is that it provides support for irregular applications, whether SPMD or MPMD. All advantages however stem from the fact that components are developed separately as independent modules and that compatible modules can be composed in any configuration without any modification.

One major task of a MP designer and programmer is to code in a program P the interactions of all processes spawned from P, in all possible positions in the process topology and for any size of the topology. Message passing APIs require specific process identifiers. The process identifiers are specified either directly (e.g. tids or ranks) or indirectly by functions, which presuppose a specific (usually regular) topology. Grids [6] impose additional requirements for program modularity, since applications may need to couple components developed by different teams. An application may 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).

Previous Ensemble implementations [1,2,3] were aiming at developing abstract component code capable of running on any API (PVM, MPI, Parix). Consequently program components could only use a limited, semantically common, subset of routines (e.g. send, receive, broadcast, barrier). In this paper we distinguish MPI and

PVM modular components fully supporting point-point as well as collective communication in each of the two APIs.

The structure of the paper is as follows: In section 2 we present the structure of modular MPI components by example (downsized atmospheric and ocean modules); in 3 we outline PVM modular components; finally in section 4 we present our conclusions and plans for future work.

2 Modular MPI Components

Modular components are the heart of the Ensemble methodology; they are implementation abstractions of MP programs. They specify communication leaving all envelope data that is related to the origin and destination of messages (i.e. context, rank, message tag and root) unspecified. We use a data structure for storing all envelope related data, namely EnvArgs. All envelope parameters in MPI calls bind to appropriate elements of EnvArgs. For convenience we use virtual envelopes and macros, which transparently bind envelope data to EnvArgs elements. Pure MPI code is generated (envelope parameters are given proper bindings) by expanding the macros and compiled to executable modular components.

The setting of actual envelope data is done dynamically for each process. When processes are spawned their EnvArgs structure is empty. After MPI_Init each process must set values to its EnvArgs by calling SetEnvArgs, which parses and interprets (a list of) command line arguments (CLAs). The CLAs are composition directives related to a specific process and are usually generated from a tool for designing and building applications (experienced Ensemble programmers produce them directly).

In the sequel we present modular MPI components without elaborating on implementation details. We design two modular components Atm and Ocn each solving a finite difference problem, assumed to be “downscaled” versions of Atmosphere and the Ocean models respectively. Each component is required to run separately as SPMD, but also may be coupled together for solving the climate model.

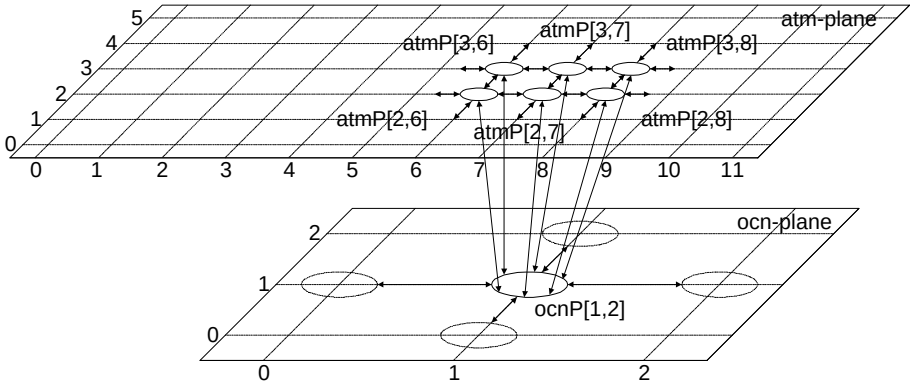

Fig. 1. The coupled climate model

Our aim is to design the Atm and Ocn components so that, on the one hand they may execute independently and, on the other to be coupled in any desirable configuration, either because of land and ocean constraints or load balancing considerations. Although, we can assume a rectangular shape for the Atm processes, this may not be true for Ocn processes, as it depends on the actual ocean-land boundaries. For this purpose we may associate Ocn processes with a variant number of Atm processes.

In figure 1 we show a possible configuration of coupling Atm and Ocn processes together solving the climate model for a given region. In the two planes we depict the two distinct SPMD applications. Only Atm processes in the eastern part of atm-plane are coupled with Ocn processes, assuming 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-six correspondence. One Ocn corresponds to six Atm processes; this is a design decision aiming to balance the load, as the atmosphere model is computationally more demanding than the ocean model.

2.1 Ensemble Atm and Ocn Components

The virtual envelope of Atm requires two contexts, Atm for processes involved in the atmospheric calculations and context X for the co-ordination of atmospheric process with corresponding ones, e.g. ocean or land calculations. Context X may involve ocean or land or be NULL; its value will depend on the application configuration. Respectively the Ocn component requires two contexts, Ocn for the ocean model calculation and Y for possible coupling with corresponding Atm processes (or some other for testing).

The actual context of each process spawned from a component will be determined at run time. As communicator types cannot be passed directly as CLAs, we use integers, which SetEnvArgs interprets as color values and creates corresponding communicators by splitting MPI_WORLD_COMM. Processes given the same color will be in the same context. The same color may be passed to processes spawned either from the same component (SPMD) or from different components (MPMD). In the former case the virtual context of the single component involved will refer to the same actual context, but in the latter the virtual context of each of the components involved will refer to the same actual context. Also processes spawned from the same component may be organized into different contexts. In this case the virtual context will not correspond to a single actual contexts; from each process's point of view however its virtual context will correspond to exactly one actual context. The composition directives determine actual contexts. In the configuration of figure 1 we have two contexts atm-plane and ocn-plane corresponding to contexts atm and ocn respectively. The groups of these two contexts consist of atm and respectively ocn processes. But groups of six atm processes with one ocn process will be in their own context, 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 actual context created from 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 atmospheric model runs independently or because a process of physical constraints.

Table 1. The Virtual Envelope and Ensemble Code of Atm in MPI

Ensemble Atm Component

```
Virtual Envelope
  Context Atm
    Ports
      North[0..1]; South[0..1]; East[0..1]; West[0..1]
  Context X
    Ports
      Down [0..1]
  Arguments
    Threshold; InputFile; OutputFile
```

```
Code
/* Declarations omitted */
MPI_Init(&argc, &argv);
SetEnvArgs(&argc, &argv);
while (!done) {
 MPI_ISend(NData,N,MPI_Float,ENVPort(North,1,Atm),&S[0]);
 MPI_ISend(SData,N,MPI_Float,ENVPort(South,1,Atm),&S[1]);
 MPI_ISend(EData,M,MPI_Float,ENVPort(East,1,Atm),&S[2]);
 MPI_ISend(WData,M,MPI_Float,ENVPort(West,1,Atm),&S[3]);
 MPI_IRecv(NData,N,MPI_Float,ENVPort(North,1,Atm),&R[0]);
 MPI_IRecv(SData,N,MPI_Float,ENVPort(South,1,Atm),&R[1]);
 MPI_IRecv(EData,M,MPI_Float,ENVPort(East,1,Atm),&R[2]);
 MPI_IRecv(WData,M,MPI_Float,ENVPort(West,1,Atm),&R[3]);
 MPI_Waitall(4,&R,&RecvStatus);

 AtmComputations(&LErr);

 MPI_AllReduce(&MaxErr,&LErr,1,MPI_Float,MPI_MAX,
                                          ENVComm(Atm));
 if (MaxErr < threshold) done=true;
/* 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(BDat,L,MPI_FLOAT,ENVport(Down,1,X),&SD);
   MPI_IRecv(BDat,L,MPI_FLOAT,ENVport(Down,1,X),&RD);
   MPI_Wait(&RD,&RDstatus); MPI_Wait(&SD,&SDstatus);
  }
 }/* End of Interactions with X */
 MPI_Waitall(4,&S,&SendStatus);
}/* while not done */
```

Within a virtual envelope roots for reductions and ports for point-point interaction are defined. A port is an abstraction of the envelope triplet (context, rank, message tag). Virtual ports with the same semantics are treated as an array of ports (MultiPort).

Table 2. The Virtual Envelope and Ensemble Code of Atm in MPI

Ensemble Ocean Component

```
Virtual Envelope
  Context Ocn
    Ports
      North[0..1]; South[0..1]; East[0..1];West[0..1]
  Context Y
    Ports
      Up[0..N]
Arguments
  Threshold; InputFile; OutputFile
```

```
Code
/* Declarations omitted */
MPI_Init(&argc, &argv);
SetEnvArgs(&argc, &argv);
while (!done) {
 /*Internal Ocean Communications similar to Atm*/

 OcnComputations(&LErr);

 MPI_AllReduce(&MaxErr,&LErr,1,MPI_Float,MPI_MAX,
                                          ENVComm(Ocn));
 if (MaxErr < threshold) done=true;
/* 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(T[i],L,MPI_FLOAT,ENVport(Up,i,Y),&U[i-1]);
    MPI_IRecv(T[i],L,MPI_FLOAT,ENVport(Up,i,Y),&Q[i-1);
   }
   MPI_Waitall(ENVportN(Up,Y),&Q,&Qstatus);
   MPI_Waitall(ENVportN(Up,Y),&U,&Ustatus);
  }
 }/* End of Interactions with Y */
 MPI_Waitall(4,&S,&SendStatus);
}/* while not done */
```

In the Atm group there are four multiports; each process may have zero or one port depending on its position on the plane. Note that in context Y of Ocn multiport Up may have up to N ports, the number of corresponding Atm processes.

Following the virtual envelope we specify arguments, which correspond to command line arguments needed in the calculations. In the case of Atm and Ocn components they specify I/O files and the criterion for termination (threshold). When

the two models are coupled the same value must be passed to all atm and ocn processes, otherwise the program will deadlock.

The code of Atm and Ocn components looks like an MPI program, but all envelope-related arguments in MPI communication and synchronization calls are expressed as macros referring to virtual envelope names. Macros in atm and ocn code are shown in italics. All other arguments have the usual bindings. All envelop arguments of point-point communication (Rank, Message Tag, Communicator) refer to virtual envelope names by a macro e.g. ENVPort(North, 1, Atm). This macro refers to port 1 of multiport North within the Atm group. Macro ENVComm(X), refers to the actual communicator corresponding to virtual context X. A third macro ENVroot(Vcomm,Vroot), which is not used here, refers to roots. Finally ENVportN(Up,Y) corresponds to the value of ports in Up. With these macros MPI calls refer to virtual envelope names and upon expansion generate proper MPI bindings. Thus all possible communication is expressed in the code (types, order and number of messages), but no information about the receiver or the originator of messages. The code resembles the task/channel model [5], but in a two stage manner. Stage one 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.

The composition of applications is specified in two levels: in a High Level Composition Tool in which symbolic names for processes, roots, groups, etc. are used. At this level the designer puts components together. In our example, the number of atm and ocn processes, as well as the shape of the two SPMD topologies are set; also the correspondence between atm and ocn processes, etc. This tool interprets the design and generates globus RSL [9] scripts, as low-level composition directives. Executing the RSL scripts the application is composed in MPICH-G [7,12]. More details may be found in [4].

3. Modular PVM Components

For PVM components the concept is the same as for MPI components: only capability of communication of processes is specified whereas all envelope data are left unspecified until process spawning. However, we need a different EnvArgs structure, reflecting the arguments of PVM calls and the relation of context, groups, roots, message tags and ports. Consequently, we need different set of macros, SetEnvArgs and argument list. These variations are reflected in the virtual envelope of PVM and MPI components. In the following section we present the virtual envelop of Atm and Ocn components in PVM, and only give some code segments demonstrating the use of the corresponding macros.

The virtual components in PVM just as their MPI counterparts consist of the envelope, the arguments and the source code. The main difference between the envelope for PVM and the envelope for MPI, results from the fact that PVM has not one integrated concept of group and context information whereas MPI has the communicator. In PVM groups may be constructed dynamically and operations inside the group do not need to have a common context. PVM contexts are not associated

with a specific group, and may be applied independently to a communication operation by calling a routine to set the appropriate context. Furthermore collective communication requires message tags and groups. As a result, the virtual envelope for PVM specifies names of ports for point-point operations, multicast operations, contexts and groups and within groups all relative information: group operations (reductions, broadcasts, barriers), roots, tags, etc. The virtual envelope of Atm and Ocn for PVM are presented in table **3**.

Table 3. Virtual Envelopes of Atm and Ocean Components in PVM

Atm Virtual Envelope	Ocean Virtual Envelop

`Port`
`  North[0..1]; South[0..1];`
`  East[0..1]; West[0..1];`
`  Down[0..1];`
`  Bcast[0..1]`
`Group Atm`
`  Reduction CalcMax`
`  Broadcast BcastMax`
`Context DEFAULT`
	`Port`
	`  North[0..1];South[0..1];`
	`  East[0..1];West[0..1];`
	`  Up[0..N];`
	`  Bcast[0..1]`
	`Group  Ocn`
	`  Reduction CalcMaximum`
	`  Broadcast BcastMaximum`
	`Context  DEFAULT`

Ports are specified independently whereas in the MPI virtual component ports are specified inside some context. Port Bcast is added here as broadcast messages in PVM are received by usual receive operations. Furthermore, a group atm (and respectively ocn) is specified, which inside contains a virtual name for the reduction and broadcast envelope (message tag). This group corresponds to the atm (and respectively ocn) communicator in MPI, but decoupled from a specific context. Only the DEFAULT context is specified, which is common for all components. No other contexts are required. Note that all virtual envelope names are local to the component.

The arguments for the PVM and MPI virtual components are identical (threshold and I/O files). The source code for the components is very close to PVM code In Table 4 some PVM communication calls are shown.

Table 4. Some PVM calls and macros

Abstract communication calls in Atm code

`pvm_send(`***`ENVPort(Down,1)`***`);`
`pvm_recv(`***`ENVPort(East,1)`***`);`
`pvm_reduce(PvmMax,&LErr,1,PVM_INT,`***`ENVRed(Atm,CalcMax)`***`);`

The macro ENVPort(East,1) refers to port 1 of multiport East. In contrast to its corresponding macro for MPI components, there is no context associated with the port. Similarly, the macro ENVRed(Atm,CalcMax) refers to the reduction envelope data (group name and message tag) specified by name CalcMax inside group Atm.

The High Level Composition Tool generates low-level composition directives for PVM, which are interpreted by a PVM program spawning all PVM processes with the appropriate command line arguments.

4 Conclusions

We presented MPI and PVM modular components under the Ensemble methodology. Ensemble modules (e.g. atm and ocn) are independent, but not necessarily independently developed. They are independent in the sense that their processes may be combined in many ways or not at all. However, their code must guarantee compatibility and correct performance. In general applications need to be modified if they are to be coupled together. Ensemble provides the architecture for developing modular components with the desired generality for composition. Components may then be composed without any further modifications. Other compatible components (e.g. land) may be also coupled with already existing ones.

Other benefits: tracing and debugging may use symbolic names of processes, roots and contexts (virtual and symbolic), e.g. “atm[2,6] sends to ocn[1,2] in context X”. No execution overhead is introduced, as envelope bindings are done at compile time.We currently develop a new composition environment, which manages components, executables, etc as grid resources and study formal support for compatibility and composition.Our future plans are to re-engineer SPMD programs as modular components; also to address dynamically configurable applications by modifying EnvArgs during execution.

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

References

1. 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, Berlin, 74-85, Chapman & Hall.
2. Cotronis, J.Y.:(1997) Message Passing Program Development by Ensemble, Proc. PVM/MPI’97, LNCS 1332, 242-249, Springer.
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. Foster, I. (1995) Designing and Building Parallel Programs, Addison-Wesley Publishing Company, ISBN 0-201-57594-9.
6. Foster, I., Kesselman, C (eds.) The Grid, Blueprint for the New Computing Infrastructure, Morgan Kaufmann, 1999.
7. 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.
8. 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.
9. Globus Quick Start Guide, Globus Software version 1.1.3 and 1.1.4, February 2001. www.globus.org
10. Gropp, W. and Lusk, E. (1999) User's Guide for mpich, a Portable Implementation of MPI, ANL/MCS-TM-ANL-96/6 Rev B
11. Message Passing Interface Forum (1994) MPI: A Message Passing Interface Standard.
12. MPICH-G2, http://www.hpclab.niu.edu/mpi/g2_body.html